SECOND EDITION

COMPUTER SECURITY

[ART and SCIENCE]

MATT BISHOP

With contributions from ELISABETH SULLIVAN and MICHELLE RUPPEL

Computer Security

Second Edition

This page intentionally left blank

Computer Security
Art and Science

Second Edition

Matt Bishop

with contributions from

Elisabeth Sullivan and Michelle Ruppel

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2018950017

ISBN-13: 978-0-321-71233-2
ISBN-10: 0-321-71233-1

1 18

To my dear Holly; our children Heidi, Steven, David, and Caroline; our grandchildren Skyler and Sage; and our friends Seaview, Tinker Belle, Stripe, Baby Windsor, Scout, Fur, Puff, Mouse, Shadow, Fuzzy, Dusty, and the rest of the menagerie.

This page intentionally left blank

Contents

PART II : FOUNDATIONS 29

This page intentionally left blank

Preface

HORTENSIO: Madam, before you touch the instrument
To learn the order of my fingering,
I must begin with rudiments of art
To teach you gamouth in a briefer sort,
More pleasant, pithy and effectual,
Than hath been taught by any of my trade;
And there it is in writing, fairly drawn.
— *The Taming of the Shrew*, III, i, 62–68.

Preface to the Second Edition

Since the first edition of this book was published, the number of computer and information security incidents has increased dramatically, as has their seriousness. In 2010, a computer worm infected the software controlling a particular type of centrifuge used in uranium-enrichment sites [1116, 1137]. In 2013, a security breach at Target, a large chain of stores in the United States, compromised 40 million credit cards [1497, 1745, 2237]. Also in 2013, Yahoo reported that an attack compromised more than 1 billion accounts [779]. In 2017, attackers spread ransomware that crippled computers throughout the world, including computers used in hospitals and telecommunications companies [1881]. Equifax estimated that attackers also compromised the personal data of over 100,000,000 people [176].

These attacks exploit vulnerabilities that have their roots in vulnerabilities of the 1980s, 1970s, and earlier. They seem more complex because systems have become more complex, and thus the vulnerabilities are more obscure and require more complex attacks to exploit. But the principles underlying the attacks, the vulnerabilities, and the failures of the systems have not changed—only the arena in which they are applied has.

Consistent with this philosophy, the second edition continues to focus on the principles underlying the field of computer and information security. Many newer examples show how these principles are applied, or not applied, today; but the principles themselves are as important today as they were in 2002, and earlier. Some have been updated to reflect a deeper understanding of people and systems. Others have been applied in new and interesting ways. But they still ring true.

That said, the landscape of security has evolved greatly in the years since this book was first published. The explosive growth of the World Wide Web, and the consequent explosion in its use, has made security a problem at the forefront of our society. No longer can vulnerabilities, both human and technological, be relegated to the background of our daily lives. It is one of the elements at the forefront, playing a role in everyone's life as one browses the web, uses a camera to take and send pictures, and turns on an oven remotely. We grant access to our personal lives through social media such as Facebook, Twitter, and Instagram, and to our homes through the Internet of Things and our connections to the Internet. To ignore security issues, or consider them simply ancillary details that "someone will fix somehow" or threats unlikely to be realized personally is dangerous at best, and potentially disastrous at worst.

Ultimately, little has changed. The computing ecosystem of our day is badly flawed. Among the manifestations of these technological flaws are that security problems continue to exist, and continue to grow in magnitude of effect. An interesting question to ponder is what might move the paradigm of security away from the cycle of "patch and catch" and "let the buyer beware" to a stable and safer ecosystem.

But we must continue to improve our understanding of, and implementation of, security. Security nihilism—simply giving up and asserting that we cannot make things secure, so why try—means we accept these invasions of our privacy, our society, and our world. Like everything else, security is imperfect, and always will be—meaning we can improve the state of the art. This book is directed towards that goal.

Updated Roadmap

The dependencies of the chapters are the same as in the first edition (see p. xl), with two new chapters added.

Chapter 7, which includes a discussion of denial of service attack models, contains material useful for Chapters 23, 24, 27, and 28. Similarly, Chapter 27 draws on material from the chapters in Part III as well as Chapters 23, 25, 26, and all of Part VIII.

In addition to the suggestions in the preface to the first edition on p. xli about topics for undergraduate classes, the material in Chapter 27 will introduce undergraduates to how attacks occur, how they can be analyzed, and what their effects are. Coupled with current examples drawn from the news, this chapter should prove fascinating to undergraduates.

As for graduate classes, the new material in Chapter 7 will provide students with some background on resilience, a topic increasing in importance. Otherwise, the recommendations are the same as for the first edition (see p. xlii).

Changes to the First Edition

The second edition has extensively revised many examples to apply the concepts to technologies, methodologies, and ideas that have emerged since the first edition was published. Here, the focus is on new material in the chapters; changes to examples are mentioned only when necessary to describe that material. In addition to what is mentioned here, much of the text has been updated.

Chapter 1, "An Overview of Computer Security": This chapter is largely unchanged.

Chapter 2, "Access Control Matrix": Section 2.2.2, "Access Controlled by History" has been changed to use the problem of preventing downloaded programs from accessing the system in unauthorized ways, instead of updating a database. Section 2.4.3, "Principle of Attenuation of Privilege," has been expanded slightly, and exercises added to point out differing forms of the principle.

Chapter 3, "Foundational Results": Definition 3–1 has been updated to make clear that "leaking" refers to a right being added to an element of the access control matrix that did not contain it initially, and an exercise has been added to demonstrate the difference between this definition and the one in the first edition. Section 3.6 discusses comparing security properties of models.

Chapter 4, "Security Policies": Section 4.5.1, "High-Level Policy Languages," now uses Ponder rather than a Java policy constraint language. Section 4.6, "Example: Academic Computer Security Policy," has been updated to reflect changes in the university policy.

Chapter 5, "Confidentiality Policies": Section 5.3.1 discusses principles for declassifying information.

Chapter 6, "Integrity Policies": Section 6.5 presents trust models.

Chapter 7, "Availability Policies": This chapter is new.

Chapter 8, "Hybrid Policies": Section 8.1.3 modifies one of the assumptions of the Chinese Wall model that is unrealistic. Section 8.3.1 expands the discussion of ORCON to include DRM. Section 8.4 adds a discussion of several types of RBAC models.

Chapter 9, "Noninterference and Policy Composition": This chapter adds Section 9.6, which presents side channels in the context of deducibility.

Chapter 10, "Basic Cryptography": This chapter has been extensively revised. The discussion of the DES (Section 10.2.3) has been tightened and the algorithm moved to Appendix F. Discussions of the AES (Section 10.2.5) and elliptic curve cryptography (Section 10.3.3) have been added, and the section on digital signatures moved from Chapter 11 to Section 10.5. Also, the number of digits in the integers used in examples for public key cryptography has been increased from 2 to at least 4, and in many cases more.

Chapter 11, "Key Management": Section 11.4.3 discusses public key infrastructures. Section 11.5.1.4, "Other Approaches," now includes a brief discussion of identity-based encryption.

Chapter 12, "Cipher Techniques": Section 12.1, "Problems," now includes a discussion of type flaw attacks. Section 12.3 discusses authenticated encryption with associated data, and presents the CCM and GCM modes of block ciphers. A new section, Section 12.5.2, discusses the Signal Protocol. Section 12.5.3, "Security at the Transport Layer: TLS and SSL," has been expanded and focuses on TLS rather than SSL. It also discusses cryptographic weaknesses in SSL, such as the POODLE attack, that have led to the use of SSL being strongly discouraged.

Chapter 13, "Authentication": A discussion of graphical passwords has been added as Section 13.3.4. Section 13.4.3 looks at quantifying password strength in terms of entropy. The discussion of biometrics in Section 13.7 has been expanded to reflect their increasing use.

Chapter 14, "Design Principles": The principle of least authority follows the principle of least privilege in Section 14.2.1, and the principle of least astonishment now supersedes the principle of psychological acceptability in Section 14.2.8.

Chapter 15, "Representing Identity": Section 15.5, "Naming and Certificates," now includes a discussion of registration authorities (RAs). Section 15.6.1.3 adds a discussion of the DNS security extensions (DNSSEC). Section 15.7.2 discusses onion routing and Tor in the context of anonymity.

Chapter 16, "Access Control Mechanisms": Section 16.2.6 discusses sets of privileges in Linux and other UNIX-like systems.

Chapter 17, "Information Flow": In contrast to the confidentiality-based context of information flow in the main part of this chapter, Section 17.5 presents information flow in an integrity context. In Section 17.6, the SPI and SNSMG examples of the first edition have been replaced by Android cell phones (Section 17.6.1) and firewalls (Section 17.6.2).

Chapter 18, "Confinement Problem": Section 18.2 has been expanded to include library operating systems (Section 18.2.1.2) and program modification techniques (Section 18.2.2).

Chapter 19, "Introduction to Assurance": Section 19.2.3, which covers agile software development, has been added.

Chapter 20, "Building Systems with Assurance": The example decomposition of Windows 2000 into components has been updated to use Windows 10.

Chapter 21, "Formal Methods": A new section, Section 21.5, discusses functional programming languages, and another new section, 21.6, discusses formally verified products.

Chapter 22, "Evaluating Systems": Sections 22.7, on FIPS 140, and 22.8, on the Common Criteria, have been extensively updated.

Chapter 23, "Malware": Section 23.5 presents botnets, and Sections 23.6.3, 23.6.4, 23.6.5, and 23.6.6 discuss adware and spyware, ransomware, and phishing. While not malware, phishing is a common vector for getting malware onto a system and so it is discussed here.

Chapter 24, "Vulnerability Analysis": Section 24.2.5 reviews several penetration testing frameworks used commercially and based on the Flaw Hypothesis Methodology. Section 24.5 presents the widely used CVE and CWE standards.

Chapter 25, "Auditing": Section 25.3.3, which discusses sanitization, has been expanded.

Chapter 26, "Intrusion Detection": Section 26.3.1 has been expanded to include several widely used machine learning techniques for anomaly detection. Incident response groups are discussed in Section 27.3.

Chapter 27, "Attacks and Responses": This chapter is new.

Chapter 28, "Network Security": The discussion of what firewalls are has been moved to Section 17.6.2, but the discussion of how the Drib configures and uses them remains in this chapter. The Drib added wireless networks, which are discussed in Section 28.3.3.1. Its analysis of using the cloud is in Section 28.3.3.2. The rest of the chapter has been updated to refer to the new material in previous chapters.

Chapter 29, "System Security": This chapter has been updated to refer to the new material in previous chapters.

Chapter 30, "User Security": Section 30.2.2 describes the two-factor authentication procedure used by the Drib. The rest of the chapter has been updated to refer to the new material in previous chapters.

Chapter 31, "Program Security": This chapter has been updated to refer to the new material in previous chapters.

Two new appendices have been added. Appendix F presents the DES and AES algorithms, and Appendix H collects the rules in Chapter 31 for easy reference. In addition, Appendix D examines some hardware enhancements to aid virtualization, and Appendix G contains the full academic security policy discussed in Section 4.6.

Preface to the First Edition[1]

On September 11, 2001, terrorists seized control of four airplanes. Three were flown into buildings, and a fourth crashed, with catastrophic loss of life. In the aftermath, the security and reliability of many aspects of society drew renewed scrutiny. One of these aspects was the widespread use of computers and their interconnecting networks. The issue is not new. In 1988, approximately 5,000 computers throughout the Internet were rendered unusable within 4 hours by a program called a worm [842].[2] While the spread, and the effects, of this program alarmed computer scientists, most people were not worried because the worm did not affect their lives or their ability to do their jobs. In 1993, more users of computer systems were alerted to such dangers when a set of programs called sniffers were placed on many computers run by network service providers and recorded login names and passwords [670].

After an attack on Tsutomu Shimomura's computer system, and the fascinating way Shimomura followed the attacker's trail, which led to his arrest [1736], the public's interest and apprehension were finally aroused. Computers were now vulnerable. Their once reassuring protections were now viewed as flimsy.

Several films explored these concerns. Movies such as *War Games* and *Hackers* provided images of people who can, at will, wander throughout computers and networks, maliciously or frivolously corrupting or destroying information it may have taken millions of dollars to amass. (Reality intruded on *Hackers* when the World Wide Web page set up by MGM/United Artists was quickly altered to present an irreverent commentary on the movie and to suggest that viewers see *The Net* instead. Paramount Pictures denied doing this [869].) Another film, *Sneakers*, presented a picture of those who test the security of computer (and other) systems for their owners and for the government.

Goals

This book has three goals. The first is to show the importance of theory to practice and of practice to theory. All too often, practitioners regard theory as irrelevant and theoreticians think of practice as trivial. In reality, theory and practice are symbiotic. For example, the theory of covert channels, in which the goal is to limit the ability of processes to communicate through shared resources, provides a mechanism for evaluating the effectiveness of mechanisms that confine processes, such as sandboxes and firewalls. Similarly, business practices in the commercial world led to the development of several security policy models such as the Clark-Wilson model and the Chinese Wall model. These models in turn help the designers of security policies better understand and evaluate the mechanisms and procedures needed to secure their sites.

[1]Chapter numbers have been updated to correspond to the chapters in the second edition.
[2]Section 23.4 discusses computer worms.

The second goal is to emphasize that computer security and cryptography are different. Although cryptography is an essential component of computer security, it is by no means the only component. Cryptography provides a mechanism for performing specific functions, such as preventing unauthorized people from reading and altering messages on a network. However, unless developers understand the context in which they are using cryptography, and unless the assumptions underlying the protocol and the cryptographic mechanisms apply to the context, the cryptography may not add to the security of the system. The canonical example is the use of cryptography to secure communications between two low-security systems. If only trusted users can access the two systems, cryptography protects messages in transit. But if untrusted users can access either system (through authorized accounts or, more likely, by breaking in), the cryptography is not sufficient to protect the messages. The attackers can read the messages at either endpoint.

The third goal is to demonstrate that computer security is not just a science but also an art. It is an art because no system can be considered secure without an examination of how it is to be used. The definition of a "secure computer" necessitates a statement of requirements and an expression of those requirements in the form of authorized actions and authorized users. (A computer engaged in work at a university may be considered "secure" for the purposes of the work done at the university. When moved to a military installation, that same system may not provide sufficient control to be deemed "secure" for the purposes of the work done at that installation.) How will people, as well as other computers, interact with the computer system? How clear and restrictive an interface can a designer create without rendering the system unusable while trying to prevent unauthorized use or access to the data or resources on the system?

Just as an artist paints his view of the world onto canvas, so does a designer of security features articulate his view of the world of human/machine interaction in the security policy and mechanisms of the system. Two designers may use entirely different designs to achieve the same creation, just as two artists may use different subjects to achieve the same concept.

Computer security is also a science. Its theory is based on mathematical constructions, analyses, and proofs. Its systems are built in accordance with the accepted practices of engineering. It uses inductive and deductive reasoning to examine the security of systems from key axioms and to discover underlying principles. These scientific principles can then be applied to untraditional situations and new theories, policies, and mechanisms.

Philosophy

Key to understanding the problems that exist in computer security is a recognition that the problems are not new. They are old problems, dating from the beginning of computer security (and, in fact, arising from parallel problems in the noncomputer world). But the locus has changed as the field of computing has changed. Before the mid-1980s, mainframe and mid-level computers dominated the market, and computer security problems and solutions were phrased in terms of securing files or processes on a single system. With the rise of networking and the Internet, the arena has changed. Workstations and servers, and the networking infrastructure that connects them, now dominate the market. Computer security problems and solutions now focus on a networked environment. However, if the workstations and servers, and the supporting network infrastructure, are viewed as a single system, the models, theories, and problem statements developed for systems before the mid-1980s apply equally well to current systems.

As an example, consider the issue of assurance. In the early period, assurance arose in several ways: formal methods and proofs of correctness, validation of policy to requirements, and acquisition of data and programs from trusted sources, to name a few. Those providing assurance analyzed a single system, the code on it, and the sources (vendors and users) from which the code could be acquired to ensure that either the sources could be trusted or the programs could be confined adequately to do minimal damage. In the later period, the same basic principles and techniques apply, except that the scope of some has been greatly expanded (from a single system and a small set of vendors to the world-wide Internet). The work on proof-carrying code, an exciting development in which the proof that a downloadable program module satisfies a stated policy is incorporated into the program itself, is an example of this expansion.[3] It extends the notion of a proof of consistency with a stated policy. It advances the technology of the earlier period into the later period. But in order to understand it properly, one must understand the ideas underlying the concept of proof-carrying code, and these ideas lie in the earlier period.

