Transmission and Propagation

A printed copy of code does nothing and threatens no one. Even executable code sitting on a disk does nothing. What triggers code to start? For malware to do its malicious work and spread itself, it must be executed to be activated. Fortunately for malware writers but unfortunately for the rest of us, there are many ways to ensure that programs will be executed on a running computer.

Setup and Installer Program Transmission

Recall the SETUP program that you run to load and install a new program on your computer. It may call dozens or hundreds of other programs, some on the distribution medium, some already residing on the computer, some in memory. If any one of these programs contains a virus, the virus code could be activated. Let us see how. Suppose the virus code were in a program on the distribution medium, such as a CD, or downloaded in the installation package; when executed, the virus could install itself on a permanent storage medium (typically, a hard disk) and also in any and all executing programs in memory. Human intervention is necessary to start the process; a human being puts the virus on the distribution medium, and perhaps another person initiates the execution of the program to which the virus is attached. (Execution can occur without human intervention, though, such as when execution is triggered by a date or the passage of a certain amount of time.) After that, no human intervention is needed; the virus can spread by itself.

Attached File

A more common means of virus activation is in a file attached to an email message or embedded in a file. In this attack, the virus writer tries to convince the victim (the recipient of the message or file) to open the object. Once the viral object is opened (and thereby executed), the activated virus can do its work. Some modern email handlers, in a drive to "help" the receiver (victim), automatically open attachments as soon as the receiver opens the body of the email message. The virus can be executable code embedded in an executable attachment, but other types of files are equally dangerous. For example, objects such as graphics or photo images can contain code to be executed by an editor, so they can be transmission agents for viruses. In general, forcing users to open files on their own rather than having an application do it automatically is a best practice; programs should not perform potentially security-relevant actions without a user's consent. However, ease-of-use often trumps security, so programs such as browsers, email handlers, and viewers often "helpfully" open files without first asking the user.

Document Viruses

A virus type that used to be quite popular is what we call the document virus, which is implemented within a formatted document, such as a written document, a database, a slide presentation, a picture, or a spreadsheet. These documents are highly structured files that contain both data (words or numbers) and commands (such as formulas, formatting controls, links). The commands are part of a rich programming language, including macros, variables and procedures, file accesses, and even system calls. The writer of a document virus uses any of the features of the programming language to perform malicious actions.

The ordinary user usually sees only the content of the document (its text or data), so the virus writer simply includes the virus in the commands part of the document, as in the integrated program virus.

Autorun

Autorun is a feature of operating systems that causes the automatic execution of code based on name or placement. An early autorun program was the DOS file autoexec.bat, a script file located at the highest directory level of a startup disk. As the system began execution, it would automatically execute autoexec.bat, so a goal of early malicious code writers was to augment or replace autoexec.bat to get the malicious code executed. Similarly, in Unix, files such as .cshrc and .profile are automatically processed at system startup (depending on version).

In Windows, the registry contains several lists of programs automatically invoked at startup, some readily apparent (in the start menu/programs/startup list) and others more hidden (for example, in the registry key software\windows\current_version\run).

One popular technique for transmitting malware is distribution via flash memory, such as a solid state USB memory stick. People love getting something for free, and handing out infected memory devices is a relatively low cost way to spread an infection. Although the spread has to be done by hand (handing out free drives as advertising at a railway station, for example), the personal touch does add to credibility: We would be suspicious of an attachment from an unknown person, but some people relax their guards for something received by hand from another person.

Propagation

Since a virus can be rather small, its code can be "hidden" inside other larger and more complicated programs. Two hundred lines of a virus could be separated into one hundred packets of two lines of code and a jump each; these one hundred packets could be easily hidden inside a compiler, a database manager, a file manager, or some other large utility.

Appended Viruses

A program virus attaches itself to a program; then, whenever the program is run, the virus is activated. This kind of attachment is usually easy to design and implement.

In the simplest case, a virus inserts a copy of itself into the executable program file before the first executable instruction. Then, all the virus instructions execute first; after the last virus instruction, control flows naturally to what used to be the first program instruction. Such a situation is shown in Figure 3-19.

This kind of attachment is simple and usually effective. The virus writer need not know anything about the program to which the virus will attach, and often the attached program simply serves as a carrier for the virus. The virus performs its task and then transfers to the original program. Typically, the user is unaware of the effect of the virus if the original program still does all that it used to. Most viruses attach in this manner.

Viruses That Surround a Program

An alternative to the attachment is a virus that runs the original program but has control before and after its execution. For example, a virus writer might want to prevent

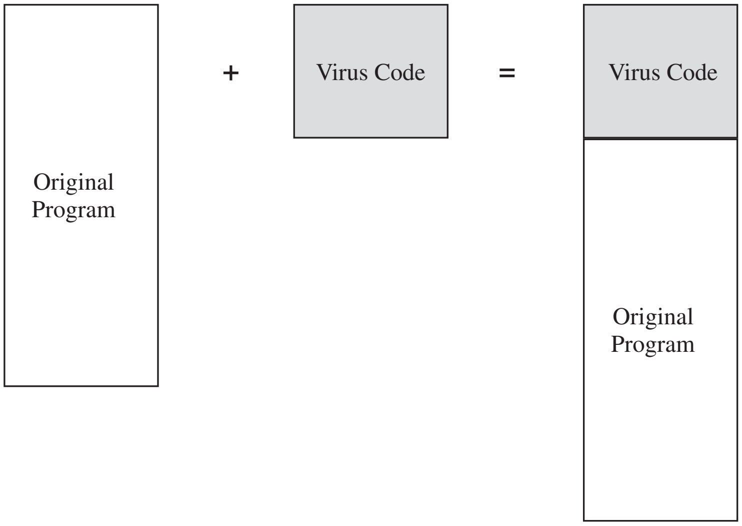


FIGURE 3-19 Virus Attachment

the virus from being detected. If the virus is stored on disk, its presence will be given away by its file name, or its size will affect the amount of space used on the disk. The virus writer might arrange for the virus to attach itself to the program that constructs the listing of files on the disk. If the virus regains control after the listing program has generated the listing but before the listing is displayed or printed, the virus could eliminate its entry from the listing and falsify space counts so that it appears not to exist. A surrounding virus is shown in Figure 3-20.

Integrated Viruses and Replacements

A third situation occurs when the virus replaces some of its target, integrating itself into the original code of the target. Such a situation is shown in Figure 3-21. Clearly, the virus writer has to know the exact structure of the original program to know where to insert which pieces of the virus.

Finally, the malicious code can replace an entire target, either mimicking the effect of the target or ignoring its expected effect and performing only the virus effect. In this case, the user may perceive the loss of the original program.

Activation

Early malware writers used document macros and scripts as the vector for introducing malware into an environment. Correspondingly, users and designers tightened controls on macros and scripts to guard in general against malicious code, so malware writers had to find other means of transferring their code.

Malware now often exploits one or more existing vulnerabilities in a commonly used program. For example, the Code Red worm of 2001 exploited an older buffer overflow program flaw in Microsoft's Internet Information Server (IIS), and Conficker.A exploited a flaw involving a specially constructed remote procedure call (RPC) request.

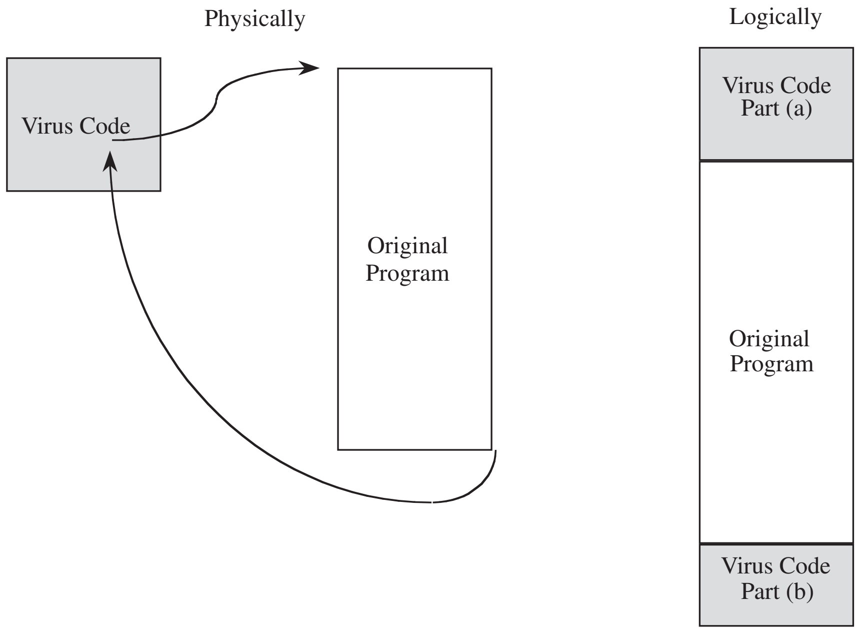


FIGURE 3-20 Surrounding Virus

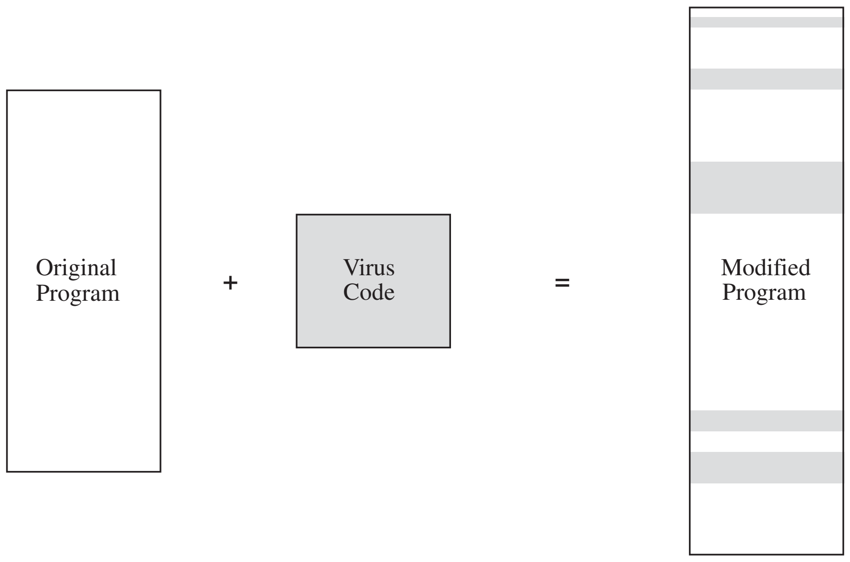


FIGURE 3-21 Virus Insertion

Although the malware writer usually must find a vulnerability and hope the intended victim has not yet applied a protective or corrective patch, each vulnerability represents a new opening for wreaking havoc against all users of a product.

Is it better to disclose a flaw and alert users that they are vulnerable or conceal it until there is a countermeasure? There is no easy answer.

Flaws happen, in spite of the best efforts of development teams. Having discovered a flaw, a security researcher—or a commercial software vendor—faces a dilemma: Announce the flaw (for which there may not yet be a patch) and alert malicious code writers of yet another vulnerability to attack, or keep quiet and hope the malicious code writers have not yet discovered the flaw. As Sidebar 3-7 describes, a vendor who cannot release an effective patch will want to limit disclosure. If one attacker finds the vulnerability, however, word will spread quickly through the underground attackers' network. Competing objectives make vulnerability disclosure a difficult issue.

SIDEBAR 3-7 Just Keep It a Secret and It's Not There

In July 2005, security researcher Michael Lynn presented information to the Black Hat security conference. As a researcher for Internet Security Systems (ISS), he had discovered what he considered serious vulnerabilities in the underlying operating system IOS on which Cisco based most of its firewall and router products. ISS had made Cisco aware of the vulnerabilities a month before the presentation, and the two companies had been planning a joint talk there but canceled it.

Concerned that users were in jeopardy because the vulnerability could be discovered by attackers, Lynn presented enough details of the vulnerability for users to appreciate its severity. ISS had tried to block Lynn's presentation or remove technical details, but he resigned from ISS rather than be muzzled. Cisco tried to block the presentation, as well, demanding that 20 pages be torn from the conference proceedings. Various sites posted the details of the presentation, lawsuits ensued, and the copies were withdrawn in settlement of the suits. The incident was a public relations fiasco for both Cisco and ISS. (For an overview of the facts of the situation, see Bank [BAN05].)

The issue remains: How far can or should a company go to limit vulnerability disclosure? On the one hand, a company wants to limit disclosure, while on the other hand users should know of a potential weakness that might affect them. Researchers fear that companies will not act quickly to close vulnerabilities, thus leaving customers at risk. Regardless of the points, the legal system may not always be the most effective way to address disclosure.

Computer security is not the only domain in which these debates arise. Matt Blaze, a computer security researcher with AT&T Labs, investigated physical locks and master keys [BLA03]; these are locks for structures such as college dormitories and office buildings, in which individuals have keys to single rooms, and a few maintenance or other workers have a single master key that opens all locks. Blaze describes a technique that can find a master key for a class of locks with relatively little effort because of a characteristic (vulnerability?) of these locks; the attack finds the master key one pin at a time. According to Schneier [SCH03] and Blaze, the characteristic was well known to locksmiths and lock-picking criminals, but not to the general public (users). A respected cryptographer, Blaze came upon his strategy naturally: His approach is analogous to a standard cryptologic attack in which one seeks to deduce the cryptographic key one bit at a time.

Blaze confronted an important question: Is it better to document a technique known by manufacturers and attackers but not to users, or to leave users with a false sense of security? He opted for disclosure. Schneier notes that this weakness has been known for over 100 years and that several other master key designs are immune from Blaze's attack. But those locks are not in widespread use because customers are unaware of the risk and thus do not demand stronger products. Says Schneier, "I'd rather have as much information as I can to make informed decisions about security."

When an attacker finds a vulnerability to exploit, the next step is using that vulnerability to further the attack. Next we consider how malicious code gains control as part of a compromise.

How Malicious Code Gains Control

To gain control of processing, malicious code such as a virus (V) has to be invoked instead of the target (T). Essentially, the virus either has to seem to be T, saying effectively "I am T," or the virus has to push T out of the way and become a substitute for T, saying effectively "Call me instead of T." A more blatant virus can simply say "invoke me [you fool]."

The virus can assume T's name by replacing (or joining to) T's code in a file structure; this invocation technique is most appropriate for ordinary programs. The virus can overwrite T in storage (simply replacing the copy of T in storage, for example). Alternatively, the virus can change the pointers in the file table so that the virus is located instead of T whenever T is accessed through the file system. These two cases are shown in Figure 3-22.

