1 Introduction to Software Security

". . . any program, no matter how innocuous it seems, can harbor security holes. . . . We thus have a firm belief that everything is guilty until proven innocent."

—William Cheswick and Steve Bellovin
Firewalls and Internet Security

Computer security is an important topic. As e-commerce blossoms, and the Internet works its way into every nook and cranny of our lives, security and privacy come to play an essential role. Computer security is moving beyond the realm of the technical elite, and is beginning to have a real impact on our everyday lives.

It is no big surprise, then, that security seems to be popping up everywhere, from headline news to TV talk shows. Because the general public doesn't know very much about security, a majority of the words devoted to computer security cover basic technology issues such as what firewalls are, what cryptography is, or which antivirus product is best. Much of the rest of computer security coverage centers around the "hot topic of the day," usually involving an out-of-control virus or a malicious attack. Historically, the popular press pays much attention to viruses and denial-of-service attacks: Many people remember hearing about the Anna Kournikova worm, the "Love Bug," or the Melissa virus ad nauseam. These topics are important, to be sure. Nonetheless, the media generally manages not to get to the heart of the matter when reporting these subjects. Behind every computer security problem and malicious attack lies a common enemy—bad software.

It's All about the Software

The Internet continues to change the role that software plays in the business world, fundamentally and radically. Software no longer simply supports back offices and home entertainment. Instead, software has become the lifeblood of our businesses and has become deeply entwined in our lives. The invisible hand of Internet software enables e-business, automates supply chains, and provides instant, worldwide access to information. At the same time, Internet software is moving into our cars, our televisions, our home security systems, and even our toasters.

The biggest problem in computer security today is that many security practitioners don't know what the problem is. Simply put, it's the software! You may have the world's best firewall, but if you let people access an application through the firewall and the code is remotely exploitable, then the firewall will not do you any good (not to mention the fact that the firewall is often a piece of fallible software itself). The same can be said of cryptography. In fact, 85% of CERT security advisories[1] could not have been prevented with cryptography [Schneider, 1998].

Data lines protected by strong cryptography make poor targets. Attackers like to go after the programs at either end of a secure communications link because the end points are typically easier to compromise. As security professor Gene Spafford puts it, "Using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench."

Internet-enabled applications, including those developed internally by a business, present the largest category of security risk today. Real attackers compromise software. Of course, software does not need to be Internet enabled to be at risk. The Internet is just the most obvious avenue of attack in most systems.

This book is about protecting yourself by building secure software. We approach the software security problem as a risk management problem. The fundamental technique is to begin early, know your threats, design for security, and subject your design to thorough objective risk analyses and testing. We provide tips and techniques that architects, developers, and managers can use to produce Internet-based code that is as secure as necessary.

1. CERT is an organization that studies Internet security vulnerabilities, and occasionally releases security advisories when there are large security problems facing the Internet. See http://www.cert.org.

A good risk management approach acknowledges that security is often just a single concern among many, including time-to-market, cost, flexibility, reusability, and ease of use. Organizations must set priorities, and identify the relative costs involved in pursuing each. Sometimes security is not a high priority.

Some people disagree with a risk management security approach. Many people would like to think of security as a yes-or-no, black-or-white affair, but it's not. You can never prove that any moderately complex system is secure. Often, it's not even worth making a system as secure as possible, because the risk is low and the cost is high. It's much more realistic to think of software security as risk management than as a binary switch that costs a lot to turn on.

Software is at the root of all common computer security problems. If your software misbehaves, a number of diverse sorts of problems can crop up: reliability, availability, safety, and security. The extra twist in the security situation is that a bad guy is actively trying to make your software misbehave. This certainly makes security a tricky proposition.

Malicious hackers don't create security holes; they simply exploit them. Security holes and vulnerabilities—the real root cause of the problem—are the result of bad software design and implementation. Bad guys build exploits (often widely distributed as scripts) that exploit the holes. (By the way, we try to refer to bad guys who exploit security holes as *malicious hackers* instead of simply *hackers* throughout this book. See the sidebar for more details.)

Hackers, Crackers, and Attackers

We'll admit it. We are hackers. But don't break out the handcuffs yet. We don't run around breaking into machines, reading other people's e-mail, and erasing hard disks. In fact, we stay firmly on this side of the law (well, we would in a world where driving really fast is always legal).

The term hacker originally had a positive meaning. It sprang from the computer science culture at the Massachusetts Institute of Technology during the late 1960s, where it was used as a badge of honor to refer to people who were exceptionally good at solving tricky problems through programming, often as if by magic. For most people in the UNIX development community, the term's preferred definition retains that meaning, or is just used to refer to someone who is an excellent and enthusiastic programmer. Often, hackers like to tinker with things, and figure out how they work.

Software engineers commonly use the term hacker in a way that carries a slightly negative connotation. To them, a hacker is still the programming world's equivalent of MacGyver—someone who can solve hard programming problems given a ball of

fishing line, a matchbook, and two sticks of chewing gum. The problem for software engineers is not that hackers (by their definition) are malicious; it is that they believe cobbling together an ad hoc solution is at best barely acceptable. They feel careful thought should be given to software design before coding begins. They therefore feel that hackers are developers who tend to "wing it," instead of using sound engineering principles. (Of course, *we* never do that! Wait! Did we say that we're hackers?)

Far more negative is the definition of hacker that normal people use (including the press). To most people, a hacker is someone who maliciously tries to break software. If someone breaks in to your machine, many people would call that person a hacker. Needless to say, this definition is one that the many programmers who consider themselves "hackers" resent.

Do we call locksmiths burglars just because they could break into our house if they wanted to do so? Of course not. But that's not to say that locksmiths can't be burglars. And, of course, there are hackers who are malicious, do break into other people's machines, and do erase disk drives. These people are a very small minority compared with the number of expert programmers who consider themselves "hackers."

In the mid 1980s, people who considered themselves hackers, but hated the negative connotations the term carried in most peoples' minds (mainly because of media coverage of security problems), coined the term **cracker**. A cracker is someone who breaks software for nefarious ends.

Unfortunately, this term hasn't caught on much outside of hacker circles. The press doesn't use it, probably because it is already quite comfortable with "hacker." And it sure didn't help that they called the movie *Hackers* instead of *Crackers.* Nonetheless, we think it is insulting to lump all hackers together under a negative light. But we don't like the term cracker either. To us, it sounds dorky, bringing images of Saltines to mind. So when we use the term **hacker**, that should give you warm fuzzy feelings. When we use the term **malicious hacker**, **attacker**, or **bad guy**, it is okay to scowl. If we say "malicious hacker," we're generally implying that the person at hand is skilled. If we say anything else, they may or may not be.

Who Is the Bad Guy?