As another example, consider Saltzer and Schroeder's principles of secure design.[4] Enunciated in 1975, they promote simplicity, confinement, and understanding. When security mechanisms grow too complex, attackers can evade or bypass them. Many programmers and vendors are learning this when attackers break into their systems and servers. The argument that the principles are old, and somehow outdated, rings hollow when the result of their violation is a nonsecure system.

The work from the earlier period is sometimes cast in terms of systems that no longer exist and that differ in many ways from modern systems. This does not vitiate the ideas and concepts, which also underlie the work done today. Once these ideas and concepts are properly understood, applying them in a multiplicity of environments becomes possible. Furthermore, the current mechanisms and technologies will become obsolete and of historical interest themselves as new forms of computing arise, but the underlying principles will live on, to underlie the next generation—indeed the next era—of computing.

The philosophy of this book is that certain key concepts underlie all of computer security, and that the study of all parts of computer security enriches

---

[3]Section 23.9.5.1 discusses proof-carrying code.
[4]Chapter 14 discusses these principles.

the understanding of all parts. Moreover, critical to an understanding of the applications of security-related technologies and methodologies is an understanding of the theory underlying those applications. Advances in the theory of computer protection have illuminated the foundations of security systems. Issues of abstract modeling, and modeling to meet specific environments, lead to systems designed to achieve a specific and rewarding goal. Theorems about composability of policies[5] and the undecidability of the general security question[6] have indicated the limits of what can be done. Much work and effort are continuing to extend the borders of those limits.

Application of these results has improved the quality of the security of the systems being protected. However, the issue is how compatibly the assumptions of the model (and theory) conform to the environment to which the theory is applied. Although our knowledge of how to apply these abstractions is continually increasing, we still have difficulty correctly transposing the relevant information from a realistic setting to one in which analyses can then proceed. Such abstraction often eliminates vital information. The omitted data may pertain to security in non-obvious ways. Without this information, the analysis is flawed.

The practitioner needs to know both the theoretical and practical aspects of the art and science of computer security. The theory demonstrates what is possible. The practical makes known what is feasible. The theoretician needs to understand the constraints under which these theories are used, how their results are translated into practical tools and methods, and how realistic are the assumptions underlying the theories. *Computer Security: Art and Science* tries to meet these needs.

Unfortunately, no single work can cover all aspects of computer security, so this book focuses on those parts that are, in the author's opinion, most fundamental and most pervasive. The mechanisms exemplify the applications of these principles.

Organization

The organization of this book reflects its philosophy. It begins with mathematical fundamentals and principles that provide boundaries within which security can be modeled and analyzed effectively. The mathematics provides a framework for expressing and analyzing the requirements of the security of a system. These policies constrain what is allowed and what is not allowed. Mechanisms provide the ability to implement these policies. The degree to which the mechanisms correctly implement the policies, and indeed the degree to which the policies themselves meet the requirements of the organizations using the system, are questions of assurance. Exploiting failures in policy, in implementation, and in assurance comes next, as well as mechanisms for providing information on the attack. The book concludes with the applications of both theory and policy focused

[5]See Chapter 9, "Noninterference and Policy Composition."
[6]See Section 3.2, "Basic Results."

on realistic situations. This natural progression emphasizes the development and application of the principles existent in computer security.

Part I, "Introduction," describes what computer security is all about and explores the problems and challenges to be faced. It sets the context for the remainder of the book.

Part II, "Foundations," deals with basic questions such as how "security" can be clearly and functionally defined, whether or not it is realistic, and whether or not it is decidable. If it is decidable, under what conditions is it decidable, and if not, how must the definition be bounded in order to make it decidable?

Part III, "Policy," probes the relationship between policy and security. The definition of "security" depends on policy. In Part III we examine several types of policies, including the ever-present fundamental questions of trust, analysis of policies, and the use of policies to constrain operations and transitions.

Part IV, "Implementation I: Cryptography," discusses cryptography and its role in security. It focuses on applications and discusses issues such as key management and escrow, key distribution, and how cryptosystems are used in networks. A quick study of authentication completes Part III.

Part V, "Implementation II: Systems," considers how to implement the requirements imposed by policies using system-oriented techniques. Certain design principles are fundamental to effective security mechanisms. Policies define who can act and how they can act, and so identity is a critical aspect of implementation. Mechanisms implementing access control and flow control enforce various aspects of policies.

Part VI, "Assurance," presents methodologies and technologies for ascertaining how well a system, or a product, meets its goals. After setting the background, to explain exactly what "assurance" is, the art of building systems to meet varying levels of assurance is discussed. Formal verification methods play a role. Part VI shows how the progression of standards has enhanced our understanding of assurance techniques.

Part VII, "Special Topics," discusses some miscellaneous aspects of computer security. Malicious logic thwarts many mechanisms. Despite our best efforts at high assurance, systems today are replete with vulnerabilities. Why? How can a system be analyzed to detect vulnerabilities? What models might help us improve the state of the art? Given these security holes, how can we detect attackers who exploit them? A discussion of auditing flows naturally into a discussion of intrusion detection—a detection method for such attacks.

Part VIII, "Practicum," presents examples of how to apply the principles discussed throughout the book. It begins with networks and proceeds to systems, users, and programs. Each chapter states a desired policy and shows how to translate that policy into a set of mechanisms and procedures that support the policy. Part VIII tries to demonstrate that the material covered elsewhere can be, and should be, used in practice.

Each chapter in this book ends with a summary, descriptions of some research issues, and some suggestions for further reading. The summary highlights the important ideas in the chapter. The research issues are current "hot topics" or are topics that may prove to be fertile ground for advancing the state of the art and science of computer security. Interested readers who wish to pursue the topics in any chapter in more depth can go to some of the suggested readings. They expand on the material in the chapter or present other interesting avenues.

Roadmap

This book is both a reference book and a textbook. Its audience is undergraduate and graduate students as well as practitioners. This section offers some suggestions on approaching the book.

Dependencies

Chapter 1 is fundamental to the rest of the book and should be read first. After that, however, the reader need not follow the chapters in order. Some of the dependencies among chapters are as follows.

Chapter 3 depends on Chapter 2 and requires a fair degree of mathematical maturity. Chapter 2, on the other hand, does not. The material in Chapter 3 is for the most part not used elsewhere (although the existence of the first section's key result, the undecidability theorem, is mentioned repeatedly). It can be safely skipped if the interests of the reader lie elsewhere.

The chapters in Part III build on one another. The formalisms in Chapter 5 are called on in Chapters 20 and 21, but nowhere else. Unless the reader intends to delve into the sections on theorem proving and formal mappings, the formalisms may be skipped. The material in Chapter 9 requires a degree of mathematical maturity, and this material is used sparingly elsewhere. Like Chapter 3, Chapter 9 can be skipped by the reader whose interests lie elsewhere.

Chapters 10, 11, and 12 also build on one another in order. A reader who has encountered basic cryptography will have an easier time with the material than one who has not, but the chapters do not demand the level of mathematical experience that Chapters 3 and 9 require. Chapter 13 does not require material from Chapter 11 or Chapter 12, but it does require material from Chapter 10.

Chapter 14 is required for all of Part V. A reader who has studied operating systems at the undergraduate level will have no trouble with Chapter 16. Chapter 15 uses the material in Chapters 10 and 11; Chapter 17 builds on material in Chapters 5, 14, and 16; and Chapter 18 uses material in Chapters 4, 14, and 17.

Chapter 19 relies on information in Chapter 4. Chapter 20 builds on Chapters 5, 14, 16, and 19. Chapter 21 presents highly mathematical concepts and uses material from Chapters 19 and 20. Chapter 22 is based on material in Chapters 5, 19, and 20; it does not require Chapter 21. For all of Part VI, a knowledge of software engineering is very helpful.

Chapter 23 draws on ideas and information in Chapters 5, 6, 10, 14, 16, and 18 (and for Section 23.8, the reader should read Section 3.1). Chapter 24 is self-contained, although it implicitly uses many ideas from assurance. It also assumes a good working knowledge of compilers, operating systems, and in some cases networks. Many of the flaws are drawn from versions of the UNIX operating system, or from Windows systems, and so a working knowledge of either or both systems will make some of the material easier to understand. Chapter 25 uses information from Chapter 4, and Chapter 26 uses material from Chapter 25.

The practicum chapters are self-contained and do not require any material beyond Chapter 1. However, they point out relevant material in other sections that augments the information and (we hope) the reader's understanding of that information.

Background

The material in this book is at the advanced undergraduate level. Throughout, we assume that the reader is familiar with the basics of compilers and computer architecture (such as the use of the program stack) and operating systems. The reader should also be comfortable with modular arithmetic (for the material on cryptography). Some material, such as that on formal methods (Chapter 21) and the mathematical theory of computer security (Chapter 3 and the formal presentation of policy models), requires considerable mathematical maturity. Other specific recommended background is presented in the preceding section. Part IX, the appendices, contains material that will be helpful to readers with backgrounds that lack some of the recommended material.

Examples are drawn from many systems. Many come from the UNIX operating system or variations of it (such as Linux). Others come from the Windows family of systems. Familiarity with these systems will help the reader understand many examples easily and quickly.

Undergraduate Level

An undergraduate class typically focuses on applications of theory and how students can use the material. The specific arrangement and selection of material depends on the focus of the class, but all classes should cover some basic material—notably that in Chapters 1, 10, and 14, as well as the notion of an access control matrix, which is discussed in Sections 2.1 and 2.2.

Presentation of real problems and solutions often engages undergraduate students more effectively than presentation of abstractions. The special topics and the practicum provide a wealth of practical problems and ways to deal with them. This leads naturally to the deeper issues of policy, cryptography, non-cryptographic mechanisms, and assurance. The following are sections appropriate for non-mathematical undergraduate courses in these topics.

- *Policy*: Sections 4.1 through 4.4 describe the notion of policy. The instructor should select one or two examples from Sections 5.1, 5.2.1, 6.2, 6.4, 8.1.1, and 8.2, which describe several policy models informally. Section 8.4 discusses role-based access control.
- *Cryptography*: Key distribution is discussed in Sections 11.1 and 11.2, and a common form of public key infrastructures (called PKIs) is discussed in Section 11.4.2. Section 12.1 points out common errors

in using cryptography. Section 12.4 shows how cryptography is used in networks, and the instructor should use one of the protocols in Section 12.5 as an example. Chapter 13 offers a look at various forms of authentication, including non-cryptographic methods.

- *Non-cryptographic mechanisms*: Identity is the basis for many access control mechanisms. Sections 15.1 through 15.4 discuss identity on a system, and Section 15.6 discusses identity and anonymity on the Web. Sections 16.1 and 16.2 explore two mechanisms for controlling access to files, and Section 16.4 discusses the ring-based mechanism underlying the notion of multiple levels of privilege. If desired, the instructor can cover sandboxes by using Sections 18.1 and 18.2, but because Section 18.2 uses material from Section 4.5, the instructor will need to go over those sections as well.
- *Assurance*: Chapter 19 provides a basic introduction to the often overlooked topic of assurance.

Graduate Level

A typical introductory graduate class can focus more deeply on the subject than can an undergraduate class. Like an undergraduate class, a graduate class should cover Chapters 1, 10, and 14. Also important are the undecidability results in Sections 3.1 and 3.2, which require that Chapter 2 be covered. Beyond that, the instructor can choose from a variety of topics and present them to whatever depth is appropriate. The following are sections suitable for graduate study.

- *Policy models*: Part III covers many common policy models both informally and formally. The formal description is much easier to understand once the informal description is understood, so in all cases both should be covered. The controversy in Section 5.4 is particularly illuminating to students who have not considered the role of policy and the nature of a policy. Chapter 9 is a highly formal discussion of the foundations of policy and is appropriate for students with experience in formal mathematics. Students without such a background will find it quite difficult.
- *Cryptography*: Part IV focuses on the applications of cryptography, not on cryptography's mathematical underpinnings.[7] It discusses areas of interest critical to the use of cryptography, such as key management and some basic cryptographic protocols used in networking.
- *Non-cryptographic mechanisms*: Issues of identity and certification are complex and generally poorly understood. Section 15.5 covers these problems. Combining this with the discussion of identity on the Web

[7]The interested reader will find a number of books covering aspects of this subject [440, 787, 788, 914, 1092, 1093, 1318, 1826].

(Section 15.6) raises issues of trust and naming. Chapters 17 and 18 explore issues of information flow and confining that flow.

- *Assurance*: Traditionally, assurance is taught as formal methods, and Chapter 21 serves this purpose. In practice, however, assurance is more often accomplished by using structured processes and techniques and informal but rigorous arguments of justification, mappings, and analysis. Chapter 20 emphasizes these topics. Chapter 22 discusses evaluation standards and relies heavily on the material in Chapters 19 and 20 and some of the ideas in Chapter 21.
- *Miscellaneous Topics*: Section 23.8 presents a proof that the generic problem of determining if a generic program is a computer virus is in fact undecidable. The theory of penetration studies in Section 24.2, and the more formal approach in Section 24.6, illuminate the analysis of systems for vulnerabilities. If the instructor chooses to cover intrusion detection (Chapter 26) in depth, it should be understood that this discussion draws heavily on the material on auditing (Chapter 25).
- *Practicum*: The practicum (Part VIII) ties the material in the earlier part of the book to real-world examples and emphasizes the applications of the theory and methodologies discussed earlier.

Practitioners

Practitioners in the field of computer security will find much to interest them. The table of contents and the index will help them locate specific topics. A more general approach is to start with Chapter 1 and then proceed to Part VIII, the practicum. Each chapter has references to other sections of the text that explain the underpinnings of the material. This will lead the reader to a deeper understanding of the reasons for the policies, settings, configurations, and advice in the practicum. This approach also allows readers to focus on those topics that are of most interest to them.

This page intentionally left blank

Acknowledgments

It is not possible to separate those who contributed to the second edition from those who contributed to the first edition, because everything done for the first edition, especially after the first printing, has contributed to the second. So these acknowledgments apply to both editions. That said . . .

Special Acknowledgments

Elisabeth Sullivan and Michelle Ruppel contributed the assurance part of this book.

For the first edition, Liz wrote several drafts, all of which reflect her extensive knowledge and experience in that aspect of computer security. I am particularly grateful to her for contributing her real-world knowledge of how assurance is managed. Too often, books recount the mathematics of assurance without recognizing that other aspects are equally important and more widely used. These other aspects shine through in the assurance section, thanks to Liz. As if that were not enough, she made several suggestions that improved the policy part of this book. I will always be grateful for her contribution, her humor, and especially her friendship.

For the second edition, Michelle stepped in to update that part based on her extensive experience and real-world knowledge as a practitioner. She was careful to maintain the tone and style of Liz's writing, and her contributions strengthened the assurance part. I am grateful to her for agreeing to step in, for the exceptional effort she put forth, and the high quality that resulted.

In summary, I am very grateful for their contributions.

Acknowledgments

Many people offered comments, suggestions, and ideas for the second edition. Thanks to Marvin Schaefer, Sean Peisert, Prof. Christian Probst, Carrie Gates, and Richard Ford for their reviews of the various chapters. I appreciate Prof. Ken

Rosen and Prof. Alfred Menezes for their help with Chapter 10, Steven Templeton and Kara Nance for their suggestions on Chapter 27, Karl Levitt for his comments on Chapter 26, and Richard Ford for his many suggestions on Chapter 23. Their advice and suggestions were invaluable in preparing this edition. Of course, any errors in the text are my responsibility, and usually occurred because I did not always follow their advice.

Thanks also to Pasquale Noce, who sent me a thorough analysis of many of the theorems, proving them constructively as opposed to how they were done in the book. He made many other helpful comments and caught some errors.

The students in Peter Reiher's COM SCI 236-80, Computer Security, class at UCLA in the Spring Quarter 2018, and the students in my ECS 153, Computer Security, classes over the past few years at UC Davis used parts of this edition in various stages of preparation. I thank them for their feedback, which also improved the book.