The virus can supplant T by altering the sequence that would have invoked T to now invoke the virus V; this invocation can replace parts of the resident operating system by modifying pointers to those resident parts, such as the table of handlers for different kinds of interrupts.

Embedding: Homes for Malware

The malware writer may find it appealing to build these qualities into the malware:

- The malicious code is hard to detect.
- The malicious code is not easily destroyed or deactivated.
- The malicious code spreads infection widely.
- The malicious code can reinfect its home program or other programs.

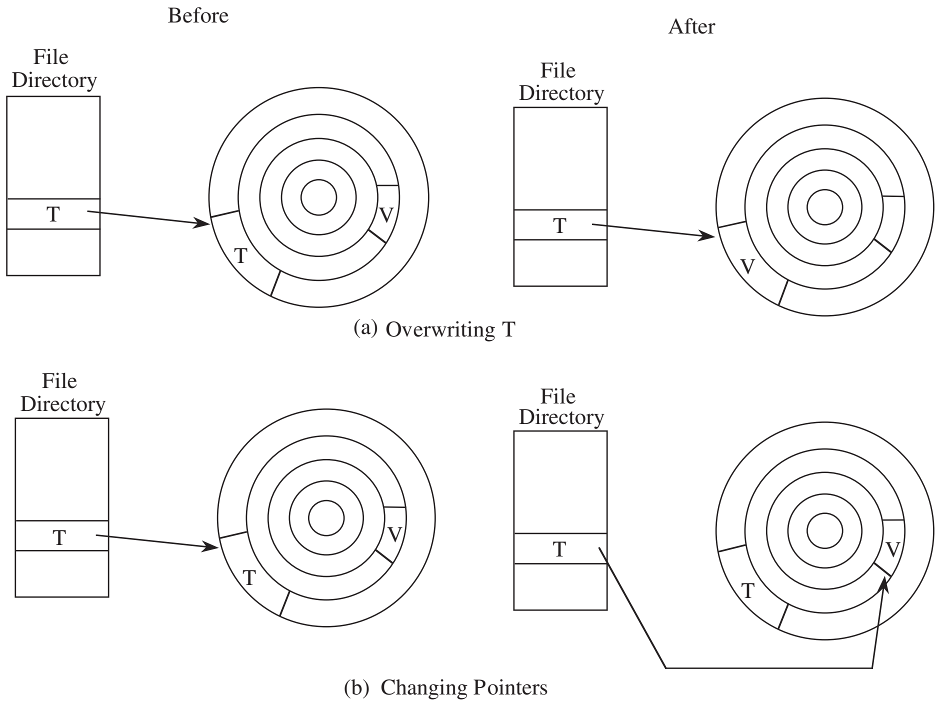


FIGURE 3-22 Virus V Replacing Target T

- The malicious code is easy to create.
- The malicious code is machine independent and operating system independent.

Few examples of malware meet all these criteria. The writer chooses from these objectives when deciding what the code will do and where it will reside.

Just a few years ago, the challenge for the virus writer was to write code that would be executed repeatedly so that the virus could multiply. Now, however, one execution is usually enough to ensure widespread distribution. Many kinds of malware are transmitted by email. For example, some examples of malware generate a new email message to all addresses in the victim's address book. These new messages contain a copy of the malware so that it propagates widely. Often the message is a brief, chatty, nonspecific message that would encourage the new recipient to open the attachment from a friend (the first recipient). For example, the subject line or message body may read "I thought you might enjoy this picture from our vacation."

One-Time Execution (Implanting)

Malicious code often executes a one-time process to transmit or receive and install the infection. Sometimes the user clicks to download a file, other times the user opens an attachment, and other times the malicious code is downloaded silently as a web page is displayed. In any event, this first step to acquire and install the code must be quick and not obvious to the user.

Boot Sector Viruses

A special case of virus attachment, but formerly a fairly popular one, is the so-called boot sector virus. Attackers are interested in creating continuing or repeated harm, instead of just a one-time assault. For continuity the infection needs to stay around and become an integral part of the operating system. In such attackers, the easy way to become permanent is to force the harmful code to be reloaded each time the system is restarted. Actually, a similar technique works for most types of malicious code, so we first describe the process for viruses and then explain how the technique extends to other types.

When a computer is started, control begins with firmware that determines which hardware components are present, tests them, and transfers control to an operating system. A given hardware platform can run many different operating systems, so the operating system is not coded in firmware but is instead invoked dynamically, perhaps even by a user's choice, after the hardware test.

Modern operating systems consist of many modules; which modules are included on any computer depends on the hardware of the computer and attached devices, loaded software, user preferences and settings, and other factors. An executive oversees the boot process, loading and initiating the right modules in an acceptable order. Putting together a jigsaw puzzle is hard enough, but the executive has to work with pieces from many puzzles at once, somehow putting together just a few pieces from each to form a consistent, connected whole, without even a picture of what the result will look like when it is assembled. Some people see flexibility in such a wide array of connectable modules; others see vulnerability in the uncertainty of which modules will be loaded and how they will interrelate.

Malicious code can intrude in this bootstrap sequence in several ways. An assault can revise or add to the list of modules to be loaded, or substitute an infected module for a good one by changing the address of the module to be loaded or by substituting a modified routine of the same name. With boot sector attacks, the assailant changes the pointer to the next part of the operating system to load, as shown in Figure 3-23.

The boot sector is an especially appealing place to house a virus. The virus gains control early in the boot process, before most detection tools are active, so that it can avoid, or at least complicate, detection. The files in the boot area are crucial parts of the operating system. Consequently, to keep users from accidentally modifying or deleting them with disastrous results, the operating system makes them "invisible" by not showing them as part of a normal listing of stored files, thereby preventing their deletion. Thus, the virus code is not readily noticed by users.

Operating systems have gotten large and complex since the first viruses. The boot process is still the same, but many more routines are activated during the boot process; many programs—often hundreds of them—run at startup time. The operating system, device handlers, and other necessary applications are numerous and have unintelligible names, so malicious code writers do not need to hide their code completely; probably a user even seeing a file named malware.exe, would more likely think the file a joke than some real malicious code. Burying the code among other system routines and placing the code on the list of programs started at computer startup are current techniques to ensure that a piece of malware is reactivated.

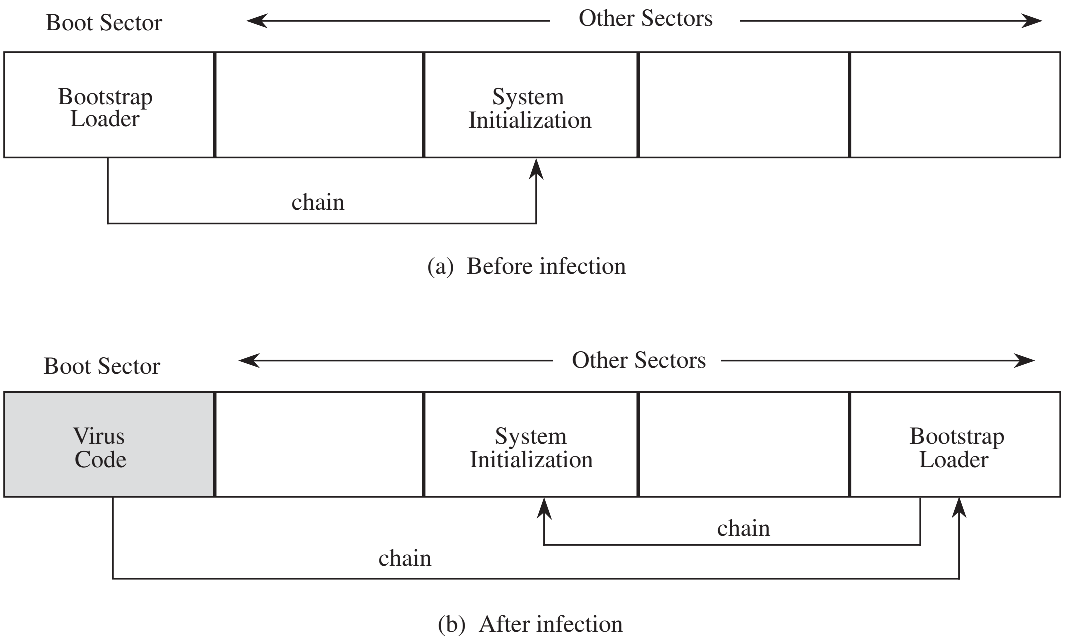


FIGURE 3-23 Boot or Initialization Time Virus

Memory-Resident Viruses

Some parts of the operating system and most user programs execute, terminate, and disappear, with their space in memory then being available for anything executed later. For frequently used parts of the operating system and for a few specialized user programs, it would take too long to reload the program each time it is needed. Instead, such code remains in memory and is called "resident" code. Examples of resident code are the routine that interprets keys pressed on the keyboard, the code that handles error conditions that arise during a program's execution, or a program that acts like an alarm clock, sounding a signal at a time the user determines. Resident routines are sometimes called TSRs or "terminate and stay resident" routines.

Virus writers also like to attach viruses to resident code because the resident code is activated many times while the machine is running. Each time the resident code runs, the virus does too. Once activated, the virus can look for and infect uninfected carriers. For example, after activation, a boot sector virus might attach itself to a piece of resident code. Then, each time the virus was activated, it might check whether any removable disk in a disk drive was infected and, if not, infect it. In this way the virus could spread its infection to all removable disks used during the computing session.

A virus can also modify the operating system's table of programs to run. Once the virus gains control, it can insert a registry entry so that it will be reinvoked each time the system restarts. In this way, even if the user notices and deletes the executing copy of the virus from memory, the system will resurrect the virus on the next system restart.

For general malware, executing just once from memory has the obvious disadvantage of only one opportunity to cause malicious behavior, but on the other hand, if the infectious code disappears whenever the machine is shut down, the malicious code is less likely to be analyzed by security teams.

Other Homes for Viruses

A virus that does not take up residence in one of these cozy establishments has to fend for itself. But that is not to say that the virus will go homeless.

You might think that application programs—code—can do things, but that data files—documents, spreadsheets, document image PDF files, or pictures—are passive objects that cannot do harmful things. In fact, however, these structured data files contain commands to display and manipulate their data. Thus, a PDF file is displayed by a program such as Adobe Reader that does many things in response to commands in the PDF file. Although such a file is not executable as a program itself, it can cause activity in the program that handles it. Such a file is called **interpretive data**, and the handler program is also called an **interpreter**. The Adobe Reader program is an interpreter for PDF files. If there is a flaw in the PDF interpreter or the semantics of the PDF interpretive language, opening a PDF file can cause the download and execution of malicious code. So even an apparently passive object like a document image can lead to a malicious code infection.

One popular home for a virus is an application program. Many applications, such as word processors and spreadsheets, have a "macro" feature, by which a user can record a series of commands and then repeat the entire series with one invocation. Such programs also provide a "startup macro" that is executed every time the application is executed. A virus writer can create a virus macro that adds itself to the startup directives for the application. It also then embeds a copy of itself in data files so that the infection spreads to anyone receiving one or more of those files. Thus, the virus writer effectively adds malware to a trusted and commonly used application, thereby assuring repeated activations of the harmful addition.

Code libraries are also excellent places for malicious code to reside. Because libraries are used by many programs, the code in them will have a broad effect. Additionally, libraries are often shared among users and transmitted from one user to another, a practice that spreads the infection. Finally, executing code in a library can pass on the viral infection to other transmission media. Compilers, loaders, linkers, runtime monitors, runtime debuggers, and even virus control programs are good candidates for hosting viruses because they are widely shared.

Stealth

The final objective for a malicious code writer is stealth: avoiding detection during installation, while executing, or even at rest in storage.

Most viruses maintain stealth by concealing their action, not announcing their presence, and disguising their appearance.

Detection

Malicious code discovery could be aided with a procedure to determine if two programs are equivalent: We could write a program with a known harmful effect, and then compare with any other suspect program to determine if the two have equivalent results. However, this equivalence problem is complex, and theoretical results in computing suggest that a general solution is unlikely. In complexity theory, we say that the general question "Are these two programs equivalent?" is undecidable (although that question *can* be answered for many specific pairs of programs).

Even if we ignore the general undecidability problem, we must still deal with a great deal of uncertainty about what equivalence means and how it affects security. Two modules may be practically equivalent but produce subtly different results that may—or may not—be security relevant. One may run faster, or the first may use a temporary file for workspace, whereas the second performs all its computations in memory. These differences could be benign, or they could be a marker of an infection. Therefore, we are unlikely to develop a screening program that can separate infected modules from uninfected ones.

Although the general case is dismaying, the particular is not. If we know that a particular virus may infect a computing system, we can check for its "signature" and detect it if it is there. Having found the virus, however, we are left with the task of cleansing the system of it. Removing the virus in a running system requires being able to detect and eliminate its instances faster than it can spread.

The examples we have just given describe several ways in which malicious code arrives at a target computer, but they do not answer the question of how the code is first executed and continues to be executed. Code from a web page can simply be injected into the code the browser executes, although users' security settings within browsers may limit what that code can do. More generally, however, code writers try to find ways to associate their code with existing programs, in ways such as we describe here, so that the "bad" code executes whenever the "good" code is invoked.

Installation Stealth

We have described several approaches used to transmit code without the user's being aware, including downloading as a result of loading a web page and advertising one function while implementing another. Malicious code designers are fairly competent at tricking the user into accepting malware.

Execution Stealth

Similarly, remaining unnoticed during execution is not too difficult. Modern operating systems often support dozens of concurrent processes, many of which have unrecognizable names and functions. Thus, even if a user does notice a program with an unrecognized name, the user is more likely to accept it as a system program than malware.

Stealth in Storage

If you write a program to distribute to others, you will give everyone a copy of the same thing. Except for some customization (such as user identity details or a product serial number) your routine will be identical to everyone else's. Even if you have different versions, you will probably structure your code in two sections: as a core routine for everyone and some smaller modules specific to the kind of user—home user, small business professional, school personnel, or large enterprise customer. Designing your code this way is the economical approach for you: Designing, coding, testing, and maintaining one entity for many customers is less expensive than doing that for each individual sale. Your delivered and installed code will then have sections of identical instructions across all copies.