We've said that some hackers are malicious. Many of those who break software protection mechanisms so that pirate copies can be easily distributed are malicious. Removing such protection mechanisms takes a fair bit of skill, which is probably just cause to label that particular person a hacker as well. However, most bad guys are not hackers; they're just kids trying to break into other people's machines.

Some hackers who are interested in security may take a piece of software and try to break it just out of sheer curiosity. If they do break it, they won't do anything malicious;

they will instead notify the author of the software, so that the problem can be fixed. If they don't tell the author in a reasonable way, then they can safely be labeled *malicious.* Fortunately, most people who find serious security flaws don't use their finds to do bad things.

At this point, you are probably wondering who the malicious attackers and bad guys really are! After all, it takes someone with serious programming experience to break software. This may be true in finding a new exploit, but not in exploiting it. Generally, bad guys are people with little or no programming ability capable of downloading, building, and running programs other people write (hackers often call this sort of bad guy a **script kiddie**). These kinds of bad guys go to a hacker site, such as (the largely defunct) http://www.rootshell.com, download a program that can be used to break into a machine, and get the program to run. It doesn't take all that much skill (other than hitting return a few times). Most of the people we see engage in this sort of activity tend to be teenage boys going through a rebellious period. Despite the fact that such people tend to be relatively unskilled, they are still very dangerous.

So, you might ask yourself, who wrote the programs that these script kiddies used to break stuff? Hackers? And if so, then isn't that highly unethical? Yes, hackers tend to write most such programs. Some of those hackers do indeed have malicious intent, and are truly bad people. However, many of these programs are made public by hackers who believe in the principle of *full disclosure.* The basic idea behind this principle is that everyone will be encouraged to write more secure software if all security vulnerabilities in software are made public. It encourages vendors to acknowledge and fix their software and can expose people to the existence of problems before fixes exist.

Of course, there are plenty of arguments against full disclosure too. Although full disclosure may encourage vendors to fix their problems quickly, there is almost always a long delay before all users of a vendor's software upgrade their software. Giving so much information to potential attackers makes it much easier for them to exploit those people. In the year 2000, Marcus Ranum sparked considerable debate on this issue, arguing that full disclosure does more harm than good. Suffice it to say, people without malicious intent sometimes do release working attack code on a regular basis, knowing that there is a potential for their code to be misused. These people believe the benefits of full disclosure outweigh the risks. We provide third-party information on these debates on the book's Web site.

We want to do our part in the war against crummy software by providing lots of information about common mistakes, and by providing good ideas that can help you build more secure code.

Dealing with Widespread Security Failures

We probably don't need to spend too much time convincing you that security holes in software are common, and that you need to watch out for them. Nonetheless, many people do not realize how widespread a problem insecure software really is.

The December 2000 issue of *The Industry Standard* (a new economy business magazine) featured an article entitled "Asleep at the Wheel" by David Lake [Lake, 2000]. The article emphasized the magnitude of today's security problem using an excellent presentation of data. Using numbers derived from Bugtraq (a mailing list dedicated to reporting security vulnerabilities), Lake created Figure 1–1, which shows the number of new vulnerabilities reported monthly, from the start of 1998 until the end of September 2000. According to these data, the number of software holes being reported is growing. These figures suggest that approximately 20 new vulnerabilities in software are made public each week. Some of these vulnerabilities are found in source-available software programs, but many are also found in proprietary code. Similarly, both UNIX and Windows programs are well represented in such vulnerability data. For example, there were more than a dozen security problems found and fixed in Microsoft Outlook during the year 2000.

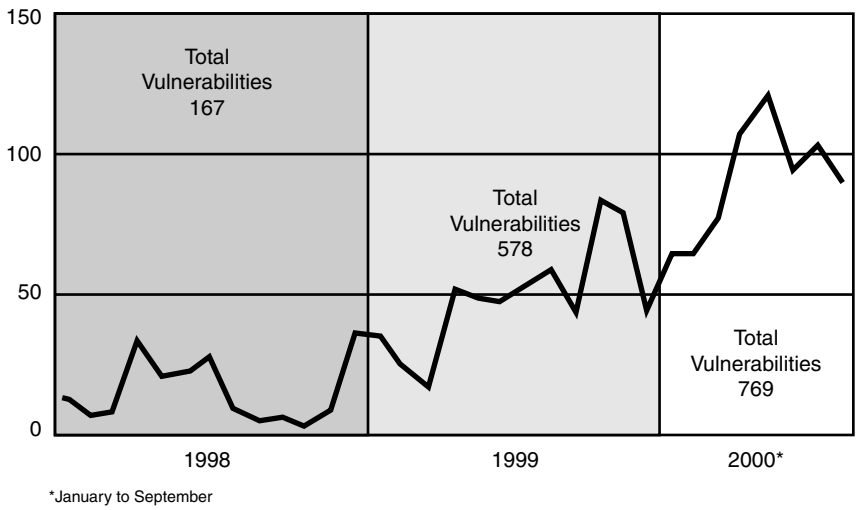

Figure 1–1 Bugtraq vulnerabilities by month from January 1998 to September 2000.

Additionally, "tried-and-true" software may not be as safe as one may think. Many vulnerabilities that have been discovered in software existed for months, years, and even decades before discovery, even when the source was available (see Chapter 4).

The consequences of security flaws vary. Consider that the goal of most malicious hackers is to "own" a networked computer (and note that most malicious attackers seem to break into computers simply because they can). Attacks tend to be either "remote" or "local." In a remote attack, a malicious attacker can break into a machine that is connected to the same network, usually through some flaw in the software. If the software is available through a firewall, then the firewall will be useless. In a local attack, a malicious user can gain additional privileges on a machine (usually administrative privileges). Most security experts agree that once an attacker has a foothold on your machine, it is incredibly difficult to keep them from getting administrative access. Operating systems and the privileged applications that are generally found in them constitute such a large and complex body of code that the presence of some security hole unknown to the masses (or at least the system administrator) is always likely.

Nonetheless, both kinds of problems are important, and it is important for companies that wish to be secure keep up with security vulnerabilities in software. There are several popular sources for vulnerability information, including Bugtraq, CERT advisories, and RISKS Digest.

Bugtraq

The Bugtraq mailing list, administered by securityfocus.com, is an e-mail discussion list devoted to security issues. Many security vulnerabilities are first revealed on Bugtraq (which generates a large amount of traffic). The signal-to-noise ratio on Bugtraq is low, so reader discretion is advised. Nevertheless, this list is often the source of breaking security news and interesting technical discussion. Plenty of computer security reporters use Bugtraq as their primary source of information.

Bugtraq is famous for pioneering the principle of **full disclosure**, which is a debated notion that making full information about security vulnerabilities public will encourage vendors to fix such problems more quickly. This philosophy was driven by numerous documented cases of vendors downplaying security problems and refusing to fix them when they believed that few of their customers would find out about any problems.