Many others contributed to this book in various ways. Special thanks to Steven Alexander, Amin Alipour, Jim Alves-Foss, Bill Arbaugh, Andrew Arcilla, Alex Aris, Rebecca Bace, Belinda Bashore, Vladimir Berman, Rafael Bhatti, Ziad El Bizri, David Bover, Logan Browne, Terry Brugger, Gordon Burns, Serdar Cabuk, Raymond Centeno, Yang Chen, Yi Chen, HakSoo Choi, Lisa Clark, Michael Clifford, Christopher Clifton, Dan Coming, Kay Connelly, Crispin Cowan, Shayne Czyzewski, Tom Daniels, Dimitri DeFigueiredo, Joseph-Patrick Dib, Till Dörges, Felix Fan, Robert Fourney, Guillermo Francia III, Jeremy Frank, Conny Francke, Martin Gagne, Nina Gholami, Ron Gove, James Hinde, James Hook, Xuxian Jiang, Jesper Johansson, Mark Jones, Calvin Ko, Mark-Neil Ledesma, Ken Levine, Karl Levitt, Luc Longpre, Yunhua Lu, Gary McGraw, Alexander Meau, Nasir Memon, Katherine Moore, Mark Morrissey, Ather Nawaz, Iulian Neamtiu, Dan Nerenburg, Kimberly Nico, Stephen Northcutt, Rafael Obelheiro, Josko Orsulic, Holly Pang, Sean Peisert, Ryan Poling, Sung Park, Ashwini Raina, Jorge Ramos, Brennen Reynolds, Peter Rozental, Christoph Schuba, night SH, David Shambroom, Jonathan Shapiro, Clay Shields, Sriram Srinivasan, Mahesh V. Tripunitara, Vinay Vittal, Tom Walcott, James Walden, Dan Watson, Guido Wedig, Chris Wee, Han Weili, Patrick Wheeler, Paul Williams, Bonnie Xu, Charles Yang, Xiaoduan Ye, Xiaohui Ye, Lara Whelan, John Zachary, Linfeng Zhang, Aleksandr Zingorenko, and to everyone in my and others' computer security classes, who (knowingly or unknowingly) helped me develop and test this material.

The Pearson folks, in particular my editors Laura Lewin and Malobika Chakraborty, and Sheri Replin, were incredibly helpful and patient. Their patience and enthusiasm ensured this second edition was completed, although a bit later than expected. The production people, especially Julie Nahil, Ramya Gangadharan, and Charles Roumeliotis, moved the book smoothly into print, and I thank them for making it as painless as possible. I owe them many thanks. Similarly, for the first edition, the Addison-Wesley folks, Kathleen Billus, Susannah Buzard, Bernie Gaffney, Amy Fleischer, Helen Goldstein, Tom Stone, Asdis Thorsteinsson, and most especially my editor, Peter Gordon, were incredibly patient and helpful, despite fears that this book would never materialize. The fact that it did so is in great measure attributable to their hard work and encouragement. I also thank the production people at Argosy, especially Beatriz Valdés and Craig Kirkpatrick, for their wonderful work.

Dorothy Denning, my advisor in graduate school, guided me through the maze of computer security when I was just beginning. Peter Denning, Barry Leiner, Karl Levitt, Peter Neumann, Marvin Schaefer, Larry Snyder, and several others influenced my approach to the subject. I hope this work reflects in some small way what they gave to me and passes a modicum of it along to my readers.

I also thank my parents, Leonard Bishop and Linda Allen. My father, a writer, gave me some useful tips on writing, which I tried to follow. My mother, a literary agent, helped me understand the process of getting the book published, and supported me throughout.

Finally, I would like to thank my family for their support throughout the writing. My wife Holly, our children Heidi, Steven, David, and Caroline, and grandchildren Skyler and Sage were very patient and understanding and made sure I had time to work on the book. They also provided delightful distractions. To them all, my love and gratitude.

This page intentionally left blank

About the Author

Matt Bishop is a professor in the Department of Computer Science at the University of California at Davis. He received his Ph.D. in computer science from Purdue University, where he specialized in computer security, in 1984. He was a systems programmer at Megatest Corporation, a research scientist at the Research Institute of Advanced Computer Science and was on the faculty at Dartmouth College.

His main research area is the analysis of vulnerabilities in computer systems, including modeling them, building tools to detect vulnerabilities, and ameliorating or eliminating them. This includes detecting and handling all types of malicious logic. He works in the areas of network security, the study of denial of service attacks and defenses, policy modeling, software assurance testing, resilience, and formal modeling of access control. He has worked extensively in electronic voting, was one of the members of the RABA study for Maryland, and was one of the two principle investigators of the California Top-to-Bottom Review, which performed a technical review of all electronic voting systems certified in the State of California.

He is active in information assurance education. He was co-chair of the Joint Task Force that developed the *Cybersecurity Curricula 2017: Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity*, released in December 2017. He teaches introductory programming, software engineering, operating systems, and (of course) computer security.

This page intentionally left blank

Chapter 31
Program Security

CLOWN: What is he that builds stronger than either
the mason, the shipwright, or the carpenter?
OTHER CLOWN: The gallows-maker; for that frame outlives
a thousand tenants.
— *Hamlet*, V, i, 42–45.

The software on systems implements many mechanisms that support security. Some of these mechanisms reside in the operating system, whereas others reside in application and system programs. This chapter discusses the design and implementation of a program to grant users increased privileges. It also presents common programming errors that create security problems, and offers suggestions for avoiding those problems. Finally, testing and distribution are discussed.

This chapter shows the development of the program from requirements to implementation, testing, and distribution.

31.1 Problem

The purpose of this chapter is to provide a glimpse of techniques that provide better than ordinary assurance that a program's design and implementation satisfy its requirements. This chapter is not a manual on applying high-assurance techniques. In terms of the techniques discussed in Part VI, "Assurance," this chapter describes low-assurance techniques.

However, given the current state of programming and software development, these low-assurance techniques enable programmers to produce significantly better, more robust, and more usable code than they could produce without these techniques. So, using a methodology similar to the one outlined in this chapter will reduce vulnerabilities and improve both the quality and the security of code.

A specific problem will illustrate the methods in this chapter. On the Drib's development network infrastructure systems, numerous system administrators must assume certain roles, such as *bin* (the installers of software), *mail* (the manager of electronic mail), and *root* (the system administrator). Each of these roles is implemented as a separate account, called a *role account*. Unfortunately, this raises the problem of password management. To avoid this problem, as well as to control when access is allowed, the Drib will implement a program that verifies a user's identity, determines if the requested change of account is allowed, and, if so, places the user in the desired role.

31.2 Requirements and Policy

The problem of sharing a password arises when a system implements administrative roles as separate user accounts.

EXAMPLE: Linux systems implement the administrator role as the account *root* (and several other accounts that have more limited functionality).[1] All individuals who share access to the account know the account's password. If the password is changed, all must be notified. All these individuals must remember to notify the other individuals should they change the password.

An alternative to using passwords is to constrain access on the basis of identity and other attributes. With this scheme, a user would execute a special program that would check the user's identity and any ancillary conditions. If all these conditions were satisfied, the user would be given access to the role account.

31.2.1 Requirements

The first requirement comes directly from the description of the alternative scheme above. The system administrators choose to constrain access through known paths (locations) and at times of day when the user is expected to access the role account.

> **Requirement 31.1.** Access to a role account is based on user, location, and time of request.

Users often tailor their environments to fit their needs. This is also true of role accounts. For example, a role account may use special programs kept in a subdirectory of the role account's home directory. This new directory must be on the role account's search path, and would typically be set in the startup file

[1]See Section 14.2.1, "Principle of Least Privilege," for an explanation of how the existence of the *root* account violates the principle of least privilege.

executed when the user logged in. A question is whether the user's environment should be discarded and replaced by the role account's environment, or whether the two environments should be merged. The requirement chosen for this program is as follows.

> **Requirement 31.2.** The settings of the role account's environment shall replace the corresponding settings of the user's environment, but the remainder of the user's environment shall be preserved.

The set of role accounts chosen for access using this scheme is critical. If unrestricted access is given (essentially, a full command interpreter), then any user in the role that maintains the access control information can change that information and acquire unrestricted access to the system. Presumably, if the access control information is kept accessible only to *root*, then the users who can alter the information—all of whom have access to *root*—are trusted. Thus:

> **Requirement 31.3.** Only *root* can alter the access control information for access to a role account.

In most cases, a user assuming a particular role will perform specific actions while in that role. For example, someone who enters the role of *oper* may perform backups but may not use other commands. This restricts the danger of commands interacting with the system to produce undesirable effects (such as security violations) and follows from the principle of least privilege.[2] This form of access is called "restricted access."

> **Requirement 31.4.** The mechanism shall allow both restricted access and unrestricted access to a role account. For unrestricted access, the user shall have access to a standard command interpreter. For restricted access, the user shall be able to execute only a specified set of commands.

Requirement 31.4 implicitly requires that access to the role account be granted to authorized users meeting the conditions in Requirement 31.1. Finally, the role account itself must be protected from unauthorized changes.

> **Requirement 31.5.** Access to the files, directories, and objects owned by any account administered by use of this mechanism shall be restricted to those authorized to use the role account, to users trusted to install system programs, and to *root*.

We next check that these requirements are complete with respect to the threats of concern.

[2] See Section 14.2.1, "Principle of Least Privilege."

31.2.2 Threats

The threats against this mechanism fall into distinct classes. We enumerate the classes and discuss the requirements that handle each threat.

31.2.2.1 Group 1: Unauthorized Users Accessing Role Accounts

There are four threats that involve attackers trying to acquire access to role accounts using this mechanism.

> **Threat 31.1.** An unauthorized user may obtain access to a role account as though she were an authorized user.
>
> **Threat 31.2.** An authorized user may use a nonsecure channel to obtain access to a role account, thereby revealing her authentication information to unauthorized individuals.
>
> **Threat 31.3.** An unauthorized user may alter the access control information to grant access to the role account.
>
> **Threat 31.4.** An authorized user may execute a Trojan horse (or other form of malicious logic),[3] giving an unauthorized user access to the role account.

Requirements 31.1 and 31.5 handle Threat 31.1 by restricting the set of users who can access a role account and protecting the access control data. Requirement 31.1 also handles Threat 31.2 by restricting the locations from which the user can request access. For example, if the set of locations contains only those on trusted or confidential networks, a passive wiretapper cannot discover the authorized user's password or hijack a session begun by an authorized user. Similarly, if an authorized user connects from an untrusted system, Requirement 31.1 allows the system administrator to configure the mechanism so that the user's request is rejected.

The access control information that Requirement 31.1 specifies can be changed. Requirement 31.3 acknowledges this but restricts changes to trusted users (defined as those with access to the root account). This answers Threat 31.3.

Threat 31.4 is more complex. This threat arises from an untrusted user, without authorization, planting a Trojan horse at some location at which an authorized user might execute it. If the attacker can write into a directory in the role account's search path, this attack is feasible. Requirement 31.2 states that the role account's search path may be selected from two other search paths: the default search path for the role account, and the user's search path altered to include those components of the role account's search path that are not present. This leads to Requirement 31.5 which states that, regardless of how the search path is

[3] See Chapter 23, "Malware."

derived, the final search path may contain only directories (and may access only programs) that trusted users or the role account itself can manipulate. In this case, the attacker cannot place a Trojan horse where someone using the role account may execute it.

Finally, if a user is authorized to use the role account but is a novice and may change the search path, Requirement 31.4 allows the administrators to restrict the set of commands that the user may execute in that role.

31.2.2.2 Group 2: Authorized Users Accessing Role Accounts

Because access is allowed here, the threats relate to an authorized user changing access permissions or executing unauthorized commands.

Threat 31.5. An authorized user may obtain access to a role account and perform unauthorized commands.

Threat 31.6. An authorized user may execute a command that performs functions that the user is not authorized to perform.

Threat 31.7. An authorized user may change the restrictions on the user's ability to obtain access to the account.

The difference between Threats 31.5 and 31.6 is subtle but important. In the former, the user deliberately executes commands that violate the site security policy. In the latter, the user executes authorized commands that perform covert, unauthorized actions as well as overt, authorized actions—the classic Trojan horse. Threat 31.6 differs from Threat 31.4 because the action may not give access to authorized users; it may simply damage or destroy the system.

Requirement 31.4 handles Threat 31.5. If the user accessing the role account should execute only a specific set of commands, then the access controls must be configured to restrict the user's access to executing only those commands.

Requirements 31.2 and 31.5 handle Threat 31.6 by preventing the introduction of a Trojan horse, as discussed in the preceding section.

Requirement 31.3 answers Threat 31.7. Because all users who have access to *root* are trusted by definition, the only way for an authorized user to change the restrictions on obtaining access to the role account is to implant a backdoor (which is equivalent to a Trojan horse) or to modify the access control information. But the requirement holds that only trusted users can do that, so the authorized user cannot change the information unless he is trusted—in which case, by definition, the threat is handled.

31.2.2.3 Summary

Because the requirements handle the threats, and because all requirements are used, the set of requirements is both necessary and sufficient. We now proceed with the design.

31.3 Design

To create this program, we build modules that fit together to supply security services that satisfy the requirements. First, we create a general framework to guide the development of each interface. Then we examine each requirement separately, and design a component for each requirement.

31.3.1 Framework

The framework begins with the user interface and then breaks down the internals of the program into modules that implement the various requirements.

31.3.1.1 User Interface

The user can run the program in two ways. The first is to request unrestricted access to the account. The second is to request that a specific program be run from the role account. Any interface must be able to handle both.

The simplest interface is a command line. Other interfaces, such as graphical user interfaces, are possible and may make the program easier to use. However, these GUIs will be built in such a way that they construct and execute a command-line version of the program.

The interface chosen is

`role` *`role_account`* `[` *`command`* `]`

where *role_account* is the name of the role account and *command* is the (optional) command to execute under that account. If the user wants unrestricted access to the role account, he omits *command*. Otherwise, the user is given restricted access and *command* is executed with the privileges of the role account.

The user need not specify the time of day using the interface, because the program can obtain such information from the system. It can also obtain the location from which the user requests access, although the method used presents potential problems (see Section 31.4.3.1). The individual modules handle the remainder of the issues.

31.3.1.2 High-Level Design

The basic algorithm is as follows.

1. Obtain the role account, command, user, location, and time of day. If the command is omitted, the user is requesting unrestricted access to the role account.

2. Check that the user is allowed to access the role account
   a. at the specified location;
   b. at the specified time; and
   c. for the specified command (or without restriction).

   If the user is not, log the attempt and quit.
3. Obtain the user and group information for the role account. Change the privileges of the process to those of the role account.
4. If the user has requested that a specific command be run, overlay the child process with a command interpreter that spawns the named command.
5. If the user has requested unrestricted access, overlay the child process with a command interpreter.

This algorithm points out an important ambiguity in the requirements. Requirements 31.1 and 31.4 do not indicate whether the ability of the user to execute a command in the given role account requires that the user work from a particular location or access the account at a particular time. This design uses the interpretation that a user's ability to run a command in a role account is conditioned on location and time.

The alternative interpretation, that access only is controlled by location and time, and that commands are restricted by role and user, is equally valid. But sometimes the ability to run commands may require that users work at particular times. For example, an operator may create the daily backups at 1 a.m. The operator is not to do backups at other times because of the load on the system. The interpretation of the design allows this. The alternative interpretation requires the backup program, or some other mechanism, to enforce this restriction. Furthermore, the design interpretation includes the alternative interpretation, because any control expressed in the alternative interpretation can be expressed in the design interpretation.

Requirement 31.4 can now be clarified. The addition is in boldface.

> **Requirement 31.6.** The mechanism shall allow both restricted access and unrestricted access to a role account. For unrestricted access, the user shall have access to a standard command interpreter. For restricted access, the user shall be able to execute only a specified set of commands. **The level of access (restricted or unrestricted) shall depend on the user, the role, the time, and the location.**

Thus, the design phase feeds back into the requirements phase, here clarifying the meaning of the requirements. It is left as an exercise for the reader to verify that the new form of this requirement counters the appropriate threats (see Exercise 2).

31.3.2 Access to Roles and Commands

The user attempting access, the location (host or terminal), the time of day, and the type of access (restricted or unrestricted) control access to the role account. The access checking module returns a value indicating success (meaning that access is allowed) or failure (meaning that access is not allowed). By the principle of fail-safe defaults, an error causes a denial of access.

We consider two aspects of the design of this module. The interface controls how information is passed to the module from its caller, and how the module returns success or failure. The internal structure of the module includes how it handles errors. This leads to a discussion of how the access control data is stored. We consider these issues separately to emphasize that the interface provides an entry point into the module, and that the entry point will remain fixed even if the internal design of the module is completely changed. The internal design and structures are hidden from the caller.