Antivirus and other malicious code scanners look for patterns because malware writers have the same considerations you would have in developing mass-market software: They want to write one body of code and distribute it to all their victims. That identical code becomes a pattern on disk for which a scanner can search quickly and efficiently.

Knowing that scanners look for identical patterns, malicious code writers try to vary the appearance of their code in several ways:

- Rearrange the order of modules.
- Rearrange the order of instructions (when order does not affect execution; for example A := 1; B := 2 can be rearranged with no detrimental effect).
- Insert instructions, (such as A := A), that have no impact.
- Insert random strings (perhaps as constants that are never used).
- Replace instructions with others of equivalent effect, such as replacing A := B −1 with A := B + (−1) or A := B + 2 − 1.
- Insert instructions that are never executed (for example, in the *else* part of a conditional expression that is always true).

These are relatively simple changes for which a malicious code writer can build a tool, producing a unique copy for every user. Unfortunately (for the code writer), even with a few of these changes on each copy, there will still be recognizable identical sections. We discuss this problem for the malware writer later in this chapter as we consider virus scanners as countermeasures to malicious code.

Now that we have explored the threat side of malicious code, we turn to vulnerabilities. As we showed in Chapter 1, a threat is harmless without a vulnerability it can exploit. Unfortunately, exploitable vulnerabilities abound for malicious code.

Introduction of Malicious Code

The easiest way for malicious code to gain access to a system is to be introduced by a user, a system owner, an administrator, or other authorized agent.

The only way to prevent the infection of a virus is not to receive executable code from an infected source. This philosophy used to be easy to follow because it was easy to tell if a file was executable or not. For example, on PCs, a *.exe* extension was a clear sign that the file was executable. However, as we have noted, today's files are more complex, and a seemingly nonexecutable file with a *.doc* extension may have some executable code buried deep within it. For example, a word processor may have commands within the document file. As we noted earlier, these commands, called macros, make it easy for the user to do complex or repetitive things, but they are really executable code embedded in the context of the document. Similarly, spreadsheets, presentation slides, other office or business files, and even media files can contain code or scripts that can be executed in various ways—and thereby harbor viruses. And, as we have seen, the applications that run or use these files may try to be helpful by automatically invoking the executable code, whether you want it to run or not! Against the principles of good security, email handlers can be set to automatically open (without performing access control) attachments or embedded code for the recipient, so your email message can have animated bears dancing across the top.

Another approach virus writers have used is a little-known feature in the Microsoft file design that deals with file types. Although a file with a *.doc* extension is expected to be a Word document, in fact, the true document type is hidden in a field at the start of the file. This convenience ostensibly helps a user who inadvertently names a Word document with a *.ppt* (PowerPoint) or any other extension. In some cases, the operating system will try to open the associated application but, if that fails, the system will switch to the application of the hidden file type. So, the virus writer creates an executable file, names it with an inappropriate extension, and sends it to the victim, describing it as a picture or a necessary code add-in or something else desirable. The unwitting recipient opens the file and, without intending to, executes the malicious code.

More recently, executable code has been hidden in files containing large data sets, such as pictures or read-only documents, using a process called steganography. These bits of viral code are not easily detected by virus scanners and certainly not by the human eye. For example, a file containing a photograph may be highly detailed, often at a resolution of 600 or more points of color (called pixels) per inch. Changing every sixteenth pixel will scarcely be detected by the human eye, so a virus writer can conceal the machine instructions of the virus in a large picture image, one bit of code for every sixteen pixels.

Steganography permits data to be hidden in large, complex, redundant data sets.

Execution Patterns

A virus writer may want a virus to do several things at the same time, namely, spread infection, avoid detection, and cause harm. These goals are shown in Table 3-4, along with ways each goal can be addressed. Unfortunately, many of these behaviors are perfectly normal and might otherwise go undetected. For instance, one goal is modifying the file directory; many normal programs create files, delete files, and write to storage media. Thus, no key signals point to the presence of a virus.

Most virus writers seek to avoid detection for themselves and their creations. Because a disk's boot sector is not visible to normal operations (for example, the contents of the boot sector do not show on a directory listing), many virus writers hide their code there. A resident virus can monitor disk accesses and fake the result of a disk operation that would show the virus hidden in a boot sector by showing the data that *should* have been in the boot sector (which the virus has moved elsewhere).

There are no limits to the harm a virus can cause. On the modest end, the virus might do nothing; some writers create viruses just to show they can do it. Or the virus can be relatively benign, displaying a message on the screen, sounding the buzzer, or playing music. From there, the problems can escalate. One virus can erase files, another an entire disk; one virus can prevent a computer from booting, and another can prevent writing to disk. The damage is bounded only by the creativity of the virus's author.

Transmission Patterns

A virus is effective only if it has some means of transmission from one location to another. As we have already seen, viruses can travel during the boot process by attaching

TABLE 3-4 Virus Effects and What They Cause

Virus Effect	How It Is Caused
Attach to executable program	• Modify file directory • Write to executable program file
Attach to data or control file	• Modify directory • Rewrite data • Append to data • Append data to self
Remain in memory	• Intercept interrupt by modifying interrupt handler address table • Load self in nontransient memory area
Infect disks	• Intercept interrupt • Intercept operating system call (to format disk, for example) • Modify system file • Modify ordinary executable program
Conceal self	• Intercept system calls that would reveal self and falsify result • Classify self as "hidden" file
Spread infection	• Infect boot sector • Infect system program • Infect ordinary program • Infect data ordinary program reads to control its execution
Prevent deactivation	• Activate before deactivating program and block deactivation • Store copy to reinfect after deactivation

to an executable file or traveling within data files. The travel itself occurs during execution of an already infected program. Since a virus can execute any instructions a program can, virus travel is not confined to any single medium or execution pattern. For example, a virus can arrive on a diskette or from a network connection, travel during its host's execution to a hard disk boot sector, reemerge next time the host computer is booted, and remain in memory to infect other diskettes as they are accessed.

Polymorphic Viruses

The virus signature may be the most reliable way for a virus scanner to identify a virus. If a particular virus always begins with the string 0x47F0F00E08 and has string 0x00113FFF located at word 12, other programs or data files are not likely to have these exact characteristics. For longer signatures, the probability of a correct match increases.

If the virus scanner will always look for those strings, then the clever virus writer can cause something other than those strings to be in those positions. Certain instructions cause no effect, such as adding 0 to a number, comparing a number to itself, or jumping to the next instruction. These instructions, sometimes called *no-ops* (for "no operation"), can be sprinkled into a piece of code to distort any pattern. For example, the virus could have two alternative but equivalent beginning words; after being installed, the virus will choose one of the two words for its initial word. Then, a virus scanner would have to look for both patterns. A virus that can change its appearance is called a **polymorphic virus**. (*Poly* means "many" and *morph* means "form.")

A two-form polymorphic virus can be handled easily as two independent viruses. Therefore, the virus writer intent on preventing detection of the virus will want either a large or an unlimited number of forms so that the number of possible forms is too large for a virus scanner to search for. Simply embedding a random number or string at a fixed place in the executable version of a virus is not sufficient, because the signature of the virus is just the unvaried instructions, excluding the random part. A polymorphic virus has to randomly reposition all parts of itself and randomly change all fixed data. Thus, instead of containing the fixed (and therefore searchable) string "HA! INFECTED BY A VIRUS," a polymorphic virus has to change even that pattern sometimes.

Trivially, assume a virus writer has 100 bytes of code and 50 bytes of data. To make two virus instances different, the writer might distribute the first version as 100 bytes of code followed by all 50 bytes of data. A second version could be 99 bytes of code, a jump instruction, 50 bytes of data, and the last byte of code. Other versions are 98 code bytes jumping to the last two, 97 and three, and so forth. Just by moving pieces around, the virus writer can create enough different appearances to fool simple virus scanners. Once the scanner writers became aware of these kinds of tricks, however, they refined their signature definitions and search techniques.

A simple variety of polymorphic virus uses encryption under various keys to make the stored form of the virus different. These are sometimes called **encrypting viruses**. This type of virus must contain three distinct parts: a decryption key, the (encrypted) object code of the virus, and the (unencrypted) object code of the decryption routine. For these viruses, the decryption routine itself or a call to a decryption library routine must be in the clear, and so that becomes the signature. (See [PFL10d] for more on virus writers' use of encryption.)

To avoid detection, not every copy of a polymorphic virus has to differ from every other copy. If the virus changes occasionally, not every copy will match a signature of every other copy.

Because you cannot always know which sources are infected, you should assume that any outside source is infected. Fortunately, you know when you are receiving code from an outside source; unfortunately, cutting off all contact with the outside world is not feasible. Malware seldom comes with a big warning sign and, in fact, as Sidebar 3-8 shows, malware is often designed to fool the unsuspecting.

SIDEBAR 3-8 Malware Non-Detector

In May 2010, the United States issued indictments against three men charged with deceiving people into believing their computers had been infected with malicious code [FBI10]. The three men set up computer sites that would first report false and misleading computer error messages and then indicate that the users' computers were infected with various forms of malware.

According to the indictment, after the false error messages were transmitted, the sites then induced Internet users to purchase software products bearing such names as "DriveCleaner" and "ErrorSafe," ranging in price from approximately $30 to $70, that the web sites claimed would rid the victims' computers of the infection, but actually did little or nothing to improve or repair computer performance. The U.S. Federal Bureau of Investigation (FBI) estimated that the sites generated over $100 million for the perpetrators of the fraud.

The perpetrators allegedly enabled the fraud by establishing advertising agencies that sought legitimate client web sites on which to host advertisements. When a victim user went to the client's site, code in the malicious web advertisement hijacked the user's browser and generated the false error messages. The user was then redirected to what is called a **scareware** web site, to scare users about a computer security weakness. The site then displayed a graphic purporting to monitor the scanning of the victim's computer for malware, of which (not surprisingly) it found a significant amount. The user was then invited to click to download a free malware eradicator, which would appear to fix only a few vulnerabilities and would then request the user to upgrade to a paid version to repair the rest.

Two of the three indicted are U.S. citizens, although one was believed to be living in Ukraine; the third was Swedish and believed to be living in Sweden. All were charged with wire fraud and computer fraud. The three ran a company called Innovative Marketing that was closed under action by the U.S. Federal Trade Commission (FTC), alleging the sale of fraudulent anti-malware software, between 2003 and 2008.

The advice for innocent users seems to be both "trust but verify" and "if it ain't broke; don't fix it." That is, if you are being lured into buying security products, your skeptical self should first run your own trusted malware scanner to verify that there is indeed malicious code lurking on your system.

As we saw in Sidebar 3-8, there may be no better way to entice a security-conscious user than to offer a free security scanning tool. Several legitimate antivirus scanners, including ones from the Anti-Virus Group (AVG) and Microsoft, are free. However, other scanner offers provide malware, with effects ranging from locking up a computer to demanding money to clean up nonexistent infections. As with all software, be careful acquiring software from unknown sources.

Natural Immunity

In their interesting paper comparing computer virus transmission with human disease transmission, Kephart et al. [KEP93] observe that individuals' efforts to keep their computers free from viruses lead to communities that are generally free from viruses because members of the community have little (electronic) contact with the outside world. In this case, transmission is contained not because of limited contact but because of limited contact outside the community, much as isolated human communities seldom experience outbreaks of communicable diseases such as measles.

For this reason, governments often run disconnected network communities for handling top military or diplomatic secrets. The key to success seems to be choosing one's community prudently. However, as use of the Internet and the World Wide Web increases, such separation is almost impossible to maintain. Furthermore, in both human and computing communities, natural defenses tend to be lower, so if an infection does occur, it often spreads unchecked. Human computer users can be naïve, uninformed, and lax, so the human route to computer infection is likely to remain important.

Malware Toolkits

A bank robber has to learn and practice the trade all alone. There is no *Bank Robbing for Dummies* book (at least none of which we are aware), and a would-be criminal cannot send off a check and receive a box containing all the necessary tools. There seems to be a form of apprenticeship as new criminals work with more experienced ones, but this is a difficult, risky, and time-consuming process, or at least it seems that way to us outsiders.

Computer attacking is somewhat different. First, there is a thriving underground of web sites for hackers to exchange techniques and knowledge. (As with any web site, the reader has to assess the quality of the content.) Second, attackers can often experiment in their own laboratories (homes) before launching public strikes. Most importantly, malware toolkits are readily available for sale. A would-be assailant can acquire, install, and activate one of these as easily as loading and running any other software; using one is easier than many computer games. Such a toolkit takes as input a target address and, when the user presses the [Start] button, it launches a probe for a range of vulnerabilities. Such toolkit users, who do not need to understand the vulnerabilities they seek to exploit, are known as script kiddies. As we noted earlier in this chapter, these toolkits often exploit old vulnerabilities for which defenses have long been publicized. Still, these toolkits are effective against many victims.

Malware toolkits let novice attackers probe for many vulnerabilities at the press of a button.

Ease of use means that attackers do not have to understand, much less create, their own attacks. For this reason, it would seem as if offense is easier than defense in computer security, which is certainly true. Remember that the defender must protect against all possible threats, but the assailant only has to find one uncovered vulnerability.

3.3 COUNTERMEASURES

So far we have described the techniques by which malware writers can transmit, conceal, and activate their evil products. If you have concluded that these hackers are clever, crafty, diligent, and devious, you are right. And they never seem to stop working. Antivirus software maker McAfee reports identifying 200 distinct, new pieces of malware *per minute*. At the start of 2012 their malware library contained slightly fewer than 100 million items and by the end of 2013 it had over 196 million [MCA14].

Faced with such a siege, users are hard pressed to protect themselves, and the security defense community in general is strained. However, all is not lost. The available countermeasures are not perfect, some are reactive—after the attack succeeds—rather than preventive, and all parties from developers to users must do their part. In this section we survey the countermeasures available to keep code clean and computing safe.

We organize this section by who must take action: users or developers, and then we add a few suggestions that seem appealing but simply do not work.

Countermeasures for Users

Users bear the most harm from malware infection, so users have to implement the first line of protection. Users can do this by being skeptical of all code, with the degree of skepticism rising as the source of the code becomes less trustworthy.

User Vigilance

The easiest control against malicious code is hygiene: not engaging in behavior that permits malicious code contamination. The two components of hygiene are avoiding points of contamination and blocking avenues of vulnerability.