If the signal-to-noise ratio on Bugtraq is too low for your tastes, the securityfocus.com Web site keeps good information on recent vulnerabilities.

CERT Advisories

The CERT Coordination Center ([CERT/CC], www.cert.org) is located at the Software Engineering Institute, a federally funded research and development center operated by Carnegie Mellon University. CERT/CC studies Internet security vulnerabilities, provides incident response services to sites that have been the victims of attack, and publishes a variety of security alerts.

Many people in the security community complain that CERT/CC announces problems much too late to be effective. For example, a problem in the TCP protocol was the subject of a CERT advisory released a decade after the flaw was first made public. Delays experienced in the past can largely be attributed to the (since-changed) policy of not publicizing an attack until patches were available. However, problems like the aforementioned TCP vulnerability are more likely attributed to the small size of CERT. CERT tends only to release advisories for significant problems. In this case, they reacted once the problem was being commonly exploited in the wild. The advantage of CERT/CC as a knowledge source is that they highlight attacks that are in actual use, and they ignore low-level malicious activity. This makes CERT a good source for making risk management decisions and for working on problems that really matter.

RISKS Digest

The RISKS Digest forum is a mailing list compiled by security guru Peter Neumann that covers all kinds of security, safety, and reliability risks introduced and exacerbated by technology. RISKS Digest is often among the first places that sophisticated attacks discovered by the security research community are announced. Most Java security attacks, for example, first appeared here. The preferred method for reading the RISKS Digest is through the Usenet News group comp.risks. However, for those without easy access to Usenet News, you can subscribe to the mailing list by sending a request to risks-request@CSL.SRI.COM.

These aren't the only sources for novel information, but they're certainly among the most popular. The biggest problem is that there are too many sources. To get the "big picture," one must sift through too much information. Often, administrators never learn of the existence of an important patch that should definitely be applied to their system. We don't really consider this problem the fault of the system administrator. As Bruce

Schneier has said, blaming a system administrator for not keeping up with patches is blaming the victim. It is very much like saying, "You deserved to get mugged for being out alone in a bad neighborhood after midnight." Having to keep up with dozens of weekly reports announcing security vulnerabilities is a Herculean task that is also thankless. People don't see the payoff.

Technical Trends Affecting Software Security

Complex systems, by their very nature, introduce multiple risks. And almost all systems that involve software are complex. One risk is that malicious functionality can be added to a system (either during creation or afterward) that extends it past its primary, intended design. As an unfortunate side effect, inherent complexity lets malicious and flawed subsystems remain invisible to unsuspecting users until it is too late. This is one of the root causes of the malicious code problem. Another risk more relevant to our purposes is that the complexity of a system makes it hard to understand, hard to analyze, and hard to secure. Security is difficult to get right even in simple systems; complex systems serve only to make security harder. Security risks can remain hidden in the jungle of complexity, not coming to light until it is too late.

Extensible systems, including computers, are particularly susceptible to complexity-driven hidden risk and malicious functionality problems. When extending a system is as easy as writing and installing a program, the risk of intentional introduction of malicious behavior increases drastically—as does the risk of introducing unintentional vulnerabilities.

Any computing system is susceptible to hidden risk. Rogue programmers can modify systems software that is initially installed on the machine. Unwitting programmers may introduce a security vulnerability when adding important features to a network-based application. Users may incorrectly install a program that introduces unacceptable risk or, worse yet, accidentally propagate a virus by installing new programs or software updates. In a multi-user system, a hostile user may install a Trojan horse to collect other users' passwords. These attack classes have been well-known since the dawn of computing, so why is software security a bigger problem now than in the past? We believe that a small number of trends have a large amount of influence on the software security problem.

One significant problem is the fact that computer networks are becoming ubiquitous. The growing connectivity of computers through

the Internet has increased both the number of attack vectors (avenues for attack) and the ease with which an attack can be made. More and more computers, ranging from home personal computers (PCs) to systems that control critical infrastructures (such as the power grid), are being connected to the Internet. Furthermore, people, businesses, and governments are increasingly dependent on network-enabled communication such as e-mail or Web pages provided by information systems. Unfortunately, because these systems are connected to the Internet, they become vulnerable to attacks from distant sources. Put simply, an attacker no longer needs physical access to a system to cause security problems.

Because access through a network does not require human intervention, launching automated attacks from the comfort of your living room is relatively easy. Indeed, the well-publicized denial-of-service attacks in February 2000 took advantage of a number of (previously compromised) hosts to flood popular e-commerce Web sites, including Yahoo!, with bogus requests automatically. The ubiquity of networking means that there are more systems to attack, more attacks, and greater risks from poor software security practice than ever before.

A second trend that has allowed software security vulnerabilities to flourish is the size and complexity of modern information systems and their corresponding programs. A desktop system running Windows/NT and associated applications depends on the proper functioning of the kernel as well as the applications to ensure that an attacker cannot corrupt the system. However, NT itself consists of approximately 35 million lines of code, and applications are becoming equally, if not more, complex. When systems become this large, bugs cannot be avoided.

Exacerbating this problem is the widespread use of low-level programming languages, such as C or C++, that do not protect against simple kinds of attacks (most notably, buffer overflows). However, even if the systems and applications codes were bug free, improper configuration by retailers, administrators, or users can open the door to attackers. In addition to providing more avenues for attack, complex systems make it easier to hide or to mask malicious code. In theory, we could analyze and prove that a small program was free of security problems, but this task is impossible for even the simplest of desktop systems today, much less the enterprise-wide systems used by businesses or governments.

A third trend exacerbating software security problems is the degree to which systems have become extensible. An extensible host accepts updates

or extensions, sometimes referred to as **mobile code**, so that the system's functionality can be evolved in an incremental fashion. For example, the plug-in architecture of Web browsers makes it easy to install viewer extensions for new document types as needed.

Given a basic intuitive grasp of the architecture of a browser (a program that runs on top of an operating system and provides basic Web interface services), it is natural to assume that a browser may be used to enhance security. In reality, it is hard to tell where the boundaries of the browser are, where the operating system fits, and how a browser can protect itself. The two most popular browsers, Netscape Navigator and Microsoft Internet Explorer (MSIE), have very fuzzy boundaries and include many more hooks to external applications than most people realize.

On the most pervasive platform (Windows 95, 98, and Millennium Edition (ME)) there is really no way for a browser to protect itself or any secrets it may be trying to keep (like client-side certificates) at all. This means that if your design requires some security features inside the browser (like an intact Java Virtual Machine [JVM] or a cryptographic secret), there is probably a real need for a more advanced operating system like Windows/NT or UNIX.

