THE ART OF SOFTWARE SECURITY ASSESSMENT

Identifying and Avoiding Software Vulnerabilities

MARK DOWD
JOHN McDONALD

THE ART OF

SOFTWARE SECURITY ASSESSMENT

This page intentionally left blank

THE ART OF

SOFTWARE SECURITY ASSESSMENT

IDENTIFYING AND PREVENTING SOFTWARE VULNERABILITIES

MARK DOWD
JOHN MCDONALD
JUSTIN SCHUH

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ISBN 0-321-44442-6
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, November 2006

Library of Congress Cataloging-in-Publication Data

Dowd, Mark.
The art of software security assessment : identifying and preventing software vulnerabilities / Mark Dowd, John McDonald, and Justin Schuh.
p. cm.
ISBN 0-321-44442-6 (pbk. : alk. paper) 1. Computer security. 2. Computer software—Development. 3. Computer networks—Security measures. I. McDonald, John, 1977- II. Schuh, Justin. III. Title.
QA76.9.A25D75 2006
005.8—dc22

2006023446

Table of Contents

About the Authors

Mark Dowd is a principal security architect at McAfee, Inc. and an established expert in the field of application security. His professional experience includes several years as a senior researcher at Internet Security Systems (ISS) X-Force, and the discovery of a number of high-profile vulnerabilities in ubiquitous Internet software. He is responsible for identifying and helping to address critical flaws in Sendmail, Microsoft Exchange Server, OpenSSH, Internet Explorer, Mozilla (Firefox), Checkpoint VPN, and Microsoft's SSL implementation. In addition to his research work, Mark presents at industry conferences, including Black Hat and RUXCON.

John McDonald is a senior consultant with Neohapsis, where he specializes in advanced application security assessment across a broad range of technologies and platforms. He has an established reputation in software security, including work in security architecture and vulnerability research for NAI (now McAfee), Data Protect GmbH, and Citibank. As a vulnerability researcher, John has identified and helped resolve numerous critical vulnerabilities, including issues in Solaris, BSD, Checkpoint FireWall-1, OpenSSL, and BIND.

Justin Schuh is a senior consultant with Neohapsis, where he leads the Application Security Practice. As a senior consultant and practice lead, he performs software security assessments across a range of systems, from embedded device firmware to distributed enterprise web applications. Prior to his employment with Neohapsis, Justin spent nearly a decade in computer security activities at the Department of Defense (DoD) and related agencies. His government service includes a role as a lead researcher with the National Security Agency (NSA) penetration testing team—the Red Team.

This page intentionally left blank

Preface

"If popular culture has taught us anything, it is that someday mankind must face and destroy the growing robot menace."

Daniel H. Wilson, *How to Survive a Robot Uprising*

The past several years have seen huge strides in computer security, particularly in the field of software vulnerabilities. It seems as though every stop at the bookstore introduces a new title on topics such as secure development or exploiting software.

Books that cover application security tend to do so from the perspective of software designers and developers and focus on techniques to prevent software vulnerabilities from occurring in applications. These techniques start with solid security design principles and threat modeling and carry all the way through to implementation best practices and defensive programming strategies. Although they serve as strong defensive foundations for application development, these resources tend to give little treatment to the nature of vulnerabilities; instead, they focus on how to avoid them. What's more, every development team can't start rebuilding a secure application from the ground up. Real people have to deal with huge existing codebases, in-place applications, and limited time and budget. Meanwhile, the secure coding mantra seems to be "If it smells bad, throw it out." That's certainly necessary in some cases, but often it's too expensive and time consuming to be reasonable. So you might turn your attention to penetration testing and ethical hacking instead. A wide range of information on this topic is available, and it's certainly useful for the acid test of a software system. However, even the most technically detailed resources have a strong focus on exploit development and little to no treatment on how to find vulnerabilities in the first place. This still leaves the hanging question of how to find issues in an existing application and how to get a reasonable degree of assurance that a piece of software is safe.

This problem is exactly the one faced by those in the field of professional software security assessment. People are growing more concerned with building and testing secure systems, but very few resources address the practice of finding vulnerabilities. After all, this process requires a deep technical understanding of some very complex issues and must include a systematic approach to analyzing an application. Without formally addressing how to find vulnerabilities, the software security industry has no way of establishing the quality of a software security assessment or training the next generation in the craft. We have written this book in the hope of answering these questions and to help bridge the gap between secure software development and practical post-implementation reviews. Although this book is aimed primarily at consultants and other security professionals, much of the material will have value to the rest of the IT community as well. Developers can gain insight into the subtleties and nuances of how languages and operating systems work and how those features can introduce vulnerabilities into an application that otherwise appears secure. Quality assurance (QA) personnel can use some of the guidelines in this book to ensure the integrity of in-house software and cut down on the likelihood of their applications being stung by a major vulnerability. Administrators can find helpful guidelines for evaluating the security impact of applications on their networks and use this knowledge to make better decisions about future deployments. Finally, hobbyists who are simply interested in learning more about how to assess applications will find this book an invaluable resource (we hope!) for getting started in application security review or advancing their current skill sets.

Prerequisites

The majority of this book has been targeted at a level that any moderately experienced developer should find approachable. This means you need to be fairly comfortable with at least one programming language, and ideally, you should be familiar with basic C/C++ programming. At several stages throughout the book, we use Intel assembly examples, but we have attempted to keep them to a minimum and translate them into approximate C code when possible. We have also put a lot of effort into making the material as platform neutral as possible, although we do cover platform specifics for the most common operating systems. When necessary, we have tried to include references to additional resources that provide background for material that can't be covered adequately in this book.

How to Use This Book

Before we discuss the use of this book, we need to introduce its basic structure. The book is divided into three different parts:

- *Part I: Introduction to Software Security Assessment (Chapters 1–4)*—These chapters introduce the practice of code auditing and explain how it fits into the software development process. You learn about the function of design review, threat modeling, and operational review—tools that are useful for evaluating an application as a whole, and not just the code. Finally, you learn some generic high-level methods for performing a code review on any application, regardless of its function or size.
- *Part II: Software Vulnerabilities (Chapters 5–13)*—These chapters shift the focus of the book toward practical implementation review and address how to find specific vulnerabilities in an application's codebase. Major software vulnerability classes are described, and you learn how to discover high-risk security flaws in an application. Numerous real-world examples of security vulnerabilities are given to help you get a feel for what software bugs look like in real code.
- *Part III: Software Vulnerabilities in Practice (Chapters 14–18)*—The final portion of the book turns your attention toward practical uses of lessons learned from the earlier chapters. These chapters describe a number of common application classes and the types of bugs they tend to be vulnerable to. They also show you how to apply the technical knowledge gained from Part II to real-world applications. Specifically, you look at networking, firewalling technologies, and Web technologies. Each chapter in this section introduces the common frameworks and designs of each application class and identifies where flaws typically occur.

You'll get the most value if you read this book straight through at least once so that you can get a feel for the material. This approach is best because we have tried to use each section as an opportunity to highlight techniques and tools that help you in performing application assessments. In particular, you should pay attention to the sidebars and notes we use to sum up the more important concepts in a section.

Of course, busy schedules and impending deadlines can have a serious impact on your time. To that end, we want to lay out a few tracks of focus for different types of reviews. However, you should start with Part 1 (Chapters 1–4) because it establishes a foundation for the rest of the book. After that, you can branch out to the following chapters:

- *UNIX track (Chapters 5–10, 13)*—This chapter track starts off by covering common software vulnerability classes, such as memory corruption, program control flow, and specially formatted data. Then UNIX-centered security problems that arise because of quirks in the various UNIX operating systems are addressed. Finally, this track ends with coverage of synchronization vulnerabilities common to most platforms.

- *Windows track (Chapters 5–8, 11–13)*—This track starts off similarly to the UNIX track, by covering platform-neutral security problems. Then two chapters specifically address Windows APIs and their related vulnerabilities. Finally, this track finishes with coverage of common synchronization vulnerabilities.
- *Web track (Chapters 8, 13, 17, 18)*—Web auditing requires understanding common security vulnerabilities as well as Web-based frameworks and languages. This track discusses the common vulnerability classes that pertain to Web-based languages, and then finishes off with the Web-specific chapters. Although the UNIX and Windows chapters aren't listed here, reading them might be necessary depending on the Web application's deployment environment.
- *Network application track (Chapters 5–8, 13, 16)*—This sequence of chapters best addresses the types of vulnerabilities you're likely to encounter with network client/server applications. Notice that even though Chapter 16 is targeted at selected application protocols, it has a section for generic application protocol auditing methods. Like the previous track, UNIX or Windows chapters might also be relevant, depending on the deployment environment.
- *Network analysis track (Chapters 5–8, 13–16)*—This track is aimed at analyzing network analysis applications, such as firewalls, IPSs, sniffers, routing software, and so on. Coverage includes standard vulnerability classes along with popular network-based technologies and the common vulnerabilities in these products. Again, the UNIX and Windows chapters would be a good addition to this track, if applicable.

Acknowledgments

Mark: To my family, friends, and colleagues, for supporting me and providing encouragement throughout this endeavor.

John: To my girlfriend Jess, my family and friends, Neohapsis, Vincent Howard, Dave Aitel, David Leblanc, Thomas Lopatic, and Howard Kirk.

Justin: To my wife Cat, my coworkers at Neohapsis, my family and friends, and everyone at a three-letter agency who kept me out of trouble.

We would collectively like to thank reviewers, friends, and colleagues who have given invaluable feedback, suggestions, and comments that helped shape this book into the finished product you see today. In particular, we would like to acknowledge Neel Mehta, Halvar Flake, John Viega, and Nishad Herath for their tireless efforts in reviewing and helping to give us technical and organizational direction. We'd also like to thank the entire publishing team at Addison-Wesley for working with us to ensure the highest-quality finished product possible.

This page intentionally left blank

PART I

INTRODUCTION TO SOFTWARE SECURITY ASSESSMENT

This page intentionally left blank

Chapter 1

Software Vulnerability Fundamentals

"Any sufficiently advanced technology is indistinguishable from magic."
Arthur C. Clarke

Introduction

The average person tends to think of software as a form of technological wizardry simply beyond understanding. A piece of software might have complexity that rivals any physical hardware, but most people never see its wheels spin, hear the hum of its engine, or take apart the nuts and bolts to see what makes it tick. Yet computer software has become such an integral part of society that it affects almost every aspect of people's daily lives. This wide-reaching effect inevitably raises questions about the security of systems that people have become so dependent on. You can't help but wonder whether the software you use is really secure. How can you verify that it is? What are the implications of a failure in software security?

Over the course of this book, you'll learn about the tools you need to understand and assess software security. You'll see how to apply the theory and practice of code auditing; this process includes learning how to dissect an application, discover security

vulnerabilities, and assess the danger each vulnerability presents. You also learn how to maximize your time, focusing on the most security-relevant elements of an application and prioritizing your efforts to help identify the most critical vulnerabilities first. This knowledge provides the foundation you need to perform a comprehensive security assessment of an application.

This chapter introduces the elements of a software vulnerability and explains what it means to violate the security of a software system. You also learn about the elements of software assessment, including motivation, types of auditing, and how an audit fits in with the development process. Finally, some distinctions are pointed out to help you classify software vulnerabilities and address the common causes of these security issues.

Vulnerabilities

There's almost an air of magic when you first see a modern remote software exploit deployed. It's amazing to think that a complex program, written by a team of experts and deployed around the world for more than a decade, can suddenly be co-opted by attackers for their own means. At first glance, it's easy to consider the process as some form of digital voodoo because it simply shouldn't be possible. Like any magic trick, however, this sense of wonder fades when you peek behind the curtain and see how it works. After all, software vulnerabilities are simply weaknesses in a system that attackers can leverage to their advantage. In the context of software security, **vulnerabilities** are specific flaws or oversights in a piece of software that allow attackers to do something malicious—expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program.

You're no doubt familiar with software **bugs**; they are errors, mistakes, or oversights in programs that result in unexpected and typically undesirable behavior. Almost every computer user has lost an important piece of work because of a software bug. In general, software vulnerabilities can be thought of as a subset of the larger phenomenon of software bugs. Security vulnerabilities are bugs that pack an extra hidden surprise: A malicious user can leverage them to launch attacks against the software and supporting systems. Almost all security vulnerabilities are software bugs, but only some software bugs turn out to be security vulnerabilities. A bug must have some security-relevant impact or properties to be considered a security issue; in other words, it has to allow attackers to do something they normally wouldn't be able to do. (This topic is revisited in later chapters, as it's a common mistake to mischaracterize a major security flaw as an innocuous bug.)

There's a common saying that security is a subset of reliability. This saying might not pass muster as a universal truth, but it does draw a useful comparison. A reliable program is one that's relatively free of software bugs: It rarely fails on

users, and it handles exceptional conditions gracefully. It's written "defensively" so that it can handle uncertain execution environments and malformed inputs. A secure program is similar to a robust program: It can repel a focused attack by intruders who are attempting to manipulate its environment and input so that they can leverage it to achieve some nefarious end. Software security and reliability also share similar goals, in that they both necessitate development strategies that focus on exterminating software bugs.

> **Note**
> Although the comparison of security flaws to software bugs is useful, some vulnerabilities don't map so cleanly. For example, a program that allows you to edit a critical system file you shouldn't have access to might be operating completely correctly according to its specifications and design. So it probably wouldn't fall under most people's definition of a software bug, but it's definitely a security vulnerability.

The process of attacking a vulnerability in a program is called **exploiting.** Attackers might exploit a vulnerability by running the program in a clever way, altering or monitoring the program's environment while it runs, or if the program is inherently insecure, simply using the program for its intended purpose. When attackers use an external program or script to perform an attack, this attacking program is often called an **exploit** or **exploit script.**

Security Policies

As mentioned, attackers can exploit a vulnerability to violate the security of a system. One useful way to conceptualize the "security of a system" is to think of a system's security as being defined by a security policy. From this perspective, a violation of a software system's security occurs when the system's security policy is violated.

> **Note**
> Matt Bishop, a computer science professor at University of California–Davis, is an accomplished security researcher who has been researching and studying computer vulnerabilities for many years. Needless to say, he's put a lot of thought into computer security from a formal academic perspective as well as a technical perspective. If these topics interest you, check out his book, *Computer Security: Art and Science* (Addison-Wesley, 2003), and the resources at his home page: http://nob.cs.ucdavis.edu/~bishop/.

For a system composed of software, users, and resources, you have a **security policy**, which is simply a list of what's allowed and what's forbidden. This policy might state, for example, "Unauthenticated users are forbidden from using the calendar service on the staging machine." A problem that allows unauthenticated users to access the staging machine's calendar service would clearly violate the security policy.

Every software system can be considered to have a security policy. It might be a formal policy consisting of written documents, or it might be an informal loose collection of expectations that the software's users have about what constitutes reasonable behavior for that system. For most software systems, people usually understand what behavior constitutes a violation of security, even if it hasn't been stated explicitly. Therefore, the term "security policy" often means the user community's consensus on what system behavior is allowed and what system behavior is forbidden. This policy could take a few different forms, as described in the following list:

- For a particularly sensitive and tightly scoped system, a security policy could be a formal specification of constraints that can be verified against the program code by mathematical proof. This approach is often expensive and applicable only to an extremely controlled software environment. You would hope that embedded systems in devices such as traffic lights, elevators, airplanes, and life support equipment go through this kind of verification. Unfortunately, this approach is prohibitively expensive or unwieldy, even for many of those applications.
- A security policy could be a formal, written document with clauses such as "C.2. Credit card information (A.1.13) should never be disclosed to a third party (as defined in A.1.3) or transferred across any transmission media without sufficient encryption, as specified in Addendum Q." This clause could come from a policy written about the software, perhaps one created during the development process. It could also come from policies related to resources the software uses, such as a site security policy, an operating system (OS) policy, or a database security policy.
- The security policy could be composed solely of an informal, slightly ambiguous collection of people's expectations of reasonable program security behavior, such as "Yeah, giving a criminal organization access to our credit card database is probably bad."

Note
The Java Virtual Machine (JVM) and .NET Common Language Runtime (CLR) have varying degrees of code access security (CAS). CAS provides a means of extensively validating a package at both load time and runtime. These validations include the

integrity of the bytecode, the software's originator, and the application of code access restrictions. The most obvious applications of these technologies include the sandbox environments for Java applets and .NET-managed browser controls.

Although CAS can be used as a platform for a rigidly formalized security model, some important caveats are associated with it. The first concern is that most developers don't thoroughly understand its application and function, so it's rarely leveraged in commercial software. The second concern is that the security provided by CAS depends entirely on the security of underlying components. Both the Java VM and the .NET CLR have been victims of vulnerabilities that could allow an application to escape the virtual machine sandbox and run arbitrary code.

In practice, a software system's security policy is likely to be mostly informal and made up of people's expectations. However, it often borrows from formal documentation from the development process and references site and resource security policies. This definition of a system security policy helps clarify the concept of "system security." The bottom line is that security is in the eye of the beholder, and it boils down to end users' requirements and expectations.

Security Expectations

Considering the possible expectations people have about software security helps determine which issues they consider to be security violations. Security is often described as resting on three components: confidentiality, integrity, and availability. The following sections consider possible expectations for software security from the perspective of these cornerstones.

Confidentiality

Confidentiality requires that information be kept private. This includes any situation where software is expected to hide information or hide the existence of information. Software systems often deal with data that contains secrets, ranging from nation- or state-level intelligence secrets to company trade secrets or even sensitive personal information.

Businesses and other organizations have plenty of secrets residing in their software. Financial information is generally expected to be kept confidential. Information about plans and performance could have strategic importance and is potentially useful for an unlawful competitive advantage or for criminal activities, such as insider trading. So businesses expect that data to be kept confidential as

well. Data involving business relationships, contracts, lawsuits, or any other sensitive content carries an expectation of confidentiality.

If a software system maintains information about people, expectations about the confidentiality of that data are often high. Because of **privacy** concerns, organizations and users expect a software system to carefully control who can view details related to people. If the information contains financial details or medical records, improper disclosure of the data might involve liability issues. Software is often expected to keep personal user information secret, such as personal files, e-mail, activity histories, and accounts and passwords.

In many types of software, the actual program code constitutes a secret. It could be a trade secret, such as code for evaluating a potential transaction in a commodities market or a new 3D graphics engine. Even if it's not a trade secret, it could still be sensitive, such as code for evaluating credit risks of potential loan applicants or the algorithm behind an online videogame's combat system.

Software is often expected to compartmentalize information and ensure that only authenticated parties are allowed to see information for which they're authorized. These requirements mean that software is often expected to use access control technology to authenticate users and to check their authorization when accessing data. Encryption is also used to maintain the confidentiality of data when it's transferred or stored.

Integrity

Integrity is the trustworthiness and correctness of data. It refers to expectations that people have about software's capability to prevent data from being altered. Integrity refers not only to the contents of a piece of data, but also to the source of that data. Software can maintain integrity by preventing unauthorized changes to data sources. Other software might detect changes to data integrity by making note of a change in a piece of data or an alteration of the data's origins.

Software integrity often involves compartmentalization of information, in which the software uses access control technology to authenticate users and check their authorization before they're allowed to modify data. Authentication is also an important component of software that's expected to preserve the integrity of the data's source because it tells the software definitively who the user is.

Typically, users hold similar expectations for integrity as they do for confidentiality. Any issue that allows attackers to modify information they wouldn't otherwise be permitted to modify is considered a security flaw. Any issue that allows users to masquerade as other users and manipulate data is also considered a breach of data integrity.

Software vulnerabilities can be particularly devastating in breaches of integrity, as the modification of data can often be leveraged to further an attackers' access into a software system and the computing resources that host the software.

Availability

Availability is the capability to use information and resources. Generally, it refers to expectations users have about a system's availability and its resilience to denial-of-service (DoS) attacks.

An issue that allows users to easily crash or disrupt a piece of software would likely be considered a vulnerability that violates users' expectations of availability. This issue generally includes attacks that use specific inputs or environmental disruptions to disable a program as well as attacks centered on exhausting software system resources, such as CPU, disk, or network bandwidth.

The Necessity of Auditing

Most people expect vendors to provide some degree of assurance about the integrity of their software. The sad truth is that vendors offer few guarantees of quality for any software. If you doubt this, just read the end user license agreement (EULA) that accompanies almost every piece of commercial software. However, it's in a company's best interests to keep clients happy; so most vendors implement their own quality assurance measures. These measures usually focus on marketable concerns, such as features, availability, and general stability; this focus has historically left security haphazardly applied or occasionally ignored entirely.

> **Note**
> Some industries do impose their own security requirements and standards, but they typically involve regulatory interests and apply only to certain specialized environments and applications. This practice is changing, however, as high-profile incidents are moving regulators and industry standards bodies toward more proactive security requirements.

The good news is that attitudes toward security have been changing recently, and many vendors are adopting business processes for more rigorous security testing. Many approaches are becoming commonplace, including automated code analysis, security unit testing, and manual code audits. As you can tell from the title, this book focuses on manual code audits.

Auditing an application is the process of analyzing application code (in source or binary form) to uncover vulnerabilities that attackers might exploit. By going through this process, you can identify and close security holes that would otherwise put sensitive data and business resources at unnecessary risk.

In addition to the obvious case of a company developing in-house software, code auditing makes sense in several other situations. Table 1-1 summarizes the most common ones.

Table 1-1

Code-Auditing Situations		
Situation	**Description**	**Advantage**
In-house software audit (prerelease)	A software company performs code audits of a new product before its release.	Design and implementation flaws can be identified and remedied before the product goes to market, saving money in developing and deploying updates. It also saves the company from potential embarrassment.
In-house software audit (postrelease)	A software company performs code audits of a product after its release.	Security vulnerabilities can be found and fixed before malicious parties discover the flaws. This process allows time to perform testing and other checks as opposed to doing a hurried release in response to a vulnerability disclosure.
Third-party product range comparison	A third party performs audits of a number of competing products in a particular field.	An objective third party can provide valuable information to consumers and assist in selecting the most secure product.
Third-party evaluation	A third party performs an independent software audit of a product for a client.	The client can gain an understanding of the relative security of an application it's considering deploying. This might prove to be the deciding factor between purchasing one technology over another.
Third-party preliminary evaluation	A third party performs an independent review of a product before it goes to market.	Venture capitalists can get an idea of the viability of a prospective technology for investment purposes. Vendors might also conduct this type of evaluation to ensure the quality of a product they intend to market.
Independent research	A security company or consulting firm performs a software audit independently.	Security product vendors can identify vulnerabilities and implement protective measures in scanners and other security devices. Independent research also functions as an industry watchdog and provides a way for researchers and security companies to establish professional credibility.

As you can see, code auditing makes sense in quite a few situations. Despite the demand for people with these skills, however, few professionals have the training and experience to perform these audits at a high standard. It's our hope that this book helps fill that gap.