To avoid contamination, you could simply not use your computer systems—not a realistic choice in today's world. But, as with preventing colds and the flu, there are several techniques for building a reasonably safe community for electronic contact, including the following:

- *Use only commercial software acquired from reliable, well-established vendors*. There is always a chance that you might receive a virus from a large manufacturer with a name everyone would recognize. However, such enterprises have significant reputations that could be seriously damaged by even one bad incident, so they go to some degree of trouble to keep their products virus free and to patch any problem-causing code right away. Similarly, software distribution companies will be careful about products they handle.
- *Test all new software on an isolated computer*. If you must use software from a questionable source, test the software first on a computer that is not connected to a network and contains no sensitive or important data. Run the software and look for unexpected behavior, even simple behavior such as unexplained figures on the screen. Test the computer with a copy of an up-to-date virus scanner created before the suspect program is run. Only if the program passes these tests should you install it on a less isolated machine.
- *Open attachments—and other potentially infected data files—only when you know them to be safe*. What constitutes "safe" is up to you, as you have probably already learned in this chapter. Certainly, an attachment from an unknown source is of questionable safety. You might also distrust an attachment from a known source but with a peculiar message or description.
- *Install software—and other potentially infected executable code files—only when you really, really know them to be safe.* When a software package asks to install software on your system (including plug-ins or browser helper objects), be really suspicious.
- *Recognize that any web site can be potentially harmful.* You might reasonably assume that sites run by and for hackers are risky, as are sites serving pornography, scalping tickets, or selling contraband. You might also be wary of sites located in certain countries; Russia, China, Brazil, Korea, and India are often

near the top of the list for highest proportion of web sites containing malicious code. A web site could be located anywhere, although a .cn or .ru at the end of a URL associates the domain with China or Russia, respectively. However, the United States is also often high on such lists because of the large number of web-hosting providers located there.

- *Make a recoverable system image and store it safely*. If your system does become infected, this clean version will let you reboot securely because it overwrites the corrupted system files with clean copies. For this reason, you must keep the image write-protected during reboot. Prepare this image now, before infection; after infection is too late. For safety, prepare an extra copy of the safe boot image.
- *Make and retain backup copies of executable system files*. This way, in the event of a virus infection, you can remove infected files and reinstall from the clean backup copies (stored in a secure, offline location, of course). Also make and retain backups of important data files that might contain infectable code; such files include word-processor documents, spreadsheets, slide presentations, pictures, sound files, and databases. Keep these backups on inexpensive media, such as CDs or DVDs, a flash memory device, or a removable disk so that you can keep old backups for a long time. In case you find an infection, you want to be able to start from a clean backup, that is, one taken before the infection.

As for blocking system vulnerabilities, the recommendation is clear but problematic. As new vulnerabilities become known you should apply patches. However, finding flaws and fixing them under time pressure is often less than perfectly effective. Zero-day attacks are especially problematic, because a vulnerability presumably unknown to the software writers is now being exploited, so the manufacturer will press the development and maintenance team hard to develop and disseminate a fix. Furthermore, systems run many different software products from different vendors, but a vendor's patch cannot and does not consider possible interactions with other software. Thus, not only may a patch not repair the flaw for which it was intended, but it may fail or cause failure in conjunction with other software. Indeed, cases have arisen where a patch to one software application has been "recognized" incorrectly by an antivirus checker to be malicious code—and the system has ground to a halt. Thus, we recommend that you should apply all patches promptly except when doing so would cause more harm than good, which of course you seldom know in advance.

Still, good hygiene and self-defense are important controls users can take against malicious code. Most users rely on tools, called virus scanners or malicious code detectors, to guard against malicious code that somehow makes it onto a system.

Virus detectors are powerful but not all-powerful.

Virus Detectors

Virus scanners are tools that look for signs of malicious code infection. Most such tools look for a signature or fingerprint, a telltale pattern in program files or memory. As we show in this section, detection tools are generally effective, meaning that they detect most examples of malicious code that are at most somewhat sophisticated. Detection tools do have two major limitations, however.

First, detection tools are necessarily retrospective, looking for patterns of known infections. As new infectious code types are developed, tools need to be updated frequently with new patterns. But even with frequent updates (most tool vendors recommend daily updates), there will be infections that are too new to have been analyzed and included in the latest pattern file. Thus, a malicious code writer has a brief window, as little as hours or a day but perhaps longer if a new strain evades notice of the pattern analysts, during which the strain's pattern will not be in the database. Even though a day is a short window of opportunity, it is enough to achieve significant harm.

Second, patterns are necessarily static. If malicious code always begins with, or even contains, the same four instructions, the binary code of those instructions may be the invariant pattern for which the tool searches. Because tool writers want to avoid misclassifying good code as malicious, they seek the longest pattern they can: Two programs, one good and one malicious, might by chance contain the same four instructions. But the longer the pattern string, the less likely a benign program will match that pattern, so longer patterns are desirable. Malicious code writers are conscious of pattern matching, so they vary their code to reduce the number of repeated patterns. Sometimes minor perturbations in the order of instructions is insignificant. Thus, in the example, the dominant pattern might be instructions A-B-C-D, in that order. But the program's logic might work just as well with instructions B-A-C-D, so the malware writer will send out half the code with instructions A-B-C-D and half with B-A-C-D. Do-nothing instructions, such as adding 0 or subtracting 1 and later adding 1 again or replacing a data variable with itself, can be slipped into code at various points to break repetitive patterns. Longer patterns are more likely to be broken by a code modification. Thus, the virus detector tool writers have to discern more patterns for which to check.

Both timeliness and variation limit the effectiveness of malicious code detectors. Still, these tools are largely successful, and so we study them now. You should also note in Sidebar 3-9 that antivirus tools can also help people who *do not* use the tools.

Symantec, maker of the Norton antivirus software packages, announced in a 4 May 2014 *Wall Street Journal* article that antivirus technology is dead. They contend that recognizing malicious code on a system is a cat-and-mouse game: Malware signatures will always be reactive, reflecting code patterns discovered yesterday, and heuristics detect suspicious behavior but must forward code samples to a laboratory for human analysis and confirmation. Attackers are getting more skillful at evading detection by both pattern matchers and heuristic detectors. Furthermore, in the article, Symantec's Senior Vice President for Information Security admitted that antivirus software catches only 45 percent of malicious code. In the past, another vendor, FireEye, has also denounced these tools as ineffective. Both vendors prefer more specialized monitoring and analysis services, of which antivirus scanners are typically a first line of defense.

Does this statistic mean that people should abandon virus checkers? No, for two reasons. First, 45 percent still represents a solid defense, when you consider that there are now over 200 million specimens of malicious code in circulation [MCA14]. Second,

SIDEBAR 3-9 Free Security

Whenever influenza threatens, governments urge all citizens to get a flu vaccine. Not everyone does, but the vaccines manage to keep down the incidence of flu nevertheless. As long as enough people are vaccinated, the whole population gets protection. Such protection is called "herd immunity," because all in the group are protected by the actions of most, usually because enough vaccination occurs to prevent the infection from spreading.

In a similar way, sometimes parts of a network without security are protected by the other parts that are secure. For example, a node on a network may not incur the expense of antivirus software or a firewall, knowing that a virus or intruder is not likely to get far if the others in the network are protected. So the "free riding" acts as a disincentive to pay for security; the one who shirks security gets the benefit from the others' good hygiene.

The same kind of free-riding discourages reporting of security attacks and breaches. As we have seen, it may be costly for an attacked organization to report a problem, not just in terms of the resources invested in reporting but also in negative effects on reputation or stock price. So free-riding provides an incentive for an attacked organization to wait for someone else to report it, and then benefit from the problem's resolution. Similarly, if a second organization experiences an attack and shares its information and successful response techniques with others, the first organization receives the benefits without bearing any of the costs. Thus, incentives matter, and technology without incentives to understand and use it properly may in fact be ineffective technology.

recognize that the interview was in the *Wall Street Journal*, a popular publication for business and finance executives. Antivirus products make money; otherwise there would not be so many of them on the market. However, consulting services can make even more money, too. The Symantec executive was making the point that businesses, whose executives read the *Wall Street Journal*, need to invest also in advisors who will study a business's computing activity, identify shortcomings, and recommend remediation. And in the event of a security incident, organizations will need similar advice on the cause of the case, the amount and nature of harm suffered, and the next steps for further protection.

Virus Signatures

A virus cannot be completely invisible. Code must be stored somewhere, and the code must be in memory to execute. Moreover, the virus executes in a particular way, using certain methods to spread. Each of these characteristics yields a telltale pattern, called a signature, that can be found by a program that looks for it. The virus's signature is important for creating a program, called a virus scanner, that can detect and, in some cases, remove viruses. The scanner searches memory and long-term storage, monitoring execution and watching for the telltale signatures of viruses. For example, a scanner looking for signs of the Code Red worm can look for a pattern containing the following characters:

```
/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3
%u7801%u9090%u6858%ucdb3%u7801%u9090%u6858
%ucbd3%u7801%u9090
%u9090%u8190%u00c3%u0003%ub00%u531b%u53ff
%u0078%u0000%u00=a HTTP/1.0
```

When the scanner recognizes a known virus's pattern, it can then block the virus, inform the user, and deactivate or remove the virus. However, a virus scanner is effective only if it has been kept up-to-date with the latest information on current viruses.

Virus writers and antivirus tool makers engage in a battle to conceal patterns and find those regularities.

Code Analysis

Another approach to detecting an infection is to analyze the code to determine what it does, how it propagates and perhaps even where it originated. That task is difficult, however.

The first difficulty with analyzing code is that the researcher normally has only the end product to look at. As Figure 3-24 shows, a programmer writes code in some high-level language, such as C, Java, or C#. That code is converted by a compiler or interpreter into intermediate object code; a linker adds code of standard library routines and packages the result into machine code that is executable. The higher-level language code uses meaningful variable names, comments, and documentation techniques to make the code meaningful, at least to the programmer.

During compilation, all the structure and documentation are lost; only the raw instructions are preserved. To load a program for execution, a linker merges called library routines and performs address translation. If the code is intended for propagation, the attacker may also invoke a packager, a routine that strips out other identifying information and minimizes the size of the combined code block.

In case of an infestation, an analyst may be called in. The analyst starts with code that was actually executing, active in computer memory, but that may represent only a portion of the actual malicious package. Writers interested in stealth clean up, purging memory or disk of unnecessary instructions that were needed once, only to install the infectious code. In any event, analysis starts from machine instructions. Using a tool called a disassembler, the analyst can convert machine-language binary instructions to their assembly language equivalents, but the trail stops there. These assembly language instructions have none of the informative documentation, variable names, structure, labels or comments, and the assembler language representation of a program is much less easily understood than its higher-level language counterpart. Thus, although the

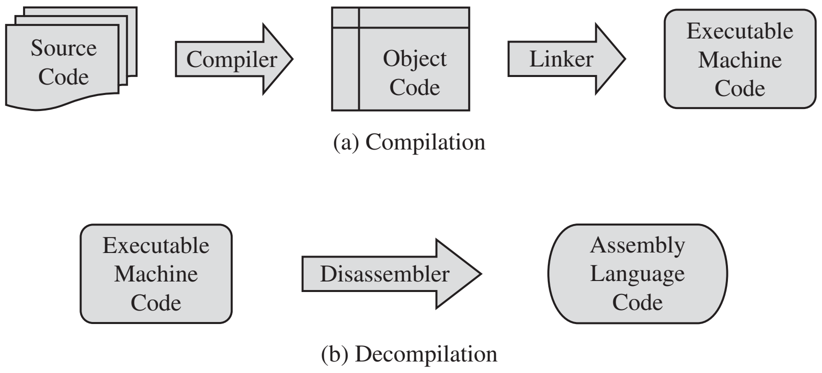


FIGURE 3-24 The Compilation Process: (a) Compilation. (b) Decompilation

analyst can determine literally what instructions a piece of code performs, the analyst has a harder time determining the broader intent and impact of those statements.

Security research labs do an excellent job of tracking and analyzing malicious code, but such analysis is necessarily an operation of small steps with microscope and tweezers. (The phrase microscope and tweezers is attributed to Jerome Saltzer in [EIC89].) Even with analysis tools, the process depends heavily on human ingenuity. In Chapter 10 we expand on teams that do incident response and analysis.

Thoughtful analysis with "microscope and tweezers" after an attack must complement preventive tools such as virus detectors.

Storage Patterns

Most viruses attach to programs that are stored on media such as disks. The attached virus piece is invariant, so the start of the virus code becomes a detectable signature. The attached piece is always located at the same position relative to its attached file. For example, the virus might always be at the beginning, 400 bytes from the top, or at the bottom of the infected file. Most likely, the virus will be at the beginning of the file because the virus writer wants to control execution before the bona fide code of the infected program is in charge. In the simplest case, the virus code sits at the top of the program, and the entire virus does its malicious duty before the normal code is invoked. In other cases, the virus infection consists of only a handful of instructions that point or jump to other, more detailed, instructions elsewhere. For example, the infected code may consist of condition testing and a jump or call to a separate virus module. In either case, the code to which control is transferred will also have a recognizable pattern. Both of these situations are shown in Figure 3-25.

A virus may attach itself to a file, in which case the file's size grows. Or the virus may obliterate all or part of the underlying program, in which case the program's size does not change but the program's functioning will be impaired. The virus writer has to choose one of these detectable effects.

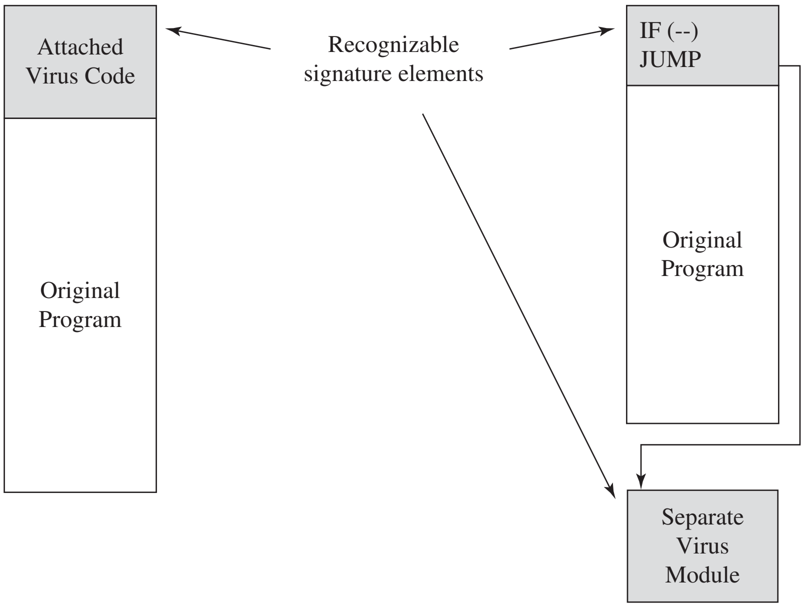


FIGURE 3-25 Recognizable Patterns in Viruses

The virus scanner can use a code or checksum to detect changes to a file. It can also look for suspicious patterns, such as a JUMP instruction as the first instruction of a system program (in case the virus has positioned itself at the bottom of the file but is to be executed first, as we saw in Figure 3-25).