Without delving into the details of how a browser is constructed, it is worth showing a general-purpose abstract architectural diagram of each. Figures 1–2 and 1–3 show Netscape and MSIE respectively. From a high-level perspective, it is clear that there are many interacting components involved in each architecture. This makes securing a browser quite a monumental task. In addition, helper applications (such as AOL Instant Messenger for Netscape and ActiveX control functionality in MSIE) introduce large security risks.

Browsers are not the only extensible systems. Today's operating systems support extensibility through dynamically loadable device drivers and modules. Current applications, such as word processors, e-mail clients, spreadsheets, and (yes) Web browsers, support extensibility through scripting, controls, components, dynamically loadable libraries, and applets.

From an economic standpoint, extensible systems are attractive because they provide flexible interfaces that can be adapted through new components. In today's marketplace, it is crucial that software be deployed as rapidly as possible to gain market share. Yet the marketplace also demands that applications provide new features with each release. An extensible architecture makes it easy to satisfy both demands by letting

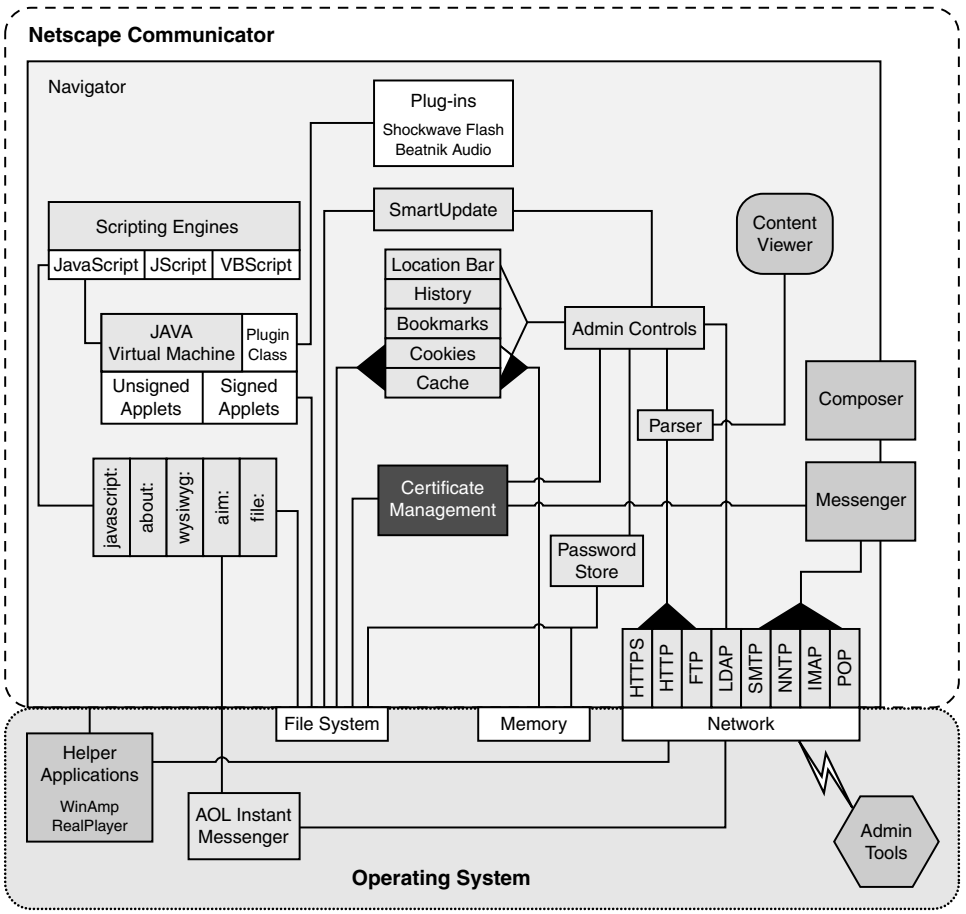

Figure 1–2 An overview of the Netscape architecture.

companies ship the base application code early, and later ship feature extensions as needed.

Unfortunately, the very nature of extensible systems makes security harder. For one thing, it is hard to prevent malicious code from slipping in as an unwanted extension. Meaning, the features designed to add extensibility to a system (such as Java's class-loading mechanism) must be designed with security in mind. Furthermore, analyzing the security of an extensible system is much harder than analyzing a complete system that can't be changed. How can you take a look at code that has yet to arrive? Better yet, how can you even begin to anticipate every kind of mobile code that may arrive?

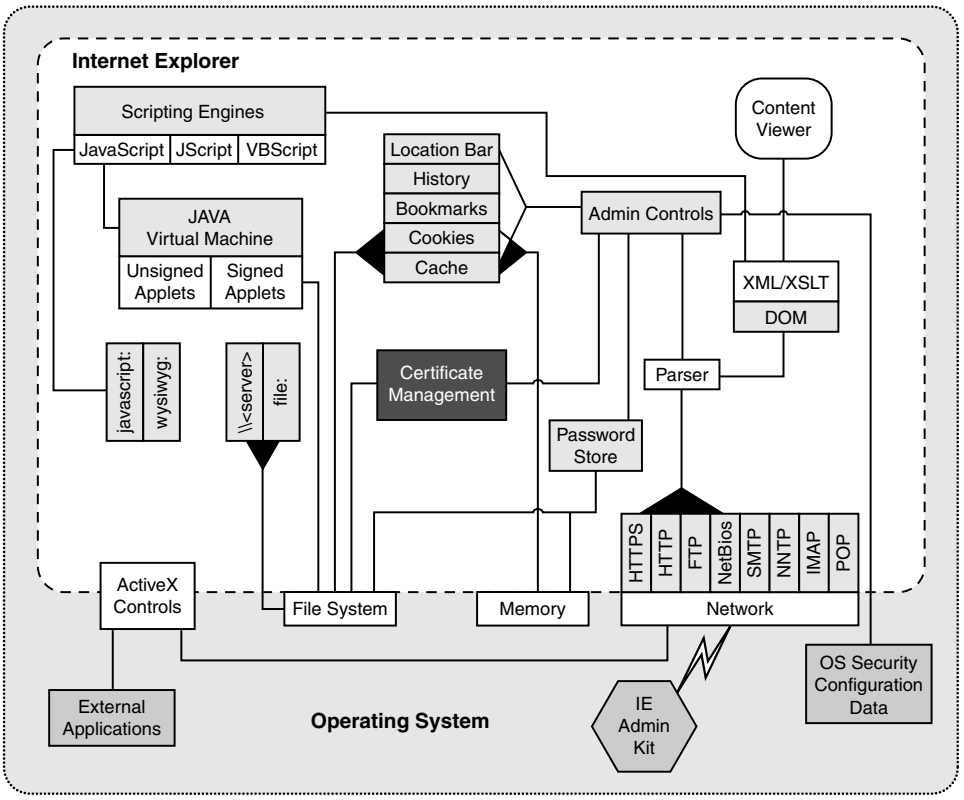

Figure 1–3 An overview of the Internet Explorer architecture.