31.3.2.1 Interface

Following the practice of hiding information among modules,[4] we minimize the amount of information to be passed to the access checking module. The module requires the user requesting access, the role to which access is requested, the location, the time, and the command (if any). The return value must indicate success or failure. The question is how this information is to be obtained.

The command (or request for unrestricted access) must come from the caller, because the caller provides the interface for the processing of that command. The command is supplied externally, so the principles of layering require it to pass through the program to the module.

The caller could also pass the other information to the module. This would allow the module to provide an access control result without obtaining the information directly. The advantage is that a different program could use this module to determine whether or not access *had been* or *would be* granted at some past or future point in time, or from some other location. The disadvantage is a lack of portability, because the interface is tied to a particular representation of the data. Also, if the caller of the module is untrusted but the module is trusted, the module might make trusted decisions based on untrusted data, violating a principle of integrity.[5] So we choose to have the module determine all of the data.

This suggests the following interface:

```
boolean accessok(role rname, command cmd);
```

where *rname* is the name of the requested role and *cmd* is the command to be executed (or is empty if unrestricted access is desired). The routine returns **true** if access is to be granted, and **false** otherwise.

---

[4]This is one aspect of the principle of least common mechanism (see Section 14.2.7).

[5]This follows from Biba's low-water-mark policy (see Section 6.2.1).

31.3.2.2 Internals

This module has three parts. The first part gathers the data on which access is to be based. The second part retrieves the access control information. The third part determines whether the data and the access control information require access to be granted.

The module queries the operating system to determine the needed data. The real user identification data is obtained through a system call, as is the current time of day. The location consists of two components: the entry point (terminal or network connection) and the remote host from which the user is accessing the local system. The latter component may indicate that the entry point is directly connected to the system, rather than using a remote host.

Part I: Obtain user ID, time of day, entry point, and remote host.

Next, the module must access the access control information. The access control information resides in a file. The file contains a sequence of records of the following form:

```
role account
user names
locations from which the role account can be accessed
times when the role account can be accessed
command and arguments
```

If the "command and arguments" line is omitted, the user is granted unrestricted access. Multiple command lines may be listed in a single record.

Part II: Obtain a handle (or descriptor) to the access control information. The programmer will use this handle to read the access control records from the access control information.

Finally, the program iterates through the access control information. If the role in the current record does not match the requested role, it is ignored. Otherwise, the user name, location, time, and command are compared with the appropriate fields of the record. If they all match, the module releases the handle and returns success.[6] If any of them does not match, the module continues on to the next record. If the module reaches the end of the access control information, the handle is released and the module returns failure. Note that records never deny access, but only grant it. The default action is to deny. Granting access requires an explicit record.[7]

If any record is invalid (for example, if there is a syntax error in one of the fields or if the user field contains a nonexistent user name), the module logs the error and ignores the record. This again follows the principle of fail-safe defaults, in which the system falls into a secure state when there is an error.

---

[6]If the time interval during which access is allowed expires after the access control check but before the access is granted, Requirement 31.1 is met (as it refers to the time of request). This eliminates a possible race condition.

[7]See Section 14.2.2, "Principle of Fail-Safe Defaults."

Part III: Iterate through the records until one matches the data or there are no more records. In the first case, return success; in the second case, return failure.

31.3.2.3 Storage of the Access Control Data

The system administrators of the local system control access to privileged accounts. To keep maintenance of this information simple, the administrators store the access control information in a file. Then they need only edit the file to change a user's ability to access the privileged account. The file consists of a set of records, each containing the components listed above. This raises the issue of expression. How should each part of the record be written?

For example, must each entry point be listed, or are wildcards acceptable? Strictly speaking, the principle of fail-safe defaults[8] says that we should list explicitly those locations from which access may be obtained. In practice, this is too cumbersome. Suppose a particular user was trusted to assume a role from any system on the Internet. Requiring the administrators to list all hosts would be time-consuming as well as infeasible. Worse, if the user were not allowed to access the role account from one system, the administrators would need to check the list to see which system was missing. This would violate the principle of least astonishment.[9] Given the dynamic nature of the Internet, this requirement would be absurd. Instead, we allow the following special host names, both of which are illegal [1365]:

any (a wildcard matching any system)
local (matches the local host name)

In BNF form, the language used to express location is

location ::= '(' *location* ')' | 'not' *location* | *location* 'or' *location* | *basic*
basic ::= '*any*' | '*local*' | '.' *domain* | *host*

where *domain* and *host* are domain names and host names, respectively. The strings in single quotation marks are literals. The parentheses are grouping operators, the "not" complements the associated locations, and the "or" allows either location.

EXAMPLE: A user is allowed to assume a role only when logged into the local system, the system "control.fixit.com", and the domain "watchu.edu". The appropriate entry would be

```
*local* | control.fixit.com | .watchu.edu
```

[8]See Section 14.2.2.
[9]See Section 14.2.8.

A similar question arises for times. Ignoring how times are expressed, how do we indicate when users may access the role account? Considerations similar to those above lead us to the following language, in which the keyword

any

allows access at any time. In BNF form, the language used to express time is

time ::= '(' *time* ')' | 'not' *time* | *time* 'or' *time* | *time time* | *time* '–' *time* | *basic*
basic ::= *day_of_year day_of_week time_of_day* | '*any*'
day_of_year ::= *month* [*day*] [','*year*] | *nmonth* '/' [*day* '/'] *year* | *empty*
day_of_week ::= 'Sunday' | ... | 'Saturday' | 'Weekend' | 'Weekday' | *empty*
time_of_day ::= *hour* [':' *min*] [':' *sec*] ['AM' | 'PM'] | *special* | *empty*
special ::= 'noon' | 'midnight' | 'morning' | 'afternoon' | 'evening'
empty ::= ''

where *month* is a string naming the month, *nmonth* is an integer naming the month, *day* is an integer naming the day of the month, and *year* is an integer specifying the year. Similarly, *hour*, *min*, and *sec* are integers specifying the hour, minute, and second. If *basic* is empty, it is treated as not allowing access.[10]

EXAMPLE: A user is allowed to assume a role between the hours of 9 o'clock in the morning and 5 o'clock in the evening on Monday through Thursday. An appropriate entry would be

Monday-Thursday 9a.m.-5p.m.

This is different than saying

Monday 9a.m.-Thursday 5p.m.

because the latter allows access on Monday at 10 p.m., whereas the former does not.

Finally, the users field of the record has a similar structure:

any

In BNF form, the language used to express the set of users who may access a role is

userlist ::= '(' *userlist* ')' | 'not' *userlist* | *userlist* ',' *userlist* | *user*

where *user* is the name of a user on the system.

[10]By the principle of fail-safe defaults (see Section 14.2.2).

These "little languages" are straightforward and simple (but incomplete; see Exercise 5). Various implementation details, such as allowing abbreviations for day and month names, can be added, as can an option to change the American expression of days of the year to an international one. These points must be considered in light of where the program is to be used. Whatever changes are made, the administrators must be able to configure times and places quickly and easily, and in a manner that a reader of the access control file can understand quickly.[11]

The listing of commands requires some thought about how to represent arguments. If no arguments are listed, is the command to be run without arguments, or should it allow any set of arguments? Conversely, if arguments are listed, should the command be run only with those arguments? Our approach is to force the administrator to indicate how arguments are to be treated.

Each command line contains a command followed by zero or more arguments. If the first word after the command is an asterisk (" * "), then the command may be run with any arguments. Otherwise, the command must be run with the exact arguments provided.

EXAMPLE: Charles is allowed to run the install command when he accesses the *bin* role. He may supply any arguments. The line in the access control file is

```
/bin/install *
```

He may also copy the file log from the current working directory to the directory */var/install*. The line for this is

```
/bin/cp log /var/install/log
```

Finally, he may run the *id* command to ensure that he is working as *bin*. He may not supply other arguments to the command, however. This would be expressed by

```
/usr/bin/id
```

The user must type the command as given in the access control file. The full path names are present to prevent the user from accidentally executing the command *id* with *bin* privileges when *id* is a command in the local directory, rather than the system *id* command.[12]

[11]See Section 14.2.8, "Principle of Least Astonishment."
[12]See Chapter 23, "Malware."

31.4 Refinement and Implementation

This section focuses on the access control module of the program. We refine the high-level design presented in the preceding section until we produce a routine in a programming language.

31.4.1 First-Level Refinement

Rather than use any particular programming language, we first implement the module in pseudocode. This requires two decisions. First, the implementation language will be block-structured, like C or Java, rather than functional, like Scheme or ML. Second, the environment in which the program will function will be a UNIX-like system such as FreeBSD or Linux.

The basic structure of the security module is

```
boolean accessok(role rname, command cmd);
    status ← false
    user ← obtain user ID
    timeday ← obtain time of day
    entry ← obtain entry point (terminal line, remote host)
    open access control file
    repeat
      currecord ← obtain next record from file; EOF if none
      if currecord ≠ EOF then
            status ← match(currecord, rname, cmd, user,
                                timeday, entry)
    until currecord = EOF or status = true
    close access control file
    return status
```

We now verify that this sketch matches the design. Clearly, the interface is unchanged. The variable *status* will contain the status of the access control file check, becoming true when a match is found. Initially, it is set to false (deny access) because of the principle of fail-safe defaults. If *status* were not set, and the access control file were empty, *status* would never be set and the returned value would be undefined.

The next three lines obtain the user ID, the current time of day, and the system entry point. The following line opens the access control file.

The routine then iterates through the records of that file. The iteration has two requirements—that if any record allows access, the routine is to return true, and that if no record grants access, the routine is to return false. From the structure of the file, one cannot create a record to deny access. By default, access is denied. Entries explicitly grant access. So, iterating over the records of the file either produces a record that grants access (in which case the match routine returns true, terminating the loop and causing *accessok* to return with a value of true) or produces no such record. In that case, *status* is false, and *currecord* is set to EOF when the records in the access control file are exhausted. The loop then terminates, and the routine returns the value of *status*, which is false. Hence, this pseudocode matches the design and, transitively, the requirements.

31.4.2 Second-Level Refinement

Now we will focus on mapping the pseudocode above to a particular language and system. The C programming language is widely available and provides a convenient interface to UNIX-like systems. Given that our target system is a UNIX-like system, C is a reasonable choice. As for the operating system, there are many variants of the UNIX operating system. However, they all have fundamental similarities. The Linux operating system will provide the interfaces discussed below, and they work on a wide variety of UNIX systems.

On these systems, roles are represented as normal user accounts. The *root* account is really a role account,[13] for example. Each user account has two distinct representations of identity:[14] an internal user type *uid_t*,[15] and a string (name). When a user specifies a role, either representation may be used. For our purposes, we will assume that the caller of the *accessok* routine provides the *uid_t* representation of the role identity. Two reasons make this representation preferable. First, the target systems are unable to address privilege in terms of names, because, within the kernel, process identity is always represented by a *uid_t*. So the routines will need to do the conversion anyway. The second reason is more complex. Roles in the access control file can be represented by numbers or names. The routine for reading the access control file records will convert the roles to *uid_t*s to ensure consistency of representation. This also allows the input routine to check the records for consistency with the system environment. Specifically, if the role name refers to a nonexistent account, the routine can ignore the record. So any comparisons would require the role from the interface to be converted to a *uid_t*.

This leads to a design decision: represent all user and role IDs as integers internally. Fortunately, none of the design decisions discussed so far depend on the representation of identity, so we need not review or change our design.

Next, consider the command. On the target system, a command consists of a program name followed by a sequence of words, which are the command-line arguments to the command. The command representation is an array of strings, in

---

[13]See Section 15.4, "Groups and Roles."

[14]See Section 15.3, "Users."

[15]On Linux systems, and on most UNIX-like systems, this is an integer.

which the first string is the program name and the other strings are the command-line arguments.

Putting this all together, the resulting interface is

```
int accessok(uid_t rname, char *cmd[])
```

Next comes obtaining the user ID. Processes in the target system have several identities, but the key ones are the *real UID* (which identifies the user running the process) and the *effective UID* (which identifies the privileges with which the process runs).[16] The effective UID of this program must have *root* privileges (see Exercise 4) regardless of who runs the process. Hence, it is useless for this purpose. Only the real UID identifies the user running the program. So, to obtain the user ID of the user running the program, we use

```
userid = getuid();
```

The time of day is obtained from the system and expressed in internal format. The internal representation can be given in seconds since a specific date and time (the *epoch*)[17] or in microseconds since that time. It is unlikely that times will need to be specified in microseconds in the access control file. For both simplicity of code and simplicity of the access control data,[18] the internal format of seconds will be used. So, to obtain the current time, we use

```
timeday = time(NULL);
```

Finally, we need to obtain the location. There is no simple method for obtaining this information, so we defer it until later by encapsulating it in a function. This also localizes any changes should we move this program to a different system (for example, the methods used on a Linux system may differ from those used on a FreeBSD system).

```
entry = getlocation();
```

Opening the access control file for reading is straightforward:

```
if ((fp = fopen(acfile, "r")) == NULL){
    logerror(errno, acfile);
    return(0);
}
```

---

[16]See Section 15.3, "Users."

[17]On Linux and most other UNIX-like systems, the epoch is midnight on January 1, 1970 (UTC).

[18]See Section 14.2.3, "Principle of Economy of Mechanism," and Section 14.2.8, "Principle of Least Astonishment."

Notice first the error checking, and the logging of information on an error. The variable `errno` is set to a code indicating the nature of the error. The variable `acfile` points to the access control file name. The processing of the access control records follows:

```
do {
    acrec = getnextacrec(fp);
    if (acrec != NULL)
        status = match(acrec, rname, cmd, user, timeday,
                       entry);
} while (acrec == NULL || status == 1);
```

Here, we read in the record—assuming that any records remain—and check the record to see if it allows permission. This looping continues until either some record indicates that permission is to be given or all records are checked. The exact internal record format is not yet specified; hence, the use of functions. The routine concludes by closing the access control file and returning status:

```
(void) fclose(fp);
return(status);
```

31.4.3 Functions

Three functions remain: the function for obtaining location, the function for getting an access control record, and the function for checking the access control record against the information of the current process. Each raises security issues.

31.4.3.1 Obtaining Location

UNIX and Linux systems write the user's account name, the name of the terminal on which the login takes place, the time of login, and the name of the remote host (if any) to the *utmp* file. Any process may read this file. As each new process runs, it may have an associated terminal. To determine the *utmp* record associated with the process, a routine may obtain the associated terminal name, open the *utmp* file, and scan through the record to find the one with the corresponding terminal name. That record contains the name of the host from which the user is working.

This approach, although clumsy, works on most UNIX and Linux systems. It suffers from two problems related to security.

1. If any process can alter the *utmp* file, its contents cannot be trusted. Several security holes have occurred because any process could alter the *utmp* file [2254].
2. A process may have no associated terminal. Such a detached process must be mapped into the corresponding *utmp* record through other means.

However, if the *utmp* record contains only the information described above, this is not possible because the user may be logged into multiple terminals. The issue does not arise if the process has an associated terminal, because only one user at a time may be logged into a terminal.

In the first case, we make a design decision that if the data in the *utmp* file cannot be trusted because any process can alter that file, we return a meaningless location. Then, unless the location specifier of the record allows access from any location, the record will not match the current process information and will not grant access. A similar approach works if the process does not have an associated terminal.

The outline of this routine is

```
hostname getlocation()
    status ← false
    myterm ← name of terminal
    obtain access control list for utmp
    if any user other than root can alter it then
        return "*nowhere*"
    open utmp
    repeat
        term ← obtain next entry from utmp; otherwise EOF
        if term ≠ EOF and myterm = term then
                status ← true
    until term = EOF or status = true
    if host field of utmp entry = empty
        host = "localhost"
    else
        host = host field of utmp entry
    close utmp
    return host
```

We omit the implementation due to space limitations.

31.4.3.2 The Access Control Record

The format of the records in the access control file affects both the reading of the file and the comparison with the process information, so we design it here.

Our approach is to consider the match routine first. Four items must be checked: the user name, the location, the time, and the command. Consider these items separately.

The user name is represented as an integer. Thus, the internal format of the user field of the access control record must contain either integers or names that the match routine can convert to integers. If a match occurs before all user names have been checked, then the program needs to convert no more names to integers. So, we adopt the strategy of representing the user field as a string read directly from the file. The match routine will parse the line and will use lazy evaluation to check whether or not the user ID is listed.

A similar strategy can be applied to the location and the set of commands in the record.