Countermeasures for Developers

Against this threat background you may well ask how anyone can ever make secure, trustworthy, flawless programs. As the size and complexity of programs grows, the number of possibilities for attack does, too.

In this section we briefly look at some software engineering techniques that have been shown to improve the security of code. Of course, these methods must be used effectively, for a good method used improperly or naïvely will not make programs better by magic. Ideally, developers should have a reasonable understanding of security, and especially of thinking in terms of threats and vulnerabilities. Armed with that mindset and good development practices, programmers can write code that maintains security.

Software Engineering Techniques

Code usually has a long shelf-life and is enhanced over time as needs change and faults are found and fixed. For this reason, a key principle of software engineering is to create a design or code in small, self-contained units, called components or modules; when a system is written this way, we say that it is **modular**. Modularity offers advantages for program development in general and security in particular.

If a component is isolated from the effects of other components, then the system is designed in a way that limits the damage any fault causes. Maintaining the system is easier because any problem that arises connects with the fault that caused it. Testing (especially regression testing—making sure that everything else still works when you make a corrective change) is simpler, since changes to an isolated component do not affect other components. And developers can readily see where vulnerabilities may lie if the component is isolated. We call this isolation **encapsulation**.

Information hiding is another characteristic of modular software. When information is hidden, each component hides its precise implementation or some other design decision from the others. Thus, when a change is needed, the overall design can remain intact while only the necessary changes are made to particular components.

Let us look at these characteristics in more detail.

Modularity

Modularization is the process of dividing a task into subtasks, as depicted in Figure 3-26. This division is usually done on a logical or functional basis, so that each component performs a separate, independent part of the task. The goal is for each component to meet four conditions:

- *single-purpose*, performs one function
- *small*, consists of an amount of information for which a human can readily grasp both structure and content

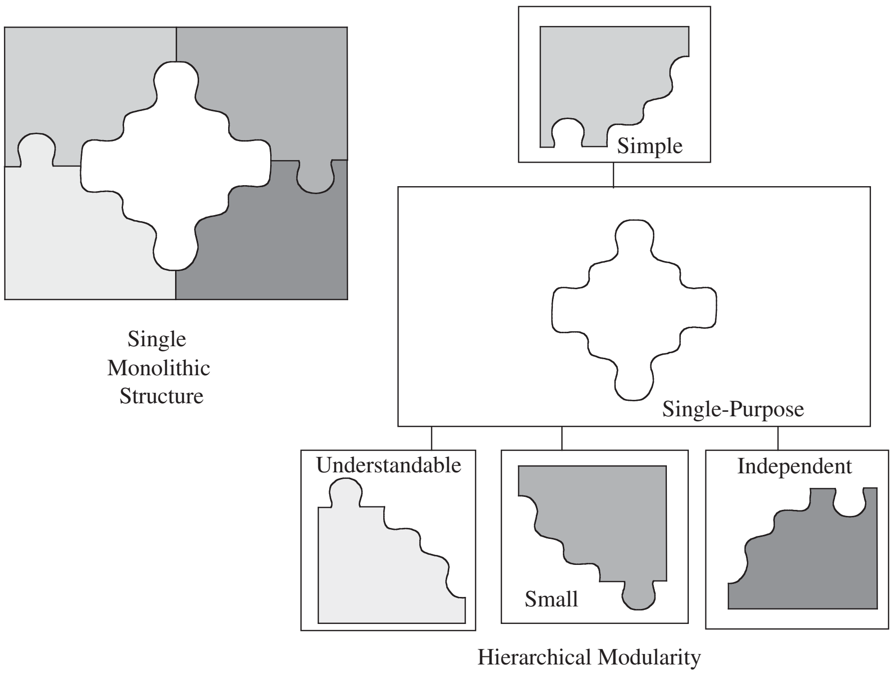


FIGURE 3-26 Modularity

- *simple*, is of a low degree of complexity so that a human can readily understand the purpose and structure of the module
- *independent*, performs a task isolated from other modules

Other component characteristics, such as having a single input and single output or using a limited set of programming constructs, indicate modularity. From a security standpoint, modularity should improve the likelihood that an implementation is correct.

In particular, smallness and simplicity help both developers and analysts understand what each component does. That is, in good software, design and program units should be only as large or complex as needed to perform their required functions. There are several advantages to having small, independent components.

- *Maintenance.* If a component implements a single function, it can be replaced easily with a revised one if necessary. The new component may be needed because of a change in requirements, hardware, or environment. Sometimes the replacement is an enhancement, using a smaller, faster, more correct, or otherwise better module. The interfaces between this component and the remainder of the design or code are few and well described, so the effects of the replacement are evident.
- *Understandability.* A system composed of small and simple components is usually easier to comprehend than one large, unstructured block of code.
- *Reuse.* Components developed for one purpose can often be reused in other systems. Reuse of correct, existing design or code components can significantly reduce the difficulty of implementation and testing.
- *Correctness.* A failure can be quickly traced to its cause if the components perform only one task each.
- *Testing.* A single component with well-defined inputs, outputs, and function can be tested exhaustively by itself, without concern for its effects on other modules (other than the expected function and output, of course).

Simplicity of software design improves correctness and maintainability.

A modular component usually has high cohesion and low coupling. By **cohesion**, we mean that all the elements of a component have a logical and functional reason for being there; every aspect of the component is tied to the component's single purpose. A highly cohesive component has a high degree of focus on the purpose; a low degree of cohesion means that the component's contents are an unrelated jumble of actions, often put together because of time dependencies or convenience.

Coupling refers to the degree with which a component depends on other components in the system. Thus, low or loose coupling is better than high or tight coupling because the loosely coupled components are free from unwitting interference from other components. This difference in coupling is shown in Figure 3-27.

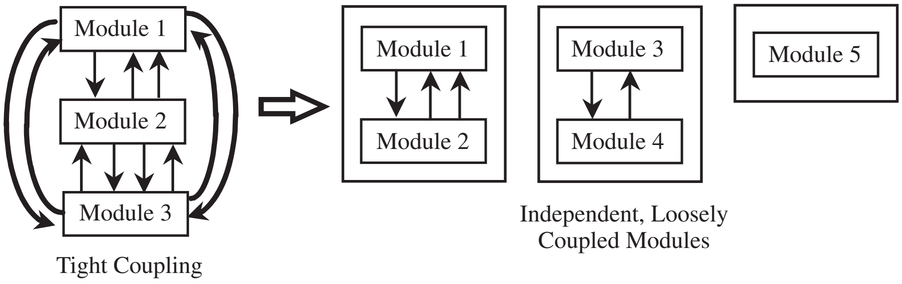


FIGURE 3-27 Types of Coupling

Encapsulation

Encapsulation hides a component's implementation details, but it does not necessarily mean complete isolation. Many components must share information with other components, usually with good reason. However, this sharing is carefully documented so that a component is affected only in known ways by other components in the system. Sharing is minimized so that the fewest interfaces possible are used.

An encapsulated component's protective boundary can be translucent or transparent, as needed. Berard [BER00] notes that encapsulation is the "technique for packaging the information [inside a component] in such a way as to hide what should be hidden and make visible what is intended to be visible."

Information Hiding

Developers who work where modularization is stressed can be sure that other components will have limited effect on the ones they write. Thus, we can think of a component as a kind of black box, with certain well-defined inputs and outputs and a well-defined function. Other components' designers do not need to know how the module completes its function; it is enough to be assured that the component performs its task in some correct manner.

Information hiding: describing what a module does, not how

This concealment is the information hiding, depicted in Figure 3-28. Information hiding is desirable, because malicious developers cannot easily alter the components of others if they do not know how the components work.

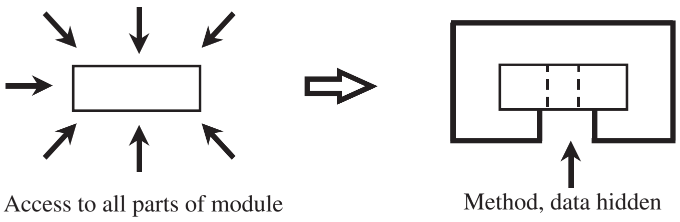


FIGURE 3-28 Information Hiding

Mutual Suspicion

Programs are not always trustworthy. Even with an operating system to enforce access limitations, it may be impossible or infeasible to bound the access privileges of an untested program effectively. In this case, the user U is legitimately suspicious of a new program P. However, program P may be invoked by another program, Q. There is no way for Q to know that P is correct or proper, any more than a user knows that of P.

Therefore, we use the concept of **mutual suspicion** to describe the relationship between two programs. Mutually suspicious programs operate as if other routines in the system were malicious or incorrect. A calling program cannot trust its called subprocedures to be correct, and a called subprocedure cannot trust its calling program to be correct. Each protects its interface data so that the other has only limited access. For example, a procedure to sort the entries in a list cannot be trusted not to modify those elements, while that procedure cannot trust its caller to provide any list at all or to supply the number of elements predicted. An example of misplaced trust is described in Sidebar 3-10.

SIDEBAR 3-10 Facebook Outage from Improper Error Handling

In September 2010 the popular social networking site Facebook was forced to shut down for several hours. According to a posting by company representative Robert Johnson, the root cause was an improperly handled error condition.

Facebook maintains in a persistent store a set of configuration parameters that are then copied to cache for ordinary use. Code checks the validity of parameters in the cache. If it finds an invalid value, it fetches the value from the persistent store and uses it to replace the cache value. Thus, the developers assumed the cache value might become corrupted but the persistent value would always be accurate.

In the September 2010 instance, staff mistakenly placed an incorrect value in the persistent store. When this value was propagated to the cache, checking routines identified it as erroneous and caused the cache controller to fetch the value from the persistent store. The persistent store value, of course, was erroneous, so as soon as the checking routines examined it, they again called for its replacement from the persistent store. This constant fetch from the persistent store led to an overload on the server holding the persistent store, which in turn led to a severe degradation in performance overall.

Facebook engineers were able to diagnose the problem, concluding that the best solution was to disable all Facebook activity and then correct the persistent store value. They gradually allowed Facebook clients to reactivate; as each client detected an inaccurate value in its cache, it would refresh it from the correct value in the persistent store. In this way,

(*continues*)

SIDEBAR 3-10 *Continued*

the gradual expansion of services allowed these refresh requests to occur without overwhelming access to the persistent store server.

A design of mutual suspicion—not implicitly assuming the cache is wrong and the persistent store is right—would have avoided this catastrophe.

Confinement

Confinement is a technique used by an operating system on a suspected program to help ensure that possible damage does not spread to other parts of a system. A **confined** program is strictly limited in what system resources it can access. If a program is not trustworthy, the data it can access are strictly limited. Strong confinement would be particularly helpful in limiting the spread of viruses. Since a virus spreads by means of transitivity and shared data, all the data and programs within a single compartment of a confined program can affect only the data and programs in the same compartment. Therefore, the virus can spread only to things in that compartment; it cannot get outside the compartment.

Simplicity

The case for simplicity—of both design and implementation—should be self-evident: simple solutions are easier to understand, leave less room for error, and are easier to review for faults. The value of simplicity goes deeper, however.

With a simple design, all members of the design and implementation team can understand the role and scope of each element of the design, so each participant knows not only what to expect others to do but also what others expect. Perhaps the worst problem of a running system is maintenance: After a system has been running for some time, and the designers and programmers are working on other projects (or perhaps even at other companies), a fault appears and some unlucky junior staff member is assigned the task of correcting the fault. With no background on the project, this staff member must attempt to intuit the visions of the original designers and understand the entire context of the flaw well enough to fix it. A simple design and implementation facilitates correct maintenance.

Hoare [HOA81] makes the case simply for simplicity of design:

> I gave desperate warnings against the obscurity, the complexity, and overambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies.

In 2014 the web site for the annual RSA computer security conference was compromised. Amit Yoran, Senior Vice President of Products and Sales for RSA, the parent company that founded the conference and supports it financially, spoke to the issue. “Unfortunately, complexity is very often the enemy of security,” he concluded, emphasizing that he was speaking for RSA and not for the RSA conference web site, a separate entity [KRE14].

"Complexity is often the enemy of security."—Amit Yoran, RSA

Genetic Diversity

At your local electronics shop you can buy a combination printer–scanner–copier–fax machine. It comes at a good price (compared to costs of buying the four components separately) because there is considerable overlap in implementing the functionality among those four. Moreover, the multifunction device is compact, and you need install only one device on your system, not four. But if any part of it fails, you lose a lot of capabilities all at once. So the multipurpose machine represents the kinds of trade-offs among functionality, economy, and availability that we make in any system design.

An architectural decision about these types of devices is related to the arguments above for modularity, information hiding, and reuse or interchangeability of software components. For these reasons, some people recommend heterogeneity or "genetic diversity" in system architecture: Having many components of a system come from one source or relying on a single component is risky, they say.

However, many systems are in fact quite homogeneous in this sense. For reasons of convenience and cost, we often design systems with software or hardware (or both) from a single vendor. For example, in the early days of computing, it was convenient to buy "bundled" hardware and software from a single vendor. There were fewer decisions for the buyer to make, and if something went wrong, only one phone call was required to initiate trouble-shooting and maintenance. Daniel Geer et al. [GEE03a] examined the monoculture of computing dominated by one manufacturer, often characterized by Apple or Google today, Microsoft or IBM yesterday, unknown tomorrow. They looked at the parallel situation in agriculture where an entire crop may be vulnerable to a single pathogen. In computing, the pathogenic equivalent may be malicious code from the Morris worm to the Code Red virus; these "infections" were especially harmful because a significant proportion of the world's computers were disabled because they ran versions of the same operating systems (Unix for Morris, Windows for Code Red).