Together, the three trends of ubiquitous networking, growing system complexity, and built-in extensibility make the software security problem more urgent than ever. There are other trends that have an impact as well, such as the lack of diversity in popular computing environments, and the tendency for people to use systems in unintended ways (for example, using Windows NT as an embedded operating system). For more on security trends, see Bruce Schneier's book *Secrets and Lies* [Schneier, 2000].

The 'ilities

Is security a feature that can be added on to an existing system? Is it a static property of software that remains the same no matter what environment the code is placed in? The answer to these questions is an emphatic *no*.

Bolting security onto an existing system is simply a bad idea. Security is not a feature you can add to a system at any time. Security is like safety, dependability, reliability, or any other software *'ility.* Each 'ility is a systemwide emergent property that requires advance planning and careful design. Security is a behavioral property of a complete system in a particular environment.

It is always better to design for security from scratch than to try to add security to an existing design. Reuse is an admirable goal, but the environment in which a system will be used is so integral to security that any change of environment is likely to cause all sorts of trouble—so much trouble that well-tested and well-understood software can simply fall to pieces.

We have come across many real-world systems (designed for use over protected, proprietary networks) that were being reworked for use over the Internet. In every one of these cases, Internet-specific risks caused the systems to lose all their security properties. Some people refer to this problem as an **environment problem**: when a system that is secure enough in one environment is completely insecure when placed in another. As the world becomes more interconnected via the Internet, the environment most machines find themselves in is at times less than friendly.

What Is Security?

So far, we have dodged the question we often hear asked: What is security? Security means different things to different people. It may even mean different things to the same person, depending on the context. For us, security boils down to enforcing a policy that describes rules for accessing resources. If we don't want unauthorized users logging in to our system, and they do, then we have a security violation on our hands. Similarly, if someone performs a denial-of-service attack against us, then they're probably violating our policy on acceptable availability of our server or product. In many cases, we don't really require an explicit security policy because our implicit policy is fairly obvious and widely shared.

Without a well-defined policy, however, arguing whether some event is really a security breach can become difficult. Is a port scan considered a security breach? Do you need to take steps to counter such "attacks?" There's no universal answer to this question. Despite the wide evidence of such gray areas, most people tend to have an implicit policy that gets them pretty far. Instead of disagreeing on whether a particular action someone takes is a security problem, we worry about things like whether the consequences are

significant or whether there is anything we can do about the potential problem at all.

Isn't That Just Reliability?

Comparing reliability with security is a natural thing to do. At the very least, reliability and security have a lot in common. Reliability is roughly a measurement of how robust your software is with respect to some definition of a bug. The definition of a bug is analogous to a security policy. Security can be seen as a measurement of how robust your software is with respect to a particular security policy. Some people argue that security is a subset of reliability, and some argue the reverse. We're of the opinion that security is a subset of reliability. If you manage to violate a security policy, then there's a bug. The security policy always seems to be part of the particular definition of "robust" that is applied to a particular product.

Reliability problems aren't always security problems, although we should note that reliability problems are security problems a lot more often than one may think. For example, sometimes bugs that can crash a program provide a potential attacker with unauthorized access to a resource. However, reliability problems can usually be considered denial-of-service problems. If an attacker knows a good, remotely exploitable "crasher" in the Web server you're using, this can be leveraged into a denial-of-service attack by tickling the problem as often as possible.

If you apply solid software reliability techniques to your software, you will probably improve its security, especially against some kinds of an attack. Therefore, we recommend that anyone wishing to improve the security of their products work on improving the overall robustness of their products as well. We won't cover that kind of material in any depth in this book. There are several good books on software reliability and testing, including the two classics *Software Testing Techniques* by Boris Beizer [Beizer, 1990] and *Testing Computer Software* by Cem Kaner et al. [Kaner, 1999]. We also recommend picking up a good text on software engineering, such as *The Engineering of Software* [Hamlet, 2001].

Penetrate and Patch Is Bad

Many well-known software vendors don't yet understand that security is not an add-on feature. They continue to design and create products at alarming rates, with little attention paid to security. They start to worry

about security only after their product has been publicly (and often spectacularly) broken by someone. Then they rush out a patch instead of coming to the realization that designing security in from the start may be a better idea. This sort of approach won't do in e-commerce or other business-critical applications.

Our goal is to minimize the unfortunately pervasive "penetrate-and-patch" approach to security, and to avoid the problem of desperately trying to come up with a fix to a problem that is being actively exploited by attackers. In simple economic terms, finding and removing bugs in a software system before its release is orders of magnitude cheaper and more effective than trying to fix systems after release [Brooks, 1995].

There are many problems to the penetrate-and-patch approach to security. Among them are the following:

- Developers can only patch problems which they know about. Attackers may find problems that they never report to developers.
- Patches are rushed out as a result of market pressures on vendors, and often introduce new problems of their own to a system.
- Patches often only fix the symptom of a problem, and do nothing to address the underlying cause.
- Patches often go unapplied, because system administrators tend to be overworked and often do not wish to make changes to a system that "works." As we discussed earlier, system administrators are generally not security professionals.

Designing a system for security, carefully implementing the system, and testing the system extensively before release, presents a much better alternative. We discuss design for security extensively in Chapter 2.

The fact that the existing penetrate-and-patch system is so poorly implemented is yet another reason why the approach needs to be changed. In an *IEEE Computer* article from December 2000 entitled "Windows of Vulnerability: A Case Study Analysis," Bill Arbaugh, Bill Fithen, and John McHugh discuss a life cycle model for system vulnerabilities that emphasizes how big the problem is [Arbaugh, 2000]. Data from their study show that intrusions increase once a vulnerability is discovered, the rate continues to increase until the vendor releases a patch, but exploits continue to occur even after the patch is issued (sometimes years after). Figure 1–4 is based on their data. It takes a long time before most people upgrade to patched versions, because most people upgrade for newer functionality, the hope of more robust software, or better performance; not because they know of a real vulnerability.