Auditing Versus Black Box Testing

Black box testing is a method of evaluating a software system by manipulating only its exposed interfaces. Typically, this process involves generating specially crafted inputs that are likely to cause the application to perform some unexpected behavior, such as crashing or exposing sensitive data. For example, black box testing an HTTP server might involve sending requests with abnormally large field sizes, which could trigger a memory corruption bug (covered in more depth later in Chapter 5, "Memory Corruption"). This test might involve a legitimate request, such as the following (assume that the "..." sequence represents a much longer series of "A" characters):

```
GET AAAAAAAAAAAAAAAAAA...AAAAAAAAAAAAAAAAAA HTTP/1.0
```

Or it might involve an invalid request, such as this one (once again, the "..." sequence represents a much longer series of "A" characters):

```
GET / AAAAAAAAAAAAAAAAAA...AAAAAAAAAAAAAAAAAA/1.0
```

Any crashes resulting from these requests would imply a fairly serious bug in the application. This approach is even more appealing when you consider that tools to automate the process of testing applications are available. This process of automated black box testing is called fuzz-testing, and fuzz-testing tools include generic "dumb" and protocol-aware "intelligent" fuzzers. So you don't need to manually try out every case you can think of; you simply run the tool, perhaps with some modifications of your own design, and collect the results.

The advantage of black box testing an application is that you can do it quickly and possibly have results almost immediately. However, it's not all good news; there are several important disadvantages of black box testing. Essentially, black box testing is just throwing a bunch of data at an application and hoping it does something it isn't supposed to do. You really have no idea what the application is doing with the data, so there are potentially hundreds of code paths you haven't explored because the data you throw at the application doesn't trigger those paths. For instance, returning to the Web server example, imagine that it has certain internal functionality if particular keywords are present in the query string of a request. Take a look at the following code snippet, paying close attention to the bolded lines:

```
struct keyval {
    char *key;
    char *value;
};

int handle_query_string(char *query_string)
{
    struct keyval *qstring_values, *ent;
    char buf[1024];

    if(!query_string)
        return 0;

    qstring_values = split_keyvalue_pairs(query_string);

    if((ent = find_entry(qstring_values, "mode")) != NULL)
    {
        sprintf(buf, "MODE=%s", ent->value);
        putenv(buf);
    }

    ... more stuff here ...
}
```

This Web server has a specialized nonstandard behavior; if the query string contains the sequence `mode=xxx`, the environment variable `MODE` is set with the value `xxx`. This specialized behavior has an implementation flaw, however; a buffer overflow caused by a careless use of the `sprintf()` function. If you aren't sure why this code is dangerous, don't worry; buffer overflow vulnerabilities are covered in depth in Chapter 5.

You can see the bug right away by examining the code, but a black box or fuzz-testing tool would probably miss this basic vulnerability. Therefore, you need to be able to assess code constructs intelligently in addition to just running testing tools and noting the results. That's why code auditing is important. You need to be able to analyze code and detect code paths that an automated tool might miss as well as locate vulnerabilities that automated tools can't catch.

Fortunately, code auditing combined with black box testing provides maximum results for uncovering vulnerabilities in a minimum amount of time. This book arms

you with the knowledge and techniques to thoroughly analyze an application for a wide range of vulnerabilities and provides insight into how you can use your understanding and creativity to discover flaws unique to a particular application.

Code Auditing and the Development Life Cycle

When you consider the risks of exposing an application to potentially malicious users, the value of application security assessment is clear. However, you need to know exactly when to perform an assessment. Generally, you can perform an audit at any stage of the **Systems Development Life Cycle (SDLC)**. However, the cost of identifying and fixing vulnerabilities can vary widely based on when and how you choose to audit. So before you get started, review the following phases of the SDLC:

1. *Feasibility study*—This phase is concerned with identifying the needs the project should meet and determining whether developing the solution is technologically and financially viable.
2. *Requirements definition*—In this phase, a more in-depth study of requirements for the project is done, and project goals are established.
3. *Design*—The solution is designed and decisions are made about how the system will technically achieve the agreed-on requirements.
4. *Implementation*—The application code is developed according to the design laid out in the previous phase.
5. *Integration and testing*—The solution is put through some level of quality assurance to ensure that it works as expected and to catch any bugs in the software.
6. *Operation and maintenance*—The solution is deployed and is now in use, and revisions, updates, and corrections are made as a result of user feedback.

Every software development process follows this model to some degree. Classical **waterfall** models tend toward a strict interpretation, in which the system's life span goes through only a single iteration through the model. In contrast, newer methodologies, such as **agile development**, tend to focus on refining an application by going through repeated iterations of the SDLC phases. So the way in which the SDLC model is applied might vary, but the basic concepts and phases are consistent enough for the purposes of this discussion. You can use these distinctions to help classify vulnerabilities, and in later chapters, you learn about the best phases in which to conduct different classes of reviews.

Classifying Vulnerabilities

A **vulnerability class** is a set of vulnerabilities that share some unifying commonality—a pattern or concept that isolates a specific feature shared by several different software flaws. Granted, this definition might seem a bit confusing, but the bottom line is that vulnerability classes are just mental devices for conceptualizing software flaws. They are useful for understanding issues and communicating that understanding with others, but there isn't a single, clean taxonomy for grouping vulnerabilities into accurate, nonoverlapping classes. It's quite possible for a single vulnerability to fall into multiple classes, depending on the code auditor's terminology, classification system, and perspective.

A rigid formal taxonomy for categorizing vulnerabilities isn't used in this book; instead, issues are categorized in a consistent, pragmatic fashion that lends itself to the material. Some software vulnerabilities are best tackled from a particular perspective. For example, certain flaws might best be approached by looking at a program in terms of the interaction of high-level software components; another type of flaw might best be approached by conceptualizing a program as a sequence of system calls. Regardless of the approach, this book explains the terms and concepts you'll encounter in security literature so that you can keep the array of terms and taxonomies the security community uses in some sort of context.

In defining general vulnerability classes, you can draw a few general distinctions from the discussion of the SDLC phases. Two commonly accepted vulnerability classes include design vulnerabilities (SDLC phases 1, 2, and 3) and implementation vulnerabilities (SDLC phases 4 and 5). In addition, this book includes a third category, operational vulnerabilities (SDLC phase 6). The security community generally accepts design vulnerabilities as flaws in a software system's architecture and specifications; implementation vulnerabilities are low-level technical flaws in the actual construction of a software system. The category of operational vulnerabilities addresses flaws that arise in deploying and configuring software in a particular environment.

Design Vulnerabilities

A **design vulnerability** is a problem that arises from a fundamental mistake or oversight in the software's design. With a design flaw, the software isn't secure because it does exactly what it was designed to do; it was simply designed to do the wrong thing! These types of flaws often occur because of assumptions made about the environment in which a program will run or the risk of exposure that program components will face in the actual production environment. Design flaws are also referred to as high-level vulnerabilities, architectural flaws, or problems with program requirements or constraints.

A quick glance at the SDLC phases reminds you that a software system's design is driven by the definition of software **requirements,** which are a list of objectives a software system must meet to accomplish the goals of its creators. Typically, an engineer takes the set of requirements and constructs design **specifications,** which focus on how to create the software that meets those goals. Requirements usually address what a software system has to accomplish—for example, "Allow a user to retrieve a transaction file from a server." Requirements can also specify capabilities the software must have—for example, "It must support 100 simultaneous downloads per hour."

Specifications are the plans for how the program should be constructed to meet the requirements. Typically, they include a description of the different components of a software system, information on how the components will be implemented and what they will do, and information on how the components will interact. Specifications could involve architecture diagrams, logic diagrams, process flowcharts, interface and protocol specifications, class hierarchies, and other technical specifications.

When people speak of a design flaw, they don't usually make a distinction between a problem with the software's requirements and a problem with the software's specifications. Making this distinction often isn't easy because many high-level issues could be explained as an oversight in the requirements or a mistake in the specifications.

For example, the TELNET protocol is designed to allow users to connect to a remote machine and access that machine as though it's connected to a local terminal. From a design perspective, TELNET arguably has a vulnerability in that it relies on unencrypted communication. In some environments, this reliance might be acceptable if the underlying network environment is trusted. However, in corporate networks and the Internet, unencrypted communications could be a major weakness because attackers sitting on the routing path can monitor and hijack TELNET sessions. If an administrator connects to a router via TELNET and enters a username and password to log in, a sniffer could record the administrator's username and password. In contrast, a protocol such as Secure Shell (SSH) serves the same basic purpose as TELNET, but it addresses the sniffing threat because it encrypts all communications.

Implementation Vulnerabilities

In an **implementation vulnerability**, the code is generally doing what it should, but there's a security problem in the way the operation is carried out. As you would expect from the name, these issues occur during the SDLC implementation phase, but they often carry over into the integration and testing phase. These problems can happen if the implementation deviates from the design to solve technical discrepancies. Mostly, however, exploitable situations are caused by technical artifacts and

nuances of the platform and language environment in which the software is constructed. Implementation vulnerabilities are also referred to as low-level flaws or technical flaws.

This book includes many examples of implementation vulnerabilities because identifying these technical flaws is one of the primary charges of the code review process. Implementation vulnerabilities encompass several well-publicized vulnerability classes you've probably heard of, such as buffer overflows and SQL injection.

Going back to the TELNET example, you can also find implementation vulnerabilities in specific versions of TELNET software. Some previous implementations of TELNET daemons didn't cleanse user environment variables correctly, allowing intruders to leverage the dynamic linking features of a UNIX machine to elevate their privileges on the machine. There were also flaws that allowed intruders to perform buffer overflows and format string attacks against various versions of TELNET daemons, often without authenticating at all. These flaws resulted in attackers being able to remotely issue arbitrary commands on the machine as privileged users. Basically, attackers could run a small exploit program against a vulnerable TELNET daemon and immediately get a root prompt on the server.

Operational Vulnerabilities

Operational vulnerabilities are security problems that arise through the operational procedures and general use of a piece of software in a specific environment. One way to distinguish these vulnerabilities is that they aren't present in the source code of the software under consideration; rather, they are rooted in how the software interacts with its environment. Specifically, they can include issues with configuration of the software in its environment, issues with configuration of supporting software and computers, and issues caused by automated and manual processes that surround the system. Operational vulnerabilities can even include certain types of attacks on users of the system, such as social engineering and theft. These issues occur in the SDLC operation and maintenance phase, although they have some overlap into the integration and testing phase.

Going back to the TELNET example, you know TELNET has a design flaw because of its lack of encryption. Say you're looking at a software system for automated securities trading. Suppose it needs a set of weighting values to be updated every night to adjust its trading strategy for the next day. The documented process for updating this data is for an administrator to log in to the machine using TELNET at the end of each business day and enter the new set of values through a simple utility program. Depending on the environment, this process could represent a major operational vulnerability because of the multiple risks associated with using TELNET, including sniffing and connection hijacking. In short, the operational procedure for maintaining the software is flawed because it exposes the system to potential fraud and attacks.

Gray Areas

The distinction between design and implementation vulnerabilities is deceptively simple in terms of the SDLC, but it's not always easy to make. Many implementation vulnerabilities could also be interpreted as situations in which the design didn't anticipate or address the problem adequately. On the flip side, you could argue that lower-level pieces of a software system are also designed, in a fashion. A programmer can design plenty of software components when implementing a specification, depending on the level of detail the specification goes into. These components might include a class, a function, a network protocol, a virtual machine, or perhaps a clever series of loops and branches. Lacking a strict distinction, in this book the following definition of a design vulnerability is used:

> In general, when people refer to design vulnerabilities, they mean high-level issues with program architecture, requirements, base interfaces, and key algorithms.

Expanding on the definition of design vulnerabilities, this book uses the following definition of an implementation vulnerability:

> Security issues in the design of low-level program pieces, such as parts of individual functions and classes, are generally considered to be implementation vulnerabilities. Implementation vulnerabilities also include more complex logical elements that are not normally addressed in the design specification. (These issues are often called logic vulnerabilities.)

Likewise, there's no clear distinction between operational vulnerabilities and implementation or design vulnerabilities. For example, if a program is installed in an environment in a fashion that isn't secure, you could easily argue that it's a failure of the design or implementation. You would expect the application to be developed in a manner that's not vulnerable to these environmental concerns. Lacking a strict distinction again, the following definition of an operational vulnerability is used in this book:

> In general, the label "operational vulnerabilities" is used for issues that deal with unsafe deployment and configuration of software, unsound management and administration practices surrounding software, issues with supporting components such as application and Web servers, and direct attacks on the software's users.

You can see that there's plenty of room for interpretation and overlap in the concepts of design, implementation, and operational vulnerabilities, so don't consider these definitions to be an infallible formal system for labeling software flaws. They are simply a useful way to approach and study software vulnerabilities.

Common Threads

So far you've learned some background on the audit process, security models, and the three common classes of vulnerabilities. This line of discussion is continued throughout the rest of this book, as you drill down into the details of specific technical issues. For now, however, take a step back to look at some common threads that underlie security vulnerabilities in software, focusing primarily on where and why vulnerabilities are most likely to surface in software.

Input and Data Flow

The majority of software vulnerabilities result from unexpected behaviors triggered by a program's response to malicious data. So the first question to address is how exactly malicious data gets accepted by the system and causes such a serious impact. The best way to explain it is by starting with a simple example of a buffer overflow vulnerability.

Consider a UNIX program that contains a buffer overflow triggered by an overly long command-line argument. In this case, the malicious data is user input that comes directly from an attacker via the command-line interface. This data travels through the program until some function uses it in an unsafe way, leading to an exploitable situation.

For most vulnerabilities, you'll find some piece of malicious data that an attacker injects into the system to trigger the exploit. However, this malicious data might come into play through a far more circuitous route than direct user input. This data can come from several different sources and through several different interfaces. It might also pass through multiple components of a system and be modified a great deal before it reaches the location where it ultimately triggers an exploitable condition. Consequently, when reviewing a software system, one of the most useful attributes to consider is the flow of data throughout the system's various components.

For example, you have an application that handles scheduling meetings for a large organization. At the end of every month, the application generates a report of all meetings coordinated in this cycle, including a brief summary of each meeting. Close inspection of the code reveals that when the application creates this summary, a meeting description larger than 1,000 characters results in an exploitable buffer overflow condition.

To exploit this vulnerability, you would have to create a new meeting with a description longer than 1,000 characters, and then have the application schedule the meeting. Then you would need to wait until the monthly report was created to see whether the exploit worked. Your malicious data would have to pass through several components of the system and survive being stored in a database, all the while avoiding being spotted by another user of the system. Correspondingly, you

have to evaluate the feasibility of this attack vector as a security reviewer. This viewpoint involves analyzing the flow of the meeting description from its initial creation, through multiple application components, and finally to its use in the vulnerable report generation code.

This process of tracing data flow is central to reviews of both the design and implementation of software. User-malleable data presents a serious threat to the system, and tracing the end-to-end flow of data is the main way to evaluate this threat. Typically, you must identify where user-malleable data enters the system through an interface to the outside world, such as a command line or Web request. Then you study the different ways in which user-malleable data can travel through the system, all the while looking for any potentially exploitable code that acts on the data. It's likely the data will pass through multiple components of a software system and be validated and manipulated at several points throughout its life span.

This process isn't always straightforward. Often you find a piece of code that's almost vulnerable but ends up being safe because the malicious input is caught or filtered earlier in the data flow. More often than you would expect, the exploit is prevented only through happenstance; for example, a developer introduces some code for a reason completely unrelated to security, but it has the side effect of protecting a vulnerable component later down the data flow. Also, tracing data flow in a real-world application can be exceedingly difficult. Complex systems often develop organically, resulting in highly fragmented data flows. The actual data might traverse dozens of components and delve in and out of third-party framework code during the process of handling a single user request.

Trust Relationships

Different components in a software system place varying degrees of trust in each other, and it's important to understand these trust relationships when analyzing the security of a given software system. **Trust relationships** are integral to the flow of data, as the level of trust between components often determines the amount of validation that happens to the data exchanged between them.

Designers and developers often consider an interface between two components to be trusted or designate a peer or supporting software component as trusted. This means they generally believe that the trusted component is impervious to malicious interference, and they feel safe in making assumptions about that component's data and behavior. Naturally, if this trust is misplaced, and an attacker can access or manipulate trusted entities, system security can fall like dominos.

Speaking of dominos, when evaluating trust relationships in a system, it's important to appreciate the **transitive** nature of trust. For example, if your software system trusts a particular external component, and that component in turn trusts a certain network, your system has indirectly placed trust in that network. If the

component's trust in the network is poorly placed, it might fall victim to an attack that ends up putting your software at risk.

Assumptions and Misplaced Trust

Another useful way of looking at software flaws is to think of them in terms of programmers and designers making unfounded assumptions when they create software. Developers can make incorrect assumptions about many aspects of a piece of software, including the validity and format of incoming data, the security of supporting programs, the potential hostility of its environment, the capabilities of its attackers and users, and even the behaviors and nuances of particular application programming interface (API) calls or language features.

The concept of inappropriate assumptions is closely related to the concept of misplaced trust because you can say that placing undue trust in a component is much the same as making an unfounded assumption about that component. The following sections discuss several ways in which developers can make security-relevant mistakes by making unfounded assumptions and extending undeserved trust.

Input

As stated earlier, the majority of software vulnerabilities are triggered by attackers injecting malicious data into software systems. One reason this data can cause such trouble is that software often places too much trust in its communication peers and makes assumptions about the data's potential origins and contents.

Specifically, when developers write code to process data, they often make assumptions about the user or software component providing that data. When handling user input, developers often assume users aren't likely to do things such as enter a 5,000-character street address containing nonprintable symbols. Similarly, if developers are writing code for a programmatic interface between two software components, they usually make assumptions about the input being well formed. For example, they might not anticipate a program placing a negative length binary record in a file or sending a network request that's four billion bytes long.

In contrast, attackers looking at input-handling code try to consider every possible input that can be entered, including any input that might lead to an inconsistent or unexpected program state. Attackers try to explore every accessible interface to a piece of software and look specifically for any assumptions the developer made. For an attacker, any opportunity to provide unexpected input is gold because this input often has a subtle impact on later processing that the developers didn't anticipate. In general, if you can make an unanticipated change in software's runtime properties, you can often find a way to leverage it to have more influence on the program.

Interfaces

Interfaces are the mechanisms by which software components communicate with each other and the outside world. Many vulnerabilities are caused by developers not fully appreciating the security properties of these interfaces and consequently assuming that only trusted peers can use them. If a program component is accessible via the network or through various mechanisms on the local machine, attackers might be able to connect to that component directly and enter malicious input. If that component is written so that it assumes its peer is trustworthy, the application is likely to mishandle the input in an exploitable manner.

What makes this vulnerability even more serious is that developers often incorrectly estimate the difficulty an attacker has in reaching an interface, so they place trust in the interface that isn't warranted. For example, developers might expect a high degree of safety because they used a proprietary and complex network protocol with custom encryption. They might incorrectly assume that attackers won't be likely to construct their own clients and encryption layers and then manipulate the protocol in unexpected ways. Unfortunately, this assumption is particularly unsound, as many attackers find a singular joy in reverse engineering a proprietary protocol.

To summarize, developers might misplace trust in an interface for the following reasons:

- They choose a method of exposing the interface that doesn't provide enough protection from external attackers.
- They choose a reliable method of exposing the interface, typically a service of the OS, but they use or configure it incorrectly. The attacker might also exploit a vulnerability in the base platform to gain unexpected control over that interface.
- They assume that an interface is too difficult for an attacker to access, which is usually a dangerous bet.

Environmental Attacks

Software systems don't run in a vacuum. They run as one or more programs supported by a larger computing environment, which typically includes components such as operating systems, hardware architectures, networks, file systems, databases, and users.

Although many software vulnerabilities result from processing malicious data, some software flaws occur when an attacker manipulates the software's underlying environment. These flaws can be thought of as vulnerabilities caused by assumptions made about the underlying environment in which the software is running. Each type of supporting technology a software system might rely on has many best practices and nuances, and if an application developer doesn't fully

understand the potential security issues of each technology, making a mistake that creates a security exposure can be all too easy.

The classic example of this problem is a type of race condition you see often in UNIX software, called a `/tmp` race (pronounced "temp race"). It occurs when a program needs to make use of a temporary file, and it creates this file in a public directory on the system, located in `/tmp` or `/var/tmp`. If the program hasn't been written carefully, an attacker can anticipate the program's moves and set up a trap for it in the public directory. If the attacker creates a symbolic link in the right place and at the right time, the program can be tricked into creating its temporary file somewhere else on the system with a different name. This usually leads to an exploitable condition if the vulnerable program is running with root (administrator) privileges.

In this situation, the vulnerability wasn't triggered through data the attacker supplied to the program. Instead, it was an attack against the program's runtime environment, which caused the program's interaction with the OS to proceed in an unexpected and undesired fashion.

Exceptional Conditions

Vulnerabilities related to handling exceptional conditions are intertwined with data and environmental vulnerabilities. Basically, an **exceptional condition** occurs when an attacker can cause an unexpected change in a program's normal control flow via external measures. This behavior can entail an asynchronous interruption of the program, such as the delivery of a signal. It might also involve consuming global system resources to deliberately induce a failure condition at a particular location in the program.

For example, a UNIX system sends a SIGPIPE signal if a process attempts to write to a closed network connection or pipe; the default behavior on receipt of this signal is to terminate the process. An attacker might cause a vulnerable program to write to a pipe at an opportune moment, and then close the pipe before the application can perform the write operation successfully. This would result in a SIGPIPE signal that could cause the application to abort and perhaps leave the overall system in an unstable state. For a more concrete example, the Network File System (NFS) status daemon of some Linux distributions was vulnerable to crashing caused by closing a connection at the correct time. Exploiting this vulnerability created a disruption in NFS functionality that persisted until an administrator can intervene and reset the daemon.

Summary

You've covered a lot of ground in this short chapter and might be left with a number of questions. Don't worry; subsequent chapters delve into more detail and provide answers as you progress. For now, it's important that you have a good understanding of what can go wrong in computer software and understand the terminology used in discussing these issues. You should also have developed an appreciation of the need for security auditing of applications and become familiar with different aspects of the process. In later chapters, you build on this foundation as you learn how to use this audit process to identify vulnerabilities in the applications you review.

This page intentionally left blank

Chapter 2

Design Review

"Sure. Each one of us is wearing an unlicensed nuclear accelerator on our back. No problem."

Bill Murray as Dr. Peter Venkman, *Ghostbusters* (1984)

Introduction

Computer security people tend to fall into one of two camps on design review. People from a formal development background are usually receptive to the design review process. This is only natural, as it maps closely to most formal software development methodologies. The design review process can also seem to be less trouble than reviewing a large application code base manually.