Diversity creates a moving target for the adversary. As Per Larson and colleagues explain [LAR14], introducing diversity automatically is possible but tricky. A compiler can generate different but functionally equivalent object code from one source file; reordering statements (where there is no functional dependence on the order), using different storage layouts, and even adding useless but harmless instructions helps protect one version from harm that might affect another version. However, different output object code can create a nightmare for code maintenance.

Diversity reduces the number of targets susceptible to one attack type.

In 2014 many computers and web sites were affected by the so-called Heartbleed malware, which exploited a vulnerability in the widely used OpenSSL software. SSL (secure socket layer) is a cryptographic technique by which browser web communications are secured, for example, to protect the privacy of a banking transaction. (We cover SSL in Chapter 6.) The OpenSSL implementation is used by the majority of web sites; two major packages using OpenSSL account for over 66 percent of sites using SSL. Because the adoption of OpenSSL is so vast, this one vulnerability affects a huge number of sites, putting the majority of Internet users at risk. The warning about lack of diversity in software is especially relevant here. However, cryptography is a delicate topic; even correctly written code can leak sensitive information, not to mention the numerous subtle ways such code can be wrong. Thus, there is a good argument for having a small number of cryptographic implementations that analysts can scrutinize rigorously. But common code presents a single or common point for mass failure.

Furthermore, diversity is expensive, as large users such as companies or universities must maintain several kinds of systems instead of focusing their effort on just one. Furthermore, diversity would be substantially enhanced by a large number of competing products, but the economics of the market make it difficult for many vendors to all profit enough to stay in business. Geer refined the argument in [GEE03], which was debated by James Whittaker [WHI03b] and David Aucsmith [AUC03]. There is no obvious right solution for this dilemma.

Tight integration of products is a similar concern. The Windows operating system is tightly linked to Internet Explorer, the Office suite, and the Outlook email handler. A vulnerability in one of these can also affect the others. Because of the tight integration, fixing a vulnerability in one subsystem can have an impact on the others. On the other hand, with a more diverse (in terms of vendors) architecture, a vulnerability in another vendor's browser, for example, can affect Word only to the extent that the two systems communicate through a well-defined interface.

A different form of change occurs when a program is loaded into memory for execution. **Address-space-layout randomization** is a technique by which a module is loaded into different locations at different times (using a relocation device similar to base and bounds registers, described in Chapter 5). However, when an entire module is relocated as a unit, getting one real address gives the attacker the key to compute the addresses of all other parts of the module.

Next we turn from product to process. How is good software produced? As with the code properties, these process approaches are not a recipe: doing these things does not guarantee good code. However, like the code characteristics, these processes tend to reflect approaches of people who successfully develop secure software.

Testing

Testing is a process activity that concentrates on product quality: It seeks to locate potential product failures before they actually occur. The goal of testing is to make the product failure free (eliminating the possibility of failure); realistically, however, testing will only reduce the likelihood or limit the impact of failures. Each software problem (especially when it relates to security) has the potential not only for making software fail but also for adversely affecting a business or a life. The failure of one control may expose a vulnerability that is not ameliorated by any number of functioning controls. Testers improve software quality by finding as many faults as possible and carefully documenting their findings so that developers can locate the causes and repair the problems if possible.

Testing is easier said than done, and Herbert Thompson points out that security testing is particularly hard [THO03]. James Whittaker observes in the Google Testing Blog, 20 August 2010, that "Developers grow trees; testers manage forests," meaning the job of the tester is to explore the interplay of many factors. Side effects, dependencies, unpredictable users, and flawed implementation bases (languages, compilers, infrastructure) all contribute to this difficulty. But the essential complication with security testing is that we cannot look at just the one behavior the program gets right; we also have to look for the hundreds of ways the program might go wrong.

Security testing tries to anticipate the hundreds of ways a program can fail.

Types of Testing

Testing usually involves several stages. First, each program component is tested on its own. Such testing, known as **module testing**, **component testing**, or **unit testing**, verifies that the component functions properly with the types of input expected from a study of the component's design. **Unit testing** is done so that the test team can feed a predetermined set of data to the component being tested and observe what output actions and data are produced. In addition, the test team checks the internal data structures, logic, and boundary conditions for the input and output data.

When collections of components have been subjected to unit testing, the next step is ensuring that the interfaces among the components are defined and handled properly. Indeed, interface mismatch can be a significant security vulnerability, so the interface design is often documented as an **application programming interface** or **API**. **Integration testing** is the process of verifying that the system components work together as described in the system and program design specifications.

Once the developers verify that information is passed among components in accordance with their design, the system is tested to ensure that it has the desired functionality. A **function test** evaluates the system to determine whether the functions described by the requirements specification are actually performed by the integrated system. The result is a functioning system.

The function test compares the system being built with the functions described in the developers' requirements specification. Then, a **performance test** compares the system with the remainder of these software and hardware requirements. During the function and performance tests, testers examine security requirements and confirm that the system is as secure as it is required to be.

When the performance test is complete, developers are certain that the system functions according to their understanding of the system description. The next step is conferring with the customer to make certain that the system works according to customer expectations. Developers join the customer to perform an **acceptance test**, in which the system is checked against the customer's requirements description. Upon completion of acceptance testing, the accepted system is installed in the environment in which it will be used. A final **installation test** is run to make sure that the system still functions as it should. However, security requirements often state that a system should not do something. As Sidebar 3-11 demonstrates, absence is harder to demonstrate than presence.

SIDEBAR 3-11 Absence vs. Presence

Charles Pfleeger [PFL97] points out that security requirements resemble those for any other computing task, with one seemingly insignificant difference. Whereas most requirements say "the system will do this," security requirements add the phrase "and nothing more." As we pointed out in Chapter 1, security awareness calls for more than a little caution when a creative developer takes liberties with the system's specification. Ordinarily, we do not worry if a programmer or designer adds a little something extra. For instance, if the requirement calls for generating a file list on a disk, the "something more" might be sorting the list in alphabetical order or displaying the date it was created. But we would never expect someone to meet the requirement by displaying the list and then erasing all the files on the disk!

If we could easily determine whether an addition were harmful, we could just disallow harmful additions. But unfortunately we cannot. For security reasons, we must state explicitly the phrase "and nothing more" and leave room for negotiation in the requirements definition on any proposed extensions.

Programmers naturally want to exercise their creativity in extending and expanding the requirements. But apparently benign choices, such as storing a value in a global variable or writing to a temporary file, can have serious security implications. And sometimes the best design approach for security is the counterintuitive one. For example, one attack on a cryptographic system depends on measuring the time it takes the system to perform an encryption. With one encryption technique, the time to encrypt depends on the key, a parameter that allows someone to "unlock" or decode the encryption; encryption time specifically depends on the size or the number of bits in the key. The time measurement helps attackers know the approximate key length, so they can narrow their search space accordingly (as described in Chapter 2). Thus, an efficient implementation can actually undermine the system's security. The solution, oddly enough, is to artificially pad the encryption process with unnecessary computation so that short computations complete as slowly as long ones.

In another instance, an enthusiastic programmer added parity checking to a cryptographic procedure. But the routine generating the keys did not supply a check bit, only the keys themselves. Because the keys were generated randomly, the result was that 255 of the 256 encryption keys failed the parity check, leading to the substitution of a fixed key—so that without warning, all encryptions were being performed under the same key!

No technology can automatically distinguish malicious extensions from benign code. For this reason, we have to rely on a combination of approaches, including human-intensive ones, to help us detect when we are going beyond the scope of the requirements and threatening the system's security.

The objective of unit and integration testing is to ensure that the code implemented the design properly; that is, that the programmers have written code to do what the designers intended. System testing has a very different objective: to ensure that the system does what the customer wants it to do. Regression testing, an aspect of system testing, is particularly important for security purposes. After a change is made to enhance the system or fix a problem, **regression testing** ensures that all remaining functions are still working and that performance has not been degraded by the change. As we point out in Sidebar 3-12, regression testing is difficult because it essentially entails reconfirming all functionality.

SIDEBAR 3-12 The GOTO Fail Bug

In February 2014 Apple released a maintenance patch to its iOS operating system. The problem involved code to implement SSL, the encryption that protects secure web communications, such as between a user's web browser and a bank's web site, for example. The code problem, which has been called the "GOTO Fail" bug, is shown in the following code fragment.

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom))
                 != 0)
            goto fail;
      if ((err = SSLHashSHA1.update(&hashCtx,
                 &signedParams)) != 0)
            goto fail;
            goto fail;
      if ((err = SSLHashSHA1.final(&hashCtx, &hashOut))
                 != 0)
            goto fail;
      ...