The time is somewhat different, because in the previous two cases, the process user ID and the location had to match one of the record entries exactly. However, the time does not have to do so. Time in the access control record is (almost always) a range. For example, the entry "May 30" means any time on the date of May 30. The day begins at midnight and ends at midnight, 24 hours later. So, the range would be from May 30 at midnight to May 31 at midnight, or in internal time (for example) between 1527638400 and 1527724800. In those rare cases in which a user may assume a role only at a precise second, the range can be treated as having the same beginning and ending points. Given this view of time as ranges, checking that the current time falls into an acceptable range suggests having the match routine parse the times and checking whether or not the internal system time falls in each range as it is constructed.

This means that the routine for reading the record may simply load the record as a sequence of strings and let the match routine do the interpretation. This yields the following structure:

```
record
    role rname
    string userlist
    string location
    string timeofday
    string commands[]
    integer numcommands
end record;
```

The *commands* field is an array of strings, each command and argument being one string, and *numcommands* containing the number of commands.

Given this information, the function used to read the access control records, and the function used to match them with the current process information, are not hard to write, but error handling does deserve some mention.

31.4.3.3 Error Handling in the Reading and Matching Routines

Assume that there is a syntax error in the access control file. Perhaps a record specifies a time incorrectly (for example, "Thurxday"), or a record divider is garbled. How should the routines handle this?

The first observation is that they cannot ignore the error. To do so violates basic principles of security (specifically, the principle of least astonishment[19]). It also defeats the purpose of the program, because access will be denied to users

[19]See Section 14.2.8, "Principle of Least Astonishment."

who need it.[20] So, the program must produce an indication of error. If it is printed, then the user will see it and should notify the system administrator maintaining the access control file. Should the user forget, the administrator will not know of the error. Hence, the error must be logged. Whether or not the user should be told why the error has occurred is another question. One school of thought holds that the more information users have, the more helpful they will be. Another school holds that information should be denied unless the user needs to know it, and in the case of an error in the access control file, the user only needs to know that access will be denied.

Hence, the routines must log information about errors. The logged information must enable the system administrator to locate the error in the file. The error message should include the access control file name and line or record number. This suggests that both routines need access to that information. Hence, the record counts, line numbers, and file name must be shared. For reasons of modularity, this implies that these two routines should be in a submodule of the access checking routine. If they are placed in their own module, no other parts of the routine can access the line or record numbers (and none need to, given the design described here). If the module is placed under the access control routine, no external functions can read records from the access control file or check data against that file's contents.

31.4.4 Summary

This section has examined the development of a program for performing a security-critical function. Beginning with a requirements analysis, the design and parts of the implementation demonstrate the need for repeated analysis to ensure that the design meets the requirements and that design decisions are documented. From the point at which the derivation stopped, the implementation is simple.

We will now discuss some common security-related programming problems. Then we will discuss testing, installation, and maintenance.

31.5 Common Security-Related Programming Problems

Unfortunately, programmers are not perfect. They make mistakes. These errors can have disastrous consequences in programs that change the protection domains. Attackers who exploit these errors may acquire extra privileges (e.g., access to a system account such as *root* or *Administrator*). They may disrupt the normal functioning of the system by deleting or altering services over which they

[20] Note that a record with a syntax error will never grant access (see Exercise 6).

should have no control. They may simply be able to read files to which they should have no access.[21] So the problem of avoiding these errors, or security holes, is a necessary issue to ensure that the programs and system function as required.

We present both management rules (installation, configuration, and maintenance) and programming rules together. Although there is some benefit in separating them, doing so creates an artificial distinction by implying that they can be considered separately. In fact, the limits on installation, configuration, and maintenance affect the implementation, just as the limits of implementation affect the installation, configuration, and maintenance procedures.

Researchers have developed several models for analyzing systems for these security holes.[22] These models provide a framework for characterizing the problems. The goal of the characterization guides the selection of the model. Because we are interested in technical modeling and not in the reason or time of introduction, many of the categories of the NRL model[23] are inappropriate for our needs. We also wish to analyze the multiple components of vulnerabilities rather than force each vulnerability into a particular point of view, as Aslam's model[24] does. So either the PA model[25] or the RISOS model[26] is appropriate. We have chosen the PA model for our analysis.

We examine each of the categories and subcategories separately. We consider first the general rules that we can draw from the vulnerability class, and then we focus on applying those rules to the program under discussion.

31.5.1 Improper Choice of Initial Protection Domain

Flaws involving improper choice of initial protection domain arise from incorrect setting of permissions or privileges. There are three objects for which permissions need to be set properly: the file containing the program, the access control file, and the memory space of the process. We will consider them separately.

31.5.1.1 Process Privileges

The principle of least privilege[27] dictates that no process have more privileges than it needs to complete its task, but the process must have enough privileges to complete its task successfully.

Ideally, one set of privileges should meet both criteria. In practice, different portions of the process will need different sets of privileges. For example, a process may need special privileges to access a resource (such as a log file) at the beginning

[21] See Chapter 24, "Vulnerability Analysis."
[22] See Section 24.4, "Frameworks."
[23] See Section 24.4.3, "The NRL Taxonomy."
[24] See Section 24.4.4, "Aslam's Model."
[25] See Section 24.4.2, "Protection Analysis Model."
[26] See Section 24.4.1, "The RISOS Study."
[27] See Section 14.2.1, "Principle of Least Privilege."

and end of its task, but may not need those privileges at other times. The process structure and initial protection domain should reflect this.

> **Implementation Rule 31.1.** Structure the process so that all sections requiring extra privileges are modules. The modules should be as small as possible and should perform only those tasks that require those privileges.

The basis for this rule lies in the reference monitor.[28] The reference monitor is verifiable, complete (it is always invoked to access the resource it protects), and tamperproof (it cannot be compromised). Here, the modules are kept small and simple (verifiable), access to the privileged resource requires the process to invoke these modules (complete), and the use of separate modules with well-defined interfaces minimizes the chances of other parts of the program corrupting the module (tamperproof).

> **Management Rule 31.1.** Check that the process privileges are set properly.

Insufficient privileges could cause a denial of service. Excessive privileges could enable an attacker to exploit vulnerabilities in the program. To avoid these problems, the privileges of the process, and the times at which the process has these privileges, must be chosen and managed carefully.

One of the requirements of this program is availability (Requirements 31.1 and 31.4). On Linux and UNIX systems, the program must change the effective identity of the user from the user's account to the role account. This requires special (setuid) privileges of either the role account or the superuser.[29] The principle of least privilege[30] says that the former is better than the latter, but if one of the role accounts is *root*, then having multiple copies of the program with limited privileges is irrelevant, because the program with privileges to access the *root* role account is the logical target of attack. After all, if one can compromise a less privileged account through this program, the same attack will probably work against the *root* account. Because the Drib plans to control access to *root* in some cases, the program requires setuid to *root* privileges.

If the program does not have root privileges initially, the UNIX protection model does not allow the process to acquire them; the permissions on the program file corresponding to the program must be changed. The process must log enough information for the system administrator to identify the problem,[31] and should notify users of the problem so that the users can notify the system administrator. An alternative is to develop a server that will periodically check the permissions on the program file and reset them if needed, or a server that the program can notify should it have insufficient privileges. The designers felt that the benefits of

---

[28]See Section 20.1.2.2, "Building Security In or Adding Security Later." Programs implemented following this rule are *not* reference monitors.

[29]See Section 15.3, "Users."

[30]See Section 14.2.1, "Principle of Least Privilege."

[31]See Section 25.3, "Designing an Auditing System."

these servers were not sufficient to warrant their development. In particular, they were concerned that the system administrators investigate any unexpected change in file permissions, and an automated server that changed the permissions back would provide insufficient incentive for an analysis of the problem.

As a result, the developers required that the program acquire *root* permission at start-up. The access control module is executed. Within that module, the privileges are reset to the user's once the log file and access control file have been opened.[32] Superuser privileges are needed only once more—to change the privileges to those of the role account should access be granted. This routine, also in a separate module, supplies the granularity required to provide the needed functionality while minimizing the time spent executing with *root* privileges.

31.5.1.2 Access Control File Permissions

Biba's models[33] emphasize that the integrity of the process relies on both the integrity of the program and the integrity of the access control file. The former requires that the program be properly protected so that only authorized personnel can alter it. The system managers must determine who the "authorized personnel" are. Among the considerations here are the principle of separation of duty[34] and the principle of least privilege.[35]

Verifying the integrity of the access control file is critical, because that file controls the access to role accounts. Some external mechanism, such as a file integrity checking tool, can provide some degree of assurance that the file has not changed. However, these checks are usually periodic, and the file might change after the check. So the program itself should check the integrity of the file when the program is run.

> **Management Rule 31.2.** The program that is executed to create the process, and all associated control files, must be protected from unauthorized use and modification. Any such modification must be detected.

In many cases, the process will rely on the settings of other files or on some other external resources. Whenever possible, the program should check these dependencies to ensure that they are valid. The dependencies must be documented so that installers and maintainers will understand what else must be maintained in order to ensure that the program works correctly.

> **Implementation Rule 31.2.** Ensure that any assumptions in the program are validated. If this is not possible, document them for the installers and maintainers, so they know the assumptions that attackers will try to invalidate.

---

[32] Section 14.2.3, "Principle of Complete Mediation," provides detail on why this works.
[33] See Section 6.2, "The Biba Model."
[34] See Section 6.1, "Goals."
[35] See Section 14.2.1, "Principle of Least Privilege."

The permissions of the program, and its containing directory, are to be set so only *root* can alter or move the program. According to Requirement 31.2, only *root* can alter the access control file. Hence, the file must be owned by *root*, and only *root* can write to it. The program should check the ownership and permissions of this file, and the containing directories, to validate that only *root* can alter it.

EXAMPLE: The naive way to check that only *root* can write to the file is to check that the owner is *root* and that the file permissions allow only the owner to write to it. But consider the group permissions. If *root* is the only member of the group, then the group permissions may allow members of the group to write to the file. The problem is that checking group membership is more complicated than looking up the members of the group. A user may belong to a group without being listed as a member, because the GID of the user is assigned from the password file, and group membership lists are contained in a different file.[36] Either the password file and the group membership list must both be checked, or the program should simply report an error if anyone other than the user can write to the file. For simplicity,[37] the designers chose the second approach.

31.5.1.3 Memory Protection

As the program runs, it depends on the values of variables and other objects in memory. This includes the executable instructions themselves. Thus, protecting memory against unauthorized or unexpected alteration is critical.

Consider sharing memory. If two subjects can alter the contents of memory, then one could change data on which the second relies. Unless such sharing is required (for example, by concurrent processes), it poses a security problem because the modifying process can alter variables that control the action of the other process. Thus, each process should have a protected, unshared memory space.

If the memory is represented by an object that processes can alter, it should be protected so that only trusted processes can access it. Access here includes not only modification but also reading, because passwords reside in memory after they are types. Multiple abstractions are discussed in more detail in the next section.

> **Implementation Rule 31.3.** Ensure that the program does not share objects in memory with any other program, and that other programs cannot access the memory of a privileged process.

---

[36]Specifically, if the group field of the password file entry for *matt* is 30, and the group file lists the members of group 30 as *root*, the user *matt* is still in group 30, but a query to the group file (the standard way to determine group membership) will show that only *root* is a member.

[37]See Section 14.2.3, "Principle of Economy of Mechanism."

Interaction with other processes cannot be eliminated. If the running process obtains input or data from other processes, then that interface provides a point through which other processes can reach the memory. The most common version of this attack is the buffer overflow.

Buffer overflows involve either altering of data or injecting of instructions that can be executed later. There are a wide variety of techniques for this [32, 706].[38] Several remedies exist. For example, if buffers reside in sections of memory that are not executable, injecting instructions will not work. Similarly, if some data is to remain unaltered, the data can be stored in read-only memory.

> **Management Rule 31.3.** Configure memory to enforce the principle of least privilege. If a section of memory is not to contain executable instructions, turn execute permission off for that section of memory. If the contents of a section of memory are not to be altered, make that section read-only.

These rules appear in three ways in our program. First, the implementers use the language constructs to flag unchanging data as constant (in the C programming language, this is the keyword *const*). This will cause compile-time errors if the variables are assigned to, or runtime errors if instructions try to alter those constants.

The other two ways involve program loading. The system's loader places data in three areas: the *data* (initialized data) segment, the *stack* (used for function calls and variables local to the functions), and the *heap* (used for dynamically allocated storage). A common attack is to trick a program into executing instructions injected into three areas. The vector of injection can be a buffer overflow,[39] for example. The characteristic under discussion does not stop such alteration, but it should prevent the data from being executed by making the segments or pages of all three areas nonexecutable. This suffices for the data and stack segments and follows Management Rule 31.3.

If the program uses dynamic loading to load functions at runtime, the functions that are loaded may change over the lifetime of the program. This means that the assumptions the programmers make may no longer be valid.[40] One solution to this problem is to compile the program in such a way that it does not use dynamic loading. This also also prevents the program from trying to load a module at runtime that may be missing. This could occur if a second process deleted the appropriate library. So disabling of dynamic loading also follows Implementation Rule 31.3.[41]

Finally, some UNIX-like systems (including the one on which this program is being developed) allow execution permission to be turned off for the stack. The boot file sets the kernel flag to enforce this.

---

[38] However, alternative techniques involving corrupting data, causing the flow of control to change improperly, do work. See Section 31.5.6, "Improper Validation."

[39] Buffer overflows can also alter data. See Section 31.5.3.1, "Memory," for an example.

[40] See Section 31.5.3.2, "Changes in File Contents."

[41] Other considerations contributed. See Section 31.5.4, "Improper Naming."

31.5.1.4 Trust in the System

This analysis overlooks several system components. For example, the program relies on the system authentication mechanisms to authenticate the user, and on the user information database to map users and roles into their corresponding UIDs (and, therefore, privileges). It also relies on the inability of ordinary users to alter the system clock. If any of this supporting infrastructure can be compromised, the program will not work correctly. The best that can be done is to identify these points of trust in the installation and operation documentation so that the system administrators are aware of the dependencies of the program on the system.

> **Management Rule 31.4.** Identify all system components on which the program depends. Check for errors whenever possible, and identify those components for which error checking will not work.

For this program, the implementers should identify the system databases and information on which the program depends, and should prepare a list of these dependencies. They should discuss these dependencies with system managers to determine if the program can check for errors. When this is not possible, or when the program cannot identify all errors, they should describe the possible consequences of the errors. This document should be distributed with the program so that system administrators can check their systems before installing the program.

31.5.2 Improper Isolation of Implementation Detail

The problem of improper isolation of implementation detail arises when an abstraction is improperly mapped into an implementation detail. Consider how abstractions are mapped into implementations. Typically, some function (such as a database query) occurs, or the abstraction corresponds to an object in the system. What happens if the function produces an error or fails in some other way, or if the object can be manipulated without reference to the abstraction?

The first rule is to catch errors and failures of the mappings. This requires an analysis of the functions and a knowledge of their implementation. The action to take on failure also requires thought. In general, if the cause cannot be determined, the program should fail by returning the relevant parts of the system to the states they were in when the program began.[42]

> **Implementation Rule 31.4.** The error status of every function must be checked. Do not try to recover unless the cause of the error, and its effects, do not affect any security considerations. The program should restore the state of the system to the state before the process began, and then terminate.

---

[42] See Section 14.2.2, "Principle of Fail-Safe Defaults."

The abstractions in this program are the notion of a user and a role, the access control information, and the creation of a process with the rights of the role. We will examine these abstractions separately.

31.5.2.1 Resource Exhaustion and User Identifiers

The notion of a user and a role is an abstraction because the program can work with role names and the operating system uses integers (UIDs). The question is how those user and role names are mapped to UIDs. Typically, this is done with a user information database that contains the requisite mapping, but the program must detect any failures of the query and respond appropriately.

EXAMPLE: A mail server allowed users to forward mail by creating a forwarding file [2225]. The forwarding file could specify files to which the mail should be appended. In this case, the mail server would deliver the letter with the privileges of the owner of the forwarding file (represented on the system as an integer UID). In some cases, the mail server would queue the message for later delivery. When it did so, it would write the name (not the UID) of the user into a control file. The system queried a database, supplying the UID, and obtaining the corresponding name. If the query failed, the mail server used a default name specified by the system administrator.

Attackers discovered how to make the queries fail. As a result, the user was set to a default user, usually a system-level user (such as *daemon*). This enabled the attackers to have the mail server append mail to any file to which the default user could write. They used this to implant Trojan horses into system programs. These Trojan horses gave them extra privileges, compromising the system.

