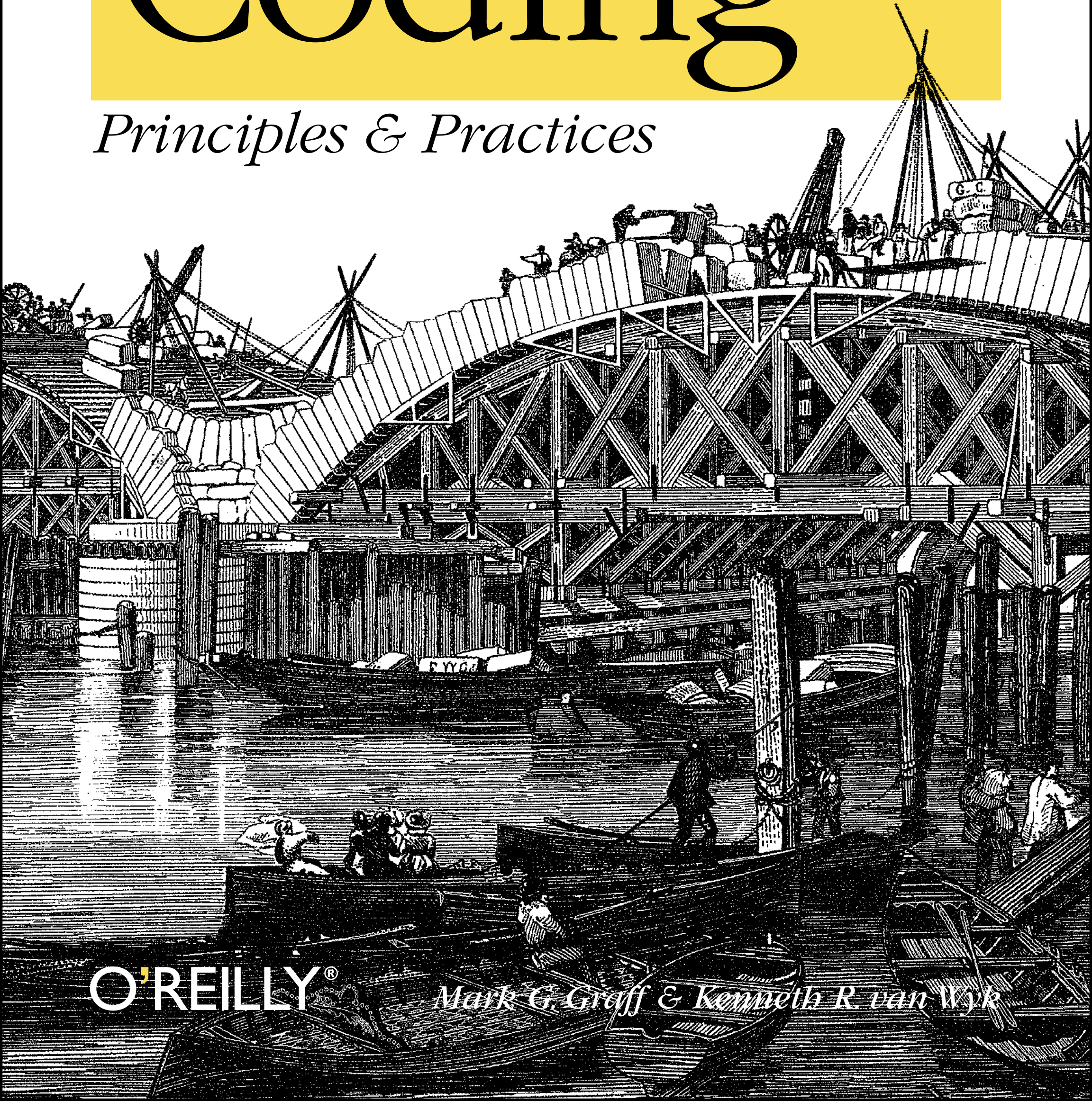
Designing & Implementing Secure Applications

Secure Coding

Principles & Practices



O'REILLY®

Mark G. Graff & Kenneth R. van Wyk

Secure Coding

Principles and Practices

Other Java™ resources from O'Reilly

Related titles

- Enterprise JavaBeans™
- Java™ & XML
- Java™ Cookbook
- Java™ Enterprise in a Nutshell
- Java™ I/O
- Java™ in a Nutshell
- Java™ Performance Tuning
- Java™ Programming with Oracle SQLJ
- Java™ Security
- JavaServer™ Pages
- Java™ Swing
- Learning Java™

Java Books Resource Center

java.oreilly.com is a complete catalog of O'Reilly's books on Java and related technologies, including sample chapters and code examples.

OnJava.com is a one-stop resource for enterprise Java developers, featuring news, code recipes, interviews, weblogs, and more.

Conferences

O'Reilly & Associates bring diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

O'REILLY NETWORK Safari® Bookshelf™

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

Secure Coding

Principles and Practices

Mark G. Graff and Kenneth R. van Wyk

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 1
No Straight Thing

Out of the crooked timber of humanity,
no straight thing can ever be made.
—Immanuel Kant

In late 1996 there were approximately 14,000,000 computers connected to the Internet. Nearly all of them relied on the Transmission Control Protocol (TCP), one of the fundamental rule sets underlying communication between computers, and the one used for most common services on the Internet. And although it was known to have security weaknesses, the protocol had been doing its work quietly for nearly two decades without a major attack against it.

But on September 1 of that year, the online magazine *Phrack* published the source code for a network attack tool that exploited the trusting way the protocol handled connection requests (see the sidebar "A Fractured Dialogue"). Suddenly, the majority of those 14,000,000 computers were now vulnerable to being taken offline—in some cases, crashed—at the whim of any malcontent capable of compiling the attack program.

It was a time when new vulnerabilities were being disclosed daily, and the article at first went unnoticed by most security professionals. It was, however, read carefully in some quarters. Within days, an ISP in New York City named Panix was repeatedly savaged using the technique. Day after day, bombarded by tens of thousands of false connection requests—known as a *SYN flood*, after the protocol element that was misapplied—Panix was helpless to service its paying customers. The security community took notice and began to mobilize; but before experts could come up with an effective defense, the attacks spread. The Internet Chess Club was clobbered several times in September. Scattered attacks troubled several more sites, mostly media outlets, in October. In November, on election night, the *New York Times* web site was disabled, and the resulting publicity opened the

A Fractured Dialogue

What happens when you call someone on the phone and they hang up before you do—and you decide not to hang up yourself? Until a few years ago (in the U.S., at least), it was possible to tie up the other person's telephone line for a long time this way.

Today we might call this trick a *denial of service attack*. It's an example of what can happen when one party to a conversation decides not to play by the rules. In the network world, a set of such rules is called a *protocol*. And the network attack known as a *TCP SYN flood* is an example of what can happen when an attacker controlling one side of a computer dialogue deliberately violates the protocol.

The Transmission Control Protocol (TCP) is used many billions of times a day on the Internet. When email is exchanged, for example, or when someone visits a web site, the dialogue between the sending and receiving computers is conducted according to these rules. Suppose that computer A wants to initiate a connection with computer B. Computer A offers to "synchronize" with computer B by sending a set of ones and zeros that fit a special pattern. One feature of this pattern is that a particular bit (the SYN flag) is set. Computer B agrees to the exchange by replying in an equally specific bit pattern, setting both the SYN flag and the ACK ("acknowledge") flag. When computer A confirms the connection by replying with its own ACK, the TCP session is open, and the email or other information begins to flow. (Figure 1-1 shows this exchange.)