In the other camp are code auditors who delight in finding the most obscure and complex vulnerabilities. This crowd tends to look at design review as an ivory-tower construct that just gets in the way of the real work. Design review's formalized process and focus on documentation come across as a barrier to digging into the code.

The truth is that design review falls somewhere between the views of these two camps, and it has value for both. Design review is a useful tool for identifying vulnerabilities in application architecture and prioritizing components for implementation review. It doesn't

replace implementation review, however; it's just a component of the complete review process. It makes identifying design flaws a lot easier and provides a more thorough analysis of the security of a software design. In this capacity, it can make the entire review process more effective and ensure the best return for the time you invest.

This chapter gives you some background on the elements of software design and design vulnerabilities, and introduces a review process to help you identify security concerns in a software design.

Software Design Fundamentals

Before you tackle the subject of design review, you need to review some fundamentals of software design. Many of these concepts tie in closely with the security considerations addressed later in the chapter, particularly in the discussion of threat modeling. The following sections introduce several concepts that help establish an application's functional boundaries with respect to security.

Algorithms

Software engineering can be summed up as the process of developing and implementing algorithms. From a design perspective, this process focuses on developing key program algorithms and data structures as well as specifying problem domain logic. To understand the security requirements and vulnerability potential of a system design, you must first understand the core algorithms that comprise a system.

Problem Domain Logic

Problem domain logic (or **business logic**) provides rules that a program follows as it processes data. A design for a software system must include rules and processes for the main tasks the software carries out. One major component of software design is the security expectations associated with the system's users and resources. For example, consider banking software with the following rules:

- A person can transfer money from his or her main account to any valid account.
- A person can transfer money from his or her money market account to any valid account.
- A person can transfer money from his or her money market account only once a month.
- If a person goes below a zero balance in his or her main account, money is automatically transferred from his or her money market account to cover the balance, if that money is available.

This example is simple, but you can see that bank customers might be able to get around the once-a-month transfer restriction on money market accounts. They could intentionally drain their main account below zero to "free" money from their monkey market accounts. Therefore, the design for this system has an oversight that bank customers could potentially exploit.

Key Algorithms

Often programs have performance requirements that dictate the choice of algorithms and data structures used to manage key pieces of data. Sometimes it's possible to evaluate these algorithm choices from a design perspective and predict security vulnerabilities that might affect the system.

For example, you know that a program stores an incoming series of records in a sorted linked list that supports a basic sequential search. Based on this knowledge, you can foresee that a specially crafted huge list of records could cause the program to spend considerable time searching through the linked list. Repeated focused attacks on a key algorithm such as this one could easily lead to temporary or even permanent disruption of a server's functioning.

Abstraction and Decomposition

Every text on software design inevitably covers two essential concepts: abstraction and decomposition. You are probably familiar with these concepts already, but if not, the following paragraphs give you a brief overview.

Abstraction is a method for reducing the complexity of a system to make it more manageable. To do this, you isolate only the most important elements and remove unnecessary details. Abstractions are an essential part of how people perceive the world around them. They explain why you can see a symbol such as ☺ and associate it with a smiling face. Abstractions allow you to generalize a concept, such as a face, and group-related concepts, such as smiling faces and frowning faces.

In software design, abstractions are how you model the processes an application will perform. They enable you to establish hierarchies of related systems, concepts, and processes—isolating the problem domain logic and key algorithms. In effect, the design process is just a method of building a set of abstractions that you can develop into an implementation. This process becomes particularly important when a piece of software must address the concerns of a range of users, or its implementation must be distributed across a team of developers.

Decomposition (or factoring) is the process of defining the generalizations and classifications that compose an abstraction. Decomposition can run in two different directions. Top-down decomposition, known as **specialization**, is the process of breaking a larger system into smaller, more manageable parts. Bottom-up decomposition,

called **generalization**, involves identifying the similarities in a number of components and developing a higher-level abstraction that applies to all of them.

The basic elements of structural software decomposition can vary from language to language. The standard **top-down** progression is application, module, class, and function (or method). Some languages might not support every distinction in this list (for example, C doesn't have language support for classes); other languages add more distinctions or use slightly different terminology. The differences aren't that important for your purposes, but to keep things simple, this discussion generally sticks to modules and functions.

Trust Relationships

In Chapter 1, "Software Vulnerability Fundamentals," the concept of trust and how it affects system security was introduced. This chapter expands on that concept to state that every communication between multiple parties must have some degree of trust associated with it. This is referred to as a **trust relationship**. For simple communications, both parties can assume complete trust—that is, each communicating party allows other parties participating in the communication complete access to its exposed functionality. For security purposes, however, you're more concerned with situations in which communicating parties should restrict their trust of one another. This means parties can access only a limited subset of each other's functionality. The limitations imposed on each party in a communication define a **trust boundary** between them. A trust boundary distinguishes between regions of shared trust, known as **trust domains**. (Don't worry if you're a bit confused by these concepts; some examples are provided in the next section.)

A software design needs to account for a system's trust domains, boundaries, and relationships; the **trust model** is the abstraction that represents these concepts and is a component of the application's security policy. The impact of this model is apparent in how the system is decomposed, as trust boundaries tend to be module boundaries, too. The model often requires that trust not be absolute; instead, it supports varying degrees of trust referred to as **privileges**. A classic example is the standard UNIX file permissions, whereby a user can provide a limited amount of access to a file for other users on the system. Specifically, users can dictate whether other users are allowed to read, write, or execute (or any combination of these permissions) the file in question, thus extending a limited amount of trust to other users of the system.

Simple Trust Boundaries

As an example of a trust relationship, consider a basic single-user OS, such as Windows 98. To keep the example simple, assume that there's no network involved. Windows 98 has basic memory protection and some notion of users but offers no

measure of access control or enforcement. In other words, if users can log in to a Windows 98 system, they are free to modify any files or system settings they please. Therefore, you have no expectation of security from any user who can log on interactively.

You can determine that there are no trust boundaries between interactive users of the same Windows 98 system. You do, however, make an implicit assumption about who has physical access to the system. So you can say that the trust boundary in this situation defines which users have physical access to the system and which do not. That leaves you with a single domain of trusted users and an implicit domain that represents all untrusted users.

To complicate this example a bit, say you've upgraded to a multiuser OS, such as Windows XP Professional. This upgrade brings with it a new range of considerations. You expect that two normally privileged users shouldn't be able to manipulate each other's data or processes. Of course, this expectation assumes you aren't running as an administrative user. So now you have an expectation of confidentiality and integrity between two users of the system, which establishes their trust relationship and another trust boundary. You also have to make allowances for the administrative user, which adds another boundary: Nonadministrative users can't affect the integrity or configuration of the system. This expectation is a natural progression that's necessary to enforce the boundary between users. After all, if any user could affect the state of the system, you would be right back to a single-user OS. Figure 2-1 is a graphical representation of this multiuser OS trust relationship.

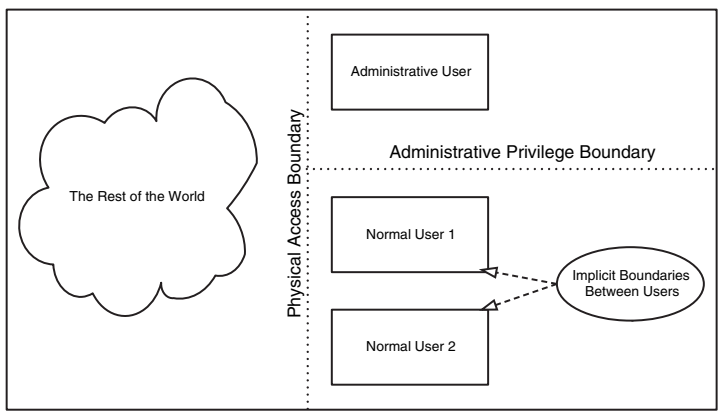

Figure 2-1 Simple trust boundaries

Now take a step back and consider something about the nature of trust. That is, every system must eventually have some absolutely **trusted authority**. There's no way

around this because someone must be responsible for the state of the system. That's why UNIX has a root account, and Windows has an administrator account. You can, of course, apply a range of controls to this level of authority. For instance, both UNIX and Windows have methods of granting degrees of administrative privilege to different users and for specific purposes. The simple fact remains, however, that in every trust boundary, you have at least one absolute authority that can assume responsibility.

Complex Trust Relationships

So far, you've looked at fairly simple trust relationships to get a sense of the problem areas you need to address later. However, some of the finer details have been glossed over. To make the discussion a bit more realistic, consider the same system connected to a network.

After you hook a system up to a network, you have to start adding a range of distinctions. You might need to consider separate domains for local users and remote users of the system, and you'll probably need a domain for people who have network access to the system but aren't "regular" users. Firewalls and gateways further complicate these distinctions and allow more separations.

It should be apparent that defining and applying a trust model can have a huge impact on any software design. The real work begins before the design process is even started. The feasibility study and requirements-gathering phases must adequately identify and define users' security expectations and the associated factors of the target environment. The resulting model must be robust enough to meet these needs, but not so complex that it's too difficult to implement and apply. In this way, security has to carefully balance the concerns of clarity with the need for accuracy. When you examine threat modeling later in this chapter, you take trust models into account by evaluating the boundaries between different system components and the rights of different entities on a system.

Chain of Trust

Chapter 1 also introduced the concept of transitive trust. Essentially, it means that if component A trusts component B, component A must implicitly trust all components trusted by component B. This concept can also be called a **chain of trust** relationship.

A chain of trust is a completely viable security construct and the core of many systems. Consider the way certificates are distributed and validated in a typical Secure Sockets Layer (SSL) connection to a Web server. You have a local database of signatures that identifies providers you trust. These providers can then issue a certificate to a certificate authority (CA), which might then be extended to other authorities. Finally, the hosting site has its certificate signed by one of these authorities. You must follow this chain of trust from CA to CA when you establish an SSL connection. The traversal is successful only when you reach an authority that's in your trusted database.

Now say you want to impersonate a Web site for some nefarious means. For the moment, leave Domain Name System (DNS) out of the picture because it's often an easy target. Instead, all you want to do is find a way to manipulate the certificate database anywhere in the chain of trust. This includes manipulating the client certificate database of visitors, compromising the target site directly, or manipulating any CA database in the chain, including a root CA.

It helps to repeat that last part, just to make sure the emphasis is clear. The transitive nature of the trust shared by every CA means that a compromise of any CA allows an attacker to impersonate any site successfully. It doesn't matter if the CA that issued the real certificate is compromised because any certificate issued by a valid CA will suffice. This means the integrity of any SSL transaction is only as strong as the weakest CA. Unfortunately, this method is the best that's available for establishing a host's identity.

Some systems can be implemented only by using a transitive chain of trust. As an auditor, however, you want to look closely at the impact of choosing this trust model and determine whether a chain of trust is appropriate. You also need to follow trusts across all the included components and determine the real exposure of any component. You'll often find that the results of using a chain of trust are complex and subtle trust relationships that attackers could exploit.

Defense in Depth

Defense in depth is the concept of layering protections so that the compromise of one aspect of a system is mitigated by other controls. Simple examples of defense in depth include using low privileged accounts to run services and daemons, and isolating different functions to different pieces of hardware. More complex examples include network demilitarized zones (DMZs), chroot jails, and stack and heap guards.

Layered defenses should be taken into consideration when you're prioritizing components for review. You would probably assign a lower priority to an intranet-facing component running on a low privileged account, inside a chroot jail, and compiled with buffer protection. In contrast, you would most likely assign a higher priority to an Internet-facing component that must run as root. This is not to say that the first component is safe and the second isn't. You just need to look at the evidence and prioritize your efforts so that they have the most impact. Prioritizing threats is discussed in more detail in "Threat Modeling" later on in this chapter.

Principles of Software Design

The number of software development methodologies seems to grow directly in proportion to the number of software developers. Different methodologies suit different needs, and the choice for a project varies based on a range of factors.

Fortunately, every methodology shares certain commonly accepted principles. The four core principles of accuracy, clarity, loose coupling, and strong cohesion (discussed in the following sections) apply to every software design and are a good starting point for any discussion of how design can affect security.

Accuracy

Accuracy refers to how effectively design abstractions meet the associated requirements. (Remember the discussion on requirements in Chapter 1.) Accuracy includes both how correctly abstractions model the requirements and how reasonably they can be translated into an implementation. The goal is, of course, to provide the most accurate model with the most direct implementation possible.

In practice, a software design might not result in an accurate translation into an implementation. Oversights in the requirements-gathering phase could result in a design that misses important capabilities or emphasizes the wrong concerns. Failures in the design process might result in an implementation that must diverge drastically from the design to meet real-world requirements. Even without failures in the process, expectations and requirements often change during the implementation phase. All these problems tend to result in an implementation that can diverge from the intended (and documented) design.

Discrepancies between a software design and its implementation result in weaknesses in the design abstraction. These weaknesses are fertile ground for a range of bugs to creep in, including security vulnerabilities. They force developers to make assumptions outside the intended design, and a failure to communicate these assumptions often creates vulnerability-prone situations. Watch for areas where the design isn't adequately defined or places unreasonable expectations on programmers.

Clarity

Software designs can model extremely complex and often confusing processes. To achieve the goal of **clarity**, a good design should decompose the problem in a reasonable manner and provide clean, self-evident abstractions. Documentation of the structure should also be readily available and well understood by all developers involved in the implementation process.

An unnecessarily complex or poorly documented design can result in vulnerabilities similar to those of an inaccurate design. In this case, weaknesses in the abstraction occur because the design is simply too poorly understood for an accurate implementation. Your review should identify design components that are inadequately documented or exceptionally complex. You see examples of this problem throughout the book, especially when variable relationships are tackled in Chapter 7, "Program Building Blocks."

Loose Coupling

Coupling refers to the level of communication between modules and the degree to which they expose their internal interfaces to each other. **Loosely coupled** modules exchange data through well-defined public interfaces, which generally leads to more adaptable and maintainable designs. In contrast, **strongly coupled** modules have complex interdependencies and expose important elements of their internal interfaces.

Strongly coupled modules generally place a high degree of trust in each other and rarely perform data validation for their communication. The absence of well-defined interfaces in these communications also makes data validation difficult and error prone. This tends to lead to security flaws when one of the components is malleable to an attacker's control. From a security perspective, you want to look out for any strong intermodule coupling across trust boundaries.

Strong Cohesion

Cohesion refers to a module's internal consistency. This consistency is primarily the degree to which a module's interfaces handle a related set of activities. Strong cohesion encourages the module to handle only closely related activities. A side effect of maintaining strong cohesion is that it tends to encourage strong intramodule coupling (the degree of coupling between different components of a single module).

Cohesion-related security vulnerabilities can occur when a design fails to decompose modules along trust boundaries. The resulting vulnerabilities are similar to strong coupling issues, except that they occur within the same module. This is often a result of systems that fail to incorporate security in the early stages of their design. Pay special attention to designs that address multiple trust domains within a single module.

Fundamental Design Flaws

Now that you have a foundational understanding, you can consider a few examples of how fundamental design concepts affect security. In particular, you need to see how misapplying these concepts can create security vulnerabilities. When reading the following examples, you'll notice quickly that they tend to result from a combination of issues. Often, an error is open to interpretation and might depend heavily on the reviewer's perspective. Unfortunately, this is part of the nature of design flaws. They usually affect the system at a conceptual level and can be difficult to categorize. Instead, you need to concentrate on the issue's security impact, not get caught up in the categorization.

Exploiting Strong Coupling

This section explores a fundamental design flaw resulting from a failure to decompose an application properly along trust boundaries. The general issue is known as the Shatter class of vulnerabilities, originally reported as part of independent research conducted by Chris Paget. The specific avenue of attack takes advantage of certain properties of the Windows GUI application programming interface (API). The following discussion avoids many details in order to highlight the design specific nature of Shatter vulnerabilities. Chapter 12, "Windows II: Interprocess Communication," provides a much more thorough discussion of the technical details associated with this class of vulnerabilities.

Windows programs use a messaging system to handle all GUI-related events; each desktop has a single message queue for all applications associated with it. So any two processes running on the same desktop can send messages to each other, regardless of the user context of the processes. This can cause an issue when a higher privileged process, such as a service, is running on a normal user's desktop.

The Windows API provides the `SetTimer()` function to schedule sending a `WM_TIMER` message. This message can include a function pointer that is invoked when the default message handler receives the `WM_TIMER` message. This creates a situation in which a process can control a function call in any other process that shares its desktop. An attacker's only remaining concern is how to supply code for execution in the target process.

The Windows API includes a number of messages for manipulating the content of window elements. Normally, they are used for setting the content of text boxes and labels, manipulating the Clipboard's content, and so forth. However, an attacker can use these messages to insert data into the address space of a target process. By combining this type of message with the `WM_TIMER` message, an attacker can build and run arbitrary code in any process on the same desktop. The result is a privilege escalation vulnerability that can be used against services running on the interactive desktop.

After this vulnerability was published, Microsoft changed the way the `WM_TIMER` message is handled. The core issue, however, is that communication across a desktop must be considered a potential attack vector. This makes more sense when you consider that the original messaging design was heavily influenced by the concerns of single-user OS. In that context, the design was accurate, understandable, and strongly cohesive.

This vulnerability demonstrates why it's difficult to add security to an existing design. The initial Windows messaging design was sound for its environment, but introducing a multiuser OS changed the landscape. The messaging queue now strongly couples different trust domains on the same desktop. The result is new types of vulnerabilities in which the desktop can be exploited as a public interface.

Exploiting Transitive Trusts

A fascinating Solaris security issue highlights how attackers can manipulate a trusted relationship between two components. Certain versions of Solaris included an RPC program, `automountd`, that ran as root. This program allowed the root user to specify a command to run as part of a mounting operation and was typically used to handle mounting and unmounting on behalf of the kernel. The `automountd` program wasn't listening on an IP network and was available only through three protected loopback transports. This meant the program would accept commands only from the root user, which seems like a fairly secure choice of interface.

Another program, `rpc.statd`, runs as root and listens on Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) interfaces. It's used as part of the Network File System (NFS) protocol support, and its purpose is to monitor NFS servers and send out a notification in case they go down. Normally, the NFS lock daemon asks `rpc.statd` to monitor servers. However, registering with `rpc.statd` requires the client to tell it which host to contact and what RPC program number to call on that host.

So an attacker can talk to a machine's `rpc.statd` and register the `automountd` program for receipt of crash notifications. Then the attacker tells `rpc.statd` that the monitored NFS server has crashed. In response, `rpc.statd` contacts the `automountd` daemon on the local machine (through the special loopback interface) and gives it an RPC message. This message doesn't match up to what `automountd` is expecting, but with some manipulation, you can get it to decode into a valid `automountd` request. The request comes from root via the loopback transport, so `automountd` thinks it's from the kernel module. The result is that it carries out a command of the attacker's choice.

In this case, the attack against a public interface to `rpc.statd` was useful only in establishing trusted communication with `automountd`. It occurred because an implicit trust is shared between all processes running under the same account. Exploiting this trust allowed remote attackers to issue commands to the `automountd` process. Finally, assumptions about the source of communication caused developers to be lenient in the format `automountd` accepts. These issues, combined with the shared trust between these modules, resulted in a remote root-level vulnerability.

Failure Handling

Proper failure handling is an essential component of clear and accurate usability in a software design. You simply expect an application to handle irregular conditions properly and provide users with assistance in solving problems. However, failure conditions can create situations in which usability and security appear to be in opposition. Occasionally, compromises must be made in an application's functionality so that security can be enforced.

Consider a networked program that detects a fault or failure condition in data it receives from a client system. Accurate and clear usability dictates that the application attempt to recover and continue processing. When recovery isn't possible, the application should assist users in diagnosing the problem by supplying detailed information about the error.

However, a security-oriented program generally takes an entirely different approach, which might involve terminating the client session and providing the minimum amount of feedback necessary. This approach is taken because a program designed around an ideal of security assumes that failure conditions are the result of attackers manipulating the program's input or environment. From that perspective, the attempt to work around the problem and continue processing often plays right into an attacker's hands. The pragmatic defensive reaction is to drop what's going on, scream bloody murder in the logs, and abort processing. Although this reaction might seem to violate some design principles, it's simply a situation in which the accuracy of security requirements supersedes the accuracy and clarity of usability requirements.

Enforcing Security Policy

Chapter 1 discussed security expectations and how they affect a system. Now you can take those concepts and develop a more detailed understanding of how security expectations are enforced in a security policy. Developers implement a security policy primarily by identifying and enforcing trust boundaries. As an auditor, you need to analyze the design of these boundaries and the code implementing their enforcement. In order to more easily address the elements of the security policy, enforcement is broken up into six main types discussed in the following sections.

Authentication

Authentication is the process by which a program determines who a user claims to be and then checks the validity of that claim. A software component uses authentication to establish the identity of a peer (client or server) when initiating communication. A classic example is requiring the user of a Web site to enter a username and password. Authentication isn't just for human peers, either, as you can see in the previous discussion of SSL certificates. In that example, the systems authenticated with each other to function safely over an untrustworthy interface.

Common Vulnerabilities of Authentication

One notable design oversight is to not require authentication in a situation that warrants it. For example, a Web application presents a summary of sensitive corporate accounting information that could be useful for insider trading. Exposing that

information to arbitrary Internet users without asking for some sort of authentication would be a design flaw. Note that "lack of authentication" issues aren't always obvious, especially when you're dealing with peer modules in a large application. Often it's difficult to determine that an attacker can get access to a presumably internal interface between two components.

Typically, the best practice is to centralize authentication in the design, especially in Web applications. Some Web applications require authentication for users who come in through a main page but don't enforce authentication in follow-on pages. This lack of authentication means you could interact with the application without ever having to enter a username or password. In contrast, centralized authentication mitigates this issue by validating every Web request within the protected domain.

Untrustworthy Credentials

Another common mistake happens when some authentication information is presented to the software, but the information isn't trustworthy. This problem often happens when authentication is performed on the client side, and an attacker can completely control the client side of the connection. For example, the SunRPC framework includes the `AUTH_UNIX` authentication scheme, which basically amounts to fully trusting the client system. The client simply passes along a record that tells the server what the user and group IDs are, and the server just accepts them as fact.

UNIX systems used to include a RPC daemon called `rexd` (remote execute daemon). The purpose of this program was to let a remote user run a program on the system as a local user. If you were to connect to the `rexd` system and tell the `rexd` program to run the `/bin/sh` command as the user bin, the program would run a shell as bin and let you interact with it. That's about all there was to it, with the exception that you couldn't run programs as the root user. Typically, getting around this restriction takes only a few minutes after you have a shell running as bin. More recently, a remote root flaw was exposed in the default installation of `sadmind` on Solaris; it treated the `AUTH_UNIX` authentication as sufficient validation for running commands on behalf of the client.

> **Note**
> The bug in `sadmind` is documented at www.securityfocus.com/bid/2354/info.