The designers and implementers decided to have the program fail if, for any reason, the query failed. This application of the principle of fail-safe defaults[43] ensured that in case of error, the users would not get access to the role account.

31.5.2.2 Validating the Access Control Entries

The access control information implements the access control policy (an abstraction). The expression of the access control information is therefore the result of mapping an abstraction to an implementation. The question is whether or not the given access control information correctly implements the policy. Answering this question requires someone to examine the implementation expression of the policy.

The programmers developed a second program that used the same routines as the role-assuming program to analyze the access control entries. This program prints the access control information in an easily readable format. It allows the system managers to check that the access control information is correct. A specific procedure requires that this information be checked periodically, and always after the file or the program is altered.

---

[43] See Section 14.2.2, "Principle of Fail-Safe Defaults."

31.5.2.3 Restricting the Protection Domain of the Role Process

Creating a role process is the third abstraction. There are two approaches. Under UNIX-like systems, the program can spawn a second, *child*, process. It can also simply start up a second program in such a way that the parent process is replaced by the new process. This technique, called *overlaying*, is intrinsically simpler than creating a child process and exiting. It allows the process to replace its own protection domain with the (possibly) more limited one corresponding to the role. The programmers elected to use this method. The new process inherits the protection domain of the original one. Before the overlaying, the original process must reset its protection domain to that of the role. The programmers do so by closing all files that the original process opened, and changing its privileges to those of the role.

EXAMPLE: The effective UIDs and GIDs[44] control privileges. Hence, the programmers reset the effective GID first, and then the effective UID (if resetting were done in the opposite order, the change to GIDs would fail because such changes require *root* privileges). However, if the UNIX-like system supports saved UIDs, an authorized user may be able to acquire *root* privileges even if the role account is not *root*. The problem is that resetting the effective UID sets the saved UID to the previous UID—namely, *root*. A process may then reacquire the rights of its saved UID. To avoid this problem, the programmers used the *setuid* system call to reset *all* of the real, effective, and saved UIDs to the UID of the role. Thus, all traces of the *root* UID are eliminated and the user cannot reacquire those privileges.

Similarly, UNIX-like systems check access permissions only when the file is opened. If a *root* process opens a privileged file and then the process drops *root* privileges, it can still read from (or write to) the file.

The components of the protection domain that the process must reset before the overlay are the open files (except for standard input, output, and error), which must be closed, the signal handlers, which must be reset to their default values, and any user-specific information, which must be cleared.

31.5.3 Improper Change

This category describes data and instructions that change over time. The danger is that the changed values may be inconsistent with the previous values. The previous values dictate the flow of control of the process. The changed values cause the program to take incorrect or nonsecure actions on that path of control.

The data and instructions can reside in shared memory, in nonshared memory, or on disk. The last includes file attribute information such as ownership and access control list.

[44]See Section 15.3, "Users."

31.5.3.1 Memory

First comes the data in shared memory. Any process that can access shared memory can manipulate data in that memory. Unless all processes that can access the shared memory implement a concurrent protocol for managing changes, one process can change data on which a second process relies. As stated above, this could cause the second process to violate the security policy.

EXAMPLE: Two processes share memory. One process reads authentication data and writes it into the shared memory space. The second process performs the authentication, and writes a boolean *true* back into the shared memory space if the authentication succeeds, and *false* if it fails. Unless the two processes use concurrent constructs to synchronize their reading and writing, the first process may read the result before the second process has completed the computation for the current data. This could allow access when it should be denied, or vice versa.

Implementation Rule 31.5. If a process interacts with other processes, the interactions should be synchronized. In particular, all possible sequences of interactions must be known and, for all such interactions, the process must enforce the required security policy.

A variant of this situation is the asynchronous exception handler. If the handler alters variables and then returns to the previous point in the program, the changes in the variables could cause problems similar to the problem of concurrent processes. For this reason, if the exception handler alters any variables on which other portions of the code depend, the programmer must understand the possible effects of such changes. This is just like the earlier situation in which a concurrent process changes another's variables in a shared memory space.

Implementation Rule 31.6. Asynchronous exception handlers should not alter any variables except those that are local to the exception handling module. An exception handler should block all other exceptions when begun, and should not release the block until the handler completes execution, unless the handler has been designed to handle exceptions within itself (or calls an uninvoked exception handler).

A second approach applies whether the memory is shared or not. A user feeds bogus information to the program, and the program accepts it. The bogus data overflows its buffer, changing other data, or inserting instructions that can be executed later.

EXAMPLE: The buffer overflow attack on *fingerd* described in Section 24.4.5.2 illustrates this approach. The return address is pushed onto the stack when the input routine is called. That address is not expected to change between its being pushed onto the stack and its being popped from the stack, but the buffer overflow changes it. When the input function returns, the address popped from the stack is that of the input buffer. Execution resumes at that point, and the input instructions are used.

This suggests one way to detect such transformations (the *stack guard approach*) [469]. Immediately after the return address is pushed onto the stack, push a random number onto the stack (the *canary*). Assume that the input overflows the buffer on the stack and alters the return address on the stack. If the canary is n bits long and has been chosen randomly, the probability of the attacker not changing that cookie is 2^{-n}. When the input procedure returns, the canary is popped and compared with the value that was pushed onto the stack. If the two differ, there has been an overflow.[45]

In terms of trust, the return address (a trusted datum) can be affected by untrusted data (from the input). This lowers the trustworthiness of the return address to that of input data. One need not supply instructions to breach security.

EXAMPLE: One (possibly apocryphal) version of a UNIX login program allocated two adjacent arrays. The first held the user's cleartext password and was 80 characters long, and the second held the password hash[46] and was 13 characters long. The program's logic loaded the password hash into the second array as soon as the user's name was determined. It then read the user's cleartext password and stored it in the first array. If the contents of the first array hashed to the contents of the second array, the user was authenticated. An attacker simply selected a random password (for example, "password") and generated a valid hash for it (here, "12CsGd8FRcMSM"). The attacker then identified herself as *root*. When asked for a password, the attacker entered "password", typed 72 spaces, and then typed "12CsGd8FRcMSM". The system hashed "password", got "12CsGd8FRcMSM", and logged the user in as *root*.

A technique in which canaries protect data, not only the return address, would work, but raises many implementation problems (see Exercise 7).

> **Implementation Rule 31.7.** Whenever possible, data that the process trusts and data that it receives from untrusted sources (such as input) should be kept in separate areas of memory. If data from a trusted source is overwritten with data from an untrusted source, a memory error will occur.

In more formal terms, the principle of least common mechanism[47] indicates that memory should not be shared in this way.

---

[45] If the goal is to alter data on the stack other than the return address, the canary will not be altered. This technique will not detect the change. (See Exercise 7.)
[46] See Section 13.2, "Passwords."
[47] See Section 14.2.7, "Principle of Least Common Mechanism."

These rules apply to our program in several ways. First, the program does not interact with any other program except through exception handling.[48] So Implementation Rule 31.5 does not apply. Exception handling consists of calling a procedure that disables further exception handling, logs the exception, and immediately terminates the program.

Illicit alteration of data in memory is the second potential problem. If the user-supplied data is read into memory that overlaps with other program data, it could erase or alter that data. To satisfy Implementation Rule 31.7, the programmers did not reuse variables into which users could input data. They also ensured that each access to a buffer did not overlap with other buffers.

The problem of buffer overflow is solved by checking all array and pointer references within the code. Any reference that is out of bounds causes the program to fail after logging an error message to help the programmers track down the error.

31.5.3.2 Changes in File Contents

File contents may change improperly. In most cases, this means that the file permissions are set incorrectly or that multiple processes are accessing the file, which is similar to the problem of concurrent processes accessing shared memory. Management Rule 31.2 and Implementation Rule 31.5 cover these two cases.

A nonobvious corollary is to be careful of dynamic loading. Dynamic load libraries are not part of this program's executable. They are loaded, as needed, when the program runs. Suppose one of the libraries is changed, and the change causes a side effect. The program may cease to function or, even worse, work incorrectly.

If the dynamic load modules cannot be altered, then this concern is minimal, but if they can be upgraded or otherwise altered, it is important. Because one of the reasons for using dynamic load libraries is to allow upgrades without having to recompile programs that depend on the library, security-related programs using dynamic load libraries are at risk.

> **Implementation Rule 31.8.** Do not use components that may change between the time the program is created and the time it is run.

This is another reason that the developers decided not to use dynamic loading.

31.5.3.3 Race Conditions in File Accesses

A race condition in this context is the *time-of-check-to-time-of-use* problem. As with memory accesses, the file being used is changed after validation but before

---

[48] If the access control information or the authentication information came from servers, then there would be interaction with other programs (the servers). The method of communication would need to be considered, as discussed above.

access.[49] To thwart it, either the file must be protected so that no untrusted user can alter it, or the process must validate the file and use it indivisibly. The former requires appropriate settings of permission, so Management Rule 31.2 applies. Section 31.5.7, "Improper Indivisibility," discusses the latter.

This program validates that the owner and access control permissions for the access control file are correct (the check). It then opens the file (the use). If an attacker can change the file after the validation but before the opening, so that the file checked is not the file opened, then the attacker can have the program obtain access control information from a file other than the legitimate access control file. Presumably, the attacker would supply a set of access control entries allowing unauthorized accesses.

EXAMPLE: The UNIX operating system allows programs to refer to files in two ways: by name and by file descriptor.[50] Once a file descriptor is bound to a file, the referent of the descriptor does not change. Each access through the file descriptor always refers to the bound file (until the descriptor is closed). However, the kernel reprocesses the file name at each reference, so two references to the same file name may refer to two *different* files. An attacker who is able to alter the file system in such a way that this occurs is exploiting a race condition. So any checks made to the file corresponding to the first use of the name may not apply to the file corresponding to the second use of the name. This can result in a process making unwarranted assumptions about the trustworthiness of the file and the data it contains.

In the *xterm* example[51] the program can be fixed by opening the file and then using the file descriptor (handle) to obtain the owner and access permissions.[52] Those permissions belong to the opened file, because they were obtained using the file descriptor. The validation is now ensured to be that of the access control file.

The program does exactly this. It opens the access control file and uses the file descriptor, which references the file attribute information directly to obtain the owner, group, and access control permissions. Those permissions are checked. If they are correct, the program uses the file descriptor to read the file. Otherwise, the file is closed and the program reports a failure.

31.5.4 Improper Naming

Improper naming refers to an ambiguity in identifying an object. Most commonly, two different objects have the same name. The programmer intends the name to refer to one of the objects, but an attacker manipulates the environment and the

[49] Section 24.3.1, "Two Security Flaws," discusses this problem in detail.
[50] See Section 15.2, "Files and Objects."
[51] See Section 24.3.1, "Two Security Flaws."
[52] The system call used would be *fstat*.

process so that the name refers to a different object. Avoiding this flaw requires that every object be unambiguously identified. This is both a management concern and an implementation concern.

Objects must be uniquely identifiable or completely interchangeable. Managing these objects means identifying those that are interchangeable and those that are not. The former objects need a controller (or set of controllers) that, when given a name, selects one of the objects. The latter objects need unique names. The managers of the objects must supply those names.

Management Rule 31.5. Unique objects require unique names. Interchangeable objects may share a name.

A name is interpreted within a context. At the implementation level, the process must force its own context into the interpretation, to ensure that the object referred to is the intended object. The context includes information about the character sets, process and file hierarchies, network domains, and any accessible variables such as the search path.

EXAMPLE: Stage 3 in Section 24.2.9 discussed an attack in which a privileged program called *loadmodule* executed a second program named *ld.so*. The attack exploited *loadmodule*'s failure to specify the context in which *ld.so* was named. *Loadmodule* used the context of the user invoking the program. Normally, this caused the correct *ld.so* to be invoked. In the example, the attacker changed the context so that another version of *ld.so* was executed. This version had a Trojan horse that would grant privileged access. When the attacker executed *loadmodule*, the Trojan horse was triggered and maximum privileges were acquired.

Implementation Rule 31.9. The process must ensure that the context in which an object is named identifies the correct object.

This program uses names for external objects in four places: the name of the access control file, the names of the users and roles, the names of the hosts, and the name of the command interpreter (the *shell*) that the program uses to execute commands in the role account.

The two file names (access control file and command interpreter) must identify specific files. Absolute path names specify the location of the object with respect to a distinguished directory called / or the "root directory." However, a privileged process can redefine / to be any directory.[53] This program does not do so. Furthermore, if the root directory is anything other than the root directory of the system, a trusted process has executed it. No untrusted user could have done so. Thus, as long as absolute path names are specified, the files are unambiguously named.

---

[53] Specifically, the system call *chroot* resets / to mean the named directory. All absolute path names are interpreted with respect to that directory. Only the superuser, *root*, may execute this system call.

The name provided may be interpreted in light of other aspects of the environment. For example, differences in the encoding of characters can transform file names. Whether characters are made up of 16 bits, 8 bits, or 7 bits can change the interpretation, and therefore the referent, of a file name. Other environment variables can change the interpretation of the path name. This program simply creates a new, known, safe environment for execution of the commands.[54]

This has two advantages over sanitization of the existing context. First, it avoids having the program analyze the environment in detail. The meaning of each aspect of the environment need not be analyzed and examined. The environment is simply replaced. Second, it allows the system to evolve without compromising the security of the program. For example, if a new environment variable is assigned a meaning that affects how programs are executed, the variable will not affect how this program executes its commands because that variable will not appear in the command's environment. So this program closes all file descriptors, resets signal handlers, and passes a new set of environment variables for the command.

These actions satisfy Implementation Rule 31.9.

The developers assumed that the system was properly maintained, so that the names of the users and roles would map into the correct UIDs. (Section 31.5.2.1 discusses this.) This applies to Management Rule 31.5.

The host names are the final set of names. These may be specified by names or IP addresses. If the former, they must be fully qualified domain names to avoid ambiguity. To see this, suppose an access control entry allows user *matt* to access the role *wheel* when logging in from the system *amelia*. Does this mean the system named *amelia* in the local domain, or any system named *amelia* from any domain? Either interpretation is valid. The former is more reasonable,[55] and applying this interpretation resolves the ambiguity. (The program implicitly maps names to fully qualified domain names using the former interpretation. Thus, *amelia* in the access control entry would match a host named *amelia* in the local domain, and not a host named *amelia* in another domain.) This implements Implementation Rule 31.9.[56]

As a side note, if the local network is mismanaged or compromised, the name *amelia* may refer to a system other than the one intended. For example, the real host *amelia* may crash or go offline. An attacker can then reset the address of his host to correspond to *amelia*. This program will not detect the impersonation.

31.5.5 Improper Deallocation or Deletion

Failing to delete sensitive information raises the possibility of another process seeing that data at a later time. In particular, cryptographic keywords, passwords,

---

[54] The principle of fail-safe defaults (see Section 14.2.2) supports this approach.

[55] According to the principle of least privilege (see Section 14.2.1).

[56] As discussed in Section 15.6.1, "Host Identity," host names can be spoofed. For reasons discussed in the preceding chapters, the Drib management and security officers are not concerned with this threat on the Drib's internal network.

and other authentication information should be discarded once they have been used. Similarly, once a process has finished with a resource, that resource should be deallocated. This allows other processes to use that resource, inhibiting denial of service attacks.

A consequence of not deleting sensitive information is that dumps of memory, which may occur if the program receives an exception or crashes for some other reason, contain the sensitive data. If the process fails to release sensitive resources before spawning unprivileged subprocesses, those unprivileged subprocesses may have access to the resource.

> **Implementation Rule 31.10.** When the process finishes using a sensitive object (one that contains confidential information or one that should not be altered), the object should be erased, then deallocated or deleted. Any resources not needed should also be released.

Our program uses three pieces of sensitive information. The first is the cleartext password, which authenticates the user. The password is hashed, and the hash is compared with the stored hash. Once the hash of the entered password has been computed, the process must delete the cleartext password. So it overwrites the array holding the password with random bytes.

The second piece of sensitive information is the access control information. Suppose an attacker wanted to gain access to a role account. The access control entries would tell the attacker which users could access that account using this program. To prevent the attacker from gaining this information, the developers decided to keep the contents of the access control file confidential. The program accesses this file using a file descriptor. File descriptors remain open when a new program overlays a process. Hence, the program closes the file descriptor corresponding to the access control file once the request has been validated (or has failed to be validated).