As early as the mid-1980s, researchers realized that if the initiating computer never completed the connection by sending that final acknowledgment, the second computer would be in a situation similar to that of the hapless telephone user whose caller never hung up. To be sure, in each case the computer programs implementing the dialogue can break the connection after a suitable period of time, freeing up the telephone line or network connection. But suppose that an attacker writes software capable of sending dozens or hundreds of false connections requests per second. Wouldn't the receiving computer be overwhelmed, keeping track of all those half-open connections? That turns out to be the foundation for a TCP SYN flood attack; and in 1996, it was deadly.[a]

[a] The version of the attack code that posed the biggest problem had an additional "feature": it produced "SYN packets" that included false sender addresses, making it much harder for the receiving computers to deal with the attack without shutting out legitimate connection requests.

floodgates. By the time an effective defense had been devised and widely deployed some weeks later, hundreds of sites around the world had been victimized. Tens of thousands more were affected, as experts and laypersons alike struggled to cope with the practical impact of this first widespread denial of service attack.

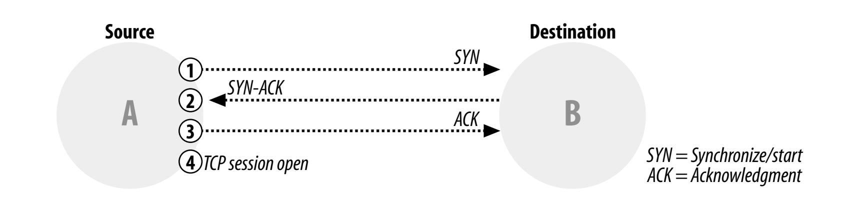


Figure 1-1. How a normal TCP network session works

Why are we telling this story? True, the attack makes for interesting reading. And true, the attackers deserve our contempt. But there are, sadly, many other Internet attacks that we might describe. Why this one?

It's partly that both of us were heavily involved in the worldwide response of the security community to the vulnerability and resulting attacks. Mark worked at Sun Microsystems then and was integrally involved in Sun's technical response to correcting their networking software. Ken worked the problem from the incident response side—he was chairman of FIRST (the international Forum of Incident Response and Security Teams) at the time.

More importantly, the TCP SYN flood attack exemplifies a multitude of different secure coding issues that we are deeply concerned about. As we wind our way through this book, we'll use this particular example to illustrate many points, ranging from the technical to the psychological and procedural.

We'll return to this story, too, when we discuss design flaws, errors in implementation, and various issues of secure program operation, because it so clearly represents failure during every phase of the software development lifecycle. If the architecture of the TCP protocol had been more defensively oriented in the first place,* the attack would never have been possible. If the request-processing code in the affected operating systems had been designed and implemented with multiple layers of defense, the attacks wouldn't have brought down the whole house of cards. If the software had been tested and deployed properly, someone would have noticed the problem before it affected thousands of Internet sites and cost millions of dollars in lost time, data, and opportunity. This "lifecycle" way of looking at security is one we'll come back to again and again throughout this book.

We'll mention several other attacks in this book, too. But our focus won't be on the details of the attacks or the attackers. We are concerned mainly with *why* these attacks have been successful. We'll explore the vulnerabilities of the code and the reasons for those vulnerabilities. Then we'll turn the tables

* We are not suggesting that the original architects of the protocol erred. Their architecture was so successful that the Internet it made possible outgrew the security provisions of that architecture.

and make our best case for how to build secure applications from the inside out. We'll ask how we can do better at all stages.

More simply, we'll try to understand why good people write bad code. Smart, motivated people have been turning out new versions of the same vulnerabilities for decades. Can "no straight thing" ever be made?

The Vulnerability Cycle

Let's consider for a moment an all-too-common sequence of events in today's security world. (Figure 1-2 illustrates it graphically.)

1. Someone uncovers and discloses a new vulnerability in a piece of software.
2. Bad guys quickly analyze the information and use the vulnerability to launch attacks against systems or networks.
3. Simultaneously, good guys (we'll include security folks who work for the vendor) start looking for a fix. They rally software development engineers in their respective organizations to analyze the vulnerability, develop a fix, test the fix in a controlled environment, and release the fix to the community of users who rely on the software.
4. If the vulnerability is serious, or the attacks are dramatic, the various media make sure that the public knows that a new battle is underway. The software developers at the organization that produced the product (and the vulnerability!) are deluged with phone calls from the media, wanting to find out what is going on.
5. Lots of folks get very worried. Pundits, cranks, finger-pointers, and copycats do their thing.
6. If a knee-jerk countermeasure is available and might do some good, we'll see a lot of it. (For example, CIOs may direct that all email coming into an enterprise be shut off.) More often than not, this type of countermeasure results in numerous and costly business interruptions at companies that rely on the software for conducting their business operations.
7. When a patch is ready, technically oriented folks who pay close attention to such matters obtain, test, and apply the patch. Everyday system administrators and ordinary business folks may get the word and follow through as well. Perhaps, for a lucky few, the patch will be installed as part of an automated update feature. But inevitably, many affected systems and networks will never be patched during the lifetime of the vulnerability—or will only receive the patch as part of a major version upgrade.

8. Security technicians, their attention focused, examine related utilities and code fragments (as well as the new patch itself!) for similar vulnerabilities. At this point, the cycle can repeat.
9. Weeks or months go by, and a piece of malicious software is released on the Internet. This software automates the exploitation of the vulnerability on unpatched systems, spreading without control across a large number of sites. Although many sites have patched their systems, many have not, and the resulting panic once again causes a great deal of business interruption across the Internet.

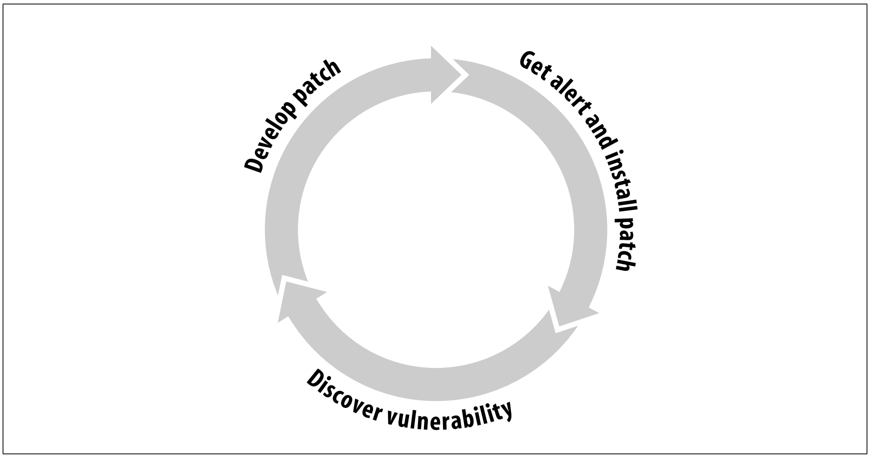


Figure 1-2. The vulnerability/patch/alarm cycle

What's so bad about this scenario? Let's consider some of the effects.

Many companies (some big, some small) just can't keep up with today's cascade of patches. To get a sense of the scope of the problem, let's assume that the Internet and its critical services run on 100 key applications. We estimate (conservatively, in our opinions) that there are 100 or so vulnerabilities per application system. If that guess is in the ballpark, that's about 10,000 security holes for hackers to exploit, just in key applications!