fail:
      SSLFreeBuffer(&signedHashes);
      SSLFreeBuffer(&hashCtx);
      return err;
```

The problem is in the seventh line. If the first two conditional statements are false, execution drops directly to the duplicate goto fail line, and exits the routine. The impact of this flaw is that even insecure web connections are treated as secure.

The origin of this error is unknown, but it appears either that another conditional statement was removed during maintenance (but not the corresponding conditional action of goto fail), or an extra goto fail statement was inadvertently pasted into the routine. Either of those possibilities is an understandable, nonmalicious programming oversight.

(*continues*)

SIDEBAR 3-12 *Continued*

Regression testing to catch such a simple programming error would require setting up a complicated test case. Programmers are often pressed during maintenance to complete fixes rapidly, so there is not time for thorough testing, which could be how this flaw became part of the standard distribution of the operating system.

The flaw is small and easy to spot when you know to look for it, although it is line 632 of a 1970-line file, where it would stand out less than in the fragment we reproduce here. The error affected mobile iPhones and iPads, as well as desktop Macintosh computers. The patches released by Apple indicate the error has been embedded in production code for some time. For more details on the flaw, see Paul Ducklin's blog posting at http://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/.

Each of the types of tests listed here can be performed from two perspectives: black box and clear box (sometimes called white box). **Black-box testing** treats a system or its components as black boxes; testers cannot "see inside" the system, so they apply particular inputs and verify that they get the expected output. **Clear-box testing** allows visibility. Here, testers can examine the design and code directly, generating test cases based on the code's actual construction. Thus, clear-box testing reveals that component X uses CASE statements and can look for instances in which the input causes control to drop through to an unexpected line. Black-box testing must rely more on the required inputs and outputs because the actual code is not available for scrutiny.

James Whittaker in his testing blog lists seven key ingredients for testing (http://googletesting.blogspot.com/2010/08/ingredients-list-for-testing-part-one.html). We summarize his posting here:

1. *Product expertise*. The tester needs to understand the requirements and functionality of the object being tested. More importantly, the tester should have sufficient familiarity with the product to be able to predict what it cannot do and be able to stress it in all its configurations.
2. *Coverage*. Testing must be complete, in that no component should be ignored, no matter how small or insignificant.
3. *Risk analysis*. Testing can never cover everything. Thus, wise testing, that is, to spend testing resources wisely and effectively, is necessary. A risk analysis answers the questions what are the most critical pieces and what can go seriously wrong? From this the priority for testing becomes clearer.
4. *Domain expertise*. A tester must understand the product being tested. Trivially, someone cannot effectively test a Fahrenheit-to-centigrade converter without understanding those two temperature scales.
5. *Common vocabulary*. There is little common vocabulary for testing; even terms like black-box testing are subject to some interpretation. More importantly,

testers need to be able to share patterns and techniques with one another, and to do that, testers need some common understanding of the larger process.

6. *Variation*. Testing is not a checklist exercise; if it were, we would automate the whole process, let a machine do it, and never have product failures. Testers need to vary their routine, test different things in different ways, and adapt to successes and failures.
7. *Boundaries*. Because testing can continue indefinitely, some concept of completeness and sufficiency is necessary. Sometimes, finite resources of time or money dictate how much testing is done. A better approach is a rational plan that determines what degree of testing is adequate.

Effectiveness of Testing

The mix of techniques appropriate for testing a given system depends on the system's size, application domain, amount of risk, and many other factors. But understanding the effectiveness of each technique helps us know what is right for each particular system. For example, Olsen [OLS93] describes the development at Contel IPC of a system containing 184,000 lines of code. He tracked faults discovered during various activities and found these differences:

- 17.3 percent of the faults were found during inspections of the system design
- 19.1 percent during component design inspection
- 15.1 percent during code inspection
- 29.4 percent during integration testing
- 16.6 percent during system and regression testing

Only 0.1 percent of the faults were revealed after the system was placed in the field. Thus, Olsen's work shows the importance of using different techniques to uncover different kinds of faults during development; we must not rely on a single method applied at one time to catch all problems.

Who does the testing? From a security standpoint, **independent testing** is highly desirable; it may prevent a developer from attempting to hide something in a routine or keep a subsystem from controlling the tests that will be applied to it. Thus, independent testing increases the likelihood that a test will expose the effect of a hidden feature.

Limitations of Testing

Testing is the most widely accepted assurance technique. As Earl Boebert [BOE92] observes, conclusions from testing are based on the actual product being evaluated, not on some abstraction or precursor of the product. This realism is a security advantage. However, conclusions based on testing are necessarily limited, for the following reasons:

- Testing can demonstrate the *existence* of a problem, but passing tests does not demonstrate the absence of problems.
- Testing adequately within reasonable time or effort is difficult because the combinatorial explosion of inputs and internal states makes complete testing complex and time consuming.

- Testing only observable effects, not the internal structure of a product, does not ensure any degree of completeness.
- Testing the internal structure of a product involves modifying the product by adding code to extract and display internal states. That extra functionality affects the product's behavior and can itself be a source of vulnerabilities or can mask other vulnerabilities.
- Testing real-time or complex systems requires keeping track of all states and triggers. This profusion of possible situations makes it hard to reproduce and analyze problems reported as testers proceed.

Ordinarily, we think of testing in terms of the developer: unit testing a module, integration testing to ensure that modules function properly together, function testing to trace correctness across all aspects of a given function, and system testing to combine hardware with software. Likewise, regression testing is performed to make sure a change to one part of a system does not degrade any other functionality. But for other tests, including acceptance tests, the user or customer administers them to determine if what was ordered is what is delivered. Thus, an important aspect of assurance is considering whether the tests run are appropriate for the application and level of security. The nature and kinds of testing reflect the developer's testing strategy: which tests address what issues.

Similarly, testing is almost always constrained by a project's budget and schedule. The constraints usually mean that testing is incomplete in some way. For this reason, we consider notions of test coverage, test completeness, and testing effectiveness in a testing strategy. The more complete and effective our testing, the more confidence we have in the software. More information on testing can be found in Pfleeger and Atlee [PFL10].

Countermeasure Specifically for Security

General software engineering principles are intended to lead to correct code, which is certainly a security objective, as well. However, there are also activities during program design, implementation, and fielding specifically to improve the security of the finished product. We consider those practices next.

Design Principles for Security

Multics (MULTiplexed Information and Computer Service) was a major secure software project intended to provide a computing utility to its users, much as we access electricity or water. The system vision involved users who could effortlessly connect to it, use the computing services they needed, and then disconnect—much as we turn the tap on and off. Clearly all three fundamental goals of computer security—confidentiality, integrity, and availability—are necessary for such a widely shared endeavor, and security was a major objective for the three participating Multics partners: M.I.T, AT&T Bell Laboratories, and GE. Although the project never achieved significant commercial success, its development helped establish secure computing as a rigorous and active discipline. The Unix operating system grew out of Multics, as did other now-common operating system design elements, such as a hierarchical file structure, dynamically invoked modules, and virtual memory.

The chief security architects for Multics, Jerome Saltzer and Michael Schroeder, documented several design principles intended to improve the security of the code they were developing. Several of their design principles are essential for building a solid, trusted operating system. These principles, well articulated in Saltzer [SAL74] and Saltzer and Schroeder [SAL75], include the following:

- *Least privilege.* Each user and each program should operate using the fewest privileges possible. In this way, damage from an inadvertent or malicious attack is minimized.
- *Economy of mechanism.* The design of the protection system should be small, simple, and straightforward. Such a protection system can be carefully analyzed, exhaustively tested, perhaps verified, and relied on.
- *Open design.* The protection mechanism must not depend on the ignorance of potential attackers; the mechanism should be public, depending on secrecy of relatively few key items, such as a password table. An open design is also available for extensive public scrutiny, thereby providing independent confirmation of the design security.
- *Complete mediation.* Every access attempt must be checked. Both direct access attempts (requests) and attempts to circumvent the access-checking mechanism should be considered, and the mechanism should be positioned so that it cannot be circumvented.
- *Permission based.* The default condition should be denial of access. A conservative designer identifies the items that should be accessible, rather than those that should not.
- *Separation of privilege.* Ideally, access to objects should depend on more than one condition, such as user authentication plus a cryptographic key. In this way, someone who defeats one protection system will not have complete access.
- *Least common mechanism.* Shared objects provide potential channels for information flow. Systems employing physical or logical separation reduce the risk from sharing.
- *Ease of use.* If a protection mechanism is easy to use, it is unlikely to be avoided.

These principles have been generally accepted by the security community as contributing to the security of software and system design. Even though they date from the stone age of computing, the 1970s, they are at least as important today. As a mark of how fundamental and valid these precepts are, consider the recently issued "Top 10 Secure Coding Practices" from the Computer Emergency Response Team (CERT) of the Software Engineering Institute at Carnegie Mellon University [CER10].

1. Validate input.
2. Heed compiler warnings.
3. Architect and design for security policies.
4. Keep it simple.

5. Default to deny.
6. Adhere to the principle of least privilege.
7. Sanitize data sent to other systems.
8. Practice defense in depth.
9. Use effective quality-assurance techniques.
10. Adopt a secure coding standard.

Of these ten, numbers 4, 5, and 6 match directly with Saltzer and Schroeder, and 3 and 8 are natural outgrowths of that work. Similarly, the Software Assurance Forum for Excellence in Code (SAFECode)[2] produced a guidance document [SAF11] that is also compatible with these concepts, including such advice as implementing least privilege and sandboxing (to be defined later), which is derived from separation of privilege and complete mediation. We elaborate on many of the points from SAFECode throughout this chapter, and we encourage you to read their full report after you have finished this chapter. Other authors, such as John Viega and Gary McGraw [VIE01] and Michael Howard and David LeBlanc [HOW02], have elaborated on the concepts in developing secure programs.

Penetration Testing for Security

The testing approaches in this chapter have described methods appropriate for all purposes of testing: correctness, usability, performance, as well as security. In this section we examine several approaches that are especially effective at uncovering security flaws.

We noted earlier in this chapter that **penetration testing** or **tiger team analysis** is a strategy often used in computer security. (See, for example, [RUB01, TIL03, PAL01].) Sometimes it is called **ethical hacking**, because it involves the use of a team of experts trying to crack the system being tested (as opposed to trying to break into the system for unethical reasons). The work of penetration testers closely resembles what an actual attacker might do [AND04, SCH00b]. The tiger team knows well the typical vulnerabilities in operating systems and computing systems. With this knowledge, the team attempts to identify and exploit the system's particular vulnerabilities.

Penetration testing is both an art and science. The artistic side requires careful analysis and creativity in choosing the test cases. But the scientific side requires rigor, order, precision, and organization. As Clark Weissman observes [WEI95], there is an organized methodology for hypothesizing and verifying flaws. It is not, as some might assume, a random punching contest.

Using penetration testing is much like asking a mechanic to look over a used car on a sales lot. The mechanic knows potential weak spots and checks as many of them as possible. A good mechanic will likely find most significant problems, but finding a problem (and fixing it) is no guarantee that no other problems are lurking in other parts

2. SAFECode is a non-profit organization exclusively dedicated to increasing trust in information and communications technology products and services through the advancement of effective software assurance methods. Its members include Adobe Systems Incorporated, EMC Corporation, Juniper Networks, Inc., Microsoft Corp., Nokia, SAP AG, and Symantec Corp.

of the system. For instance, if the mechanic checks the fuel system, the cooling system, and the brakes, there is no guarantee that the muffler is good.

In the same way, an operating system that fails a penetration test is known to have faults, but a system that does not fail is not guaranteed to be fault-free. All we can say is that the system is likely to be free only from the types of faults checked by the tests exercised on it. Nevertheless, penetration testing is useful and often finds faults that might have been overlooked by other forms of testing.

A system that fails penetration testing is known to have faults; one that passes is known only not to have the faults tested for.

One possible reason for the success of penetration testing is its use under real-life conditions. Users often exercise a system in ways that its designers never anticipated or intended. So penetration testers can exploit this real-life environment and knowledge to make certain kinds of problems visible.

Penetration testing is popular with the commercial community that thinks skilled hackers will test (attack) a site and find all its problems in days, if not hours. But finding flaws in complex code can take weeks if not months, so there is no guarantee that penetration testing will be effective.

Indeed, the original military "red teams" convened to test security in software systems were involved in 4- to 6-month exercises—a very long time to find a flaw. Anderson et al. [AND04] elaborate on this limitation of penetration testing. To find one flaw in a space of 1 million inputs may require testing all 1 million possibilities; unless the space is reasonably limited, the time needed to perform this search is prohibitive. To test the testers, Paul Karger and Roger Schell inserted a security fault in the painstakingly designed and developed Multics system, to see if the test teams would find it. Even after Karger and Schell informed testers that they had inserted a piece of malicious code in a system, the testers were unable to find it [KAR02]. Penetration testing is not a magic technique for finding needles in haystacks.

Proofs of Program Correctness

A security specialist wants to be certain that a given program computes a particular result, computes it correctly, and does nothing beyond what it is supposed to do. Unfortunately, results in computer science theory indicate that we cannot know with certainty that two programs do exactly the same thing. That is, there can be no general procedure which, given any two programs, determines if the two are equivalent. This difficulty results from the "halting problem," which states that there can never be a general technique to determine whether an arbitrary program will halt when processing an arbitrary input. (See [PFL85] for a discussion.)

In spite of this disappointing general result, a technique called **program verification** can demonstrate formally the "correctness" of certain specific programs. Program verification involves making initial assertions about the program's inputs and then checking to see if the desired output is generated. Each program statement is translated into a logical description about its contribution to the logical flow of the program. Then, the terminal statement of the program is associated with the desired output. By applying a logic analyzer, we can prove that the initial assumptions, plus the implications of the program statements, produce the terminal condition. In this way, we can show that a particular program achieves its goal. Sidebar 3-13 presents the case for appropriate use of formal proof techniques.

Proving program correctness, although desirable and useful, is hindered by several factors. (For more details see [PFL94].)

- Correctness proofs depend on a programmer's or logician's ability to translate a program's statements into logical implications. Just as programming is prone to errors, so also is this translation.
- Deriving the correctness proof from the initial assertions and the implications of statements is difficult, and the logical engine to generate proofs runs slowly. The

SIDEBAR 3-13 Formal Methods Can Catch Difficult-to-See Problems

Formal methods are sometimes used to check various aspects of secure systems. There is some disagreement about just what constitutes a formal method, but there is general agreement that every formal method involves the use of mathematically precise specification and design notations. In its purest form, development based on formal methods involves refinement and proof of correctness at each stage in the life cycle. But all formal methods are not created equal.

Shari Lawrence Pfleeger and Les Hatton [PFL97a] examined the effects of formal methods on the quality of the resulting software. They point out that, for some organizations, the changes in software development practices needed to support such techniques can be revolutionary. That is, there is not always a simple migration path from current practice to inclusion of formal methods. That's because the effective use of formal methods can require a radical change right at the beginning of the traditional software life cycle: how we capture and record customer requirements. Thus, the stakes in this area can be particularly high. For this reason, compelling evidence of the effectiveness of formal methods is highly desirable.

Susan Gerhart et al. [GER94] point out:

> There is no simple answer to the question: do formal methods pay off? Our cases provide a wealth of data but only scratch the surface of information available to address these questions. All cases involve so many interwoven factors that it is impossible to allocate payoff from formal methods versus other factors, such as quality of people or effects of other methodologies. Even where data was collected, it was difficult to interpret the results across the background of the organization and the various factors surrounding the application.

Indeed, Pfleeger and Hatton compare two similar systems: one system developed with formal methods and one not. The former has higher quality than the latter, but other possibilities explain this difference in quality, including that of careful attention to the requirements and design.

speed of the engine degrades as the size of the program increases, so proofs of correctness become less appropriate as program size increases.

- As Marv Schaefer [SCH89a] points out, too often people focus so much on the formalism and on deriving a formal proof that they ignore the underlying security properties to be ensured.
- The current state of program verification is less well developed than code production. As a result, correctness proofs have not been consistently and successfully applied to large production systems.

Program verification systems are being improved constantly. Larger programs are being verified in less time than before. Gerhart [GER89] succinctly describes the advantages and disadvantages of using formal methods, including proof of correctness. As program verification continues to mature, it may become a more important control to ensure the security of programs.

Validation

Formal verification is a particular instance of the more general approach to assuring correctness. There are many ways to show that each of a system's functions works correctly. **Validation** is the counterpart to verification, assuring that the system developers have implemented all requirements. Thus, validation makes sure that the developer is building the right product (according to the specification), and verification checks the quality of the implementation. For more details on validation in software engineering, see Shari Lawrence Pfleeger and Joanne Atlee [PFL10].

A program can be validated in several different ways:

- *Requirements checking.* One technique is to cross-check each system requirement with the system's source code or execution-time behavior. The goal is to demonstrate that the system does each thing listed in the functional requirements. This process is a narrow one, in the sense that it demonstrates only that the system does everything it should do. As we have pointed out, in security, we are equally concerned about prevention: making sure the system does *not* do the things it is not supposed to do. Requirements-checking seldom addresses this aspect of requirements compliance.