Many network daemons use the source IP address of a network connection or packet to establish a peer's identity. By itself, this information isn't a sufficient credential and is susceptible to tampering. UDP can be trivially spoofed, and TCP connections can be spoofed or intercepted in various situations. UNIX provides

multiple daemons that honor the concept of trusted hosts based on source address. These daemons are `rshd` and `rlogind`, and even `sshd` can be configured to honor these trust relationships. By initiating, spoofing, or hijacking a TCP connection from a trusted machine on a privileged port, an attacker can exploit the trust relationship between two machines.

Insufficient Validation

An authentication system can be close to sufficient for its environment but still contain a fundamental design flaw that leaves it exposed. This problem isn't likely to happen with the typical authentication design of requiring username/password/mom's maiden name, as it's easy to think through the consequences of design decisions in this type of system.

You're more likely to see this kind of design flaw in programmatic authentication between two systems. If a program makes use of existing authentication mechanisms, such as certificates, design-level problems can arise. First, many distributed client/server applications authenticate in only one direction: by authenticating only the client or only the server. An attacker can often leverage this authentication scheme to masquerade as the unauthenticated peer and perform subtle attacks on the system.

Homemade authentication with cryptographic primitives is another issue you might encounter. From a conceptual standpoint, making your own authentication seems simple. If you have a shared secret, you give the peer a challenge. The peer then sends back a value that could be derived only from a combination of the challenge and shared secret. If you're using public and private keys, you send a challenge to a peer, encrypting it with the peer's public key, and anticipate a response that proves the peer was able to decrypt it.

However, there's plenty of room for error when creating authentication protocols from scratch. Thomas Lopatic found an amusing vulnerability in the FWN/1 protocol of Firewall-1. Each peer sends a random number R1 and a hash of that random number with a shared key, Hash(R1+K). The receiving peer can look at the random number that was sent, calculate the hash, and compare it with the transmitted value. The problem is that you can simply replay the R1 and Hash(R1+K) values to the server because they're made using the same shared symmetric key.

Authorization

Authorization is the process of determining whether a user on the system is permitted to perform a specific operation within a trust domain. It works in concert with authentication as part of an **access control** policy: Authentication establishes who a user is, and authorization determines what that user is permitted to do. There are many formal designs for access control systems, including discretionary access control, mandatory access control, and role-based access control. In addition, several technologies are available for centralizing access control into various frameworks,

operating systems, and libraries. Because of the complexity of different access control schemes, it's best to begin by looking at authorization from a general perspective.

Common Vulnerabilities of Authorization

Web applications are notorious for missing or insufficient authorization. Often, you find that only a small fraction of a Web site's functionality does proper authorization checks. In these sites, pages with authorization logic are typically main menu pages and major subpages, but the actual handler pages omit authorization checks. Frequently, it's possible to find a way to log in as a relatively low-privileged user, and then be able to access information and perform actions that don't belong to your account or are intended for higher-privileged users.

Authorities That Aren't Secure

Omitting authorization checks is obviously a problem. You can also run into situations in which the logic for authorization checks is inconsistent or leaves room for abuse. For example, say you have a simple expense-tracking system, and each user in the company has an account. The system is preprogrammed with the corporate tree so that it knows which employees are managers and who they manage. The main logic is data driven and looks something like this:

```
Enter New Expense
for each employee you manage
    View/Approve Expenses
```

This system is fairly simple. Assuming that the initial corporate tree is populated correctly, managers can review and approve expenses of their subordinates. Normal employees see only the `Enter New Expense` menu entry because they aren't in the system as managing other employees.

Now say that you constantly run into situations in which employees are officially managed by one person, but actually report to another manager for day-to-day issues. To address this problem, you make it possible for each user to designate another user as his or her "virtual" manager. A user's virtual manager is given view and approve rights to that user's expenses, just like the user's official manager. This solution might seem fine at first glance, but it's flawed. It creates a situation in which employees can assign any fellow employee as their virtual manager, including themselves. The resulting virtual manager could then approve expenses without any further restrictions.

This simple system with an obvious problem might seem contrived, but it's derived from problems encountered in real-world applications. As the number of users and groups in an application grows and the complexity of the system grows, it becomes easy for designers to overlook the possibility of potential abuse in the authorization logic.

Accountability

Accountability refers to the expectation that a system can identify and log activities that users of the system perform. **Nonrepudiation** is a related term that's actually a subset of accountability. It refers to the guarantee that a system logs certain user actions so that users can't later deny having performed them. Accountability, along with authorization and authentication, establishes a complete **access control policy**. Unlike authentication and authorization, accountability doesn't specifically enforce a trust boundary or prevent a compromise from occurring. Instead, accountability provides data that can be essential in mitigating a successful compromise and performing forensic analysis. Unfortunately, accountability is one of the most overlooked portions of secure application design.

Common Vulnerabilities of Accountability

The most common accountability vulnerability is a system's failure to log operations on sensitive data. In fact, many applications provide no logging capability whatsoever. Of course, many applications don't handle sensitive data that requires logging. However, administrators or end users—not developers—should determine whether logging is required.

The next major concern for accountability is a system that doesn't adequately protect its log data. Of course, this concern might also be an authorization, confidentiality, or integrity issue. Regardless, any system maintaining a log needs to ensure the security of that log. For example, the following represents a simple text-based log, with each line including a timestamp followed by a log entry:

```
20051018133106 Logon Failure: Bob
20051018133720 Logon Success: Jim
20051018135041 Logout: Jim
```

What would happen if you included user-malleable strings in the log entry? What's to prevent a user from intentionally sending input that looks like a log entry? For instance, say a user supplied "`Bob\n20051018133106 Logon Success: Greg`" as a logon name. It looks like a harmless prank, but it could be used for malicious activity. Attackers could use fake entries to cover malicious activity or incriminate an innocent user. They might also be able to corrupt the log to the point that it becomes unreadable or unwriteable. This corruption could create a denial-of-service condition or open pathways to other vulnerabilities. It might even provide exploitable pathways in the logging system itself.

Manipulating this log isn't the only problem. What happens when attackers can read it? At the very least, they would know at what times every user logged in and logged out. From this data, they could deduce login patterns or spot which users have a habit of forgetting their passwords. This information might seem harmless, but it

can be useful in staging a larger attack. Therefore, unauthorized users shouldn't be able to read or modify the contents of a system log.

Confidentiality

Chapter 1 described confidentiality as the expectation that only authorized parties can view data. This requirement is typically addressed through access control mechanisms, which are covered by authentication and authorization. However, additional measures must be taken when communication is performed over a channel that's not secure. In these cases, encryption is often used to enforce confidentiality requirements.

Encryption is the process of encoding information so that it can't be read by a third party without special knowledge, which includes the encryption process and usually some form of key data. Key data is a piece of data known only to the parties who are authorized to access the information.

The topic of validating cryptographic algorithms and processes is not covered in this book because the mathematics involved are extremely complex and encompass an entire field of study. However, the knowledge you need to identify certain vulnerabilities in implementing and applying cryptography is covered throughout this book, including memory management issues in cryptographic message handling and how to validate specification requirements against an implementation.

Your biggest concern from a design perspective is in determining if a particular cryptographic protocol is applied correctly. The protocol must be strong enough for the data it's protecting and must be used in a secure manner. If you're interested in more information on the appropriate use of cryptography, you can read *Practical Cryptography* (Wiley, 2003) by Bruce Schneier and Niels Ferguson. If your interest lies in algorithms and implementation, consider Bruce Schneier's other book, *Applied Cryptography* (Wiley, 1996).

Encryption Algorithms

Encryption has a long history, dating all the way back to ancient cultures. However, because you're concerned with modern cryptographic protocols that can be used to protect data communications effectively, this chapter focuses on two major classes of encryption: symmetric and asymmetric.

Symmetric encryption (or **shared key encryption**) refers to algorithms in which all authorized parties share the same key. Symmetric algorithms are generally the simplest and most efficient encryption algorithms. Their major weakness is that they require multiple parties to have access to the same shared secret. The alternative is to generate and exchange a unique key for each communication relationship, but this solution quickly results in an untenable key management situation. Further, asymmetric encryption has no means for verifying the sender of a message among any group of shared key users.

Asymmetric encryption (or **public key encryption**) refers to algorithms in which each party has a different set of keys for accessing the same encrypted data. This is done by using a public and private key pair for each party. Any parties wanting to communicate must exchange their public keys in advance. The message is then encrypted by combining the recipient's public key and the sender's private key. The resulting encrypted message can be decrypted only by using the recipient's private key.

In this manner, asymmetric encryption simplifies key management, doesn't require exposing private keys, and implicitly verifies the sender of a message. However, these algorithms are more complex and tend to be computationally intensive. Therefore, asymmetric algorithms are typically used to exchange a symmetric key that's then used for the duration of a communication session.

Block Ciphers

Block ciphers are symmetric encryption algorithms that work on fixed-size blocks of data and operate in a number of modes. You should be aware of some considerations for their use, however. One consideration is whether the block cipher encrypts each block independently or uses output from the previous block in encrypting the current block. Ciphers that encrypt blocks independently are far more vulnerable to cryptanalytic attacks and should be avoided whenever possible. Therefore, a **cipher block chaining (CBC)** mode cipher is the only appropriate fixed-block cipher in general use. It performs an XOR operation with the previous block of data, resulting in negligible performance overhead and much higher security than modes that handle blocks independently.

Stream Ciphers

One of the most inconvenient aspects of block ciphers is that they must handle fixed-size chunks of data. Any data chunks larger than the block size must be fragmented, and anything smaller must be padded. This requirement can add complexity and overhead to code that handles something like a standard TCP socket.

Fortunately, block ciphers can run in modes that allow them to operate on arbitrarily sized chunks of data. In this mode, the block cipher performs as a **stream cipher**. The **counter (CTR)** mode cipher is the best choice for a stream cipher. Its performance characteristics are comparable to CBC mode, but it doesn't require padding or fragmentation.

Initialization Vectors

An **initialization vector (IV)** is a "dummy" block of data used to start a block cipher. An IV is necessary to force the cipher to produce a unique stream of output, regardless of identical input. The IV doesn't need to be kept private, although it must be different for every new cipher initialization with the same key. Reusing an IV causes

information leakage with a CBC cipher in only a limited number of scenarios; however, it severely degrades the security of other block ciphers. As a general rule, IV reuse should be considered a security vulnerability.

Key Exchange Algorithms

Key exchange protocols can get complicated, so this section just provides some simple points to keep in mind. First, the implementation should use a standard key exchange protocol, such as RSA, Diffie-Hellman, or El Gamal. These algorithms have been extensively validated and provide the best degree of assurance.

The next concern is that the key exchange is performed in a secure manner, which means both sides of the communication must provide some means of identification to prevent **man-in-the-middle attacks**. All the key exchange algorithms mentioned previously provide associated signature algorithms that can be used to validate both sides of the connection. These algorithms require that both parties have already exchanged public keys or that they are available through some trusted source, such as a Public Key Infrastructure (PKI) server.

Common Vulnerabilities of Encryption

Now that you have some background on the proper use of encryption, it's important to understand what can go wrong. Homemade encryption is one of the primary causes of confidentiality-related vulnerabilities. Encryption is extremely complicated and requires extensive knowledge and testing to design and implement properly. Therefore, most developers should restrict themselves to known algorithms, protocols, and implementations that have undergone extensive review and testing.

Storing Sensitive Data Unnecessarily

Often a design maintains sensitive data without any real cause, typically because of a misunderstanding of the system requirements. For instance, validating a password doesn't require storing the password in a retrievable form. You can safely store a hash of the password and use it for comparison. If it's done correctly, this method prevents the real password from being exposed. (Don't worry if you aren't familiar with hashes; they are introduced in "Hash Functions" later in this chapter.)

Clear-text passwords are one of the most typical cases of storing data unnecessarily, but they are far from the only example of this problem. Some application designs fail to classify sensitive information properly or just store it for no understandable reason. The real issue is that any design needs to classify the sensitivity of its data correctly and store sensitive data only when absolutely required.

Lack of Necessary Encryption

Generally, a system doesn't provide adequate confidentiality if it's designed to transfer clear-text information across publicly accessible storage, networks, or unprotected shared memory segments. For example, using TELNET to exchange

sensitive information would almost certainly be a confidentiality-related design vulnerability because TELNET does not encrypt its communication channel.

In general, any communication with the possibility of containing sensitive information should be encrypted when it travels over potentially compromised or public networks. When appropriate, sensitive information should be encrypted as it's stored in a database or on disk. Encryption requires a key management solution of some sort, which can often be tied to a user-supplied secret, such as a password. In some situations, especially when storing passwords, hashed values of sensitive data can be stored in place of the actual sensitive data.

Insufficient or Obsolete Encryption

It's certainly possible to use encryption that by design isn't strong enough to provide the required level of data security. For example, 56-bit single Digital Encryption Standard (DES) encryption is probably a bad choice in the current era of inexpensive multigigahertz computers. Keep in mind that attackers can record encrypted data, and if the data is valuable enough, they can wait it out while computing power advances. Eventually, they will be able to pick up a 128 q-bit quantum computer at Radio Shack, and your data will be theirs (assuming that scientists cure the aging problem by 2030, and everyone lives forever).

Jokes aside, it's important to remember that encryption implementations do age over time. Computers get faster, and mathematicians find devious new holes in algorithms just as code auditors do in software. Always take note of algorithms and key sizes that are inadequate for the data they protect. Of course, this concern is a moving target, so the best you can do is keep abreast of the current recommended standards. Organizations such as the National Institute for Standards and Technology (NIST; www.nist.gov) do a good job of publishing generally accepted criteria for algorithms and key sizes.

Data Obfuscation Versus Data Encryption

Some applications—and even industry-wide security standards—don't seem to differentiate between data obfuscation and data encryption. Put simply, data is obfuscated when attackers have access to all the information they need to recover encoded sensitive data. This situation typically occurs when the method of encoding data doesn't incorporate a unique key, or the key is stored in the same trust domain as the data. Two common examples of encoding methods that don't incorporate a unique key are ROT13 text encoding and simple XOR mechanisms.

The problem of keys stored in the same context as data is a bit more confusing but not necessarily less common. For example, many payment-processing applications store sensitive account holder information encrypted in their databases, but all the processing applications need the keys. This requirement means that stealing the backup media might not give attackers the account data, but compromising any payment server can get them the key along with the encrypted data. Of course, you

could add another key to protect the first key, but all the processing applications would still require access. You could layer as many keys as you like, but in the end, it's just an obfuscation technique because each processing application needs to decrypt the sensitive data.

> **Note**
> The PCI (Payment Card Industry) 1.0 Data Security Requirement is part of an industry-wide standard to help ensure safe handling of payment card data and transactions. These requirements are a forward-thinking move for the industry, and many of them are consistent with best security practices. However, the standard contains requirements that create exactly the confidentiality issue described in this chapter. In particular, the requirements allow storing encrypted data and the key in the same context, as long as the key is encrypted by another key residing in the same context.

One final point is that security by obscurity (or obfuscation) has earned a bad reputation in the past several years. On its own, it's an insufficient technique for protecting data from attackers; it simply doesn't provide a strong enough level of confidentiality. However, in practice, obfuscation can be a valuable component of any security policy because it deters casual snoopers and can often slow down dedicated attackers.

Integrity

Chapter 1 defined integrity as the expectation that only authorized parties are able to modify data. This requirement, like confidentiality, is typically addressed through access control mechanisms. However, additional measures must be taken when communication is performed over a channel that's not secure. In these cases, certain cryptographic methods, discussed in the following sections, are used to ensure data integrity.

Hash Functions

Cryptographic data integrity is enforced through a variety of methods, although **hash functions** are the basis of most approaches. A hash function (or "message digest function") accepts a variable-length input and generates a fixed-size output. The effectiveness of a hash function is measured primarily by three requirements. The first is that it must not be reversible, meaning that determining the input based only on the output should be computationally infeasible. This requirement is known as the "no pre-image" requirement. The second requirement is that the function not have a second pre-image, which means that given the input and the

output, generating an input with the same output is computationally infeasible. The final requirement, and the strongest, is that a hash must be relatively collision free, meaning that intentionally generating the same output for differing inputs should be computationally infeasible.

Hash functions provide the foundation of most programmatic integrity protection. They can be used to associate an arbitrary set of data with a unique, fixed-size value. This association can be used to avoid retaining sensitive data and to vastly reduce the storage required to validate a piece of data. The simplest forms of hash functions are **cyclic redundancy check (CRC)** routines. They are fast and efficient and offer a moderate degree of protection against unintentional data modification. However, CRC functions aren't effective against intentional modification, which makes them unusable for security purposes. Some popular CRC functions include CRC-16, CRC-32, and Adler-32.

The next step up from CRC functions are **cryptographic hash functions**. They are far more computationally intensive, but they offer a high degree of protection against intentional and unintentional modification. Popular hash functions include SHA-1, SHA-256, and MD5. (Issues with MD5 are discussed in more detail in "Bait-and-Switch Attacks" later in this chapter.)

Salt Values

Salt values are much the same as initialization vectors. The "salt" is a random value added to a message so that two messages don't generate the same hash value. As with an IV, a salt value *must not* be duplicated between messages. A salt value must be stored in addition to the hash so that the digest can be reconstructed correctly for comparison. However, unlike an IV, a salt value should be protected in most circumstances.

Salt values are most commonly used to prevent precomputation-based attacks against message digests. Most password storage methods use a salted hash value to protect against this problem. In a precomputation attack, attackers build a dictionary of all possible digest values so that they can determine the original data value. This method works only for fairly small ranges of input values, such as passwords; however, it can be extremely effective.

Consider a salt value of 32 random bits applied to an arbitrary password. This salt value increases the size of a password precomputation dictionary by four billion times its original value (2^{32}). The resulting precomputation dictionary would likely be too large for even a small subset of passwords. Rainbow tables, developed by Philippe Oechslin, are a real-world example of how a lack of a salt value leaves password hashes vulnerable to pre-computation attacks. Rainbow tables can be used to crack most password hashes in seconds, but the technique works only if the hash does not include a salt value. You can find more information on rainbow tables at the Project RainbowCrack website: http://www.antsight.com/zsl/rainbowcrack/.

Originator Validation

Hash functions provide a method of validating message content, but they can't validate the message source. Validating the source of a message requires incorporating some form of private key into the hash operation; this type of function is known as a **hash-based message authentication code (HMAC) function**. A MAC is a function that returns a fixed-length value computed from a key and variable-length message.

An HMAC is a relatively fast method of validating a message's content and sender by using a shared secret. Unfortunately, an HMAC has the same weakness as any shared key system: An attacker can impersonate any party in a conversation by compromising only one party's key.

Cryptographic Signatures

A **cryptographic signature** is a method of associating a message digest with a specific public key by encrypting the message digest with the sender's public and private key. Any recipient can then decrypt the message digest by using the sender's public key and compare the resulting value against the computed message digest. This comparison proves that the originator of the message must have had access to the private key.

Common Vulnerabilities of Integrity

Integrity vulnerabilities are similar to confidentiality vulnerabilities. Most integrity vulnerabilities can, in fact, be prevented by addressing confidentiality concerns. However, some integrity-related design vulnerabilities, discussed in the following sections, merit special consideration.

Bait-and-Switch Attacks

Commonly used hashing functions must undergo a lot of public scrutiny. However, over time, weaknesses tend to appear that could result in exploitable vulnerabilities. The **bait-and-switch attack** is typically one of the first weaknesses found in an aging hash function. This attack takes advantage of a weak hash function's tendency to generate collisions over certain ranges of input. By doing this, an attacker can create two inputs that generate the same value.

For example, say you have a banking application that accepts requests to transfer funds. The application receives the request, and if the funds are available, it signs the transfer and passes it on. If the hashing function is vulnerable, attackers could generate two fund transfers that produce the same digest. The first request would have a small value, and the second would be much larger. Attackers could then open an account with a minimum balance and get the smaller transfer approved. Then they would submit the larger request to the next system and close out their accounts before anyone was the wiser.

Bait-and-switch attacks have been a popular topic lately because SHA-1 and MD5 are starting to show some wear. The potential for collision vulnerabilities in MD5 was identified as early as 1996, but it wasn't until August 2004 that Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu published a paper describing successful collisions with the MD5 algorithm. This paper was followed up in March 2005 by Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. They successfully generated a colliding pair of X.509 certificates with different public keys, which is the certificate format used in SSL transactions. More recently, Vlastimil Klima published an algorithm in March 2006 that's capable of finding MD5 collisions in an extremely short time.

The SHA family of algorithms is also under close scrutiny. A number of potential attacks against SHA-0 have been identified; however, SHA-0 was quickly superseded by SHA-1 and never saw significant deployment. The SHA-0 attack research has provided the foundation for identifying vulnerabilities in the SHA-1 algorithm, although at the time of this writing, no party has successfully generated a SHA-1 collision. However, these issues have caused several major standards bodies (such as the U.S.-based NIST) to initiate phasing out SHA-1 in favor of SHA-256 (also known as SHA-2).

Of course, finding random collisions is much harder than finding collisions that are viable for a bait-and-switch attack. However, by their nature, cryptographic algorithms should be chosen with the intention that their security will be viable far beyond the applicable system's life span. This reasoning explains the shift in recent years from hashing algorithms that had previously been accepted as relatively secure. The impact of this shift can even be seen in password-hashing applications, which aren't directly susceptible to collision-based attacks, but are also being upgraded to stronger hash functions.

Availability

Chapter 1 defined availability as the capability to use a resource when expected. This expectation of availability is most often associated with reliability, and not security. However, there are a range of situations in which the availability of a system should be viewed as a security requirement.

Common Vulnerabilities of Availability

There is only one type of general vulnerability associated with a failure of availability—the denial-of-service (DoS) vulnerability. A DoS vulnerability occurs when an attacker can make a system unavailable by performing some unanticipated action.

The impact of a DoS attack can be very dependant on the situation in which it occurs. A critical system may include an expectation of constant availability, and outages would represent an unacceptable business risk. This is often the case with core business systems such as centralized authentication systems or flagship websites. In both of these cases, a successful DoS attack could correspond

directly to a significant loss of revenue due to the business's inability to function properly without the system.

A lack of availability also represents a security risk when an outage forces requirements to be addressed in a less secure manner. For example, consider a point-of-sale (PoS) system that processes all credit card transactions via a central reconciliation server. When the reconciliation server is unavailable, the PoS system must spool all of the transactions locally and perform them at a later time. An attacker may have a variety of reasons for inducing a DoS between a PoS system and the reconciliation server. The DoS condition may allow an attacker to make purchases with stolen or invalid credit cards, or it may expose spooled cardholder information on a less secure PoS system.