Here's a rough calculation relating to operating systems. Noted "secure coder" Wietse Venema estimates that there is roughly one security bug per 1000 lines in his source code. Given that desktop operating systems such as Linux or Windows represent some 100 million lines of code, this translates into hundreds of thousands of potential security bugs. According to CERT statistics, collectively we will probably discover roughly 5000 bugs in 2003. At this rate it could take 20 years per operating system to find all the security bugs. Fixing them will take a little longer; our experience is that, using today's common practices, 10% to 15% of all security patches themselves introduce security vulnerabilities! (It is only fair to point out here that these numbers are anything but scientific, but we believe they're not far from correct and the underlying point remains the same.)

Applying patches over and over—as though system administrators had nothing else to do—is never going to give us a secure Internet-based infrastructure. As society's reliance on Internet services grows, it's only a matter of time before catastrophe strikes. The software so many of us depend on every day is frighteningly open to attack.

What is an Attack?

In a general sense, an *attack* on a system is any maliciously intended act against a system or a population of systems. There are two very important concepts in this definition that are worth pointing out. First, we only say that the act is performed with malicious intent, without specifying any goals or objectives. Second, some attacks are directed at a particular system, while others have no particular target or victim.* Let's look at these concepts and terms one by one:

Goals
: The immediate goal of an attack can vary considerably. Most often, though, an attack goal is to damage or otherwise hurt the target, which may include stealing money, services, and so on.

Subgoals
: Achieving one or more of the goals above may require first reaching a subgoal, such as being granted elevated privileges or authorizations on the system.

Activities
: The activities that an attacker engages in are the things that he does that could help him achieve one or more of his subgoals. These could include using stolen login credentials (e.g., username and password); masquerading as a different computer, user, or device; flooding a network with malformed packets; and so on.

Events
: The activities mentioned above may result in attack events—improper access could be granted, request processing could be suspended, storage space could be exhausted, or a system or program could be halted.

* Note also that in this definition we don't limit ourselves to events that take place in an electronic realm. An attack against an application could well involve a physical act, such as carrying a hard drive out of a data center in a briefcase. For the most part, though, we'll concentrate on electronic attacks in this book.

Consequences

A further concept, often confused with an attack event, is the business consequence. By this term we mean the direct result of the events, such as financial balance sheets being incorrect, or a computer system being unavailable for a business process.

Impacts

Lastly, the impact of the attack is the business effect. This could include the tarnishing of a company's reputation, lost revenue, and other effects.

The distinction between the attack event and its business consequence is an important one. The business consequence of an attack depends on the business purpose of the affected system, not on the specific attack actions or events. A direct consequence, for example, might be an inability to process a business transaction, resulting in an impact of loss of revenue. An indirect impact might be the tarnishing of the reputation of the business owner, resulting in an erosion of customer confidence. Figure 1-3 illustrates an example attack, showing the goals, subgoals, and activities of the attacker, along with the events, consequences, and impacts from the perspective of the target enterprise.

We've trotted out this terminology because we've found that it's critical to think clearly and precisely about attacks if we are to prevent them. Does it surprise you to hear that the potential business impact of an attack may be relevant to its prevention? It is. Knowing what is at stake is an essential part of making good design decisions about which defense mechanisms you will use.

How Would You Attack?

How do attackers attack systems? Part of the *how* depends on the *why*. Some want to probe, but do no harm. Others are out to steal. Some seek to embarrass. A few want only to destroy or win bragging rights with their cronies. While we can't anticipate all possible motivations, we will try to think with you about how someone only moderately clever might approach the problem of compromising the security of your application.

Consider a safecracker. If he is a professional, he probably owns specialized safecracking tools (a stethoscope—we are told—comes in handy). He probably also has a thorough knowledge of each target vault's construction and operation, and access to useful technical documentation. He uses that knowledge to his advantage in manipulating the safe's combination knob, its internal tumblers, and so on, until he manages (or fails) to open the safe.

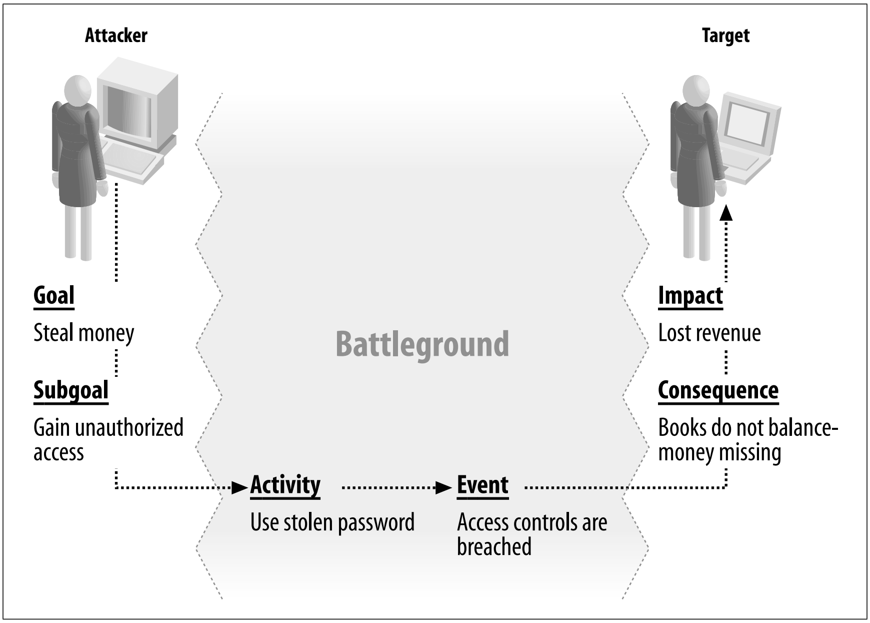


Figure 1-3. Attack activities, events, goals, and business consequences

In an analogous attack on an application system, the miscreant arms himself with knowledge of a system (and tools to automate the application of the knowledge) and attempts to crack the target system.

Ah, but there are so many ways into a safe! A bank robber who doesn't have the finesse of a safecracker can still put on a ski mask and enter the bank during business hours with a gun. If we were planning such an attack, we might masquerade as a pair of armored car security officers and enter the vault with the full (albeit unwitting) assistance of the bank staff. Bribery appeals to us—hypothetically, of course—as well. How about you? Would you blast your way in?

There have been many good studies about the motivations and mind-sets of the kinds of people and personalities who are likely to attack your software. That's a book in itself, and in the Appendix, we point you to a few good ones. In this chapter, we'll simply encourage you to keep in mind the many facets of software and of the minds of your attackers. Once you have begun to ask *what* can happen, and *how* (and maybe *why*), we believe you're on your way to enforcing application security. In the case studies at the end of this chapter, we'll provide examples of constructive worrying we encourage you to do, as well as examples of what could happen if you don't worry enough.

Attacks and Defenses

In the following sections, we'll list quite a few types of attacks that your applications may have to withstand. We've divided the discussion into three categories, according to which stage of development the vulnerability relates:

Architecture/design
While you are thinking about the application

Implementation
While you are writing the application

Operations
After the application is in production

The attacks, of course, will usually—not always—take place while the program is running. In each case, we'll try to make a clear distinction.

At the end of each description, we'll discuss briefly how an application developer might approach defending against the attack. We'll develop these ideas in greater depth in subsequent chapters, as we make our case that application security is an essential element at every stage of development.

> In these sections we describe only a very few of the many, many ways that security can be breached. We refer you again to the Appendix for pointers to more complete discussions, as well as pointers to formal attack taxonomies.

Architecture/design-level attacks

As a general rule, the hardest vulnerabilities to fix are those resulting from architectural or design decisions. You may be surprised at how many of the vulnerabilities you have heard of we ascribe to errors at "pure think" time.