The third piece of sensitive information is the log file. The program alters this file. If an unprivileged program such as one run by this program were to inherit the file descriptor, it could flood the log. Were the log to fill up, the program could no longer log failures. So the program also closes the log file before spawning the role's command.

31.5.6 Improper Validation

The problem of improper validation arises when data is not checked for consistency and correctness. Ideally, a process would validate the data against the more abstract policies to ensure correctness. In practice, the process can check correctness only by looking for error codes (indicating failure of functions and procedures) or by looking for patently incorrect values (such as negative numbers when positive ones are required).

As the program is designed, the developers should determine what conditions must hold at each interface and each block of code. They should then validate that these conditions hold.

What follows is a set of validations that are commonly overlooked. Each program requires its own analysis, and other types of validation may be critical to the correct, secure functioning of the program, so this list is by no means complete.

31.5.6.1 Bounds Checking

Errors of validation often occur when data is supposed to lie within bounds. For example, a buffer may contain entries numbered from 0 to 99. If the index used to access the buffer elements takes on a value less than 0 or greater than 99, it is an invalid operand because it accesses a nonexistent entry. The variable used to access the element may not be an integer (for example, it may be a set element or pointer), but in any case it must reference an existing element.

> **Implementation Rule 31.11.** Ensure that all array references access existing elements of the array. If a function that manipulates arrays cannot ensure that only valid elements are referenced, do not use that function. Find one that does, write a new version, or create a wrapper.

In this example program, all loops involving arrays compare the value of the variable referencing the array against the indexes (or addresses) of both the first and last elements of the array. The loop terminates if the variable's value is outside those two values. This covers all loops within the program, but it does not cover the loops in the library functions.

For loops in the library functions, bounds checking requires an analysis of the functions used to manipulate arrays. The most common type of array for which library functions are used is the character string, which is a sequence of characters (bytes) terminating with a 0 byte. Because the length of the string is not encoded as part of the string, functions cannot determine the size of the array containing the string. They simply operate on all bytes until a 0 byte is found.

EXAMPLE: The program sometimes must copy character strings (defined in C as arrays of character data terminating with a byte containing 0). The canonical function for copying strings does no bounds checking. This function, `strcpy(x, y)`, copies the string from the array *y* to the array *x*, even if the string is too long for *x*. A different function, `strncpy(x, y, n)`, copies at most *n* characters from array *y* to array *x*. However, unlike *strcpy*, *strncpy* may not copy the terminating 0 byte.[57] The program must take two actions when *strncpy* is called. First, it must insert a 0 byte at the end of the *x* array. This ensures that the contents of *x* meet the definition of a string in C. Second, the process must check that both *x* and *y* are arrays of characters, and that *n* is a positive integer.

The programmers use only those functions that bound the sizes of arrays. In particular, the function *fgets* is used to read input, because it allows the

[57] If the string in *y* is longer than *n* characters, *strncpy* will not add a 0 byte to the characters copied into *x*.

programmer to specify the maximum number of characters to be read. (This solves the problem that plagued *fingerd*.[58])

31.5.6.2 Type Checking

Failure to check types is another common validation problem. If a function parameter is an integer, but the actual argument passed is a floating point number, the function will interpret the bit pattern of the floating point number as an integer and will produce an incorrect result.

> **Implementation Rule 31.12.** Check the types of functions and parameters.

A good compiler and well-written code will handle this particular problem. All functions should be declared before they are used. Most programming languages allow the programmer to specify the number and types of arguments, as well as the type of the return value (if any). The compiler can then check the types of the declarations against the types of the actual arguments and return values.

> **Implementation Rule 31.13.** When compiling programs, ensure that the compiler reports inconsistencies in types. Investigate all such warnings and either fix the problem or document the warning and why it is spurious.

31.5.6.3 Error Checking

A third common problem involving improper validation is failure to check return values of functions. For example, suppose a program needs to determine ownership of a file. It calls a system function that returns a record containing information from the file attribute table. The program obtains the owner of the file from the appropriate field of the record. If the function fails, the information in the record is meaningless. So, if the function's return status is not checked, the program may act erroneously.

> **Implementation Rule 31.14.** Check all function and procedure executions for errors.

This program makes extensive use of system and library functions, as well as its own internal functions (such as the access control module). Every function returns a value, and the value is checked for an error before the results of the function are used. For example, the function that obtains the ownership and access permissions of the access control file would return meaningless information should the function fail. So the function's return value is checked first for an error; if no error has occurred, then the file attribute information is used.

---

[58] See Section 24.4.5.2, "The *fingerd* Buffer Overflow."

As another example, the program opens a log file. If the open fails, and the program tries to write to the (invalid) file descriptor obtained from the function that failed, the program will terminate as a result of an exception. Hence, the program checks the result of opening the log file.

31.5.6.4 Checking for Valid, Not Invalid, Data

Validation should apply the principle of fail-safe defaults.[59] This principle requires that valid values be known, and that all other values be rejected. Unfortunately, programmers often check for invalid data and assume that the rest is valid.

EXAMPLE: A *metacharacter* is a character that is interpreted as something other than itself. For example, to the UNIX shells, the character "?" is a metacharacter that represents all single character files. A vendor upgraded its version of the command interpreter for its UNIX system. The new command interpreter (shell) treated the character " `" (back quote) as a delimiter for a command (and hence a metacharacter). The old shell treated the back quote as an ordinary character. Included in the distribution was a program for executing commands on remote systems. The set of allowed commands was restricted. This program carefully checked that the command was allowed, and that it contained no metacharacters, before sending it to a shell on the remote system. Unfortunately, the program checked a list of metacharacters to be rejected, rather than checking a list of characters that were allowed in the commands. As a result, one could embed a disallowed command within a valid command request, because the list of metacharacters was not updated to include the back quote.

Implementation Rule 31.15. Check that a variable's values are valid.

This program checks that the command to be executed matches one of the authorized commands. It does not have a set of commands that are to be denied. The program will detect an invalid command as one that is not listed in the set of authorized commands for that user accessing that role at the time and place allowed.

As discussed in Section 31.3.2.3, it is possible to allow all users *except some specific users* access to a role by an appropriate access control entry (using the keyword *not*). The developers debated whether having this ability was appropriate because its use could lead to violations of the principle of fail-safe defaults. On one key system, however, the only authorized users were system administrators and one or two trainees. The administrators wanted the ability to shut the trainees out of certain roles. So the developers added the keyword and recommended against its use except in that single specific situation.

[59] See Section 14.2.2, "Principle of Fail-Safe Defaults."

Implementation Rule 31.16. If a trade-off between security and other factors results in a mechanism or procedure that can weaken security, document the reasons for the decision, the possible effects, and the situations in which the compromise method should be used. This informs others of the trade-off and the attendant risks.

31.5.6.5 Checking Input

All data from untrusted sources must be checked. Users are untrusted sources. The checking done depends on the way the data is received: into an input buffer (bounds checking) or read in as an integer (checking the magnitude and sign of the input).

Implementation Rule 31.17. Check all user input for both form and content. In particular, check integers for values that are too big or too small, and check character data for length and valid characters.

The program determines what to do on the basis of at least two pieces of data that the user provides: the role name and the command (which, if omitted, means unrestricted access).[60] Users must also authenticate themselves appropriately. The program must first validate that the supplied password is correct. It then checks the access control information to determine whether the user is allowed access to the role at that time and from that location.

The length of the input password must be no longer than the buffer in which it is placed. Similarly, the lines of the access control file must not overflow the buffer allocated for it. The contents of the lines of the access control file must make up a valid access control entry. This is most easily done by constraining the format of the contents of the file, as discussed in the next section.

An excellent example of the need to constrain user input comes from formatted print statements in C.

EXAMPLE: The *printf* function's first parameter is a character string that indicates how *printf* is to format output data. The following parameters contain the data. For example,

```
printf("%d %d\n", i, j);
```

prints the values of *i* and *j*. Some versions of this library function allow the user to store the number of characters printed at any point in the string. For example, if *i* contains 2, *j* contains 21, and *m* and *n* are integer variables,

```
printf("%d %d%n %d\n%n", i, j, &m, i, &n);
```

prints

```
 2 21 2
```

[60]See Section 14.2.6, "Principle of Separation of Privilege."

and stores 4 in *m* and 7 in *n*, because four characters are printed before the first "%n" and seven before the second "%n" (the sequence "\n" is interpreted as a single character, the newline). Now, suppose the user is asked for a file name. This input is stored in the array *str*. The program then prints the file name with

```
printf(str);
```

If the user enters the file name "log%n", the function will overwrite some memory location with the integer 3. The exact location depends on the contents of the program stack, and with some experimentation it is possible to cause the program to change the return address stored on the stack. This leads to the buffer overflow attack described earlier.

31.5.6.6 Designing for Validation

Sometimes data cannot be validated completely. For example, in the C programming language, a programmer can test for a NULL pointer (meaning that the pointer does not hold the address of any object), but if the pointer is not NULL, checking the validity of the pointer may be very difficult (or impossible). Using a language with strong type checking is another example.

The consequence of the need for validation requires that data structures and functions be designed and implemented in such a way that they can be validated. For example, because C pointers cannot be properly validated, programmers should not pass pointers or use them in situations in which they must be validated. Methods of data hiding, type checking, and object-oriented programming often provide mechanisms for doing this.

Implementation Rule 31.18. Create data structures and functions in such a way that they can be validated.

An example will show the level of detail necessary for validation. The entries in the access control file are designed to allow the program to detect obvious errors. Each access control entry consists of a block of information in the following format:

```
role name
    user comma-separated list of users
    location comma-separated list of locations
    time comma-separated list of times
    command program and arguments
    . . .
    command program and arguments
endrole
```

This defines each component of the entry. (The lines need not be in any particular order.) The syntax is well-defined, and the access control module in the program checks for syntax errors. The module also performs other checks, such as searching for invalid user names in the **user** field and requiring that the full path names of all commands be specified. Finally, note that the module computes the number of commands for the module's internal record. This eliminates a possible source of error—namely, that the user may miscount the number of commands.

In case of any error, the process logs the error, if possible, and terminates. It does not allow the user to access the role.

31.5.7 Improper Indivisibility

Improper indivisibility[61] arises when an operation is considered as one unit (indivisible) in the abstract but is implemented as two units (divisible). The race conditions discussed in Section 31.5.3.3 provide one example. The checking of the access control file attributes and the opening of that file are to be executed as one operation. Unfortunately, they may be implemented as two separate operations, and an attacker who can alter the file after the first but before the second operation can obtain access illicitly. Another example arises in exception handling. Often, program statements and system calls are considered as single units or operations when the implementation uses many operations. An exception divides those operations into two sets: the set before the exception, and the set after the exception. If the system calls or statements rely on data not changing during their execution, exception handlers must not alter the data.

Section 31.5.3 discusses handling of these situations when the operations cannot be made indivisible. Approaches to making them indivisible include disabling interrupts and having the kernel perform operations. The latter assumes that the operation is indivisible when performed by the kernel, which may be an incorrect assumption.

> **Implementation Rule 31.19.** If two operations must be performed sequentially without an intervening operation, use a mechanism to ensure that the two cannot be divided.

In UNIX systems, the problem of divisibility arises with root processes such as the program under consideration. UNIX-like systems do not enforce the principle of complete mediation.[62] For *root*, access permissions are not checked. Recall the *xterm* example in Section 24.3.1. A user needed to log information from the execution of *xterm*, and specified a log file. Before appending to that file, *xterm* needed to ensure that the real UID could write to the log file. This required an extra system call. As a result, operations that should have been indivisible (the access check followed by the opening of the file) were actually divisible. One way to make these operations indivisible on UNIX-like systems is to drop privileges to those of the real UID, then open the file. The access checking is done in the kernel as part of the open.

---

[61] This is often called "atomicity."

[62] See Section 14.2.4, "Principle of Complete Mediation."

Improper indivisibility arises in our program when the access control module validates and then opens the access control file. This should be a single operation, but because of the semantics of UNIX-like systems, it must be performed as two distinct operations. It is not possible to ensure the indivisibility of the two operations. However, it is possible to ensure that the target of the operations does not change, as discussed in Section 31.5.3, and this suffices for our purposes.

31.5.7.1 Improper Sequencing

Improper sequencing means that operations are performed in an incorrect order. For example, a process may create a lock file and then write to a log file. A second process may also write to the log file, and then check to see if the lock file exists. The first program uses the correct sequence of calls; the second does not (because that sequence allows multiple writers to access the log file simultaneously).

> **Implementation Rule 31.20.** Describe the legal sequences of operations on a resource or object. Check that all possible sequences of the program(s) involved match one (or more) legal sequences.

In our program, the sequence of operations in the design shown in Section 31.3.1.2 follows a proper order. The user is first authenticated. Then the program uses the access control information to determine if the requested access is valid. If it is, the appropriate command is executed using a new, safe environment.

A second sequence of operations occurs when privileges to the role are dropped. First, group privileges are changed to those of the role. Then all user identification numbers are changed to those of the role. A common error is to switch the user identification numbers first, followed by the change in group privileges. Because changing group privileges requires *root* privileges, the change will fail. Hence, the programmers used the stated ordering.

31.5.8 Improper Choice of Operand or Operation

Preventing errors of choosing the wrong operand or operation requires that the algorithms be thought through carefully (to ensure that they are appropriate). At the implementation level, this requires that operands be of an appropriate type and value, and that operations be selected to perform the desired functions. The difference between this type of error and improper validation lies in the program. Improper implementation refers to a validation failure. The operands may be appropriate, but no checking is done. In this category, even though the operands may have been checked, they may still be inappropriate.

EXAMPLE: The UNIX program *su* allows a user to substitute another user's identity, obtaining the second user's privileges. According to an apocryphal story, one version of this program granted the user *root* privileges if the user information database did not exist (see Exercise 10 in Chapter 14). If the program could not open the user information database file, it assumed that the database did not exist. This was an inappropriate choice of operation because one could block access to the file even when the database existed.

Assurance techniques[63] help detect these problems. The programmer documents the purpose of each function and then checks (or, preferably, others check) that the algorithms in the function work properly and that the code correctly implements the algorithms.

> **Management Rule 31.6.** Use software engineering and assurance techniques (such as documentation, design reviews, and code reviews) to ensure that operations and operands are appropriate.

Within our program, many operands and operations control the granting (and denying) of access, the changing to the role, and the execution of the command. We first focus on the access part of the program, and afterwards we consider two other issues.

First, a user is granted access only when an access control entry matches all characteristics of the current session. The relevant characteristics are the role name, the user's UID, the role's name (or UID), the location, the time, and the command. We begin by checking that if the characteristics match, the access control module returns *true* (allowing access). We also check that the caller grants access when the module returns true and denies access when the module returns *false*.

Next, we consider the user's UID. That object is of type *uid_t*. If the interface to the system database returns an object of a different type, conversion becomes an issue. Specifically, many interfaces treat the UID as an integer. The difference between the types *int* and *uid_t* may cause problems. On the systems involved, *uid_t* is an unsigned integer. Since we are comparing signed and unsigned integers, C simply converts the signed integers to unsigned integers, and the comparison succeeds. Hence, the choice of operation (comparison here) is proper.

Checking location requires the program to derive the user's location, as discussed above, and pass it to the validator. The validator takes a string and determines whether it matches the pattern in the location field of the access control entry. If the string matches, the module should continue; otherwise, it should terminate and return false.

Unlike the location, a variable of type *time_t* contains the current time. The time checking portion of the module processes the string representing the allowed times and determines if the current time falls in the range of allowed times. Checking time is different than checking location because legal times are ranges, except in one specific situation: when an allowed time is specified to the exact second. A specification of an exact time is useless, because the program may not obtain the time at the exact second required. This would lead to a denial of service, violating Requirement 31.4. Also, allowing exact times leads to ambiguity.

---

[63]See Chapter 20, "Building Systems with Assurance."

EXAMPLE: The system administrator specifies that user *matt* is allowed access to the role *mail* at 9 a.m. on Tuesdays. Should this be interpreted as *exactly* 9 a.m. (that is, 9:00:00 a.m.) or as *sometime during* the 9 a.m. hour (that is, from 9:00:00 to 9:59:59 a.m.)? The latter interprets the specification as a range rather than an exact time, so the access control module uses that interpretation.

The use of signal handlers provides a second situation in which an improper choice of operation could occur. A signal indicates either an error in the program or a request from the user to terminate, so a signal should cause the program to terminate. If the program continues to run, and then grants the user access to the role account, either the program has continued in the face of an error or it has overridden the user's attempt to terminate the program.