Threat Modeling

By now, you should have a good idea of how design affects the security of a software system. A system has defined functionality that's provided to its users but is bound by the security policy and trust model. The next step is to turn your attention to developing a process for applying this knowledge to an application you've been tasked to review. Ideally, you need to be able to identify flaws in the design of a system and prioritize the implementation review based on the most security-critical modules. Fortunately, a formalized methodology called **threat modeling** exists for just this purpose.

In this chapter, you use a specific type of threat modeling that consists of a five-phase process:

- Information collection
- Application architecture modeling
- Threat identification
- Documentation of findings
- Prioritizing the implementation review

This process is most effectively applied during the design (or a refactoring) phase of development and is updated as modifications are made in later development phases. It can, however, be integrated entirely at later phases of the SDLC. It can also be applied after development to evaluate an application's potential exposure. The phase you choose depends on your own requirements, but keep in mind that the design review is just a component of a complete application review. So make sure you account for the requirements of performing the implementation and operational review of the final system.

This approach to threat modeling should help establish a framework for relating many of the concepts you've already learned. This process can also serve as a

roadmap for applying many concepts in the remainder of this book. However, you should maintain a willingness to adapt your approach and alter these techniques as required to suit different situations. Keep in mind that processes and methodologies can make good servants but are poor masters.

> **Note**
> This threat-modeling process was originally introduced in *Writing Secure Code, 2nd Edition* (Microsoft Press, 2002) by Michael Howard and David Le Blanc. It was later expanded and refined in *Threat Modeling* (Microsoft Press, 2004) by Frank Swiderski and Window Snyder.

Information Collection

The first step in building a threat model is to compile all the information you can about the application. You shouldn't put too much effort into isolating security-related information yet because at this phase you aren't certain what's relevant to security. Instead, you want to develop an understanding of the application and get as much information as possible for the eventual implementation review. These are the key areas you need to identify by the end of this phase:

- *Assets*—Assets include anything in the system that might have value to attackers. They could be data contained in the application or an attached database, such as a database table of user accounts and passwords. An asset can also be access to some component of the application, such as the capability to run arbitrary code on a target system.
- *Entry points*—Entry points include any path through which an attacker can access the system. They include any functionality exposed via means such as listening ports, Remote Procedure Call (RPC) endpoints, submitted files, or any client-initiated activity.
- *External entities*—External entities communicate with the system via its entry points. These entities include all user classes and external systems that interact with the application.
- *External trust levels*—External trust levels refer to the privileges granted to an external entity, as discussed in "Trust Relationships" earlier in this chapter. A complex system might have several levels of external trust associated with different entities, whereas a simple application might have nothing more than a concept of local and remote access.
- *Major components*—Major components define the structure of an application design. Components can be internal to the application, or they might represent

external module dependencies. The threat-modeling process involves decomposing these components to isolate their security-relevant considerations.

- *Use scenarios*—Use scenarios cover all potential applications of the system. They include a list of both authorized and unauthorized scenarios.

Developer Interviews

In many situations, you can save yourself a lot of time by going straight to the horse's mouth, as it were. So if you have access to the developers, be sure to use this access to your advantage. Of course, this option might not be available. For instance, an independent vulnerability researcher rarely has access to the application's developers.

When you approach a system's developers, you should keep a few points in mind. First, you're in a position to criticize work they have put a lot of time and effort into. Make it clear that your goal is to help improve the security of their application, and avoid any judgmental or condescending overtones in your approach. After you have a decent dialogue going, you still need to verify any information you get against the application's implementation. After all, the developers might have their own misconceptions that could be a contributing factor to some vulnerabilities.

Developer Documentation

A well-documented application can make the review process faster and more thorough; however, there's one major catch to this convenience. You should always be cautious of any design documentation for an existing implementation. The reason for this caution isn't usually deceitful or incompetent developers; it's just that too many things change during the implementation process for the result to ever match the specifications perfectly.

A number of factors contribute to these inconsistencies between specifications and the implementation. Extremely large applications can often drift drastically from their specifications because of developer turnover and minor oversights compounded over time. Implementations can also differ simply because two people rarely have exactly the same interpretation of a specification. The bottom line is that you should expect to validate everything you determine from the design against the actual implementation.

Keeping this caveat in mind, you still need to know how to wring everything you can out of the documentation you get. Generally, you want anything you can get your hands on, including design (diagrams, protocol specifications, API documentation, and so on), deployment (installation guides, release notes, supplemental configuration information, and so forth), and end-user documentation. In binary (and some source code) reviews, end-user documentation is all you can get, but don't underestimate its value. This documentation is "customer-facing" literature, so it tends to be fairly accurate and can offer a process-focused view that makes the system easier to understand.

Standards Documentation

If you're asked to examine an application that uses standardized network protocols or file formats, a good understanding of how those protocols and file formats are structured is necessary to know how the application should function and what deficiencies might exist. Therefore, acquiring any published standards and related documentation created by researchers and authors is a good idea. Typically, Internet-related standards documents are available as requests for comments (RFCs, available at www.ietf.org/rfc/). Open-source implementations of the same standards can be particularly useful in clarifying ambiguities you might encounter when researching the technology a target application uses.

Source Profiling

Access to source code can be extremely helpful when you're trying to gather information on an application. You don't want to go too deep at this phase, but having the source code can speed up a lot of the initial modeling process. Source code can be used to initially verify documentation, and you can determine the application's general structure from class and module hierarchies in the code. When the source does not appear to be laid out hierarchically, you can look at the application startup to identify how major components are differentiated at initialization. You can also identify entry points by skimming the code to find common functions and objects, such as `listen()` or `ADODB`.

System Profiling

System profiling requires access to a functional installation of the application, which gives you an opportunity to validate the documentation review and identify elements the documentation missed. Threat models performed strictly from documentation need to skip this step and validate the model entirely during the implementation review.

You can use a variety of methods for profiling an application. Here are a few common techniques:

- *File system layout*—Look at the application's file system layout and make notes of any important information. This information includes identifying the permission structure, listing all executable modules, and identifying any relevant data files.
- *Code reuse*—Look for any application components that might have come from another library or package, such as embedded Web servers or encryption libraries. These components could present their own unique attack surface and require further review.
- *Imports and exports*—List the function import and export tables for every module. Look closely for any libraries used for establishing or managing external connections or RPC interfaces.

- *Sandboxing*—Run the application in a sandbox so that you can identify every object it touches and every activity it performs. Use a sniffer and application proxies to record any network traffic and isolate communication. In Windows environments, the Filemon, Regmon, WinObj, and Process Explorer utilities (from www.sysinternals.com) are helpful for this activity.
- *Scanning*—Probe the application on any listening ports, RPC interfaces, or similar external interfaces. Try grabbing banners to validate the protocols in use and identify any authentication requirements. For HTTP applications, try spidering links and identifying as many unique entry points as possible.

Application Architecture Modeling

After you have some background information, you need to begin examining the application architecture. This phase involves familiarizing yourself with how the software is structured and what components can affect its overall security. These steps help identify design concerns and let you know where to focus your energies during the implementation review. You build this knowledge by reviewing existing documentation of the application model and developing new models as required. Every piece of software is modeled to some extent during its development; the only difference is whether the models are ever formally recorded. So you need to understand the types of modeling in common use and how you can develop your own.

Unified Markup Language

Unified Markup Language (UML) is a specification developed by the Object Management Group (OMG; www.omg.org/uml/) to describe many different aspects of how an application operates from a fairly high level. It includes diagrams to describe information flow, interaction between components, different states the application can be in, and more. Of particular interest in this phase are class diagrams, component diagrams, and use cases. The following list briefly describes these types of diagrams so that you get a feel for what they're trying to convey. If you're unfamiliar with UML, picking up one of the myriad books available on the subject is strongly recommended. Because of UML's complexity, explaining it in depth is far beyond the scope of this chapter.

> **Note**
> UML has gone through several revisions. The currently accepted standard is UML 2.0.

- *Class diagrams*—A **class diagram** is a UML diagram for modeling an object-oriented (OO) solution. Each object class is represented by a rectangle that

includes the methods and attributes in the class. Relationships between objects are then represented by lines between classes. Lines with arrows on one end define parents in an inheritance hierarchy; unadorned lines (no arrows) with numbers near the ends indicate a cardinality relationship.

Class diagrams can be helpful when you're trying to understand relationships in a complex module. They essentially spell out how an application is modeled and how classes interact with each other. Realistically, however, you won't encounter them all that often unless you're performing in-house code reviews. By analyzing an OO solution, you can roughly construct class diagrams. Although doing so might seem like a waste of time, they can be useful when you need to come back and review the same software later or when you perform an initial high-level review and then hand off various code-auditing tasks to other members of a team.

- *Component diagrams*—**Component diagrams** divide a solution into its constituent components, with connectors indicating how they interact with each other. A component is defined as an opaque subsystem that provides an independent function for a solution. Examples of a component include a database, a parser of some description, an ordering system, and so forth. A component diagram offers a less complex view of a system than class diagrams do because components generally represent a complete self-contained subsystem, often implemented by many classes and modules.

  A component diagram exposes interfaces (denoted by protruding circles) and uses interfaces of other components (denoted by an empty semicircle). Components are tied together through these interface exposures or by means of association lines, which indicate that two components are inherently interrelated and don't rely on exposed interfaces. Component diagrams also allow two components to be joined together by **realization**. A realization simply means that the functionality required by one component is a subset of the functionality exposed by an interface of another component. Realization is represented by a dotted line.

  In an assessment, a component diagram can be valuable for defining the high-level view of a system and its intercomponent relationships. It can be especially useful when you're trying to develop the initial context of a threat model because it eliminates much of a system's complexity and allows you to focus on the big picture.

- *Use cases*—A use case is possibly the most nebulous component of the UML standard. There are no strict requirements for what a use case should look like or include. It can be represented with text or graphics, and developers choose which they prefer. Fundamentally, a **use case** is intended to describe how an

application should be used, so a good set of use cases can come in handy. After all, when you know what an application should be doing, addressing what it shouldn't be doing is easier. When reviewing use cases, keep an eye out for any developer assumptions about the system's behavior.

Data Flow Diagrams

A number of diagramming tools can aid in understanding a system, but the **data flow diagram (DFD)** is one of the most effective for security purposes. These diagrams are used to map how data moves through a system and identify any affected elements. If done properly, the DFD modeling process accounts not only for the application functionality exposed directly to external sources, but also the functionality that's exposed indirectly. This modeling process also accounts for mitigating factors in a system's design, such as additional security measures enforcing trust boundaries. Figure 2-2 shows the five main elements of a DFD, which are summarized in the following list:

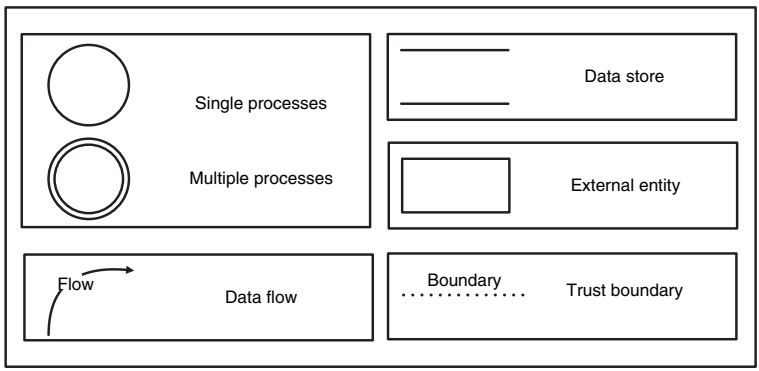

Figure 2-2 DFD elements

- *Processes*—**Processes** are opaque logic components with well-defined input and output requirements. They are represented with a circle, and groups of related processes are represented by a circle with a double border. Multiple process groups can be further decomposed in additional DFDs for each single process. Although processes aren't typically assets, they can be in some contexts.
- *Data stores*—**Data stores** are information resources the system uses, such as files and databases. They are represented by open-ended rectangular boxes. Usually, anything represented in this way in a DFD is considered a system asset.
- *External entities*—These elements, described previously in "Information Collection," are "actors" and remote systems that communicate with the system over its entry points. They are represented by closed rectangles. Identifying external

entities helps you isolate system entry points quickly and determine what assets are externally accessible. External entities might also represent assets that need to be protected, such as a remote server.

- *Data flow*—The flow of data is represented by arrows. It indicates what data is sent through what parts of the system. These elements can be useful for discovering what user-supplied data can reach certain components so that you can target them in the implementation review.
- *Trust boundary*—Trust boundaries are the boundaries between different entities in the system or between entire systems. They are represented by a dotted line between the two components.

Figure 2-3 shows how you can use DFD elements to model a system. It represents a simplified model of a basic Web application that allows users to log in and access resources stored in a database. Of course, DFDs look different at various levels of an application. A simple, high-level DFD that encapsulates a large system is referred to as a **context diagram**. The Web site example is a context diagram because it represents a high-level abstraction that encapsulates a complex system.

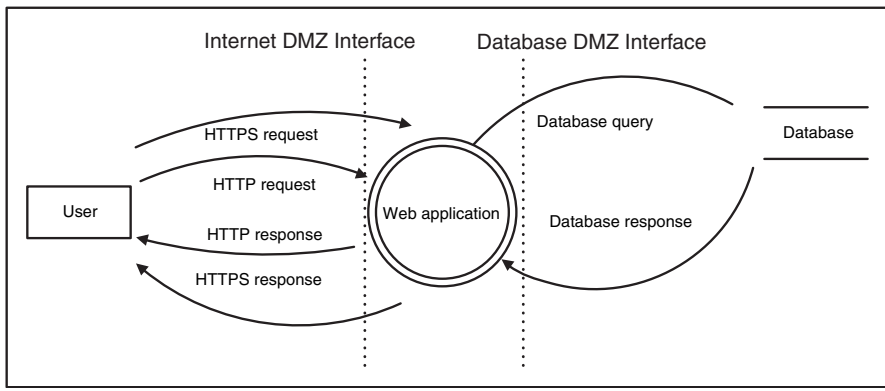

Figure 2-3 A DFD context diagram

However, your analysis generally requires you to decompose the system further. Each successive level of decomposition is labeled numerically, starting from zero. A level-0 diagram identifies the major application subsystems. The major subsystems in this Web application are distinguished by the user's authentication state. This distinction is represented in the level-0 diagram in Figure 2-4.

Depending on the complexity of a system, you may need to continue decomposing. Figure 2-5 is a level-1 diagram of the Web application's login process. Normally, you would only progress beyond level-0 diagrams when modeling complex subsystems. However, this level-1 diagram provides a useful starting point for using DFDs to isolate design vulnerabilities.

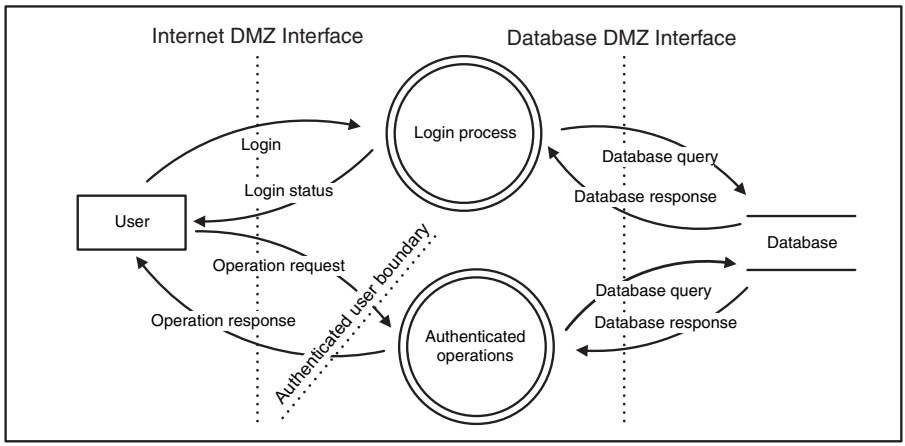

Figure 2-4 A DFD level-0 diagram of the login process

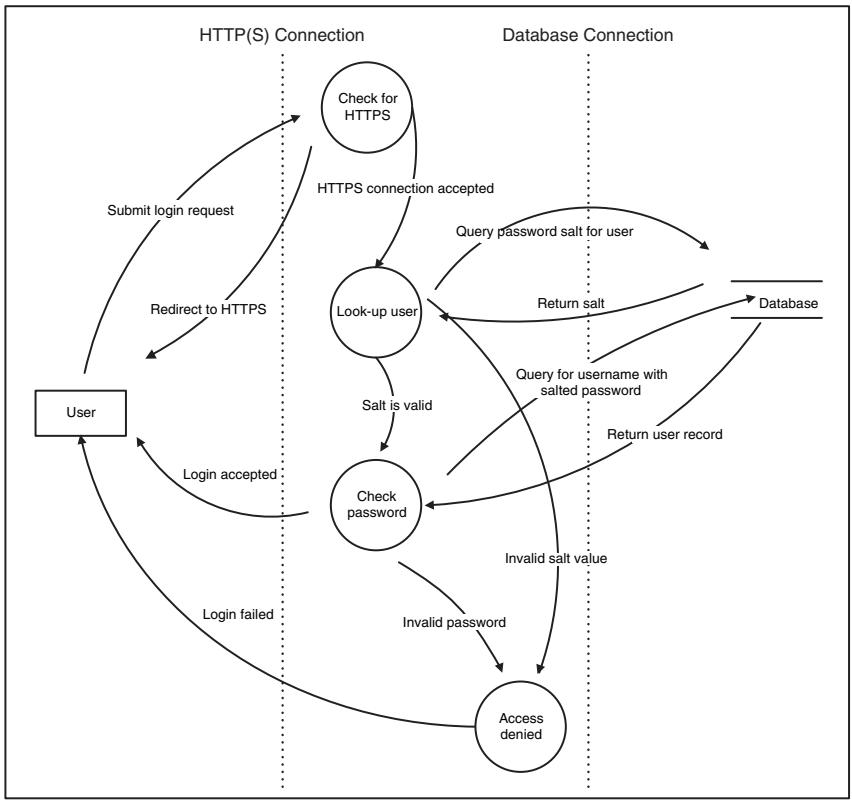

Figure 2-5 A DFD level-0 diagram of the login process

When preparing for an implementation review, you can use these diagrams to model application behavior and isolate components. For instance, Figure 2-6 shows the login process altered just a bit. Can you see where the vulnerability is? The way the login process handles an invalid login has been changed so that it now returns the result of each phase directly back to the client. This altered process is vulnerable because attackers can identify valid usernames without logging in successfully, which can be extremely useful in attempting a brute-force attack against the authentication system.

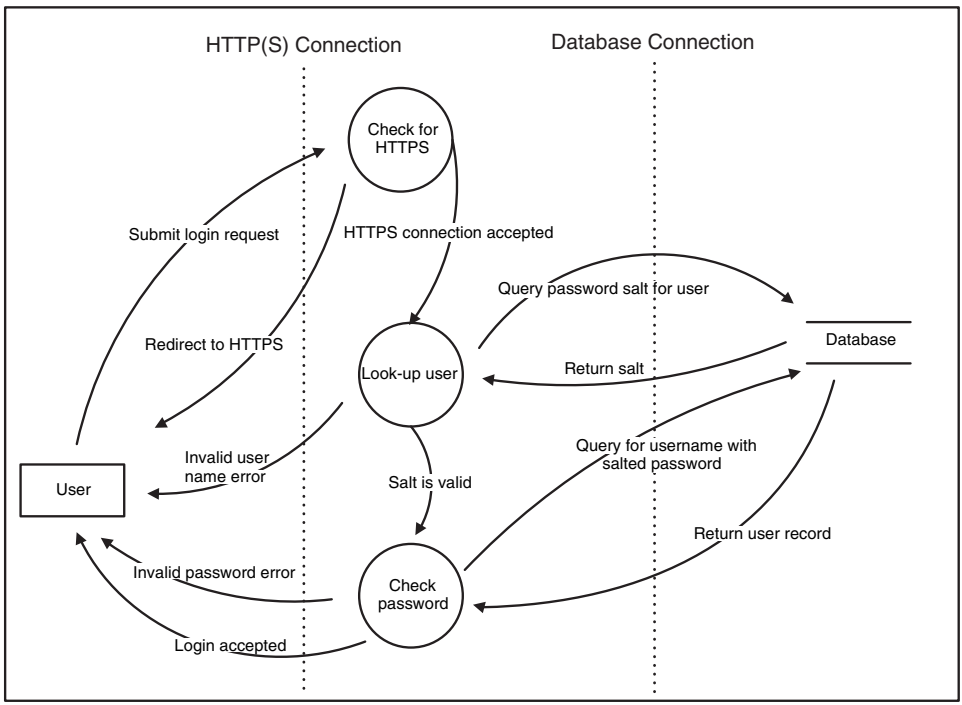

Figure 2-6 A DFD showing a login vulnerability

By diagramming this system, you can more easily identify its security components. In this example, it helped you isolate a vulnerability in the way the system authenticates. Of course, the login example is still fairly simple; a more complex system might have several layers of complexity that must be encapsulated in multiple DFDs. You probably don't want model all these layers, but you should decompose different components until you've reached a point that isolates the security-relevant considerations. Fortunately, there are tools to assist in this process. Diagramming applications such as Microsoft Visio are useful, and the Microsoft Threat Modeling Tool is especially helpful in this process.

Threat Identification

Threat identification is the process of determining an application's security exposure based on your knowledge of the system. This phase builds on the work you did in previous phases by applying your models and understanding of the system to determine how vulnerable it is to external entities. For this phase, you use a new modeling tool called **attack trees** (or **threat trees**), which provide a standardized approach for identifying and documenting potential attack vectors in a system.

Drawing an Attack Tree

The structure of an attack tree is quite simple. It consists of a root node, which describes the attacker's objective, and a series of subnodes that indicate ways of achieving that objective. Each level of the tree breaks the steps into more detail until you have a realistic map of how an attacker can exploit a system. Using the simple Web application example from the previous section, assume it's used to store personal information. Figure 2-7 shows a high-level attack tree for this application.