At the end of this section we list two types of attacks, *session hijacking* and *session killing*, that are unusually difficult to defend against from within an application. What's the point of mentioning them? As we argue in Chapter 3, the fact that you as a developer may not be able to institute adequate defenses against an attack does not relieve you of responsibility for thinking about, planning for, and considering how to minimize the impact of such an occurrence.

It's worth mentioning that architectural and design-level attacks are not always based on a *mistake* per se. For example, you may have used the *telnet* program to move from one computer to another. It does well what it was designed to do. The fact that its design causes your username and password to be sent along the cables between computers, unencrypted, is not a "flaw" in and of itself. Its design does make it an unattractive choice for using in a hostile environment, however. (We use *ssh* instead.)

The following are the main attacks we've observed at the architecture/design level:

Man-in-the-middle attack

A man-in-the-middle (or *eavesdropping*) attack occurs when an attacker intercepts a network transmission between two hosts, then masquerades as one of the parties involved in the transaction, perhaps inserting additional directives into the dialogue.

Defense: Make extensive use of encryption—in particular, strong cryptographic authentication. In addition, use session checksums and shared secrets such as cookies. (You might, for example, use *ssh* instead of *telnet*, and encrypt your files using utilities such as *PGP* or *Entrust*.)

Race condition attack

Certain operations common to application software are, from the computer's point of view, comprised of discrete steps (though we may think of them as unitary). One example is checking whether a file contains safe shell (or "batch") commands and then (if it does) telling the host system to execute it. Sometimes, the time required to complete these separate steps opens a window during which an attacker can compromise security. In this example, there may be a very brief window of opportunity for an attacker to substitute a new file (containing attack code) for the previously checked file. Such a substitution can trick an application into running software of the attacker's choosing. Even if the resulting window of opportunity is too short for a human to reliably exploit it, a program might well be able to repeatedly test and then execute the attacker's program at just the right moment in time. Because the result often depends upon the order in which two or more parallel processes complete, this problem is known as a "race" condition.*

Defense: Understand the difference between *atomic* (indivisible) and *non-atomic* operations, and avoid the latter unless you are sure there are no security implications. (Sample actions that do have security implications include opening a file, invoking a subprogram, checking a password, and verifying a username.) If you are not sure whether an operation is atomic, assume that it is not—that is, that the operating system may execute it in two or more interruptible steps.

Replay attack

If an attacker can capture or obtain a record of an entire transaction between, say, a client program and a server, it might be possible to

* The example would also be described in the technical literature as a "late binding" problem.

"replay" part of the conversation for subversive purposes. Impersonating either the client or the server could have serious security implications.

Defense: Same as for the man-in-the-middle attack; in addition, consider introducing into any dialog between software elements some element (e.g., a sequence identifier) that must differ from session to session, so that a byte-for-byte replay will fail.*

Sniffer attack

A program that silently records all traffic sent over a local area network is called a *sniffer*. Sniffers are sometimes legitimate diagnostic tools, but they are also useful to attackers who wish to record usernames and passwords transmitted in the clear along a subnet.

Defense: This attack is best addressed at the network level, where its impact can be diminished (but not removed) by careful configuration and the use of "switched" network routers. Still, as an application writer, you can render sniffers fairly harmless by making maximal effective use of encryption.

Session hijacking attack

By exploiting weaknesses in the TCP/IP protocol suite, an attacker inside the network might be able to *hijack* or take over an already established connection. Many tools have been written and distributed on the Internet to implement this rather sophisticated technical attack.

Defense: This network-level attack is quite difficult to defend against from within application software. Encryption, of course, is a help (although a discussion of its limitations is beyond our scope here). And some operational procedures can help detect a hijacking after the fact, if careful logging provides enough information about the session.

Session killing attack

Legitimate TCP/IP sessions can be terminated when either of the communicating parties sends along a *TCP reset* packet. Unfortunately, an attacker inside the network might be able to forge the originating address on such a packet, prematurely resetting the connection. Such an attack could be used either to disrupt communications or, potentially, to assist in an attempt to take over part of a transaction (see the description of session hijacking above).†

* You might consider, for example, using the (trustworthy) current time as an example of a sequence identifier, such as the Kerberos 5-minute overlap requirement.

† Note, too, that such attacks can be successful even if the attacker is not on the local segment, if the network is not doing any ingress filtering—that is, if there's no check to see if a data packet is really coming from the destination listed inside the packet.

Defense: Like session hijacking attacks, session killing attacks are difficult to defend against from within an application. Unfortunately, we believe that deterrence from within the application is not possible. However, your application may be able to compensate after the fact by either reasserting the connection or reinitializing the interrupted transaction.

Implementation-level attacks

We suspect that the kinds of errors we list in this section are the ones most folks have in mind when we talk about security vulnerabilities. In general, they're easier to understand and fix than design errors. There are many varieties of implementation-level attacks. Here are three common examples:

Buffer overflow attack

Many programming languages (C, for example) allow or encourage programmers to allocate a buffer of fixed length for a character string received from the user, such as a command-line argument. A buffer overflow condition occurs when the application does not perform adequate bounds checking on such a string and accepts more characters than there is room to store in the buffer. In many cases, a sufficiently clever attacker can cause a buffer to be overflowed in such a way that the program will actually execute unauthorized commands or actions.

Defense: Code in a language (such as Java) that rules out buffer overflows by design. Alternatively, avoid reading text strings of indeterminate length into fixed-length buffers unless you can safely read a substring of a specific length that will fit into the buffer.

Back door attack

Many application systems have been compromised as the result of a kind of attack that might be said to take place while the software is being written! You may have read about cases in which a rogue programmer writes special code directly into an application that allows access control to be bypassed later on—for example, by using a "magic" user name. Such special access points are called *back doors*.

Defense: Adopt quality assurance procedures that check all code for back doors.

Parsing error attack

Applications often accept input from remote users without properly checking the input data for malicious content. The parsing, or checking, of the input data for safety is important to block attacks. (Further, industrial-strength parsing of program input with robust error checking can greatly reduce all kinds of security flaws, as well as operational software flaws.)

One famous example of parsing errors involved web servers that did not check for requests with embedded "../" sequences, which could enable the attacker to traverse up out of the allowed directory tree into a part of the filesystem that should have been prohibited.

While parsing input URLs for "../" sequences may sound simple, the developers failed repeatedly at catching all possible variants, such as hexadecimal or Unicode-encoded strings.

Defense: We recommend arranging to employ existing code, written by a specialist, that has been carefully researched, tested, and maintained. If you must write this code yourself, take our advice: it is much safer to check to see if (among other things) every input character appears on a list of "safe" characters than to compare each to a list of "dangerous" characters. (See Chapter 4, for a fuller discussion.)

Operations-level attacks

Most attacks, as we have said, take place after an application has been released. But there is a class of special problems that can arise as a result of decisions made after development, while the software is in production. We will have much more to say about this subject in later chapters. Here is a preview.

Denial-of-service attack

An application system, a host, or even a network can often be rendered unusable to legitimate users via a cascade of service requests, or perhaps a high-frequency onslaught of input. When this happens, the attacker is said to have "denied service" to those legitimate users. In a large-scale denial-of-service attack, the malefactors may make use of previously compromised hosts on the Internet as relay platforms for the assault.

Defense: Plan and allocate resources, and design your software so that your application makes moderate demands on system resources such as disk space or the number of open files. When designing large systems, include a way to monitor resource utilization, and consider giving the system a way to shed excess load. Your software should not just complain and die in the event that resources are exhausted.