31.5.9 Summary

This type of top-down analysis differs from the more usual approach of taking a checklist of common vulnerabilities and using it to examine code. There is a place for each of these approaches. The top-down approach presented here is a design approach, and should be applied at each level of design and implementation. It emphasizes documentation, analysis, and understanding of the program, its interfaces, and the environment in which it executes. A security analysis document should describe the analysis and the reasons for each security-related decision. This document will help other analysts examine the program and, more importantly, will provide future developers and maintainers of the program with insight into potential problems they may encounter in porting the program to a different environment, adding new features, or changing existing features.

Once the appropriate phase of the program has been completed, the developers should use a checklist to validate that the design or implementation has no common errors. Given the complexity of security design and implementation, such checklists provide valuable confirmation that the developers have taken common security problems into account.

Appendix H lists the implementation and management rules in a convenient form.

31.6 Testing, Maintenance, and Operation

Testing provides an informal validation of the design and implementation of the program. The goal of testing is to show that the program meets the stated requirements. When design and implementation are driven by the requirements, as in the method used to create the program under discussion, testing is likely to uncover only minor problems, but if the developers do not have well-articulated requirements, or if the requirements are changed during development, testing may uncover major problems, requiring changes up to a complete redesign and reimplementation of a program. The worst mistake managers and developers can make is to take a program that does not meet the security requirements and add features to it to meet those requirements. The problem is that the basic design does not meet the security requirements. Adding security features will not ameliorate this fundamental flaw.

Once the program has been written and tested, it must be installed. The installation procedure must ensure that when a user starts the process, the environment in which the process is created matches the assumptions embodied in the design. This constrains the configuration of the program parameters as well as the manner in which the system is configured to protect the program. Finally, the installers must enable trusted users to modify and upgrade the program and the configuration files and parameters.

31.6.1 Testing

The results of testing a program are most useful if the tests are conducted in the environment in which the program will be used (the production environment). So, the first step in testing a program is to construct an environment that matches the production environment. This requires the testers to know the intended production environment. If there are a range of environments, the testers must test the programs in all of them. Often there is overlap between the environments, so this task is not so daunting as it might appear.

The production environment should correspond to the environment for which the program was developed. A symptom of discrepancies between the two environments is repeated failures resulting from erroneous assumptions. This indicates that the developers have implicitly embedded information from the development environment that is inconsistent with the testing environment. This discrepancy must be reconciled.

The testing process begins with the requirements. Are they appropriate? Do they solve the problem? This analysis may be moot (if the task is to write a program meeting the given requirements), but if the task is phrased in terms of a problem to be solved, the problem drives the requirements. Because the requirements drive the design of the program, the requirements must be validated before designing begins.

As many of the software life cycle models indicate, this step may be revisited many times during the development of the program. Requirements may prove to be impossible to meet, or may produce problems that cannot be solved without changing the requirements. If the requirements are changed, they must be reanalyzed and verified to solve the problem.

Then comes the design. Section 31.4 discusses the stepwise refinement of the program. The decomposition of the program into modules allows us to test the program as it is being implemented. Then, once it has been completed, the testing of the entire program should demonstrate that the program meets its requirements in the given environment.

The general philosophy of testing is to execute all possible paths of control and compare the results with the expected results. In practice, the paths of control are too numerous to test exhaustively. Instead, the paths are analyzed and ordered. Test data is generated for each path, and the testers compare the results obtained from the actual data with the expected results. This continues until as many paths as possible have been tested.

For security testing, the testers must test not only the most commonly used paths but also the *least commonly used* paths.[64] The latter often create security problems that attackers can exploit. Because they are relatively unused, traditional testing places them at a lower priority than that of other paths. Hence, they are not as well scrutinized, and vulnerabilities are missed.

The ordering of the paths relies on the requirements. Those paths that perform multiple security checks are more critical than those that perform single (or no) security checks because they introduce interfaces that affect security requirements. The other paths affect security, of course, but there are no interfaces.

First, we examine a module that calls no other module. Then we examine the program as a composition of modules. We conclude by testing the installation, configuration, and use instructions.

31.6.1.1 Testing the Module

The module may invoke one or more functions. The functions return results to the caller, either directly (through return values or parameter lists) or indirectly (by manipulation of the environment). The goal of this testing is to ensure that the module exhibits correct behavior regardless of what the functions returns.

The first step is to define "correct behavior." During the design of the program, the refinement process led to the specification of the module and the module's interface. This specification defines "correct behavior," and testing will require us to check that the specification holds.

We begin by listing all interfaces to the module. We will then use this list to execute four different types of tests. The types of test are as follows:

1. *Normal data tests*. These tests provide unexceptional data. The data should be chosen to exercise as many paths of control through the module as possible.
2. *Boundary data tests*. These tests provide data that tests any limits to the interfaces. For example, if the module expects a string of up to 256 characters to be passed in, these tests invoke the module and pass in arrays of 255, 256, and 257 characters. Longer strings should also be used in an effort to overflow internal buffers. The testers can examine the source code to determine what to try. Limits here do not apply simply to arrays or strings. In the program under discussion, the lowest allowed UID is 0, for *root*. A good test would be to try a UID of -1 to see what happens. The module should report an error.

[64] See Section 20.3.3.1, "Security Testing."

EXAMPLE: One UNIX system had UIDs of 16 bits. The system used a file server that would not allow a client's *root* user to access any files. Instead, it remapped root's UID to the public UID of −2. Because that UID was not assigned to any user, the remapped root could access only those files that were available to all users. The limit problem arose because one user, named Mike, had the UID 65534. Because 65534 = −2 in two's complement 16-bit arithmetic, the remote root user could access all of Mike's files—even those that were not publicly available.

3. *Exception tests*. These tests determine how the program handles interrupts and traps. For example, many systems allow the user to send a signal that causes the program to trap to a signal handler, or to take a default action such as dumping the contents of memory to a core file. These tests determine if the module leaves the system in a nonsecure state—for example, by leaving sensitive information in the memory dump. They also analyze what the process does if ordinary actions (such as writing to a file) fail.

EXAMPLE: An FTP server ran on a system that kept its authentication information confidential. An attacker found that she could cause the system to crash by sending an unexpected sequence of commands, causing multiple signals to be generated before the first signal could be handled. The crash resulted in a core dump. Because the server would be restarted automatically, the attacker simply connected again and downloaded the core dump. From that dump, she extracted the authentication information and used a dictionary attack[65] to obtain the passwords of several users.

4. *Random data tests*. These tests supply inputs generated at random and observe how the module reacts. They should not corrupt the state of the system. If the module fails, it should restore the system to a safe state.[66]

EXAMPLE: In a study of UNIX utilities [1345], approximately 30% crashed when given random inputs. In one case, an unprivileged program caused the system to crash. In 1995, a retest showed some improvement, but still "significant rates of failure" [1346, p. 1]. Other tested systems fared little better [705, 1344].

Throughout the testing, the testers should keep track of the paths taken. This allows them to determine how complete the testing is. Because these tests are highly informal, the assurance they provide is not as convincing as the techniques discussed in Chapter 20. However, it is more than random tests, or no tests, would provide.

---

[65] See Section 13.4, "Attacking Passwords."

[66] See Section 14.2.2, "Principle of Fail-Safe Defaults."

31.6.2 Testing Composed Modules

Now consider a module that calls other modules. Each of the invoked modules has a specification describing its actions. So, in addition to the tests discussed in the preceding section, one other type of test should be performed.

5. *Error handling tests*. These tests assume that the called modules violate their specifications in some way. The goal of these tests is to determine how robust the caller is. If it fails gracefully, and restores the system to a safe state, then the module passes the test. Otherwise, it fails and must be rewritten.

EXAMPLE: Assume that a security-related program, running with *root* privileges, logs all network connections to a UNIX system. It also sends mail to the network administrator with the name of the connecting host on the subject line. To do this, it executes a command such as

```
mail -s hostname netadmin
```

where *hostname* is the name of the connecting host. This module obtains *hostname* from a different module that is passed the connecting host's IP address and uses the Domain Name Service to find the corresponding host name. A serious problem arose because the DNS did not verify that *hostname* was composed of legal characters. The effects were discovered when one attacker changed the name of his host to

```
hi nobody; rm -rf *; true
```

causing the security-related program to delete critical files. Had the calling module expected failure, and checked for it, the error would have been caught before any damage was done.

31.6.3 Testing the Program

Once the testers have assembled the program and its documentation, the final phase of testing begins. The testers have someone follow the installation and configuration instructions. This person should not be a member of the testing team, because the testing team has been working with the program and is familiar with it. The goal of this test is to determine if the installation and configuration instructions are correct and easy to understand. The principle of least astonishment[67] requires that the tool be as easy to install and use as possible. Because most installers and users will not have experience with the program, the

[67] See Section 14.2.8, "Principle of Least Astonishment."

testers need to evaluate how they will understand the documentation and whether or not they can install the program correctly by following the instructions. An incorrectly installed security tool does not provide security; it may well detract from it. Worse, it gives people a false sense of security.

31.7 Distribution

Once the program has been completed, it must be distributed. Distribution involves placing the program in a repository where it cannot be altered except by authorized people, and from which it can be retrieved and sent to the intended recipients. This requires a policy for distribution. Specific factors to be considered are as follows.

1. *Who can use the program*? If the program is licensed to a specific organization, or to a specific host, then each copy of the program that is distributed must be tied to that organization or host so it cannot be redistributed or pirated. This is an originator controlled policy.[68] One approach is to provide the licensee with a secret key and encipher the software with the same key. This prevents redistribution without the licensee's consent, unless the attacker breaks the cryptosystem or steals the licensee's key.[69]
2. *How can the integrity of the master copy be protected*? If an attacker can alter the master copy, from which distribution copies are made, then the attacker can compromise all who use the program.

EXAMPLE: The program *tcp_wrappers* provides host-level access control for network servers. It is one of the most widely used programs in the UNIX community. In 1996, attackers broke into the site from which that program could be obtained [2238]. They altered the program to allow all connections to succeed. More than 50 groups obtained the program before the break-in was detected.

Part of the problem is credibility. If an attacker can pose as the vendor, then all who obtain the program from the attacker will be vulnerable to attack. This tactic undermines trust in the program and can be surprisingly hard to counter. It is analogous to generating a cryptographic checksum for a program infected with a computer virus.[70] When an uninfected program is obtained, the integrity checker complains because the checksum is wrong. In our example, when the real vendor

[68]See Section 8.3, "Originator Controlled Access Control."
[69]See Section 14.2.5, "Principle of Open Design."
[70]See Section 23.9.1, "Scanning Defenses."

contacts the duped customer, the customer usually reacts with disbelief, or is unwilling to concede that his system has been compromised.

3. *How can the availability of the program be ensured*? If the program is sent through a physical medium, such as a read-only DVD, availability is equivalent to the availability of mail or messenger services between the vendor and the buyer. If the program is distributed through electronic means, however, the distributor must take precautions to ensure that the distribution site is available. Denial of service attacks such as SYN flooding may hamper the availability.

Like a program, the distribution is controlled by a policy. All considerations that affect a security policy affect the distribution policy as well.

31.8 Summary

This chapter discussed informal techniques for writing programs that enforce security policies. The process began with a requirements analysis and continued with a threat analysis to show that the requirements countered the threats. The design process came next, and it fed back into the requirements to clarify an ambiguity. Once the high-level design was accepted, we used a stepwise refinement process to break the design down into modules and a caller. The categories of flaws in the program analysis vulnerability helped find potential implementation problems. Finally, issues of testing and distribution ensured that the program did what was required.

31.9 Research Issues

The first research issue has to do with analysis of code. How can one analyze programs to discover security flaws? This differs from the sort of analysis that is performed in the development of high-assurance systems, because the program and system are already in place. The goal is to determine what, and where, the problems are. Some researchers are developing analysis tools for specific problems such as buffer overflows and race conditions. Others are using flow analysis tools to study the program for a wide variety of vulnerabilities.

Related to this issue is the development of languages that are safer with respect to security. For example, some languages automatically create an exception if a reference is made beyond the bounds of an array. How much overhead does this add? Can the language use special-purpose hardware to minimize the impact of checking the references? What else should a language constrain, and how should it do so?

31.10 Further Reading

Robust programming—the art of writing programs that work correctly and handle errors gracefully—is a topic of great interest, often in the guise of "secure programming." Kernighan and Plauger's book [1039] describes the principles and ideas underlying good programming style. Kernighan and Pike [1040] also discuss style and other elements of good programming. Stavely's book [1819] combines formalisms with informal steps. Maguire's book [1234] is much more informal, and is a collection of tips on how to write robust programs. Martin [1257] focuses on robust practices for agile programming, while McConnell [1277] discusses robust programming in the general context of software construction.

Howard and LeBlanc [926] discuss secure coding, emphasizing the Windows and .NET environment. Howard, LeBlanc, and Viega's book [927] describes 24 serious but common software flaws and how programmers can avoid them.

Much focus is on the C and C++ programming languages, because of their wide use, lack of type-safe features, and ability to manipulate memory directly. Seacord [1704] and Viega and Messier [1935] discuss ways to make programs in these languages more robust and secure. Sutter and Alexandrescu [1843] present a set of coding standards for C++. Similarly, developing robust, secure web applications is critical, and several books [119, 1241, 1393, 1734] discuss how to do so.

Graff and van Wyk [804] provide a general overview of principles and practice, and much sound advice. Viega and McGraw's book [1932] is also general, with many examples focusing on UNIX and Linux systems. Its design principles give good advice. McGraw [1287] expands on these in a later book. Garfinkel, Schwartz, and Spafford [747] has a wonderful chapter on trust, which is must reading for anyone interested in security-related programming. Wheeler [2000] also provides valuable information and insight.

31.11 Exercises

1. Consider the two interpretations of a time field that specifies "1 a.m." One interpretation says that this means exactly 1:00 a.m. and no other time. The other says that this means any time during the 1 a.m. hour.
   a. How would you express the time of "exactly 1 a.m." in the second interpretation?
   b. How would you express "any time during the 1 a.m. hour" in the first interpretation?
   c. Which is more powerful? If they are equally powerful, which do you think is least astonishing? Why?

2. Verify that the modified version of Requirement 31.4 shown as Requirement 31.6 on page 1105 counters the appropriate threats.

3. Assume the alternative interpretation of Requirement 31.4 given in Section 31.3.1.2, so that access only is controlled by location and time, and that commands are restricted by role and user. This means that if a user is authorized to run a command, she can run it from any location he is authorized to use. How would you change the way information is stored in the access control file described in Section 31.3.2.2?

4. Currently, the program described in this chapter is to have setuid-to-*root* privileges. Someone observed that it could be equally well-implemented as a server, in which case the program would authenticate the user, connect to the server, send the command and role, and then let the server execute the command.

   a. What are the advantages of using the server approach rather than the single program approach?
   b. If the server responds only to clients on the local machine, using interprocess communication mechanisms on the local system, which approach would you use? Why?
   c. If the server were listening for commands from the network, would that change your answer to the previous question? Why or why not?
   d. If the client sent the password to the server, and the server authenticated, would your answers to any of the three previous parts change? Why or why not?

5. The little languages presented in Section 31.3.2.3 have ambiguous semantics. For example, in the location language, does "not host1 or host2" mean "neither at host1 nor at host2" or "at host2 or not at host1"?

   a. Rewrite the BNF of the location language to make the semantics reflect the second meaning (i.e., the precedence of "not" is lower than that of "or"). Are the semantics unambiguous now? Why or why not?
   b. Rewrite the BNF of the time language to make the semantics reflect the second meaning (i.e., the precedence of "not" is higher than that of "or"). Are the semantics unambiguous now? Why or why not?

6. Suppose an access control record is malformed (for example, it has a syntax error). Show that the access control module would deny access.

7. The canary for StackGuard simply detects overflow that might change the return address. This exercise asks you to extend the notion of a canary to detection of buffer overflow.

   a. Assume that the canary is placed directly after the array, and that after every array, access is checked to see if it has changed. Would this detect a buffer overflow? If so, why do you think this is not suitable for use in practice? If not, describe an attack that could change a number beyond the buffer without affecting the canary.

b. Now suppose that the canary was placed directly after the buffer but—like the canary for StackGuard—was only checked just before a function return. How effective do you think this method would be?

Index

B

C

D

E

F

G

H

M

N

Q

R

S

T

V

W