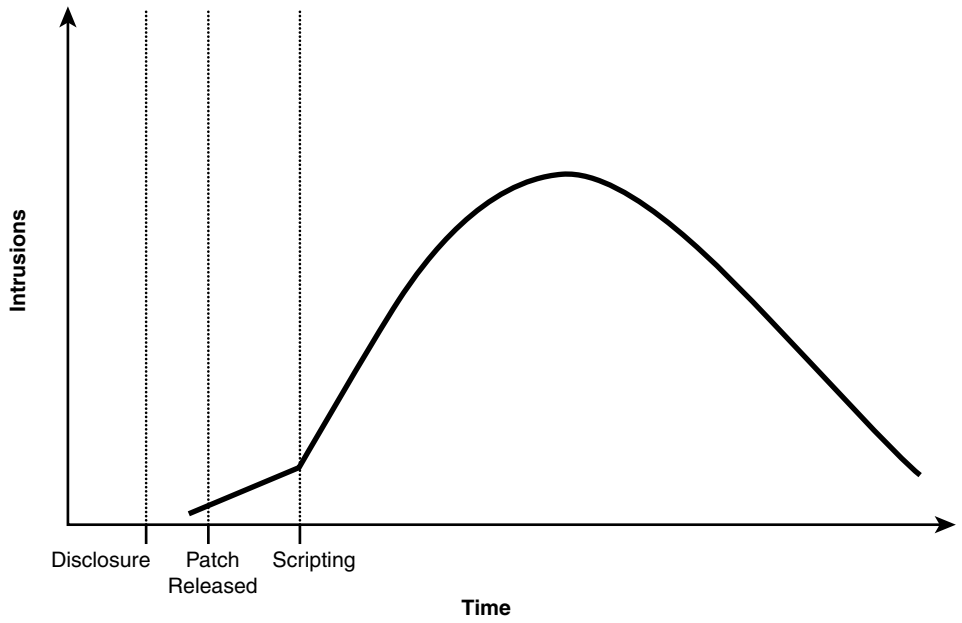

Figure 1–4 Average number of intrusions for a security bug over time.

On Art and Engineering

Software that is properly engineered goes through a well-structured process from requirements design, through detailed specification, to actual implementation. In the world of consumer-ware (software created for the mass consumer market, like browsers), pressure to be first to market and retain what is known as "mind share" compresses the development process so much that software engineering methods are often thrown out the window. This is especially true of testing, which regularly ends up with no scheduled time and few resources. An all-too-common approach is to leave rigorous testing to users in the field (sometimes even paying users when they find bugs!). We think this is just awful.

The *Internet time phenomenon* has exacerbated the software engineering problem. These days, Internet years rival dog years in shortness of duration. Given the compressed development schedules that go along with this accelerated kind of calendar, the fact that specifications are often very poorly written (if they exist at all) is not surprising. It is not uncommon to encounter popular consumer-oriented systems that have no specifications.

Java makes a good case study of the complex interrelation between secure design and secure implementation. One of the most common misconceptions about Java security holes is that they are all simple implementation errors and that the specification has been sound and complete since day one. Threads in the newsgroup `comp.lang.java.security` and other newsgroups often repeat this fallacy as people attempt to trivialize Java's security holes. The truth is that many of the holes described in books like *Securing Java* are simple implementation bugs (the code-signing hole from April 1997 comes to mind), but others, like problems discovered in Java class loaders, are not [McGraw, 1999]. Sometimes the specification is just plain wrong and must be changed. As an example, consider the Java specification for class loading, which has evolved as inadequacies have come to light. In any case, the much-hyped Java security model focuses primarily on protecting against malicious mobile code. Java is still susceptible to a large number of the same problems as other languages.

Often it is hard to determine whether a security hole is an implementation problem or a specification problem. Specifications are notoriously vague. Given a vague specification, who is to blame when a poor implementation decision is made? Specifications are also often silent; that is, when a hole is discovered and the specification is consulted, there is nothing said about the specific problem area. These sorts of omissions certainly lead to security problems, but are the resulting problems specification problems or implementation problems? In the end, the holes are fixed, regardless of whether they are implementation bugs or design-level problems. This leads to a more robust system. Of course, designing, implementing, and testing things properly in the first place is the least expensive and most successful approach. We discuss how to use software engineering to do these things in Chapter 2.

Security Goals

What does it mean for a software system to be secure? Does it even make sense to make claims like "Java is secure"? How can a program be secure?

Security is not a static feature on which everyone agrees. It's not something than can be conveniently defined away in a reductionist move. For most developers and architects, security is like pornography is to the United States Supreme Court: They may not be able to define it, but they think they know it when they see it.

The problem is, security is relative. Not only is there no such thing as 100% security, even figuring out what "secure" means differs according to

context. A key insight about security is to realize that any given system, no matter how "secure," can probably be broken. In the end, security must be understood in terms of a simple question: *Secure against what and from whom?*

Understanding security is best understood by thinking about goals. What is it we are trying to protect? From whom are we protecting it? How can we get what we want?

Prevention

As in today's criminal justice system, much more attention is paid to security after something bad happens than before. In both cases, an ounce of prevention is probably worth a pound of punishment.

Internet time compresses not only the software development life cycle (making software risk management a real challenge), it also directly affects the propagation of attacks. Once a successful attack on a vulnerability is found, the attack spreads like wildfire on the Internet. Often, the attack is embedded in a simple script, so that an attacker requires no more skill than the ability to hit return in order to carry it out.

Internet time is the enemy of software security. Automated Internet-based attacks on software are a serious threat that must be factored into the risk management equation. This makes prevention more important than ever.

Traceability and Auditing

Because there is no such thing as 100% security, attacks will happen. One of the keys to recovering from an attack is to know who did what, and when they did it. Although auditing is not a direct prevention technology, knowing that there is a system for accountability may in some cases dissuade potential attackers.

Auditing is well understood by accountants, who have practiced double-entry bookkeeping for more than 500 years. Banks and other financial institutions have entire divisions devoted to auditing. Most businesses audit their inventories. Every public company has its books audited by a designated accounting firm to meet Security Exchange Commission regulations. Any system in which security is important should seriously consider including auditing.

Good auditing and traceability measures are essential for forensics. They help detect, dissect, and demonstrate an attack. They show who did what when, and provide critical evidence in court proceedings.

Software auditing is a technological challenge. Bits stored on disk as an audit log are themselves susceptible to attack. Verification of an audit log is thus tricky. Nevertheless, auditing is an essential part of software security.

Monitoring

Monitoring is real-time auditing. Intrusion detection systems based on watching network traffic or poring over log files are simple kinds of monitoring systems. These systems are relative newcomers to the commercial security world, and getting shrink-wrapped products to be useful is not easy because of the alarming number of false alarms.

Monitoring a program is possible on many levels, and is an idea rarely practiced today. Simple approaches can watch for known signatures, such as dangerous patterns of low-level system calls that identify an attack in progress. More complex approaches place monitors in the code itself in the form of assertions.

Often, simple burglar alarms and trip wires can catch an attack in progress and can help prevent serious damage.

Privacy and Confidentiality

Privacy and confidentiality are deeply intertwined. There are clear reasons for business, individuals, and governments to keep secrets. Businesses must protect trade secrets from competitors. Web users often want to protect their on-line activities from the invasive marketing machines of AOL, Amazon.com, Yahoo!, and DoubleClick. Governments have classified military secrets to protect.