Default accounts attack

Many operating systems and application programs are configured, by default, with "standard" usernames and passwords. These default usernames and passwords, such as "guest/guest" or "field/service", offer easy entry to potential attackers who know or can guess the values.

Defense: Remove all such default accounts (or make sure that system and database administrators remove them). Check again after installing new software or new versions of existing software. Installation scripts sometimes reinstall the default accounts!

Password cracking attack

Attackers routinely guess poorly chosen passwords by using special *cracking* programs. The programs use special algorithms and dictionaries of common words and phrases to attempt hundreds or thousands of password guesses. Weak passwords, such as common names, birthdays, or the word "secret" or "system", can be guessed programmatically in a fraction of a second.

Defense: As a user, choose clever passwords. As a programmer, make use of any tools available to require robust passwords.* Better yet, try to avoid the whole problem of reusable passwords at design time (if feasible). There are many alternative methods of authentication, including biometric devices and smart cards.

Why Good People Write Bad Code

Now that we've walked on the dark side, looking at all kinds of things that can go wrong with our software, let's turn our attention back to root causes: why do software flaws occur? Why do good people write bad code?

A great many people believe that vulnerabilities are the spawn of stupid (and probably slothful) programmers. Some adherents to this credo have been customers of ours. Although we have listened respectfully to the arguments for many hours, we disagree.

We believe that, by and large, programmers want to write good software. They surely don't set out with the intention of putting security flaws in their code. Furthermore, because it's possible for a program to satisfy a stringent functional specification and nevertheless bring a vulnerability to life, many (if not most) such flaws have been coded up by people who do their best and are satisfied with (even rewarded for) the result.

What's so hard about writing secure code? Why do vulnerabilities exist at all, let alone persist for decades? Why can't the vendors get it right?

We believe there are three sets of factors that work against secure coding:

* The best passwords are easy to remember and hard to guess. Good password choices might be, for example, obscure words in uncommon languages you know, or letter combinations comprised of the initial (or final) letters of each word in a phrase you can remember. Consider also including punctuation marks and numbers in your passwords.

Technical factors
The underlying complexity of the task itself

Psychological factors
The "mental models," for example, that make it hard for human beings to design and implement secure software

Real-world factors
Economic and other social factors that work against security quality

This is a hard problem. After a close look at our examples, we think you will come to agree that wiping out security vulnerabilities by just doing a better job of coding is a monumental—perhaps hopeless—task. Improved coding is critical to progress, of course. But some vulnerabilities seem to arise without any direct human help at all. We engineers will have to adapt our tools and systems, our methods, and our ways of thinking. Beyond this, our companies, our institutions, and our networked society itself will need to face up to the danger before this scourge can pass away.

Technical Factors

Truly secure software is intrinsically difficult to produce. A true story may help show why.

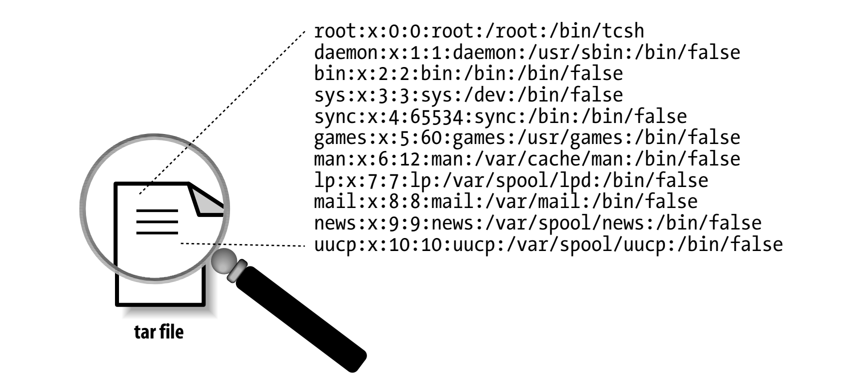
The Sun tarball story

While Mark worked at Sun back in 1993, he received one of those middle-of-the-night phone calls from CERT he used to dread so much. Jim Ellis told him they had received and verified a report that every tarball produced under Solaris 2.0 contained fragments of the */etc/passwd* file.* If this were true, Mark thought, Sun and its customers were in terrible trouble: the password file was a fundamental part of every system's security, the target of an attacker's "capture the flag" fantasy. Was Sun giving it away? Was their software actually shipping out the password file to be deposited on archival backup tapes, FTP and web sites, and countless CD-ROMs?

Jim had passed along a program he had put together to examine *tar* archive files for */etc/passwd* fragments (see Figure 1-4), so it didn't take long for Mark to confirm his report. Soon he was pulling vice presidents out of meetings and mobilizing the troops—pulling the metaphorical red fire alarm handle for all he was worth. What worried him was the possibility that some

* A "tarball" is an archive file produced by the Unix *tar* (Tape Archive) utility. Originally designed to copy blocks of disk storage onto magnetic tape, it's still in worldwide use today, the predominant method of transferring files between Unix systems.

devious, forward-looking mole might have inserted the vulnerability into the Sun code tree, several years earlier, with the intent of reaping the customer's password files much later—after the buggy code had distributed thousands of them around the Internet.



```
root:x:0:0:root:/root:/bin/tcsh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
bin:x:2:2:bin:/bin:/bin/false
sys:x:3:3:sys:/dev:/bin/false
sync:x:4:65534:sync:/bin:/bin/false
games:x:5:60:games:/usr/games:/bin/false
man:x:6:12:man:/var/cache/man:/bin/false
lp:x:7:7:lp:/var/spool/lpd:/bin/false
mail:x:8:8:mail:/var/mail:/bin/false
news:x:9:9:news:/var/spool/news:/bin/false
uucp:x:10:10:uucp:/var/spool/uucp:/bin/false
```

Figure 1-4. The Sun tarball vulnerability

The story has a happy ending. Mark was able to track down the change that introduced the bug and satisfy himself that it was inadvertent. Coincidentally, beginning with this release, the password file was no longer critical to system security: Solaris 2 introduced into Sun's product the idea of the *shadow password file*, so the */etc/passwd* file no longer contained user passwords. He fixed the bug, built a patch, issued a security advisory (Sun Security Bulletin 122, issued 21 October 1993), and breathed a sigh of relief. But Mark has never shaken the concern that such a longitudinal attack may in fact have been launched against some vendor many years ago and is silently doing its work still today.

Let's take a step back and look at some of the technical details of this particular bug. They may help illuminate the more general problems of writing unsecure code.

Material was relayed in 512-byte *blocks* from a disk source to the archive file. A read-a-block/write-a-block cycle was repeated over and over, until the entire source file was saved. However, the buffer to which the disk source block was read was not zeroed first by the programmer before the read. So the part of the block that extended past the end of the file on the last read did not come from the file, but rather from whatever was in the memory buffer before the disk read.

The usual result from such an error would be random junk at the end of the archive file. So why were fragments of the password file being written? It turned out that the buffer to which the disk block was read happened to already contain a part of the user password file—every time, without fail. Why did this happen?

The buffer always held leftover information from the password file because, as part of the read/write cycle, the *tar* program looked up some information about the user running the program. The system call used to look up the user information worked by reading parts of the */etc/passwd* file into memory. The *tar* program obtained memory for this purpose from the system "heap" and released it back to the heap when the check was done. Because the heap manager also did not zero out blocks of memory when it allocated them, any process requesting storage from the heap immediately after that system call was executed would receive a block with parts of the */etc/passwd* file in it. It was a coincidence that *tar* made the system call just before allocating the "read-a-block" buffer.