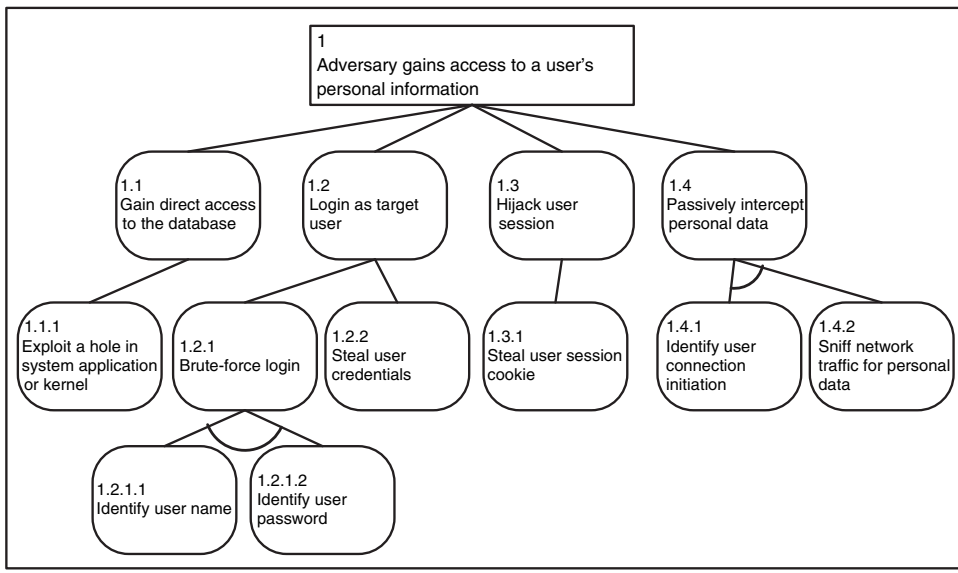

Figure 2-7 Attack tree example

As you can see, the root node is at the top with several subnodes underneath. Each subnode states an attack methodology that could be used to achieve the goal stated in the root node. This process is further decomposed, as necessary, into subnodes that eventually define an attack. Looking at this diagram, you should start to notice the similarities between attack trees and DFDs. After all, an attack tree

isn't developed in a vacuum. It's best created by walking through a DFD and using the attack tree to note specific concerns. As an example, notice how the branch leading to subnode 1.2.1 follows the same reasoning pattern used previously in analyzing the DFD of the flawed login process.

As with DFDs, you want to continue decomposing attack trees only along security-relevant paths. You need to use your judgment and determine what paths constitute reasonable attack vectors and what vectors are unlikely. Before getting into that topic, however, continue to the next section for a more detailed description of the attack tree structure.

Node Types

You might have noticed some strange markings in the lines connecting each node to its children (such as nodes 1.2.1.1 and 1.2.1.2). The arc between these node connectors indicates that the child nodes are AND nodes, meaning both conditions of the child node must be met to continue evaluating the vector. A node without an arc is simply an OR node, meaning either branch can be traversed without any additional condition. Referring to Figure 2-7, look at the brute-force login vector in node 1.2.1. To traverse past this node, you must meet the following conditions in the two subnodes:

- Identify username
- Identify user password

Neither step can be left out. A username with no password is useless, and a password without the associated username is equally useless. Therefore, node 1.2.1 is an AND node.

Conversely, OR nodes describe cases in which an objective can be reached by achieving any one of the subnodes. So the condition of just a single node must be met to continue evaluating the child nodes. Referring to Figure 2-7 again, look at the objective "Log in as target user" in node 1.2. This objective can be achieved with either of the following approaches:

- Brute-force login
- Steal user credentials

To log in as the user, you don't have to achieve both goals; you need to achieve only one. Therefore, they are OR nodes.

Textual Representation

You can represent attack trees with text as well as graphics. Text versions convey identical information as the graphical versions but sometimes aren't as easy to visualize (although they're more compact). The following example shows how you would represent the attack tree from Figure 2-7 in a text format:

```
1. Adversary gains access to a user's personal information
   OR  1.1 Gain direct access to the database
           1.1.1 Exploit a hole in system application or kernel
       1.2 Log in as target user
           OR  1.2.1 Brute-force login
               AND 1.2.1.1 Identify username
                   1.2.1.2 Identify user password
               1.2.2 Steal user credentials
       1.3 Hijack user session
           1.3.1 Steal user session cookie
       1.4 Passively intercept personal data
           AND 1.4.1 Identify user connection initiation
               1.4.2 Sniff network traffic for personal data
```

As you can see, all the same information is present. First, the root node objective is stated as the heading of the attack tree, and its immediate descendants are numbered and indented below the heading. Each new level is indented again and numbered below its parent node in the same fashion. The AND and OR keywords are used to indicate whether nodes are AND or OR nodes.

Threat Mitigation

Part of the value of an attack tree is that it allows you to track potential threats. However, tracking threats isn't particularly useful if you have no way of identifying how they are mitigated. Fortunately, attack trees include a special type of node for addressing that concern: a circular node. Figure 2-8 shows a sample attack tree with mitigating factors in place.

Three mitigation nodes have been added to this attack tree to help you realize that these vectors are less likely avenues of attack than the unmitigated branches. The dashed lines used in one mitigation node are a shorthand way to identify a branch as an unlikely attack vector. It doesn't remove the branch, but it does encourage you to direct your focus elsewhere.

One final note on mitigation: You don't want to look for it too early. Identifying mitigating factors is useful because it can prevent you from pursuing an unlikely attack vector. However, you don't want to get lulled into a false sense of security and miss a likely branch. So consider mitigation carefully, and make sure you perform some validation before you add it to your attack tree.

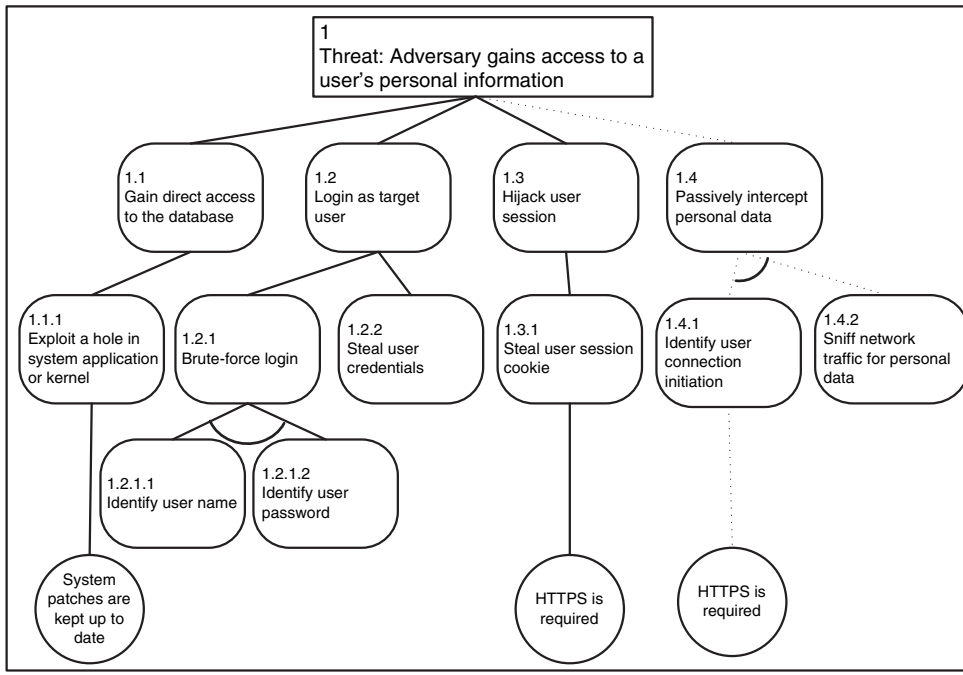

Figure 2-8 An attack tree with mitigation nodes

Documentation of Findings

Now that the investigative work is done, you need to document what you discovered. In the documentation phase, you will review the threats you uncovered in the previous phase and present them in a formal manner. For each threat you uncovered, you need to provide a brief summary along with any recommendations for eliminating the threat. To see how this process works, use the "Brute-force login" threat (node 1.2.1) from your sample attack tree. This threat could allow an attacker to log in with another user's credentials. The documentation of your threat summary would look similar to Table 2-1.

Table 2-1

Threat Summary	
Threat	Brute-force login.
Affected Component	Web application login component.
Description	Clients can brute-force attack usernames and passwords by repeatedly connecting and attempting to log in. This threat is increased because the application returns different error messages for invalid username and passwords, making usernames easier to identify.

Result	Untrusted clients can gain access to a user account and, therefore, read or modify sensitive information.
Mitigation Strategies	Make error messages ambiguous so that an attacker doesn't know whether the username or password is invalid. Lock the user account after repeated failed login attempts. (Three or five attempts would be appropriate.)

All the information for the brute-force login threat is neatly summarized in a table. In the next part of this phase, you extend this table to include some additional information on the risk of the threat.

DREAD Risk Ratings

Real-world applications are generally much larger and more complex in both design and implementation than the examples used in this chapter. Increased size and complexity creates a broad spectrum of attack vectors in a variety of user classes. As a result, you can usually come up with a long list of potential threats and possible recommendations to help mitigate those threats. In a perfect world, designers could systematically go about addressing each threat and fixing potential issues, closing each attack vector as necessary. However, certain business realities might not allow mitigating every identified vector, and almost certainly not all at once. Clearly, some sort of prioritization is needed to help address the more serious vectors before worrying about the less important ones. By assigning a threat severity rating, you can rank each uncovered threat based on the risk it poses to the security of the application and associated systems. This rating can then be used as a guideline for developers to help decide which issues take precedence.

You can choose to rate threats in a number of different ways. What's most important is that you incorporate the exposure of the threat (how easy is it to exploit and who the vector is available to) and the amount of damage incurred during a successful exploit. Beyond that, you might want to add components that are more pertinent to your environment and business processes. For this chapter's threat-modeling purposes, the DREAD rating system developed by Microsoft is used. No model is perfect, but this one provides a fairly good balance of commonly accepted threat characteristics. These characteristics are briefly summarized as follows:

- *Damage potential*—What are the repercussions if the threat is exploited successfully?
- *Reproducibility*—How easy is it to reproduce the attack in question?
- *Exploitability*—How difficult is it to perform the attack?
- *Affected users*—If a successful attack is carried out, how many users would be affected and how important are they?
- *Discoverability*—How difficult is it to spot the vulnerability?

Each category can be given a score between 1 and 10 (1 being the lowest, 10 the highest). Category scores are then totaled and divided by 5 for an overall threat rating. A rating of 3 or below can be considered a low-priority threat, 4 to 7 as a medium-priority threat, and 8 or greater as a high-priority threat.

> **Note**
> The DREAD model is also useful in rating implementation and operational vulnerabilities. In fact, you can use DREAD as your general-purpose rating system over the entire course of an application review.

One of the benefits of the DREAD rating system is that it provides a range of detail you can use when presenting results to business decision makers. You can give them a concise threat assessment, with just the total threat rating and the category it falls into. You could also present more detailed information, such as individual scores for the five threat categories. You might even want to give them a full report, including the model documentation and an explanation of how you arrived at the scores for each category. Regardless of your choice, it's a good idea to have information available at each level of detail when making a presentation to clients or senior management.

Table 2-2 is an example of applying a DREAD rating to the brute-force login threat.

Table 2-2

Threat Summary with DREAD Rating	
Threat	Brute-force login.
Affected Component	Web application login component.
Description	Clients can brute-force attack usernames and passwords by repeatedly connecting and attempting to log in. This threat is increased because the application returns a different error message for an invalid username than a valid one, making usernames easier to identify.
Result	Untrusted clients can gain access to a user account and, therefore, read or modify sensitive information.
Mitigation Strategies	Make error messages ambiguous so that an attacker doesn't know whether the username or password is invalid. Lock the user account after repeated failed login attempts. (Three to five attempts would be appropriate.)
Risk	Damage potential: 6 Reproducibility: 8 Exploitability: 4 Affected users: 5 Discoverability: 8 Overall: 6.2

Automatic Threat-Modeling Documentation

As you can see, quite a lot of documentation is involved in the threat-modeling process (both text and diagrams). Thankfully, Frank Swiderski (co-author of the previously mentioned *Threat Modeling*) has developed a tool to help with creating various threat-modeling documents. It's available as a free download at http://msdn.microsoft.com/security/securecode/threatmodeling/. The tool makes it easy to create DFDs, use cases, threat summaries, resource summaries, implementation assumptions, and many other documents you're going to need. Furthermore, the documentation is organized into a tree structure that's easy to navigate and maintain. The tool can output all your documentation as HTML or another output form of your choosing, using Extensible Stylesheet Language Transformations (XSLT) processing. Familiarizing yourself with this tool for threat-modeling documentation is strongly recommended.

Prioritizing the Implementation Review

Now that you've completed and scored your threat summaries, you can finally turn your attention to structuring the implementation review. When developing your threat model, you should have decomposed the application according to a variety of factors, including modules, objects, and functionality. These divisions should be reflected in the Affected Components entry in each individual threat summary. The next step is to make a list of components at the appropriate level of decomposition; exactly what level is determined by the size of the application, number of reviewers, time available for review, and similar factors. However, it's usually best to start at a high level of abstraction, so you only need to consider a handful of components. In addition to the component names, you need another column on your list for risk scores associated with each component.

After you have this component list, you simply identify which component a threat summary belongs to and add the risk score for that summary to the associated component. After you've totaled your list of summaries, you'll have a score for the risk associated with each component. Generally, you want to start your assessment with the highest scoring component and continue proceeding from highest to lowest. You might also need to eliminate some components due to time, budget, or other constraints. So it's best to start eliminating from the lowest scoring components. You can apply this scoring process to the next level of decomposition for a large application; although that starts to get into the implementation review process, which is covered in detail in Chapter 4, "Application Review Process."

Using a scoring list can make it a lot easier to prioritize a review, especially for a beginner. However, it isn't necessarily the best way to get the job done. An experienced auditor will often be able to prioritize the review based on their understanding of similar applications. Ideally, this should line up with the threat summary scores,

but sometimes that isn't the case. So it's important to take the threat summaries into account, but don't cling to them when you have a reason to follow a better plan.

Summary

This chapter has examined the essential elements of application design review. You've seen that security needs to be a fundamental consideration in application design and learned how decisions made in the design process can dramatically affect an application's security. You have also learned about several tools for understanding the security and vulnerability potential of an application design.

It's important that you not treat the design review process as an isolated component. The results of the design review should progress naturally into the implementation review process, discussed in depth in Chapter 4.

Chapter 3

Operational Review

"Civilization advances by extending the number of important operations which we can perform without thinking."

Alfred North Whitehead

Introduction

Operational vulnerabilities are the result of issues in an application's configuration or deployment environment. These vulnerabilities can be a direct result of configuration options an application offers, such as default settings that aren't secure, or they might be the consequence of choosing less secure modes of operation. Sometimes these vulnerabilities are caused by a failure to use platform security measures properly, such as file system and shared object permissions. Finally, an operational vulnerability could be outside the developer's direct control. This problem occurs when an application is deployed in a manner that's not secure or when the base platform inherits vulnerabilities from the deployment environment.

The responsibility for preventing these vulnerabilities can fall somewhere between the developer and the administrative personnel who deploy and maintain the system.

Shrink-wrapped commercial software might place most of the operational security burden on end users. Conversely, you also encounter special-purpose systems, especially embedded devices and turnkey systems, so tightly packaged that developers control every aspect of their configuration.

This chapter focuses on identifying several types of operational vulnerabilities and preventive measures. Concrete examples should help you understand the subtle patterns that can lead to these vulnerabilities. The goal is to help you understand how to identify these types of vulnerabilities, not present an encyclopedia of potential issues. Technologies are varied and change often, but with a little practice, you should be able to spot the commonalities in any operational vulnerability, which helps you establish your own techniques for identifying vulnerabilities in the systems you review.

Exposure

When reviewing application security, you need to consider the impact of the deployment environment. This consideration might be simple for an in-house application with a known target. Popular commercial software, on the other hand, could be deployed on a range of operating systems with unknown network profiles. When considering operational vulnerabilities, you need to identify these concerns and make sure they are adequately addressed. The following sections introduce the elements of an application's environment that define its degree of exposure to various classes of users who have access to and, therefore, are able to attack the application.

Attack Surface

Chapter 2, "Design Review," covered the threat-modeling concepts of assets and entry points. These concepts can be used to define an application's **attack surface**, the collection of all entry points that provide access to an asset. At the moment, how this access is mitigated isn't a concern; you just need to know where the attack surface is.

For the purposes of this chapter, the discussions of trust models and threats have been simplified because operational vulnerabilities usually occur when the attack surface is exposed unnecessarily. So it helps to bundle the complexities into the attack surface and simply look for where it can be eliminated.

The actual process of minimizing the attack surface is often referred to as "host hardening" or "application hardening." Hardening specific platforms isn't covered in this book, as better resources are dedicated to hardening a particular platform. Instead, this chapter focuses on several general operational vulnerabilities that occur because software deployment and configuration aren't secure.

Insecure Defaults

Insecure defaults are simply preconfigured options that create an unnecessary risk in a deployed application. This problem tends to occur because a software or device vendor is trying to make the deployment as simple and painless as possible—which brings you back to the conflict between usability and security.

Any reader with a commercial wireless access point has probably run into this same issue. Most of these devices are preconfigured without any form of connection security. The rationale is that wireless security is buggy and difficult to configure. That's probably true to an extent, but the alternative is to expose your wireless communications to anyone within a few hundred yards. Most people would rather suffer the inconvenience of struggling with configuration than expose their wireless communications.

As a reviewer, two types of vulnerable default settings should concern you the most. The first is the application's default settings, which include any options that can reduce security or increase the application's attack surface without the user's explicit consent. These options are discussed in more detail in the remainder of this chapter, but a few obvious installation considerations are prompting for passwords versus setting defaults, enabling more secure modes of communication, and enforcing proper access control.

You also need to consider the default settings of the base platform and operating system. Examples of this measure include ensuring that the installation sets adequate file and object permissions or restricting the verbs allowed in a Web request. The process can get a bit complicated if the application is portable across a range of installation targets, so be mindful of all potential deployment environments. In fact, one of main contributors to insecure defaults in an application is that the software is designed and built to run on many different operating systems and environments; a safe setting on one operating system might not be so safe on another.

Access Control

Chapter 2 introduced access control and how it affects an application's design. The effects of access control, however, don't stop at the design. Internally, an application can manage its own application-specific access control mechanisms or use features the platform provides. Externally, an application depends entirely on the access controls the host OS or platform provides (a subject covered in more depth later in Chapter 9, "Unix I: Privileges and Files," and Chapter 11, "Windows I: Objects and the File System").

Many developers do a decent amount of scripting; so you probably have a few scripting engines installed on your system. On a Windows system, you might have noticed that most scripting installations default to a directory right off the root. As an example, in a typical install of the Python interpreter on a Windows system, the

default installation path is `C:\Python24`, so it's installed directly off the root directory of the primary hard drive (C:). This installation path alone isn't an issue until you take into account default permissions on a Windows system drive. These permissions allow any user to write to a directory created off the root (permission inheritance is explained in more detail in Chapter 11). Browsing to `C:\Python24`, you find `python.exe` (among other things), and if you look at the imported dynamic link libraries (DLLs) that `python.exe` uses, you find `msvcr71.dll` listed.

> **Note**
> For those unfamiliar with basic Windows binary layout, an import is a required library containing routines the application needs to function correctly. In this example, `python.exe` needs routines implemented in the `msvcr71` library. The exact functions `python.exe` requires are also specified in the imports section.

Chapter 11 explains the particulars of how Windows handles imported. What's important to this discussion is that you can write your own `msvcr71.dll` and store it in the `C:\Python24` directory, and then it's loaded when anyone runs `python.exe`. This is possible because the Windows loader searches the current directory for named DLLs before searching system directories. This Windows feature, however, could allow an attacker to run code in the context of a higher privileged account, which would be particularly useful on a terminal server, or in any shared computing environment.

You could have the same problem with any application that inherits permissions from the root drive. The real problem is that historically, Windows developers have often been unaware of the built-in access control mechanisms. This is only natural when you consider that Windows was originally a single-user OS and has since evolved into a multiuser system. So these problems might occur when developers are unfamiliar with additional security considerations or are trying to maintain compatibility between different versions or platforms.

Unnecessary Services

You've probably heard the saying "Idle hands are the devil's playthings." You might not agree with it in general, but it definitely applies to unnecessary services. Unnecessary services include any functionality your application provides that isn't required for its operation. These capabilities often aren't configured, reviewed, or secured correctly.

These problems tend to result from insecure default settings but might be caused by the "kitchen sink mentality," a term for developers and administrators who include every possible capability in case they need it later. Although this approach might seem convenient, it can result in a security nightmare.

When reviewing an application, make sure you can justify the need for each component that's enabled and exposed. This justification is especially critical when you're reviewing a deployed application or turnkey system. In this case, you need to look at the system as a whole and identify anything that isn't needed.

The Internet Information Services (IIS) HTR vulnerabilities are a classic example of exposing a vulnerable service unnecessarily. HTR is a scripting technology Microsoft pioneered that never gained much following, which can be attributed to the release of the more powerful Active Server Pages (ASP) shortly after HTR. Any request made to an IIS server for a filename with an `.htr` extension is handled by the HTR Internet Server API (ISAPI) filter.

> **Note**
> ISAPI filters are IIS extension modules that can service requests based on file extensions.

From 1999 through 2002, a number of researchers identified HTR vulnerabilities ranging from arbitrary file reading to code execution. None of these vulnerabilities would have been significant, however, if this rarely used handler had simply been disabled in the default configuration.

Secure Channels

A **secure channel** is any means of communication that ensures confidentiality between the communicating parties. Usually this term is used in reference to encrypted links; however, even a named pipe can be considered a secure channel if access control is used properly. In either case, what's important is that only the correct parties can view or alter meaningful data in the channel, assuming, of course, that the parties have already been authenticated by some means.

Sometimes the need for secure channels can be determined during the design of an application. You might know before deployment that all communications must be conducted over secure channels, and the application must be designed and implemented in this way. More often, however, the application design must account for a range of possible deployment requirements.

The most basic example of a secure channel vulnerability is simply not using a secure channel when you should. Consider a typical Web application in which you authenticate via a password, and then pass a session key for each following transaction. (This topic is explained in more detail in Chapter 17, "Web Applications.") You expect password challenges to be performed over Secure Sockets Layer (SSL), but what about subsequent exchanges? After all, attackers would like to retrieve your password, but they can still get unrestricted access to your session if they get the session cookie.

This example shows that the need for secure channels can be a bit subtle. Everyone can agree on the need to protect passwords, but the session key might not be considered as important, which is perfectly acceptable sometimes. For example, most Web-based e-mail providers use a secure password exchange, but all remaining transactions send session cookies in the clear. These providers are offering a free service with a minimal guarantee of security, so it's an acceptable business risk. For a banking application, however, you would expect that all transactions occur over a secure channel.

Spoofing and Identification

Spoofing occurs whenever an attacker can exploit a weakness in a system to impersonate another person or system. Chapter 2 explained that authentication is used to identify users of an application and potentially connected systems. However, deploying an application could introduce some additional concerns that the application design can't address directly.

The TCP/IP standard in most common use doesn't provide a method for preventing one host from impersonating another. Extensions and higher layer protocols (such as IPsec and SSL) address this problem, but at the most basic level, you need to assume that any network connection could potentially be impersonated.

Returning to the SSL example, assume the site allows only HTTPS connections. Normally, the certificate for establishing connections would be signed by a trusted authority already listed in your browser's certificate database. When you browse to the site, the name on the certificate is compared against the server's DNS name; if they match, you have a reasonable degree of certainty that the site hasn't been spoofed.