Of these three groups, individuals probably least understand how important privacy can be. But they are beginning to clue in. Privacy groups such as the Privacy Foundation (www.privacyfoundation.org) and the Electronic Privacy Information Center (www.epic.org) are beginning to improve the awareness of the general public. Interestingly, the European Union has a large head start over the United States in terms of privacy laws.

In any case, there are often lots of reasons for software to keep secrets and to ensure privacy. The problem is, software is not really designed to do this. Software is designed to run on a machine and accomplish some useful work. This means that the machine on which a program is running can pry out every secret a piece of software may be trying to hide.

One very simple, useful piece of advice is to avoid storing secrets like passwords in your code, especially if that code is likely to be mobile.

Multilevel Security

Some kinds of information are more secret than others. Most governments have multiple levels of information classification, ranging from Unclassified and merely For Official Use Only, through Secret and Top Secret, all the way to Top Secret/Special Compartmentalized Intelligence.

Most corporations have data to protect too—sometimes from their own employees. Having a list of salaries floating around rarely makes for a happy company. Some business partners will be trusted more than others. Technologies to support these kinds of differences are not as mature as we wish.

Different levels of protection are afforded different levels of information. Getting software to interact cleanly with a multilevel security system is tricky.

Anonymity

Anonymity is a double-edge sword. Often there are good social reasons for some kinds of anonymous speech (think of AIDS patients discussing their malady on the Internet), but just as often there are good social reasons not to allow anonymity (think of hate speech, terrorist threats, and so on). Today's software often makes inherent and unanticipated decisions about anonymity. Together with privacy, decisions about anonymity are important aspects of software security.

Microsoft's Global Identifier tracks which particular copy of Microsoft Office originated a document. Sound like Big Brother? Consider that this identifier was used to help tie David L. Smith, the system administrator who created and released the Melissa virus, to his malicious code.[2] What hurts us can help us too.

Often, technology that severely degrades anonymity and privacy turns out to be useful for law enforcement. The FBI's notorious Carnivore system is set up to track who sends e-mail to whom by placing a traffic monitoring system at an Internet service provider (ISP) like AOL. But why should we trust the FBI to stop at the headers? Especially when we know that the supposedly secret Echelon system (also run by the US government) regularly scans international communications for keywords and patterns of activity!

Cookies are used with regularity by e-commerce sites that want to learn more about the habits of their customers. Cookies make buying airline

2. Steve Bellovin tells us that, although the global identifier was found, Smith was first tracked down via phone records and ISP logs.

tickets on-line faster and easier, and they remember your Amazon.com identity so you don't have to type in your username and password every time you want to buy a book. But cookies can cross the line when a single collection point is set up to link cross-Web surfing patterns. DoubleClick collects reams of surfing data about individuals across hundreds of popular Web sites, and they have publicly announced their intention to link surfing data with other demographic data. This is a direct marketer's dream, and a privacy advocate's nightmare.

Software architects and developers, along with their managers, should think carefully about what may happen to data they collect in their programs. Can the data be misused? How? Does convenience outweigh potential privacy issues?

Authentication

Authentication is held up as one of the big three security goals (the other two being confidentiality and integrity). Authentication is crucial to security because it is essential to know who to trust and who not to trust. Enforcing a security policy of almost any sort requires knowing *who* it is that is trying to do something to the *what* we want to protect.

Software security almost always includes authentication issues. Most security-critical systems require users to log in with a password before they can do anything. Then, based on the user's role in the system, he or she may be disallowed from doing certain things. Not too many years ago, PCs did very little in the way of authentication. Physical presence was good enough for the machine to let you do anything. This simplistic approach fails to work in a networked world.

Authentication on the Web is in a sorry state these days. Users tend to trust that a universal resource locator (URL) displayed on the status line means they are looking at a Web site owned by a particular business or person. But a URL is no way to foster trust! Who's to say that yourfriendlybank.com is really a bank, and is really friendly?

People falsely believe that when the little lock icon on their browser lights up that they have a "secure connection." Secure socket layer (SSL) technology uses cryptography to protect the data stream between the browser and the server to which it is connected. But from an authentication standpoint, the real question to ponder is to *whom* are you connected? Clicking on the little key may reveal a surprise.

When you buy an airline ticket from ual.com (the real United Airlines site as far as we can tell), a secure SSL connection is used. Presumably this

secures your credit card information on its travels across the Internet. But, click on the lock and you'll see that the site with which you have a secure channel is not ual.com, it's itn.net (and the certificate belongs to GetThere.com of Menlo Park, CA). Who the heck are they? Can they be trusted with your credit card data?

To abuse SSL in a Web spoofing attack (as described in *Web Spoofing* [Felten, 1997]), a bad guy must establish a secure connection with the victim's browser at the right times. Most users never bother to click the lock (and sophisticated attackers can make the resulting screens appear to say whatever they want anyway).

Authentication in software is a critical software security problem to take seriously. And there will be literally hundreds of different ways to solve it (see Chapter 3). Stored-value systems and other financial transaction systems require very strong approaches. Loyalty programs like frequent flyer programs don't. Some authentication schemes require anonymity, and others require strict and detailed auditing. Some schemes involve sessions (logging in to your computer in the morning) whereas others are geared toward transactions (payment systems).

Integrity

Last but certainly not least comes integrity. When used in a security context, integrity refers to *staying the same*. By contrast to authentication, which is all about who, when, and how, integrity is about whether something has been modified since its creation.

There are many kinds of data people rely on to be correct. Stock prices are a great example. The move toward Internet-enabled devices like WAP (Wireless Application Protocol) phones or iMode devices is often advertised with reference to real-time trading decisions made by a busy person watching the stock market on her phone as she walks through the airport or hears a stockholder's presentation. What if the data are tampered with between the stock exchange (where the information originates) and the receiver?

Stock price manipulation via misinformation is more common than you may think. Famous cases include Pairgain Technologies, whose stock was intentionally run up in 1999 by a dishonest employee, and Emulex Corporation, a fiber channel company whose stock price was similarly manipulated with fake wire stories by a junior college student in California.

Digital information is particularly easy to fake. Sometimes it can be harder to print counterfeit money than to hack a stored-value smart card

with differential power analysis (DPA) and to add some electronic blips (see the book Web site for links on DPA). The more the new economy comes to rely on information, the more critical information integrity will become.

Know Your Enemy: Common Software Security Pitfalls

There are many excellent software developers and architects in the world building all sorts of exciting things. The problem is that although these developers and architects are world class, they have not had the opportunity to learn much about security. Part of the issue is that most security courses at universities emphasize network security (that is, if the university teaches anything about security at all). Very little attention is paid to building secure software. Another part of the problem is that although some information exists covering software security, a comprehensive, practical guide to the topic has never really been available. We hope to change all that with this book.