Why didn't Sun notice this problem years before? In previous versions of the software, the system call relating to the check of usernames happened much earlier. Other allocations and deallocations of the buffer intervened. But when a programmer removed extraneous code while fixing a different bug, the security vulnerability was introduced. That program modification moved the system call and the disk read closer together so that the buffer reuse now compromised system security.

Once all this analysis was done, the fix was simple—from something like this:

```
char *buf = (char *) malloc(BUFSIZ);
```

to something like this:

```
char *buf = (char *) calloc(BUFSIZ, 1);
```

Editing just a few characters (making the code now invoke the "clear allocate" routine, which allocates a buffer and then zeroes it) "fixed" the problem and closed the vulnerability.*

The reason we tell this story in so much detail is to illustrate that critical security vulnerabilities can often result not from coding or design mistakes, but merely from unanticipated interactions between system elements that by themselves are neither unsecure nor badly engineered.

In the next chapter, we'll discuss architectural principles that (if followed) could have rendered this particular bug harmless. Please note, however, that

* While this code "works," it is probably not the best way to fix this problem. In Chapter 3, we'll display some alternatives in the discussion of security in "Performing Code Maintenance."

a program with "harmless" bugs is not really secure. It's more like a person who has a deadly disease under control. We'll discuss this issue in more detail a little later on, when we talk about the effects of system complexity.

Effects of composition

Here is a related effect: application systems are often composed from multiple separate components, each of which may be perfectly secure by itself. However, when components are taken together, they may create a hole that can be exploited. A famous example of this class of problem was the Unix "rlogin -l -froot" bug. It was caused by the composition of an *rlogin* server from one source and a *login* program from another. The problem was that the *login* program accepted preauthenticated logins if passed an argument –f <username>, assuming that the invoking program had done the authentication. The *rlogin* server program, however, did not know about the -f argument, and passed a username of -froot on to the *login* program, expecting it to do the authentication.

Neither program was wrong, exactly; but taken together they allowed any remote attacker to log in as system administrator without authentication. In other fields, the whole may be greater than the sum of the parts; in computer security, the sum of the parts is often a hole.

As a bridge-playing expert that we know observed after a disastrous tournament result, "No one made any mistakes. Only the result was ridiculous."

Other effects of extreme complexity

In addition, spontaneous security failures seem to occur from time to time. Why does this happen? Consider the following explanation, from James Reason's masterful *Human Error*. He draws a surprising analogy:

> There appear to be similarities between latent failures in complex technological systems and resident pathogens in the human body.
>
> The resident pathogen metaphor emphasizes the significance of casual factors present in the system before an accident sequence actually begins. All man-made systems contain potentially destructive agencies, like the pathogens within the human body. At one time, each complex system will have within it a certain number of latent failures, whose effects are not immediately apparent but that can serve both to promote unsafe acts and to weaken its defense mechanisms. For the most part, they are tolerated, detected and corrected, or kept in check by protective measures (the auto-immune system). But every now and again, a set of external circumstances—called here local triggers—arises that combines with these resident pathogens in subtle and often unlikely ways to thwart the system's defenses and bring about its catastrophic breakdown.

We believe that it's in the very complexity of the computer systems we engineers work with that the seeds of security failure are sown. It's not just that an algorithm too complex for the skill of the eventual coder will engender bugs. Perfect reliability—in this context, a complex system with no security vulnerabilities—may not in fact be achievable. (We'll leave that to the academics.) We certainly have never seen one; and between the two of us, we have studied hundreds of complex software systems.

Ah, but the situation gets worse. Do you know any mistake-proof engineers? We'll look at the human side of failure in the next section.

Psychological Factors

Programmers are people, a fact that many security analysts seem to overlook when examining the causes of vulnerabilities. Oh, everybody agrees that "to err is human," and it's common to lament the fallibility of software engineers. But we've seen little in the way of careful thinking about the influence human psychology has on the frequency and nature of security vulnerabilities.*

Risk assessment problems

Programming is a difficult and frustrating activity. When we or our colleagues perform a security analysis on software, we've noticed that (unless we take special precautions to the contrary) the kinds of errors we find are the ones we're looking for, the ones we understand, and the ones we understand how to fix. This factor (the tarball vulnerability we described earlier illustrates it) is one of the best arguments we know for automated security tests that require one to run and respond to a whole range of errors, both familiar and unfamiliar.

Here's another factor. When we ourselves do design work, we find that we are uncomfortable thinking about some of our colleagues/coworkers/customers/fellow human beings as crooks. Yet, that is exactly the mindset you as a developer need to adopt. Never trust anyone until his trustworthiness has been verified by an acceptably trustworthy source—that's the rule. Most of us find that to be an uncomfortable mental posture; and that's a real complication.

* If this subject interests you, we recommend that you follow up with the best text we know, *Psychology of Computer Programming* by Gerald Weinberg. It's a remarkable book, which has just been reprinted for its 25th anniversary. There are a few other authors who have made a good start on the study of human error as well. See the Appendix for details.

Another difficulty is that human beings tend to be bad at particular kinds of risk assessment—for example, determining how hard you need to try to protect passwords against snooping on your network. Your judgments are going to be made using a brain design that seems to have been optimized for cracking skulls together on an African savannah. However we got here, our brains certainly haven't been reengineered for Internet times. Your trust decisions are going to be influenced by your own personal experiences with various kinds of bad guys. The evaluations you make about the relative likelihood of possible attacks will be influenced by physical proximity to the attack sources. The impact of these outdated natural tendencies will be felt in every design you produce.

> This fact is one of the reasons we strongly recommend the use of checklists, which can be prepared once (and specially designed to concentrate on such perceptual problems) and utilized ever after while in a more everyday frame of mind.

Mental models

During the design stage of a project, another of our most interesting human foibles is most evident: the concept of psychological "set," which is the adoption of mental models or metaphors. It's an abstract topic, for sure, and most developers probably never consider it. But we think it bears a little examination here.

All of us use mental models every day as an aid in executing complex tasks. For example, when you're driving a car, you are probably not conscious of the roadway itself, of the asphalt and the paint and the little plastic bumps you might find to help guide your way. Instead, you accept the painted lines and the roadway contours, berms, and culverts as mental channels, constraining your actions and simplifying your choices. You can manage to keep your car between two painted lines (that is, stay in your "lane") more easily than you could calculate the necessary angles and minute real-time adjustments without them. Painted driving lanes are, in fact, an engineering achievement that takes into account this exact human trait.

Designing a piece of software—putting a mental conception into terms the computer can execute—is a complex mental activity as well. All the software engineers we know make extensive use of mental models and metaphors to simplify the task.

In fact, one of the characteristics of an excellent engineer may be that very ability to accept for the moment such a metaphor, to put oneself in the frame of mind in which, for example, a "stream of input characters is what the user is saying to us about what actions the program should take." If you take a second look at that last phrase, we think you will agree with us that extensive metaphorical imagery comes very naturally when people are talking about programs.

Enter the bad guy. Attackers can often succeed by purposely looking only at the asphalt, without seeing the lanes. To find security holes, think like an alien: look at everything fresh, raw, and without socialized context. (See the later sidebar "The Case of the Mouse Driver" for an example of this in action.) Similarly, to avoid security vulnerabilities in your code, you must develop the habit of suspending, from time to time, your voluntary immersion in the program's metaphors. You must train yourself (or be goaded by checklists and aided by specialized tests) to examine the ones and zeroes for what they are, surrendering their interpretation as identification numbers, or inventory elements, or coordinates on a screen.

Ways of thinking about software

In order for your applications to stand up against a determined attack, you will need to build in several layers of defense. You don't want an exploitable weakness at any level. To weed those out, you will need a thorough understanding of what a program *is*—of the worlds in which your software lives.