Now change the example a bit and assume that the certificate isn't signed by a default trusted authority. Instead, the site's developer has signed the certificate. This practice is fairly common and perfectly acceptable if the site is on a corporate intranet. You simply need to ensure that every client browser has the certificate added to its database.

If that same site is on the public Internet with a developer-signed certificate, however, it's no longer realistic to assume you can get that certificate to all potential clients. The client, therefore, has no way of knowing whether the certificate can be trusted. If users browse to the site, they get an error message stating that the certificate isn't signed by a trusted authority; the only option is to accept the untrusted certificate or terminate the connection. An attacker capable of spoofing the server could exploit this situation to stage man-in-the-middle attacks and then hijack sessions or steal credentials.

Network Profiles

An application's network profile is a crucial consideration when you're reviewing operational security. Protocols such as Network File System (NFS) and Server Message Block (SMB) are acceptable inside the corporate firewall and generally are an absolute necessity. However, these same types of protocols become an unacceptable liability when they are exposed outside the firewall. Application developers often don't know the exact environment an application might be deployed in, so they need to choose intelligent defaults and provide adequate documentation on security concerns.

Generally, identifying operational vulnerabilities in the network profile is easier for a deployed application. You can simply look at what the environment is and identify any risks that are unacceptable, and what protections are in place. Obvious protections include deploying Internet-facing servers inside demilitarized zones (DMZs) and making sure firewall rule sets are as strict as reasonably possible.

Network profile vulnerabilities are more difficult to tackle when the environment is unknown. As a reviewer, you need to determine the most hostile potential environment for a system, and then review the system from the perspective of that environment. You should also ensure that the default configuration supports a deployment in this type of environment. If it doesn't, you need to make sure the documentation and installer address this problem clearly and specifically.

Web-Specific Considerations

The World Wide Web—more specifically, HTTP and HTTPS services—has become one of the most ubiquitous platforms for application development. The proliferation of Web services and applications is almost single-handedly responsible for the increased awareness of network security and vulnerabilities. For this reason, Web security warrants certain special considerations.

HTTP Request Methods

A Web application can be tightly restricted in which requests and operations are allowed; however, in practice, this restriction often isn't applied. For example, the server might support a number of HTTP methods, but all the application requires is the HTTP `GET`, `POST`, and `HEAD` requests. When reviewing a deployed or embedded Web application, you should ensure that only the necessary request methods are allowed. In particular, question whether `TRACE`, `OPTIONS`, and `CONNECT` requests should be allowed. If you are unfamiliar with these methods, you can find a lot more information in Chapter 17.

Directory Indexing

Many Web servers enable directory indexing by default. This setting has no effect in directories that provide an index file; however, it can expose valuable information to directories with no index. Often, these directories contain include and configuration files, or other important details on the application's structure, so directory indexing should be disabled by default.

File Handlers

When you try to run a file, it's obvious if the proper handler hasn't been installed. The server simply won't run the file, and instead it returns the source or binary directly. However, handler misconfiguration could happen in a number of less obvious situations. When machines are rebuilt or replaced, the correct handlers might not be installed before the application is deployed. Developers might also establish conventions for naming include files with different extensions. For example, Classic ASP and PHP: Hypertext Processor (PHP) include files are often named with an `.inc` extension, which is not interpreted by the default handlers in PHP or ASP. Because the include file isn't intended to be requested directly, developers and administrators might not realize it's vulnerable.

Both situations can result in a source or binary file disclosure, which allows attackers to download the raw source or binary code and get detailed information on the application's internal structure. In addition, PHP and other scripting languages commonly use include files to provide database account credentials and other sensitive information, which can make source disclosure vulnerabilities particularly dangerous.

This problem needs to be approached from three sides. First, developers need to choose a set of extensions to be used for all source and binary files. Second, the Web server should be configured with handlers for all appropriate file types and extensions. Finally, the only files in the Web tree should be those that must be retrieved by Web requests. Include files and supporting libraries should be placed outside the Web tree. This last step prevents attackers from requesting files directly that are only intended to be included.

An important extension to the last step is applicable when Web applications deal with uploaded content from clients. Applications commonly allow clients to upload files, but doing so has potentially dangerous consequences, especially if the directory where files are uploaded is within the Web tree. In this case, clients might be able to request the file they just uploaded; if the file is associated with a handler, they can achieve arbitrary execution. As an example, consider a PHP application that stores uploaded files in `/var/www/webapp/tmpfiles/`, which can be browsed via the HTTP URI `/webapp/tmpfiles/`. If the client uploads a file called `evil.php` and then requests `/webapp/tmpfiles/evil.php` in a browser, the Web server will likely recognize that the file is a PHP application and run code within the file's PHP tags.

Authentication

Web applications might not perform authentication internally; this process might be handled externally through the HTTP authentication protocol, an authenticating reverse proxy, or a **single sign-on (SSO)** system. With this type of authentication, it is especially important to make sure the external authentication mechanism is configured correctly and performs authentication in a safe manner. For example, a reverse-proxy device might add headers that include the current account name and user information. However, attackers could discover a request path that doesn't pass through the reverse proxy, which would allow them to set the account headers to whatever they want and impersonate any user on the system.

Default Site Installations

Some Web servers include a number of sample sites and applications as part of a default installation. The goal is to provide some reference for configuring the server and developing modules. In practice, however, these sample sites are a rather severe case of unnecessary services and insecure defaults. Numerous security problems have been caused by installing sample Web applications and features. For example, ColdFusion's Web-scripting technologies used to install several sample applications by default that allowed clients to upload files and run arbitrary code on the system.

> **Note**
> This ColdFusion bug ties in with some of the previous discussion on spoofing and identification. The sample applications were accessible only to clients who connected from the same machine where ColdFusion was installed. However, the way they verified whether the client was connecting locally was to check the HTTP HOST variable, which is completely controlled by the client. As a result, any client could claim to be connecting locally and access sample scripts with the dangerous functionality. This bug is documented at www.securityfocus.com/bid/3154/info.

Overly Verbose Error Messages

Most Web servers return fairly verbose error messages that assist in diagnosing any problems you encounter. Web application platforms also provide detailed exception information to assist developers in debugging code. These capabilities are essential when developing a system, but they can be a serious operational vulnerability in a deployed system.

The burden of end-user error reporting should rest primarily on application developers. The application level has the correct context to determine what information is

appropriate to display to end users. Configuration of the base platform should always be performed under the assumption that the application is filtering and displaying any end-user error information. This way, the deployed system can be configured to report the minimum necessary information to client users and redirect any required details to the system log.

Public-Facing Administrative Interfaces

Web-based administration has become popular for Web applications and network devices. These administrative interfaces are often convenient, but they are rarely implemented with potentially malicious users in mind. They might use weak default passwords, not perform sufficient authentication, or have any number of other vulnerabilities. Therefore, they should be accessible only over restricted network segments when possible and never exposed to Internet-facing connections.

Protective Measures

A range of additional protective measures can affect an application's overall security. In consultant speak, they are often referred to as **mitigating factors** or **compensating controls**; generally, they're used to apply the concept of defense in depth mentioned in Chapter 2. These measures can be applied during or after the development process, but they tend to exist outside the software itself.

The following sections discuss the most common measures, but they don't form an exhaustive list. For convenience, these measures have been separated into groups, depending on whether they're applied during development, to the deployed host, or in the deployed network. One important consideration is that most of these measures include software, so they could introduce a new attack surface or even vulnerabilities that weren't in the original system.

Development Measures

Development protective measures focus on using platforms, libraries, compiler options, and hardware features that reduce the probability of code being exploited. These techniques generally don't affect the way code is written, although they often influence the selection of one platform over another. Therefore, these measures are viewed as operational, not implementation measures.

Nonexecutable Stack

The classic stack buffer overflow is quite possibly the most often-used software vulnerability in history, so hardware vendors are finally trying to prevent them at the lowest possible level by enforcing the nonexecutable protection on memory pages.

This technique is nothing new, but it's finally becoming common in inexpensive commodity hardware, such as consumer PCs.

A nonexecutable stack can make it harder to exploit a memory management vulnerability, but it doesn't necessarily eliminate it because the exploit might not require running code from the stack. It might simply involve patching a stack variable or the code execution taking advantage of a return to `libc` style attack. These vulnerabilities are covered in more detail in Chapter 5, "Memory Corruption," but for now, it's important to understand where the general weaknesses are.

Stack Protection

The goal of the classic stack overflow is to overwrite the instruction pointer. Stack protection prevents this exploit by placing a random value, called a "canary," between stack variables and the instruction pointer. When a function returns, the canary is checked to ensure that it hasn't changed. In this way, the application can determine whether a stack overflow has occurred and throw an exception instead of running potentially malicious code.

Like a nonexecutable stack, stack protection has its share of weaknesses. It also doesn't protect against stack variable patching (although some implementations reorder variables to prevent the likelihood of this problem). Stack protection mechanisms might also have issues with code that performs certain types of dynamic stack manipulation. For instance, LibSafePlus can't protect code that uses the `alloca()` call to resize the stack; this problem can also be an undocumented issue in other implementations. Worse yet, some stack protections are vulnerable to attacks that target their implementation mechanisms directly. For example, an early implementation of Microsoft's stack protection could be circumvented by writing past the canary and onto the current exception handler.

No form of stack protection is perfect, and every implementation has types of overflows that can't be detected or prevented. You have to look at your choices and determine the advantages and disadvantages. Another consideration is that it's not uncommon for a development team to enable stack protection and have the application stop functioning properly. This problem happens because of stack overflows occurring somewhere in the application, which may or may not be exploitable. Unfortunately, developers might have so much trouble tracking down the bugs that they choose to disable the protection entirely. You might need to take this possibility into account when recommending stack protection as an easy fix.

Heap Protection

Most program heaps consist of a doubly linked list of memory chunks. A generic heap exploit attempts to overwrite the list pointers so that arbitrary data can be written somewhere in the memory space. The simplest form of heap protection involves checking that list pointers reference valid heap chunks before performing any list management.

Simple heap protection is fairly easy to implement and incurs little performance overhead, so it has become common in the past few years. In particular, Microsoft's recent OS versions include a number of heap consistency-checking mechanisms to help minimize the damage heap overwrites can do. The GNU `libc` also has some capabilities to protect against common exploitation techniques; the memory management routines check linked list values and validate the size of chunks to a certain degree. Although these mechanisms are a step in the right direction, heap overflows can still be exploited by manipulating application data rather than heap structures.

Address Space Layout Randomization

When an application is launched in most contemporary operating systems, the loader organizes the program and required libraries into memory at the same locations every time. Customarily, the program stack and heap are put in identical locations for each program that runs. This practice is useful for attackers exploiting a memory corruption vulnerability; they can predict with a high degree of accuracy the location of key data structures and program components they want to manipulate or misuse. **Address space layout randomization (ASLR)** technologies seek to remove this advantage from attackers by randomizing where different program components are loaded at in memory each time the application runs. A data structure residing at address 0x12345678 during one program launch might reside at address 0xABCD5678 the next time the program is started. Therefore, attackers can no longer use hard-coded addresses to reliably exploit a memory corruption flaw by targeting specific structures in memory. ASLR is especially effective when used with other memory protection schemes; the combination of multiple measures can turn a bug that could previously be exploited easily into a very difficult target. However, ASLR is limited by a range of valid addresses, so it is possible for an attacker to perform a repeated sequence of exploit attempts and eventually succeed.

Registered Function Pointers

Applications might have long-lived functions pointers at consistent locations in a process's address space. Sometimes these pointers are defined at compile time and never change for a given binary; exception handlers are one of the most common examples. These properties make long-lived function pointers an ideal target for exploiting certain classes of vulnerabilities. Many types of vulnerabilities are similar, in that they allow only a small value to be written to one arbitrary location, such as attacks against heap management functions.

Function pointer registration is one attempt at preventing the successful exploit of these types of vulnerabilities. It's implemented by wrapping function pointer calls in some form of check for unauthorized modification. The exact details of the

check might vary in strength and how they're performed. For example, the compiler can place valid exception handlers in a read-only memory page, and the wrapper can just make a direct comparison against this page to determine whether the pointer is corrupt.

Virtual Machines

A virtual machine (VM) platform can do quite a bit to improve an application's basic security. Java and the .NET Common Language Runtime (CLR) are two popular VM environments, but the technology is even more pervasive. Most popular scripting languages (such as Perl, Python, and PHP) compile first to bytecode that's then interpreted by a virtual machine.

Virtual machine environments are typically the best choice for most common programming tasks. They generally provide features such as sized buffers and strings, which prevent most memory management attacks. They might also include additional protection schemes, such as the code access security (CAS) mentioned in Chapter 1. These approaches usually allow developers to create more secure applications more quickly.

The downside of virtual machines is that their implicit protection stops at low-level vulnerabilities. VM environments usually have no additional protections against exploiting vulnerabilities such as race conditions, formatted data manipulation, and script injection. They might also provide paths to low-level vulnerabilities in the underlying platform or have their own vulnerabilities.

Host-Based Measures

Host-based protections include OS features or supporting applications that can improve the security of a piece of software. They can be deployed with the application or be additional measures set up by end users or administrators. These additional protective measures can be useful in preventing, identifying, and mitigating successful exploits, but remember that these applications are pieces of software. They might contain vulnerabilities in their implementations and introduce new attack surface to a system.

Object and File System Permissions

Permission management is the first and most obvious place to try reducing the attack surface. Sometimes it's done programmatically, such as permissions on a shared memory object or process synchronization primitive. From an operational perspective, however, you're concerned with permissions modified during and after application installation.

As discussed earlier in this chapter, permission assignment can be complicated. Platform defaults might not provide adequate security, or the developer might not be aware of how a decision could affect application security. Typically, you need to perform at least a cursory review of all files and objects included in a software installation.

Restricted Accounts

Restricted accounts are commonly used for running an application with a public-facing service. The intent of using this type of account is not to prevent a compromise but to reduce the impact of the compromise. Therefore, these accounts have limited access to the system and can be monitored more closely.

On Windows systems, a restricted account usually isn't granted network access to the system, doesn't belong to default user groups, and might be used with restricted tokens. Sudhakar Govindavajhala and Andrew W. Appel of Princeton University published an interesting paper, "Windows Access Control Demystified," in which they list a number of considerations and escalation scenarios for different group privileges and service accounts. This paper is available at http://www.cs.princeton.edu/~sudhakar/papers/winval.pdf.

Restricted accounts generally don't have a default shell on UNIX systems, so attackers can't log in with that account, even if they successfully set a password through some application flaw. Furthermore, they usually have few to no privileges on the system, so if they are able to get an interactive shell somehow, they can't perform operations with much consequence. Having said that, attackers simply having access to the system is often dangerous because they can use the system to "springboard" to other previously inaccessible hosts or perform localized attacks on the compromised system to elevate privileges.

Restricted accounts are useful, but they can be deployed carelessly. You need to ensure that restricted accounts contain no unnecessary rights or privileges. It's also good to follow the rule of one account to one service because of the implicit shared trust between all processes running under the same account, as discussed in Chapter 2.

Chroot Jails

UNIX operating systems use the `chroot` command to change the root directory of a newly executed process. This command is normally used during system startup or when building software. However, `chroot` also has a useful security application: A nonroot process can be effectively jailed to a selected portion of the file system by running it with the `chroot` command.

This approach is particularly effective because of UNIX's use of the file system as the primary interface for all system activity. An attacker who exploits a jailed process is still restricted to the contents of the jailed file system, which prevents access to most of the critical system assets.

A `chroot` jail can improve security quite a bit; however, there are caveats. Any process running under root privileges can usually escape the jail environment by using other system mechanisms, such as the PTRACE debugging API, setting system variables with `sysctl`, or exploiting some other means to allow the system to run a new arbitrary process that's not constrained to the chroot jail. As a result, `chroot` jails are more effective when used with a restricted account. In addition, a `chroot` jail doesn't restrict network access beyond normal account permissions, which could still allow enough attack surface for a follow-on attack targeted at daemons listening on the localhost address.

System Virtualization

Security professionals have spent the past several years convincing businesses to run one public-facing service per server. This advice is logical when you consider the implicit shared trusts between any processes running on the same system. However, increases in processing power and growing numbers of services have made this practice seem unnecessarily wasteful.

Fortunately, virtualization comes to the rescue. **Virtualization** allows multiple operating systems to share a single host computer. When done correctly, each host is isolated from one another and can't affect the integrity of other hosts except through standard network interfaces. In this way, a single host can provide a high level of segmentation but still make efficient use of resources.

Virtualization is nothing new; it's been around for decades in the mainframe arena. However, most inexpensive microcomputers haven't supported the features required for true hardware virtualization—these features are known as the Popek and Goldberg virtualization requirements. True hardware virtualization involves capabilities that hardware must provide to virtualize access without requiring software emulation. Software virtualization works, of course, but only recently has commodity hardware become powerful enough to support large-scale virtualization.

Virtualization will continue to grow, however. New commodity processors from vendors such as Intel and AMD now have full hardware virtualization support, and software virtualization has become more commonplace. You can now see a handful of special cases where purpose-built operating systems and software are distributed as virtual machine disk images. These concepts have been developing for more than a decade through research in exokernels and para-virtualization, with commercial products only now becoming available.

For auditors, virtualization has advantages and disadvantages. It could allow an application to be distributed in a strictly configured environment, or it might force a poorly configured black box on users. The best approach is to treat a virtualized system as you would any other system and pay special attention to anywhere the virtual segmentation is violated. As virtualization grows more popular, however, it will almost certainly introduce new and unique security concerns.

Enhanced Kernel Protections

All operating systems must provide some mechanism for user land applications to communicate with the kernel. This interface is typically referred to as the **system call gateway**, and it should be the only interface for manipulating base system objects. The system call gateway is a useful trust boundary, as it provides a choke-point into kernel operations. A kernel module can then intercept requested operations (or subsequent calls) to provide a level of access control that is significantly more granular than normal object permissions.

For example, you might have a daemon that you need to run as root, but this daemon shouldn't be able to access arbitrary files or load kernel modules. These restrictions can be enforced only by additional measures taken inside the kernel. An additional set of permissions can be mapped to the executable and user associated with the process. In this case, the kernel module would refuse the call if the executable and user match the restricted daemon. This approach is an example of a simple type of enhanced kernel protection; however, a number of robust implementations are available for different operating systems. SELinux is a popular module for Linux and BSD systems, and Core Force (from Core Security) is a freely available option for Windows 2000 and XP systems.

There's no question that this approach offers fine-grained control over exactly what a certain process is allowed to do. It can effectively stop a compromise by restricting the rights of even the most privileged accounts. However, it's a fairly new approach to security, so implementations vary widely in their capabilities and operation. This approach can also be difficult to configure correctly, as most applications aren't designed with the expectation of operating under such tight restrictions.

Host-Based Firewalls

Host-based firewalls have become extremely popular in recent years. They often allow fine-grained control of network traffic, including per-process and per-user configuration. This additional layer of protection can help compensate for any overlooked network attack surface. These firewalls can also mitigate an attack's effect by restricting the network access of a partially compromised system.

For the most part, you can view host-based firewalls in the same manner as standard network firewalls. Given their limited purpose, they should be much less complicated than a standard firewall, although per-process and per-user rules can increase their complexity somewhat.

Antimalware Applications

Antimalware applications include antivirus and antispyware software. They are usually signature-based systems that attempt to identify behaviors and attributes associated with malicious software. They might even incorporate a degree of enhanced kernel protection, host-based firewalling, and change monitoring. For the

most part, however, these applications are useful at identifying known malware applications. Typically, they have less value in handling more specialized attacks or unknown malware.

Antimalware applications generally have little effect when auditing software systems. The primary consideration is that a deployed system should have the appropriate software installed and configured correctly.

File and Object Change Monitoring

Some security applications have methods of monitoring for changes in system objects, such as configuration files, system binaries, and sensitive Registry keys. This monitoring can be an effective way to identify a compromise, as some sensitive portion of the system is often altered as a result of an exploit. More robust monitoring systems actually maintain digests (or hashes) of sensitive files and system objects. They can then be used to assist in forensic data analysis in the event of a serious compromise.

Change monitoring is a fairly reactive process by nature, so generally it isn't useful in preventing compromises. It can, however, prove invaluable in identifying, determining the extent of, and mitigating a successful compromise. The most important consideration for auditors is that most change-monitoring systems are configured by default to monitor only base system objects. Adding monitoring for application-specific components usually requires changes to the default configuration.

Host-Based IDSs/IPSs

Host-based **intrusion detection systems (IDSs)** and **intrusion prevention systems (IPSs)** tend to fall somewhere between host-based firewalls and antimalware applications. They might include features of both or even enhanced kernel protections and file change monitoring. The details vary widely from product to product, but typically these systems can be viewed as some combination of the host-based measures presented up to this point.

Network-Based Measures

An entire book could be devoted to the subject of secure network architecture. After all, security is only one small piece of the puzzle. A good network layout must account for a number of concerns in addition to security, such as cost, usability, and performance. Fortunately, a lot of reference material is available on the topic, so this discussion has been limited to a few basic concepts in the following sections. If you're not familiar with network fundamentals, you should start with a little research on TCP/IP and the Open Systems Interconnection (OSI) model and network architecture.

Segmentation

Any discussion of network security needs to start with segmentation. **Network segmentation** describes how communication over a network is divided into groupings at different layers. TCP/IP networks are generally segmented for only two reasons: security and performance. For the purposes of this discussion, you're most concerned with the security impact of network segmentation.

You can view network segmentation as a method of enforcing trust boundaries. This enforcement is why security is an important concern when developing a network architecture. You should also consider what OSI layer is used to enforce a security boundary. Generally, beginning with the lowest layer possible is best. Each higher layer should then reinforce the boundary, as appropriate. However, you always encounter practical constraints on how much network security can be provided and limitations on what can be enforced at each layer.

Layer 1: Physical

The security of the physical layer is deceptively simple. Segmentation of this layer is literally physical separation of the transmission medium, so security of the physical layer is simply keeping the medium out of attackers' hands. In the past, that meant keeping doors locked, running cables through conduit, and not lighting up unconnected ports. If any transmission media were outside your immediate control, you just added encryption or protected at higher layers.

Unfortunately, the rapid growth of wireless networking has forced many people to reevaluate the notion of physical layer security. When you deploy a wireless network, you expose the attack surface to potentially anyone in transmission range. With the right antenna and receiver, an attacker could be a mile or more away. When you consider this possibility with the questionable protection of the original Wired Equivalent Privacy (WEP) standard, it should be apparent that physical layer security can get more complicated.

Layer 2: Data Link

Segmentation at the data link layer is concerned with preventing spoofing (impersonating) hosts and sniffing traffic (capturing data transmitted by other hosts). Systems at this layer are identified via Media Address Control (MAC) addresses, and the Address Resolution Protocol (ARP) is used to identify MAC addresses associated with connected hosts. **Switching** is then used to route traffic to only the appropriate host.