The aphorism "Keep your friends close and your enemies closer" applies quite aptly to software security. Software security is really risk management. The key to an effective risk assessment is expert knowledge of security. Being able to recognize situations in which common attacks can be launched is half the battle. The first step in any analysis is recognizing the risks.

Software security risks come in two main flavors: architectural problems and implementation errors. We'll cover both kinds of security problems and their associated exploits throughout the book. Most software security material focuses on the implementation errors leading to problems such as buffer overflows (Chapter 7), race conditions (Chapter 9), randomness problems (Chapter 10), and a handful of other common mistakes. These issues are important, and we devote plenty of space to them.

But there is more to software security than avoiding the all-too-pervasive buffer overflow. Building secure software is like building a house. The kinds of bricks you use are important. But even more important (if you want to keep bad things out) is having four walls and a roof in the design. The same thing goes for software: The system calls that you use and how you use them are important, but overall design properties often count for more. We devote the first part of this book to issues that primarily apply at design time, including integrating security into your software engineering methodology, general principles for developing secure software systems, and dealing with security when performing security assessments.

It is important to understand what kinds of threats your software will face. At a high level, the answer is more or less any threat you can imagine from the real world. Theft, fraud, break-ins, and vandalism are all alive and well on the Internet.

Getting a bit more specific, we should mention a few of the more important types of threats of which you should be wary. One significant category of high-level threat is the compromise of information as it passes through or resides on each node in a network. Client/server models are commonly encountered in today's software systems, but things can get arbitrarily more complex. The kinds of attacks that are possible at each node are virtually limitless. We spend much of the book discussing different kinds of attacks. The attacks that can be launched vary, depending on the degree to which we trust the people at each node on the network.

One important kind of problem that isn't necessarily a software problem per se is *social engineering,* which involves talking someone into giving up important information, leading to the compromise of a system through pure charisma and chutzpah. Attackers often get passwords changed on arbitrary accounts just by calling up technical support, sounding angry, and knowing some easily obtained (usually public) information. See Ira Winkler's book *Corporate Espionage* for more information on social engineering [Winkler, 1997].

On the server side of software security, many threats boil down to *malicious input problems*. Chapter 12 is devoted to this problem. *Buffer overflows* (discussed in Chapter 7) are probably the most famous sort of malicious input problem.

Besides worrying about the nodes in a data communication topology, we have to worry about data being compromised on the actual communication medium itself. Although some people assume that such attacks aren't possible (perhaps because they don't know how to perform them), network-based attacks actually turn out to be relatively easy in practice. Some of the most notable and easiest to perform network attacks include

- **Eavesdropping.** The attacker watches data as they traverse a network. Such attacks are sometimes possible even when strong cryptography is used. See the discussion on "man-in-the-middle" attacks in Appendix A.
- **Tampering.** The attacker maliciously modifies data that are in transit on a network.
- **Spoofing.** The attacker generates phony network data to give the illusion that valid data are arriving, when in reality the data are bogus.

- **Hijacking.** The attacker replaces a stream of data on a network with his or her own stream of data. For example, once someone has authenticated a remote connection using TELNET, an attacker can take over the connection by killing packets from the client, and submitting "attack" packets. Such attacks generally involve spoofing.
- **Capture/replay.** An attacker records a stream of data, and later sends the exact same traffic in an attempt to repeat the effects, with undesirable consequences. For example, an attacker may capture a transaction in which someone sells 100 shares of Microsoft stock. If the victim had thousands more, an attacker could wait for the stock price to dip, then replay the sale ad nauseam until the target had none remaining.

Cryptography can be used to solve these network problems to a varying degree. For those without exposure to cryptography basics, we provide an overview in Appendix A. In Chapter 11, we look at the practical side of using cryptography in applications.

Our simple list of threats is by no means comprehensive. What we present are some of the most important and salient concepts we want to expose you to up front. We discuss each of these threats, and many more, in detail throughout the book.

Software Project Goals

The problem with the security goals we discussed earlier is that they directly clash with many software project goals. What are these goals like, and why are they important? And what on earth could be more important than security (ha-ha)?

We won't spend much time discussing software project goals in great detail here, because they are covered in many other places. But mentioning some of the more relevant is worth a few words.

Key software project goals include

- **Functionality.** This is the number-one driver in most software projects, which are obsessed with meeting functional requirements. And for good reason too. Software is a tool meant to solve a problem. Solving the problem invariably involves getting something done in a particular way.
- **Usability.** People like programs to be easy to use, especially when it comes to conducting electronic commerce. Usability is very important,

because without it software becomes too much of a hassle to work with. Usability affects reliability too, because human error often leads to software failure. The problem is, security mechanisms, including most uses of cryptography, elaborate login procedures, tedious audit requirements, and so on, often cause usability to plummet. Security concerns regularly impact convenience too. Security people often deride cookies, but cookies are a great user-friendly programming tool!

- **Efficiency.** People tend to want to squeeze every ounce out of their software (even when efficiency isn't needed). Efficiency can be an important goal, although it usually trades off against simplicity. Security often comes with significant overhead. Waiting around while a remote server somewhere authenticates you is no fun, but it can be necessary.
- **Time-to-market.** "Internet time" happens. Sometimes the very survival of an enterprise requires getting mind share fast in a quickly evolving area. Unfortunately, the first thing to go in a software project with severe time constraints is any attention to software risk management. Design, analysis, and testing corners are cut with regularity. This often introduces grave security risks. Fortunately, building secure software does not have to be slow. Given the right level of expertise, a secure system can sometimes be designed more quickly than an ad hoc cousin system with little or no security.
- **Simplicity.** The good thing about simplicity is that it is a good idea for both software projects *and* security. Everyone agrees that keeping it simple is good advice.

Conclusion

Computer security is a vast topic that is becoming more important because the world is becoming highly interconnected, with networks being used to carry out critical transactions. The environment in which machines must survive has changed radically since the popularization of the Internet. Deciding to connect a local area network (LAN) to the Internet is a security-critical decision. The root of most security problems is software that fails in unexpected ways. Although software security as a field has much maturing to do, it has much to offer to those practitioners interested in striking at the heart of security problems. The goal of this book is to familiarize you with the current best practices for keeping security flaws out of your software.

Good software security practices can help ensure that software behaves properly. Safety-critical and high-assurance system designers have always taken great pains to analyze and to track software behavior. Security-critical system designers must follow suit. We can avoid the Band-Aid-like penetrate-and-patch approach to security only by considering security as a crucial system property. This requires integrating software security into your entire software engineering process—a topic that we take up in the next chapter.