Many of us have spent our whole working lives dealing with software. We design, write, adapt, fix, and use the stuff. When we do, what are we manipulating? You have probably gestured at a printout or a display of letters on a screen, for example, and referred to that as a program. But what is a computer program, really?

Here is a list of ways that you might think about the nature of software. We invite you to try to imagine how you as an attacker might try to exploit a program in each "plane of existence" we list. You can think of software as:

- An arrangement of abstract algorithms
- Lines of text on a sheet of paper or screen
- A series of instructions for a particular computer processor
- A stream of ones and zeros in computer memory, or stored on magnetic or optical media
- A series of interlinked library routines, third-party code, and original application software
- A stream of electronic and optical signals along electromechanical and other kinds of pathways
- Running or residing on a host as an element of a hardware network

All of the above are fairly straightforward. But here are a few other ways that may not be so straightforward. You'll want to consider your application as:

- A set of "vertical" layers, such as transport, protocol, and presentation. (These are elements that, in a way, can be thought of as being built on top of one another.)
- A set of "horizontal" stages, such as firewall, GUI (Graphical User Interface), business logic, and database server. (These are "peer" elements that operate at the same level and communicate with each other.)
- A series of events that takes place in designated time slices and in a controlled order.
- Executing at a disparate set of locations. Think about it: when an application is running, where are the user, the code itself, the host, the server, the database, the firewall, and the ISP located? They can all be in different locations, spread around the world.

It's remarkable to us, but true, that we have seen successful attacks based on each of the points of view listed in this section! It is mind-bending considerations like these that make effective application security such a tremendous challenge.

Here are a couple of examples of how some of these unusual considerations can affect security. On the "good guy" side, one of the most intriguing security patents of recent years uses the physical location of a person (as indicated by a global positioning system device) to help decide whether that person should be allowed to log into a system. This approach uses a characteristic that is seldom considered—precise physical location—to enhance the accuracy of authentication and authorization decisions. On the other hand, some of the most difficult software vulnerabilities we've ever had to fix had to do with subtle timing effects involving events—just a few milliseconds apart—that could occur in two slightly different orders.

For an illustration of how "mental" aspects of software can lead to vulnerabilities, see the following sidebar.

Real-World Factors

Enough theory. Let's come back to the real world now, and consider for a moment how software is actually produced. We'll start with a few points that are sure to offend some of our colleagues.

The source of our source code

Do you know who wrote most of the software the Internet runs on? Amateurs originally wrote many of the systems programs that have the worst

The Case of the Mouse Driver

One of our favorite security bugs helps illustrate how attackers think outside the programming metaphors. In this case, an attacker found that he was able to take control of a Unix workstation by manipulating a piece of system software known as a *mouse driver*. The designer of this program certainly never intended it to be invoked by a real user. It was called as part of a chain of execution by another program. Still, probably because convenient library routines were available for the purpose—or perhaps because it made it easy to debug the program during development—input to the driver was supplied in the form of parameters on a command line. The job of the mouse driver was to position the cursor on the screen in a spot corresponding to movements of the mouse. The X and Y coordinates at which the cursor was to be positioned were supplied as integral values from, say, 0 to 1023. In normal use, the command line provided by the invoking screen-control software would look something like "driver 100 100".

The program, because it needed to manipulate the screen cursor, was installed with high system privileges. And this design worked perfectly well for years, until one day someone with malevolent intent found a way to subvert it. By invoking the program directly and by supplying X and Y coordinates that were so large as to be meaningless, the manipulator was able to deliberately overflow the buffers allocated for the coordinates and use the program's privileges to take control of the system.

This vulnerability came into existence precisely because the engineer successfully "channelized" his thinking. The attacker succeeded by ignoring the purpose for which the program was designed, rejecting the metaphor underlying the design and instead looking straight at the bits. It's a skill to be cultivated by those who want to understand how software can be subverted, though, and as we mentioned, it's a skill that's perhaps antithetical to the skills that facilitate software design itself.

vulnerabilities. (Don't worry, we'll excoriate the professionals soon enough.) One reason for this is that Berkeley undergraduates first developed much of Unix—in particular, the TCP/IP networking subsystem. Thus, we owe many of the Internet's design and architectural decisions, and a surprising amount of code, to a collection of students of widely varying abilities using techniques that were current in the mid-1970s![*]

[*] Professor Eugene H. Spafford describes the history well in "UNIX and Security: The Influences of History," *Information Systems Security*, Auerbach Publications; 4(3), pp. 52-60, Fall 1995.

The democratization of development

The problem of amateurs writing code is not simply a historic one. Much of today's new software is being written by folks with no training at all in software engineering. A good example is the fact that many CGI scripts used extensively on the Net (some on which other folks have built entire businesses) have been clearly thrown together by people with no background at all in software. (That is, in fact, one of the design goals of HTML.) Don't get us wrong. We think it's terrific that practically anybody with the will to learn the basics can put together an online service, or a library, or a form-based database. But there is a cost.

Of course, we don't really believe that most of the security problems on the Net arise because gross amateurs are writing the programs. We professionals deserve most of the blame. So we're going to shift gears again and look at a few reasons why, even with the best training and the best intentions, doing software engineering securely in the real world remains a very challenging undertaking.

Production pressures

Almost all software is produced under some schedule pressure. Software engineers don't work in a vacuum—even if they care passionately about secure coding and work not for profit-seeking software houses, but as part of an open source effort. Testing time is limited. The chance to research how someone else has approached a problem may not come before it's time to freeze and ship. The real world impinges, sometimes in unpredictable ways.

The plight of the software engineer who wants to produce secure code is never easy. Sometimes we have to give up on the best possible result, and settle for the best result possible. And sometimes that best result (from the point of view of the individual engineer, or his or her management) has or may have security weaknesses.

Just secure enough

It is often hard for people who understand technical security issues, but have not worked as full-time software engineers, to understand how companies comprised of their colleagues can produce deeply flawed and insecure products.* One of the hopes we have for this book is that it will provide

* We have in mind comments such as one by Karl Strickland, a convicted computer attacker and member of the "8LGM" group, which posted exploit scripts on the Internet in the late 1990s. "I don't see the problem. One bug fix, one person. Two bugfixes [sic], two people. Three bugfixes [sic], three people, working simultaneously on different bugs. How hard can that be?" —Usenet *comp.security.unix* discussion thread, May 1994.

some insight here—not by way of making excuses for anyone, but rather by helping to foster a level of understanding that can help remove the root causes of these problems.

Suppose that you are a software vendor in a competitive marketplace. Your profit margins are tight, and your marketing team believes that security is not a deciding factor for customers in your product space. In this kind of environment, wouldn't you be likely to produce software that is "just secure enough"? Secure enough, we mean, not to alienate the majority of your customer base.

A friend of ours was "security coordinator" for one of the major Internet software producers. Often buttonholed by customers at security conferences and asked questions like, "When are you guys going to stop shipping this crap?" he claims the answer he is proudest of was, "Sometime soon after you folks stop buying it." It's a point to consider.

Let's assume that the vendor's goal is to expend minimal resources to forestall show-stopping vulnerabilities, prevent loss of sales, and keep the company's name out of the news. What are some other factors that keep corporations from investing heavily in security quality?

The main reason, we think, is that whatever time and effort is spent on finding, verifying, and fixing security bugs means that fewer engineers are available for adding new features.