Network switches, however, run the gamut in terms of features and quality. They might be vulnerable to a variety of ARP spoofing attacks that allow attackers to impersonate another system or sniff traffic destined for other systems. Address filtering can be used to improve security at this layer, but it should never be relied on as the sole measure.

Wireless media creates potential concerns at this layer, too, because they add encryption and authentication to compensate for their inability to segment the physical layer adequately. When choosing a wireless protection protocol, you have a few options to consider. Although proprietary standards exist, open standards are more popular, so this section focuses on them.

WEP was the original standard for wireless authentication and encryption; however, its design proved vulnerable to cryptanalytic attacks that were further aggravated by weaknesses in a number of implementations. Wi-Fi Protected Access (WPA) is a more robust standard that provides more secure key handling with the same base encryption capabilities as WEP (which allows it to operate on existing hardware). However, WPA was intended as only an interim measure and has been superseded by WPA2, which retains the essential key-handling improvements of WPA and adds stronger encryption and digest capabilities.

Layer 3: Network

Security and segmentation at the network layer are typically handled via IP filtering and, in some cases, the IP Security (IPsec) protocol. Any meaningful discussion of IPsec is beyond the scope of this book, but it's important to note exactly what it is. IPsec is a component of the IPv6 specification that has been back-ported to the current IPv4. It provides automatic encryption and authentication for TCP/IP connections at the network layer. Although IPsec does have some appealing security capabilities, its adoption has been slow, and different technologies have been developed to address many of the areas it was intended for. However, adoption is continuing to grow, and a properly deployed IPsec environment is extremely effective at preventing a range of network attacks, including most sniffing and spoofing attacks.

IP filtering is a fairly simple method of allowing or denying packets based only on the protocol, addresses, and ports. This method allows traffic to be segmented according to its function, not just the source and destination. This type of filtering is easy to implement, provides fast throughput, and has fairly low overhead. At this point, IP filtering is practically a default capability expected in almost any network-enabled system, such as a router or an OS. The disadvantage of IP filtering is that it maintains no connection state. It can't discriminate based on which side is establishing the connection or whether the communication is associated with an active connection. Therefore, a simple IP filter must allow inbound traffic to any port where it allows outbound traffic.

Layer 4: Transport

The transport layer is what most people think of when they discuss network security architecture. This layer is low enough to be common to all TCP/IP applications but high enough that you can determine connection state. The addition of state allows a firewall to determine which side is initiating the connection and establishes the fundamental concept of an internal and external network.

Firewalls, which are devices that filter traffic at the network and transport layers, are the primary method of segmenting a network for security purposes. The simplest firewall has only two interfaces: inside and outside. The simplest method of firewalling is to deny all inbound traffic and allow all outbound traffic. Most host-based firewalls and personal firewalls are configured this way by default.

Firewalls get interesting, however, when you use them to divide a network according to functional requirements. For example, say you know that employees on your network need only outbound Web access. You can allow only TCP ports 80 and 443 outbound and deny all the rest. The company Web site is hosted locally, so you need to add TCP port 80 inbound to the Web server. (Note: A number of other likely services, most notably DNS, have been ignored to keep this example simple.) However, you don't like the idea of having an opening straight into the internal network via TCP port 80. The solution is to deploy the Web server inside a **demilitarized zone (DMZ)**. A DMZ uses a third interface from the firewall containing its own set of rules. First, assume that the DMZ is configured to deny any connections by default, which lets you start with a clean slate. Next, you need to move the Web server into the DMZ, remove the deny inbound rule for port 80, and replace it with a rule that allows inbound traffic from the external network to the Web server in the DMZ on TCP port 80. Figure 3-1 shows an example of this network.

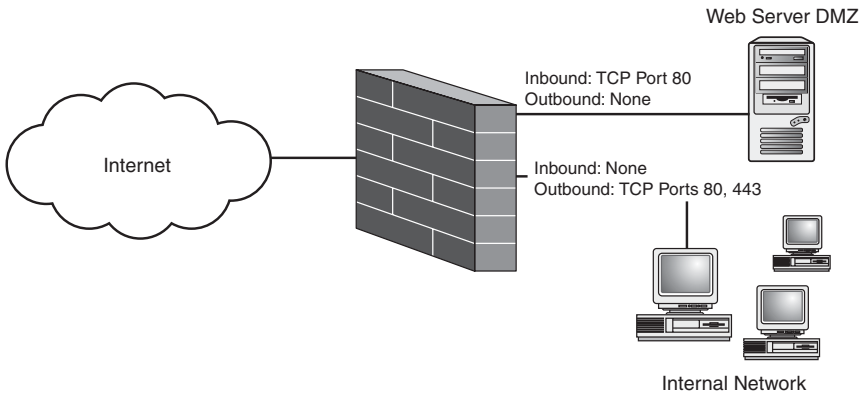

Figure 3-1 Simple DMZ example

This example, although simple, conveys the basics of transport-layer segmentation. What's important to understand is that the network should be segmented by function as much as reasonably possible. Continuing the example, what if the Web server is backed by a database on a separate system? The database might contain particularly sensitive customer information that shouldn't be located inside the DMZ. However, migrating the database to the internal network requires opening

connectivity from the DMZ into the internal network, which might not be an acceptable risk, either. In this case, adding a second DMZ containing a data tier for the Web front end might be necessary.

When reviewing an in-place application, you need to take these environmental considerations into account. There will always be legitimate reasons to prevent a deployment from having the ideal segmentation. However, you should aware of these contributing factors and determine whether the environment is adequately segmented for the application's security requirements.

Layers 5 and 6: Session and Presentation

Some layers of the OSI model don't map cleanly to TCP/IP; for example, the session and presentation layer generally get pushed up into the TCP/IP application layer. However, collectively these layers provide a useful distinction for certain application protocols. Platform-specific features, such as RPC interfaces and named pipes, are generally accepted as session- and presentation-layer protocols. Security on these interfaces is typically handled programmatically and should be addressed via the platform's native access control mechanisms.

Secure Socket Layer/Transport Layer Security (SSL/TLS) is another protocol that's more appropriately discussed in terms of the session or presentation layer. The "Secure Channels" section earlier in this chapter discussed how SSL can be used to create a secure encrypted channel. SSL/TLS also supports certificate-based authentication, which can reduce an application's attack surface by enforcing authentication below the application layer.

Layer 7: Application

Application-layer security is an interesting mix, and most of this book is devoted to it. However, application-layer proxies fall squarely into the category of operational protective measures. If you've spent any time in network security, you've probably heard numerous discussions of the value of heterogeneous (or mixed) networks. On the positive side, a heterogeneous environment is much less prone to silver bullet attacks, in which an attacker can compromise the bulk of a network by taking advantage of a single vulnerability. However, a homogeneous environment is usually easier and less expensive to manage.

Application-layer gateways are interesting because they add extra network diversity in just the right location. Some of the first popular application gateways were simply validating HTTP reverse proxies. They sat in front of vulnerability-prone Web servers and denied malformed Web traffic, which provided moderate protection against Web server attacks. Newer Web application gateways have added a range of capabilities, including sitewide authentication, exploit detection, and fine-grained rule sets.

Overall, application gateways are no substitute for properly coded applications. They have significant limitations, and configuring rules for the most effective

protection requires more effort than assessing and fixing a potentially vulnerable application. However, these gateways can increase a network's diversity, provide an extra layer of assurance, and add a layer of protection over a questionable third-party application.

Network Address Translation (NAT)

Network Address Translation (NAT) provides a method of mapping a set of internal addresses against a different set of external addresses. It was originally developed to make more efficient use of the IPv4 address space by mapping a larger number of private, internal network addresses to a much smaller number of external addresses.

NAT wasn't intended to provide security, but it does have some implicit security benefits. A NAT device must be configured with explicit rules to forward inbound connections; this configuration causes inbound connectivity to be implicitly denied. NAT also conceals the internal address space from the external network, ensuring some extra security against internal network mapping.

NAT can offer additional protection, but generally, this isn't its intended purpose. Depending on the implementation, NAT devices might allow attacks that establish internal connections, spoof internal addresses, or leak addresses on the private network. Therefore, NAT shouldn't be relied on alone; it should be viewed as a supplementary measure.

Virtual Private Networks (VPNs)

A **virtual private network (VPN)** provides a virtual network interface connected to a remote network over an encrypted tunnel. This approach has become popular and is quickly replacing dial-in business connections and leased lines. The advantage of a VPN is that it presents an interface that's almost identical to that of a directly connected user, which makes it convenient for end users and network administrators.

The main disadvantage of a VPN is that typically, the client system is outside of the network administrators' physical control, which creates the potential for a much larger attack surface than a normal internal system does. VPN segments need to be monitored more closely, and administrators must enforce additional client precautions. These precautions usually include denying VPN clients access to their local network (split tunneling) while connected and restricting access to certain internal resources over the VPN.

Network IDSs/IPSs

Network IDSs and IPSs are devices that attempt to identify malicious network traffic and potentially terminate or deny connectivity based on detected hostile activity. The first systems were primarily signature-based engines that looked for

specific traffic associated with known attacks. Newer systems attempt to identify and alert administrators to anomalous traffic patterns in addition to known hostile patterns.

There's quite a bit of literature and debate on the proper approach to IDS and IPS deployment and configuration. The details are specific to the network environment. However, the best generally accepted practices require segmenting the network first to isolate different functional areas and points of highest risk. IDS sensors are then deployed to take advantage of segmentation in identifying potential attacks or compromises.

Summary

Application security extends beyond the code to encompass the operational environment and mode in which applications function. In this chapter, you have looked at external system details that affect how secure an application is in a deployment environment. When conducting audits on an application, you need to consider the target deployment environment (if one is available) and the application's default configuration parameters. Unsafe or unnecessary exposure of the application can lead to vulnerabilities that are entirely independent of the program code.

This page intentionally left blank

Chapter 4

Application Review Process

"Ah, my ridiculously circuitous plan is one quarter complete!"

Robot Devil, *Futurama*

Introduction

You no doubt purchased this book with the expectation of delving into the technical details of application security vulnerabilities, but first you need to understand the process of application review and its logistical and administrative details. After all, technical prowess doesn't matter if a review is structured so poorly that it neglects the important application attack surface and vulnerable code paths. Having some degree of structured process in planning and carrying out an application assessment is essential. Of course, your review may have some unique requirements, but this chapter gives you a framework and tools you can adapt to your own process. By incorporating these elements, you should be able to get the best results for the time you invest in any application review.

Overview of the Application Review Process

Conducting an application security review can be a daunting task; you're presented with a piece of software you aren't familiar with and are expected to quickly reach a zenlike communion with it to extract its deepest secrets. You must strike a balance in your approach so that you uncover design, logic, operational, and implementation flaws, all of which can be difficult to find. Of course, you will rarely have enough time to review every line of an application. So you need understand how to focus your efforts and maintain good coverage of the most security-relevant code.

Rationale

To be successful, any process you adopt must be pragmatic, flexible, and results driven. A rigid methodology that provides a reproducible detailed step-by-step procedure is definitely appealing, especially for people trying to manage code reviews or train qualified professionals. For a number of reasons, however, such a rigid approach isn't realistic. It's borne out of a fundamental misunderstanding of code review because it overlooks two simple truths. The first is that *code review is a fundamentally creative process.*

It might seem as though this point couldn't possibly be true because reading other people's code doesn't seem particularly creative. However, to find vulnerabilities in applications, you must put yourself in the developer's shoes. You also need to see the unexpressed possibilities in the code and constantly brainstorm for ways that unexpected things might happen.

The second truth is that *code review is a skill.* Many people assume that code review is strictly a knowledge problem. From this perspective, the key to effective code review is compiling the best possible list of all things that could go wrong. This list is certainly an important aspect of code review, but you must also appreciate the considerable skill component. Your brain has to be able to read code in a way that you can infer the developer's intentions yet hypothesize ways to create situations the developer didn't anticipate.

Furthermore, you have to be proficient and flexible with programming languages so that you can feel at home quickly in someone else's application. This kind of aptitude takes years to develop fully, much like learning a foreign language or playing a musical instrument. There's considerable overlap with related skills, such as programming, and other forms of systems security analysis, but this aptitude has unique elements as well. So it's simply unrealistic to expect even a seasoned developer to jump in and be a capable auditor.

Accepting these truths, having a process is still quite valuable, as it makes you more effective. There's a lot to be done in a typical security review, and it's easy to overlook tasks when you're under a time crunch. A process gives your review structure,

which helps you prioritize your work and maintain a consistent level of thoroughness in your analysis. It also makes your assessments approachable from a business perspective, which is critical when you need to integrate your work with timelines and consulting or development teams.

Process Outline

The review process described in this chapter is open ended, and you can adapt it as needed for your own requirements. This discussion should arm you with the tools and knowledge you need to do a formal review, but it's left flexible enough to handle real-world application assessments. This application review process is divided into four basic phases:

1. *Preassessment*—This phase includes planning and scoping an application review, as well as collecting initial information and documentation.
2. *Application review*—This phase is the primary phase of the assessment. It can include an initial design review of some form, and then proceed to a review of the application code, augmented with live testing, if appropriate. The review isn't rigidly structured into distinct design, logic, implementation, and operational review phases. Instead, these phases are simultaneous objectives reached by using several strategies. The reason for this approach is simply that the assessment team learns a great deal about the application over the course of the assessment.
3. *Documentation and analysis*—This phase involves collecting and documenting the results of the review as well as helping others evaluate the meaning of the results by conducting risk analysis and suggesting remediation methods and their estimated costs.
4. *Remediation support*—This phase is a follow-up activity to assist those who have to act based on your findings. It includes working with developers and evaluating their fixes or possibly assisting in reporting any findings to a third party.

This process is intended to apply to reviews that occur with some form of schedule, perhaps as part of a consulting engagement, or reviews of an in-house application by developers or security architects. However, it should be easy to apply to more free-form projects, such as an open-ended, ongoing review of an in-house application or self-directed vulnerability research.

Preassessment

Before you perform the actual review, you need to help scope and plan the assessment. This process involves gathering key pieces of information that assist you in

later phases of your review. By gathering as much information as you can before starting the assessment, you can construct a better plan of attack and achieve more thorough coverage.

Scoping

When tasked with an application security review, first you need to ask what your goal is. This question might seem simple, but numerous answers are possible. Generally, a vulnerability researcher's goal is to find the most significant vulnerability in the shortest time. In contrast, an application security consultant is usually concerned with getting the best application coverage the project's budget allows. Finally, a developer or security architect might have a more generous schedule when conducting internal reviews and use that time to be as thorough as possible.

The goal of a review might also be heavily colored by business concerns or less tangible factors, such as company image. A company certainly isn't inclined to devote extensive time to a product that's close to or even past its end of life (EOL). However, a review might be required to meet regulatory concerns. That same company might also want a thorough review of its newest flagship financial management application.

When businesses commit to more thorough reviews, often you find that their interests aren't what you expect. A business is sometimes more concerned with easy-to-detect issues, regardless of their severity. Their goal is more to avoid the negative stigma of a published security issue than to address the ultimate technical security of their product or service. So you aren't meeting your client's (or employer's) needs if you spend all your time on complex issues and miss the low-risk but obvious ones. Focusing on low-risk issues seems like blasphemy to most technical security people, but it's often a reasonable business decision. For example, assume you're performing a source-code-based assessment on a bank's Web-facing account management application. What is the likelihood of someone blindly finding a subtle authentication bypass that you found only by tracing through the source code carefully? In contrast, think of how easily an attacker can find cross-site scripting vulnerabilities—just with normal user access. So which issue do you think is more likely to be identified and leveraged by a third party? The obvious answer is cross-site scripting vulnerabilities, but that's not what many auditors go after because they want to focus on the more interesting vulnerabilities.

That's not to say you should ignore complex issues and just get the easy stuff. After all, that advice would make this book quite short. However, you need to understand the goals of your review clearly. You also need to have an appreciation for what you can reasonably accomplish in a given timeframe and what confidence you can have in your results. These details are influenced by two major factors: the type of access you have to the application and the time you have available for review.

Application Access

Application access is divided into the five categories listed in Table 4-1. These distinctions are not, of course, absolute. There are always minor variations, such as limited source access or inconsistencies between test environments and deployment environments. However, these distinctions work well enough to cover most possibilities.

Table 4-1

Categories of Application Access

Category	Description
Source only	Only the source code has been supplied, with no build environment or application binaries. You might be able to build a working binary with some effort, although some required components typically aren't available. As a result, the review is generally done using only static analysis. This type of access is common for contracted application reviews, when the client can provide source but not a functional build or testing environment.
Binary only	Application binaries have been supplied, but no source code is provided. The application review focuses on live analysis and reverse engineering. This type of access is common when performing vulnerability research on closed-source commercial software.
Both source and binary access	Both a source tree and access to a working application build are available. This type of access provides the most efficient review possible. It's most common for in-house application assessments, although security- and cost- conscious clients provide this access for contracted reviews, too.
Checked build	You have an application binary and no source code, but the application binary has additional debugging information. This approach is often taken for contracted code reviews when a client is unwilling to provide source but does want to expedite the review process somewhat.
Strict black box	No direct access to the application source or binary is available. Only external, blind testing techniques, such as black box and fuzz- testing, are possible with this type of access. It's common when assessing Web applications (discussed more in Chapter 17, "Web Applications").

This book focuses primarily on source-code-based application review. Although the techniques discussed in this chapter can be applied to other types of reviews, more information is generally better. The ideal assessment environment includes source-based analysis augmented with access to functioning binaries and a live QA environment (if appropriate). This environment offers the widest range of assessment possibilities and results in the most time-effective review. The remaining types of access in Table 4-1 are all viable techniques, but they generally

require more time for the same degree of thoroughness or have an upper limit on the degree of thoroughness you can reasonably hope to achieve.

Timelines

In addition to application access, you need to determine how much time can be allotted to a review. The timeline is usually the most flexible part of a review, so it's a good way to adjust the thoroughness. The most commonly used measure of application size is thousands of lines of code (KLOC). It's not an ideal way to measure an application's complexity and size, but it's a reasonable metric for general use. A good reviewer ranges between 100 to 1,000 lines of code an hour, depending on experience and details of the code. The best way to establish an effective baseline for yourself is to keep track of how much time you spend reviewing different components and get a feel for your own pacing.

Code type and quality have a big impact on your review speed. Languages such as C/C++ generally require close examination of low-level details because of the subtle nature of many flaws. Memory-safe languages, such as Java, address some of these issues, but they might introduce higher-level complexity in the form of expansive class hierarchies and excessive layering of interfaces. Meanwhile, the quality of internal documentation and comments is a language-independent factor that can seriously affect your review pacing. For this reason, you should look at some samples of the application code before you attempt to estimate for your pace for a specific codebase.

Overall code size affects the pace at which you can effectively review an application. For instance, reviewing a 100KLOC application doesn't usually take twice as much time as a 50KLOC application. The reason is that the first 50KLOC give you a feel for the code, allow you to establish common vulnerability patterns, and let you pick up on developer idioms. This familiarity enables you to review the remainder of the application more efficiently. So be sure to account for these economies of scale when determining your timelines.

In the end, balancing coverage with cost is usually the ultimate factor in determining your timeline. In a perfect world, every application should be reviewed as thoroughly as possible, but this goal is rarely feasible in practice. Time and budgetary constraints force you to limit the components you can review and the depth of coverage you can devote to each component. Therefore, you need to exercise considerable judgment in determining where to focus your efforts.

Information Collection

The first step in reviewing an application is learning about the application's purpose and function. The discussion of threat modeling in Chapter 2 included a number of sources for information collection. This component of your review should encapsulate that portion of the threat model. To recap, you should focus on collecting information from these sources:

- Developer interviews
- Developer documentation
- Standards documentation
- Source profiling
- System profiling

Application Review

People's natural inclination when approaching code review is to try to structure it like a waterfall-style development process. This means starting with a structured design review phase and adhering to a formal process, including DFDs and attack trees. This type of approach should give you all the information you need to plan and perform an effective targeted review. However, it doesn't necessarily result in the most time-effective identification of high and intermediate level design and logic vulnerabilities, as it overlooks a simple fact about application reviews: *The time at which you know the least about an application is the beginning of the review.*

This statement seems obvious, but people often underestimate how much one learns over the course of a review; it can be a night and day difference. When you first sit down with the code, you often can't see the forest for the trees. You don't know where anything is, and you don't know how to begin. By the end of a review, the application can seem almost like a familiar friend. You probably have a feel for the developers' personalities and can identify where the code suffers from neglect because everyone is afraid to touch it. You know who just read a book on design patterns and decided to build the world's most amazing flexible aspect-oriented turbo-logging engine—and you have a good idea which developer was smart enough to trick that guy into working on a logging engine.

The point is that the time you're best qualified to find more abstract design and logic vulnerabilities is toward the end of the review, when you have a detailed knowledge of the application's workings. A reasonable process for code review should capitalize on this observation.

A design review is exceptional for starting the process, prioritizing how the review is performed, and breaking up the work among a review team. However, it's far from a security panacea. You'll regularly encounter situations, such as the ones in the following list, where you must skip the initial design review or throw out the threat model because it doesn't apply to the implementation:

- You might not have any design documentation to review. Unfortunately, this happens all the time.

- The design documentation might be so outdated that it's useless. Unfortunately, this happens all the time, too—particularly if the design couldn't be reasonably implemented or simply failed to be updated with the ongoing application development.
- There might be a third party who doesn't want to give you access to design information for one reason or another (usually involving lawyers).
- The developers might not be available for various reasons. They might even consider you the enemy.
- Clients don't want to pay for a design review. This isn't a surprise, as clients rarely *want* to pay for anything. It's more or less up to you as a professional to make sure they get the best bang for their buck—in spite of themselves. Time is expensive in consulting and development environments, so you'd better be confident that what you're doing is the best use of your time.

Accepting all the preceding points, performing a design review and threat model first, whenever realistically possible, is still encouraged. If done properly, it can make the whole assessment go more smoothly.

Avoid Drowning

This process has been structured based on extensive experience in performing code reviews. Experienced auditors (your authors in particular) have spent years experimenting with different methodologies and techniques, and some have worked out better than others. However, the most important thing learned from that experience is that it's best to use several techniques and switch between them periodically for the following reasons:

- You can concentrate intensely for only a limited time.
- Different vulnerabilities are easier to find from different perspectives.
- Variety helps you maintain discipline and motivation.
- Different people think in different ways.

Iterative Process

The method for performing the review is a simple, iterative process. It's intended to be used two or three times over the course of a work day. Generally, this method works well because you can switch to a less taxing auditing activity when you start to feel as though you're losing focus. Of course, your work day, constitution, and preferred schedule might prompt you to adapt the process further, but this method should be a reasonable starting point.