A second reason may be that some companies act as if downplaying, denying, or delaying acknowledgment of security vulnerabilities will give them an edge over the competition. Think about it. If you were the CEO and no one was forcing you to face up to the security flaws in your products, wouldn't you be focusing on positive angles, on new features and services that bring in the revenue? You *would* overlook flaws in your product if you could get away with it, wouldn't you? Most of us would at least be tempted (and we're not battered about by stockholders and litigation-wary attorneys).

The tragedy of the commons

We'd like to think that, even if marketing factors (and common decency) don't suffice, considerations of citizenship and business ethics might compel corporate software producers to clean up their act in security matters. Unfortunately, it doesn't seem to work that way. This might be explained by the so-called "tragedy of the commons," an idea first brought to wide attention in a seminal article by Garrett Hardin in 1968:

> The tragedy of the commons develops in this way. Picture a pasture open to all. It is to be expected that each herdsman will try to keep as many cattle as possible on the commons.

> As a rational being, each herdsman seeks to maximize his gain...The rational herdsman concludes that the only sensible course for him to pursue is to add another animal to his herd. And another…But this is the conclusion reached by each and every rational herdsman sharing a commons. Therein is the tragedy. Each man is locked into a system that compels him to increase his herd without limit—in a world that is limited.[*]

In our context, the Internet is the common resource. Each vulnerability is a kind of pollution. Adding one more bug to the world's security burden is in the shortsighted economic interest of each company. So long as fixing bugs will divert resources that can be used to individual advantage elsewhere, profit-seeking companies will not invest in wholesale secure coding practices. As Hardin observed, "The inherent logic of the commons remorselessly generates tragedy."

The Lesson of Y2K

Many security experts, including your authors, have lobbied for years for "blanket code sweeps" for security vulnerabilities at some of the big software houses[a]. A careful one-time effort would be no substitute for the revolution in secure coding that seems to be called for, but it would be a giant step forward. Why do you think such pleas have always failed? A similar effort for the remediation of Y2K bugs succeeded notably.

We can think of three reasons:

1. In the case of Y2K, there was a definite, unchangeable deadline.
2. The worldwide focus on possible Y2K catastrophes meant that any company that failed to fix their code was guaranteed a mass of highly unfavorable headlines.
3. In the case of security, it's hard to see where the one-time budget allocation for the sweep would come from. Hope springs eternal, of course!

[a] Again, see Dr. Eugene H. Spafford's article, "UNIX and Security: The Influences of History," as previously cited.

A Call to Arms

You probably knew that the security of Internet software was a mess before you started this book. How do we extricate ourselves?

[*] See Garrett Hardin, "The Tragedy of the Commons," *Science*, 162(1968):1243-1248.

In addition to advocating the widespread adoption of the techniques and practices described in this book, we also call for advances in three particular areas: education, standards, and metrics.

Education

Clearly, we must do a better job of educating engineers about the principles and techniques of secure coding.[*]

We must also ensure that the public understands the demonstrably poor security of Internet software today, and that the various facets of government comprehend the magnitude of the disasters that can strike us if we don't make drastic improvements.

We also need to convince the press that those who attack systems are not geniuses; they're merely criminals. It would help, too, if the media would stop publicizing dramatic names for the various vulnerabilities and exploitation programs, such as (to invent an example) the "Red Slammer." Will it take a decade or more of severe or deadly incidents to change public attitudes about computer attackers?

Standards

Many people have compared the software vulnerability situation today to the carnage endured before the advent of mandatory seat belts in private automobiles.

Having reached the point where we agree, we now call for the development of true secure coding standards—standards that can be used by companies, governments, and consumers to promote prosperity and ensure our safety. It is the only way we can see to get software vendors to *invest in quality*.[†] If every company is forced to participate, none will be able to make the excuse that they can't afford to divert resources from more competitive pursuits.

Metrics

A critical step in the widespread adoption of safe programming techniques and standards is the development of competent security metrics. Until we can apply an accepted measurement tool to two programs (or two versions of the same program) and determine which has fewer security vulnerabilities, we can expect very slow progress in this field.

[*] Professor Spafford told Congress the state of security education means we are facing "a national crisis." See "One View of A Critical National Need: Support for Information Security Education and Research," 1997.

[†] To quote Garrett Hardin again, "Ruin is the destination toward which all men rush, each pursuing his own best interest in a society that believes in the freedom of the commons. Freedom in a commons brings ruin to all."

> Until we have reliable security metrics, consumers will lack the means to reward manufacturers who produce good code and punish those whose products reek with vulnerabilities. Governments will lack the confidence to develop standards, and citizens may never be sure that they are justified in goading government to enforce the laws and requirements that do exist. Engineers will still struggle to refine their own techniques, and hesitate to condemn their colleagues.*
>
> Toward this end, you'll find in the final chapter of this book a discussion of some of the automated tools and techniques available today that can help you flag and fix security bugs. We also discuss briefly a simple script we've used for the rudimentary "security scoring" of application software.

Summary

In this first chapter, we hope we've challenged you with some new ideas about security vulnerabilities. We particularly hope that you may now consider that the blame for security vulnerabilities belongs, to some degree, to all of us who buy and use the seriously flawed programs available today.

This point of view does not minimize or try to mitigate the responsibility of software producers for security quality. They should be held to the highest standards and hung out to dry if they fail. But it does in fact "take two to tango," and customers (particularly, the U.S. government, the biggest software customer, so far as we know, in the world) bear some responsibility to demand secure software.

Those among us who produce software, of course, have a special responsibility and a unique opportunity to improve matters. Our discipline has not reached the state of understanding and sound practice exemplified by those bridge builders shown on the cover of this book, but the folks driving their virtual vehicles over our structures rely on us nevertheless to keep them safe.

In Chapter 2, we'll exhibit the most important architectural principles and engineering concepts you can employ to make your software as secure as possible. In that chapter, we'll try to pass along some distilled security wisdom from the generation of coders that built the Internet.

* Lord Kelvin, the great engineer who formulated the absolute (Kelvin) temperature scale and engineered the laying of the transatlantic cable, said: "I often say that when you can measure what you are speaking about and express it in numbers, you know something about it. But when you cannot measure it, when you cannot express it in numbers...you have scarcely in your thoughts advanced to the state of science, whatever the matter may be."

Questions

- Have you ever written a program section with a security hole? Really? How do you know? And, if you are sure you haven't, why haven't you?
- Do programmers writing code today know more about security than programmers writing code 30 years ago?
- If you accept the principle of writing code that is "just secure enough" for your own applications, do you think it is socially responsible for software vendors to do the same?
- Visualize one of your favorite programs. What is it? Are you seeing a series of lines on a computer screen or piece of paper? Or is the "program" the series of machine-language instructions? Is it perhaps the algorithm or heuristic, or maybe the very input-to-output transformations that do the useful work? Now consider: in which of these various forms do most vulnerabilities appear? Also, will the same bug-fighting techniques succeed in all of these instantiations?
- Which are more dangerous: cars without seat belts or Internet-capable programs with bad security? If the former, for how long will that be true? Is that within the lifetime of software you are working on, or will work on some day?
- Suppose you were responsible for the security of a web server. Which would make you feel safer: keeping the server in a room around the corner from your office or keeping it in another office building (also owned by your company) around the world? Why? Would it make a difference if you "knew"—had physically met—one or more workers in that remote building?
- Are the people you know more trustworthy than those you don't?
- Are you and your friends better engineers than we are?
- What are you doing to make the software you use more secure?
- Can you think of a safe way for software vendors to ensure that their customers install security patches? Should the process be automated? Should vendors be launching patch-installation worms that exploit a vulnerability in order to install a fix for it?
- Should software vendors be shielded from product liability?