- *Design and code reviews.* As described earlier in this chapter, design and code reviews usually address system correctness (that is, verification). But a review can also address requirements implementation. To support validation, the reviewers scrutinize the design or the code to assure traceability from each requirement to design and code components, noting problems along the way (including faults, incorrect assumptions, incomplete or inconsistent behavior, or faulty logic). The success of this process depends on the rigor of the review.
- *System testing.* The programmers or an independent test team select data to check the system. These test data can be organized much like acceptance testing, so behaviors and data expected from reading the requirements document can be confirmed in the actual running of the system. The checking is done methodically to ensure completeness.

Other authors, notably James Whittaker and Herbert Thompson [WHI03a], Michael Andrews and James Whittaker [AND06], and Paco Hope and Ben Walther [HOP08], have described security-testing approaches.

Defensive Programming

The aphorism "offense sells tickets; defense wins championships" has been attributed to legendary University of Alabama football coach Paul "Bear" Bryant, Jr., Minnesota high school basketball coach Dave Thorson, and others. Regardless of its origin, the aphorism has a certain relevance to computer security as well. As we have already shown, the world is generally hostile: Defenders have to counter all possible attacks, whereas attackers have only to find one weakness to exploit. Thus, a strong defense is not only helpful, it is essential.

Program designers and implementers need not only write correct code but must also anticipate what could go wrong. As we pointed out earlier in this chapter, a program expecting a date as an input must also be able to handle incorrectly formed inputs such as 31-Nov-1929 and 42-Mpb-2030. Kinds of incorrect inputs include

- *value inappropriate for data type*, such as letters in a numeric field or M for a true/false item
- *value out of range for given use*, such as a negative value for age or the date 30 February
- *value unreasonable,* such as 250 kilograms of salt in a recipe
- *value out of scale or proportion,* for example, a house description with 4 bedrooms and 300 bathrooms.
- *incorrect number of parameters*, because the system does not always protect a program from this fault
- *incorrect order of parameters*, for example, a routine that expects age, sex, date, but the calling program provides sex, age, date

Program designers must not only write correct code but must also anticipate what could go wrong.

As Microsoft says, secure software must be able to withstand attack itself:

> Software security is different. It is the property of software that allows it to continue to operate as expected even when under attack. Software security is not a specific library or function call, nor is it an add-on that magically transforms existing code. It is the holistic result of a thoughtful approach applied by all stakeholders throughout the software development life cycle. [MIC10a]

Trustworthy Computing Initiative

Microsoft had a serious problem with code quality in 2002. Flaws in its products appeared frequently, and it released patches as quickly as it could. But the sporadic nature of patch releases confused users and made the problem seem worse than it was.

The public relations problem became so large that Microsoft President Bill Gates ordered a total code development shutdown and a top-to-bottom analysis of security and coding practices. The analysis and progress plan became known as the Trusted Computing Initiative. In this effort all developers underwent security training, and secure software development practices were instituted throughout the company.

The effort seemed to have met its goal: The number of code patches went down dramatically, to a level of two to three critical security patches per month.

Design by Contract

The technique known as **design by contract**™ (a trademark of Eiffel Software) or **programming by contract** can assist us in identifying potential sources of error. The trademarked form of this technique involves a formal program development approach, but more widely, these terms refer to documenting for each program module its preconditions, postconditions, and invariants. Preconditions and postconditions are conditions necessary (expected, required, or enforced) to be true before the module begins and after it ends, respectively; invariants are conditions necessary to be true throughout the module's execution. Effectively, each module comes with a contract: It expects the preconditions to have been met, and it agrees to meet the postconditions. By having been explicitly documented, the program can check these conditions on entry and exit, as a way of defending against other modules that do not fulfill the terms of their contracts or whose contracts contradict the conditions of this module. Another way of achieving this effect is by using **assertions**, which are explicit statements about modules. Two examples of assertions are "this module accepts as input *age*, expected to be between 0 and 150 years" and "input *length* measured in meters, to be an unsigned integer between 10 and 20." These assertions are notices to other modules with which this module interacts and conditions this module can verify.

The calling program must provide correct input, but the called program must not compound errors if the input is incorrect. On sensing a problem, the program can either halt or continue. Simply halting (that is, terminating the entire thread of execution) is usually a catastrophic response to seriously and irreparably flawed data, but continuing is possible only if execution will not allow the effect of the error to expand. The programmer needs to decide on the most appropriate way to handle an error detected by a check in the program's code. The programmer of the called routine has several options for action in the event of incorrect input:

- *Stop*, or signal an error condition and return.
- *Generate an error message* and wait for user action.
- *Generate an error message* and reinvoke the calling routine from the top (appropriate if that action forces the user to enter a value for the faulty field).
- *Try to correct it* if the error is obvious (although this choice should be taken only if there is only one possible correction).
- *Continue, with a default or nominal value*, or *continue computation without the erroneous value*, for example, if a mortality prediction depends on age, sex, amount of physical activity, and history of smoking, on receiving an

inconclusive value for sex, the system could compute results for both male and female and report both.

- *Do nothing*, if the error is minor, superficial, and is certain not to cause further harm.

For more guidance on defensive programming, consult Pfleeger et al. [PFL02].

In this section we presented several characteristics of good, secure software. Of course, a programmer can write secure code that has none of these characteristics, and faulty software can exhibit all of them. These qualities are not magic; they cannot turn bad code into good. Rather, they are properties that many examples of good code reflect and practices that good code developers use; the properties are not a cause of good code but are paradigms that tend to go along with it. Following these principles affects the mindset of a designer or developer, encouraging a focus on quality and security; this attention is ultimately good for the resulting product.

Countermeasures that Don't Work

Unfortunately, a lot of good or good-sounding ideas turn out to be not so good on further reflection. Worse, humans have a tendency to fix on ideas or opinions, so dislodging a faulty opinion is often more difficult than concluding the opinion the first time.

In the security field, several myths remain, no matter how forcefully critics denounce or disprove them. The penetrate-and-patch myth is actually two problems: People assume that the way to really test a computer system is to have a crack team of brilliant penetration magicians come in, try to make it behave insecurely and if they fail (that is, if no faults are exposed) pronounce the system good.

The second myth we want to debunk is called security by obscurity, the belief that if a programmer just doesn't tell anyone about a secret, nobody will discover it. This myth has about as much value as hiding a key under a door mat.

Finally, we reject an outsider's conjecture that programmers are so smart they can write a program to identify all malicious programs. Sadly, as smart as programmers are, that feat can be proven to be impossible.

Penetrate-and-Patch

Because programmers make mistakes of many kinds, we can never be sure all programs are without flaws. We know of many practices that can be used during software development to lead to high assurance of correctness. Let us start with one technique that seems appealing but in fact does *not* lead to solid code.

Early work in computer security was based on the paradigm of **penetrate-and-patch**, in which analysts searched for and repaired flaws. Often, a top-quality tiger team (so called because of its ferocious dedication to finding flaws) would be convened to test a system's security by attempting to cause it to fail. The test was considered to be a proof of security; if the system withstood the tiger team's attacks, it must be secure, or so the thinking went.

Unfortunately, far too often the attempted proof instead became a process for generating counterexamples, in which not just one but several serious security problems were uncovered. The problem discovery in turn led to a rapid effort to "patch" the system to repair or restore the security. However, the patch efforts were largely useless, generally making the system *less* secure, rather than more, because they frequently introduced new faults even as they tried to correct old ones. (For more discussion on the futility of penetrating and patching, see Roger Schell's analysis in [SCH79].) There are at least four reasons why penetrate-and-patch is a misguided strategy.

- The pressure to repair a specific problem encourages developers to take a narrow focus on the fault itself and not on its context. In particular, the analysts often pay attention to the immediate cause of the failure and not to the underlying design or requirements faults.
- The fault often has nonobvious side effects in places other than the immediate area of the fault. For example, the faulty code might have created and never released a buffer that was then used by unrelated code elsewhere. The corrected version releases that buffer. However, code elsewhere now fails because it needs the buffer left around by the faulty code, but the buffer is no longer present in the corrected version.
- Fixing one problem often causes a failure somewhere else. The patch may have addressed the problem in only one place, not in other related places. Routine A is called by B, C, and D, but the maintenance developer knows only of the failure when B calls A. The problem appears to be in that interface, so the developer patches B and A to fix the issue, tests, B, A, and B and A together with inputs that invoke the B–A interaction. All appear to work. Only much later does another failure surface, that is traced to the C–A interface. A different programmer, unaware of B and D, addresses the problem in the C–A interface that, not surprisingly generates latent faults. In maintenance, few people see the big picture, especially not when working under time pressure.
- The fault cannot be fixed properly because system functionality or performance would suffer as a consequence. Only some instances of the fault may be fixed or the damage may be reduced but not prevented.

Penetrate-and-patch fails because it is hurried, misses the context of the fault, and focuses on one failure, not the complete system.

In some people's minds penetration testers are geniuses who can find flaws mere mortals cannot see; therefore, if code passes review by such a genius, it must be perfect. Good testers certainly have a depth and breadth of experience that lets them think quickly of potential weaknesses, such as similar flaws they have seen before. This wisdom of experience—useful as it is—is no guarantee of correctness.

People outside the professional security community still find it appealing to find and fix security problems as single aberrations. However, security professionals recommend a more structured and careful approach to developing secure code.

Security by Obscurity

Computer security experts use the term **security by** or **through obscurity** to describe the ineffective countermeasure of assuming the attacker will not find a vulnerability. Security by obscurity is the belief that a system can be secure as long as nobody outside its implementation group is told anything about its internal mechanisms. Hiding account passwords in binary files or scripts with the presumption that nobody will ever find them is a prime case. Another example of faulty obscurity is described in Sidebar 3-14, in which deleted text is not truly deleted. System owners assume an attacker will never guess, find, or deduce anything not revealed openly. Think, for example, of the dialer program described earlier in this chapter. The developer of that utility might have thought that hiding the 100-digit limitation would keep it from being found or used. Obviously that assumption was wrong.

Things meant to stay hidden seldom do. Attackers find and exploit many hidden things.

SIDEBAR 3-14 Hidden, But Not Forgotten

When is something gone? When you press the delete key, it goes away, right? Wrong.

By now you know that deleted files are not really deleted; they are moved to the recycle bin. Deleted mail messages go to the trash folder. And temporary Internet pages hang around for a few days in a history folder waiting for repeated interest. But you expect keystrokes to disappear with the delete key.

Microsoft Word saves all changes and comments since a document was created. Suppose you and a colleague collaborate on a document, you refer to someone else's work, and your colleague inserts the comment "this research is rubbish." You concur, so you delete the reference and your colleague's comment. Then you submit the paper to a journal for review and, as luck would have it, your paper is sent to the author whose work you disparaged. Then the reviewer happens to turn on change marking and finds not just the deleted reference but also your colleague's deleted comment. (See [BYE04].) If you really wanted to remove that text, you should have used the Microsoft Hidden Data Removal Tool. (Of course, inspecting the file with a binary editor is the only way you can be sure the offending text is truly gone.)

The Adobe PDF document format is a simpler format intended to provide a platform-independent way to display (and print) documents. Some people convert a Word document to PDF to eliminate hidden sensitive data.

That does remove the change-tracking data. But it preserves even invisible output. Some people create a white box to paste over data to be hidden, for example, to cut out part of a map or hide a profit column in a table. When you print the file, the box hides your sensitive information. But the PDF format preserves all layers in a document, so your recipient can effectively peel off the white box to reveal the hidden content. The NSA issued a report detailing steps to ensure that deletions are truly deleted [NSA05].

Or if you want to show that something *was* there and has been deleted, you can do that with the Microsoft Redaction Tool, which, presumably, deletes the underlying text and replaces it with a thick black line.

Auguste Kerckhoffs, a Dutch cryptologist of the 19th century, laid out several principles of solid cryptographic systems [KER83]. His second principle[3] applies to security of computer systems, as well:

> The system must not depend on secrecy, and security should not suffer if the system falls into enemy hands.

Note that Kerckhoffs did not advise giving the enemy the system, but rather he said that if the enemy should happen to obtain it by whatever means, security should not fail. There is no need to give the enemy an even break; just be sure that when (not if) the enemy learns of the security mechanism, that knowledge will not harm security. Johansson and Grimes [JOH08a] discuss the fallacy of security by obscurity in greater detail.

The term **work factor** means the amount of effort necessary for an adversary to defeat a security control. In some cases, such as password guessing, we can estimate the work factor by determining how much time it would take to test a single password, and multiplying by the total number of possible passwords. If the attacker can take a shortcut, for example, if the attacker knows the password begins with an uppercase letter, the work factor is reduced correspondingly. If the amount of effort is prohibitively high, for example, if it would take over a century to deduce a password, we can conclude that the security mechanism is adequate. (Note that some materials, such as diplomatic messages, may be so sensitive that even after a century they should not be revealed, and so we would need to find a protection mechanism strong enough that it had a longer work factor.)

We cannot assume the attacker will take the slowest route for defeating security; in fact, we have to assume a dedicated attacker will take whatever approach seems to be fastest. So, in the case of passwords, the attacker might have several approaches:

- Try all passwords, exhaustively enumerating them in some order, for example, shortest to longest.
- Guess common passwords.
- Watch as someone types a password.

3. "Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi."

- Bribe someone to divulge the password.
- Intercept the password between its being typed and used (as was done at Churchill High School).
- Pretend to have forgotten the password and guess the answers to the supposedly secret recovery.
- Override the password request in the application.

If we did a simple work factor calculation on passwords, we might conclude that it would take x time units times y passwords, for a work factor of $x*y/2$ assuming, on average, half the passwords have to be tried to guess the correct one. But if the attacker uses any but the first technique, the time could be significantly different. Thus, in determining work factor, we have to assume the attacker uses the easiest way possible, which might take minutes, not decades.

Security by obscurity is a faulty countermeasure because it assumes the attacker will always take the hard approach and never the easy one. Attackers are lazy, like most of us; they will find the labor-saving way if it exists. And that way may involve looking under the doormat to find a key instead of battering down the door. We remind you in later chapters when a countermeasure may be an instance of security by obscurity.

A Perfect Good–Bad Code Separator

Programs can send a man to the moon, restart a failing heart, and defeat a former champion of the television program Jeopardy. Surely they can separate good programs from bad, can't they? Unfortunately, not.

First, we have to be careful what we mean when we say a program is good. (We use the simple terms good and bad instead of even more nuanced terms such as secure, safe, or nonmalicious.) As Sidebar 3-11 explains, every program has side effects: It uses memory, activates certain machine hardware, takes a particular amount of time, not to mention additional activities such as reordering a list or even presenting an output in a particular color. We may see but not notice some of these. If a designer prescribes that output is to be presented in a particular shade of red, we can check that the program actually does that. However, in most cases, the output color is unspecified, so the designer or a tester cannot say a program is nonconforming or bad if the output appears in red instead of black. But if we cannot even decide whether such an effect is acceptable or not, how can a program do that? And the hidden effects (computes for 0.379 microseconds, uses register 2 but not register 4) are even worse to think about judging. Thus, we cannot now, and probably will never be able to, define precisely what we mean by good or bad well enough that a computer program could reliably judge whether other programs are good or bad.

Even if we could define "good" satisfactorily, a fundamental limitation of logic will get in our way. Although well beyond the scope of this book, the field of decidability or computability looks at whether some things can ever be programmed, not just today or using today's languages and machinery, but ever. The crux of computability is the so-called **halting problem**, which asks whether a computer program stops execution or runs forever. We can certainly answer that question for many programs. But the British mathematician Alan Turing[4] proved in 1936 (notably, well before the advent of modern computers) that it is impossible to write a program to solve the halting problem for any possible program and any possible stream of input. Our good program checker would fall into the halting problem trap: If we could identify all good programs we would solve the halting problem, which is provably unsolvable. Thus, we will never have a comprehensive good program checker.

This negative result does not say we cannot examine certain programs for goodness. We can, in fact, look at some programs and say they are bad, and we can even write code to detect programs that modify protected memory locations or exploit known security vulnerabilities. So, yes, we can detect *some* bad programs, just not all of them.

CONCLUSION

In this chapter we have surveyed programs and programming: errors programmers make and vulnerabilities attackers exploit. These failings can have serious consequences, as reported almost daily in the news. However, there are techniques to mitigate these shortcomings, as we described at the end of this chapter.

The problems recounted in this chapter form the basis for much of the rest of this book. Programs implement web browsers, website applications, operating systems, network technologies, cloud infrastructures, and mobile devices. A buffer overflow can happen in a spreadsheet program or a network appliance, although the effect is more localized in the former case than the latter. Still, you should keep the problems of this chapter in mind as you continue through the remainder of this book.

In the next chapter we consider the security of the Internet, investigating harm affecting a user. In this chapter we have implicitly focused on individual programs running on one computer, although we have acknowledged external actors, for example, when we explored transmission of malicious code. Chapter 4 involves both a local user and remote Internet of potential malice.

EXERCISES

1. Suppose you are a customs inspector. You are responsible for checking suitcases for secret compartments in which bulky items such as jewelry might be hidden. Describe the procedure you would follow to check for these compartments.
2. Your boss hands you a microprocessor and its technical reference manual. You are asked to check for undocumented features of the processor. Because of the number of possibilities, you cannot test every operation code with every combination of operands. Outline the strategy you would use to identify and characterize unpublicized operations.
3. Your boss hands you a computer program and its technical reference manual. You are asked to check for undocumented features of the program. How is this activity similar to the task of the previous exercises? How does it differ? Which is the more feasible? Why?

4. Alan Turing was also a vital contributor to Britain during World War II when he devised several techniques that succeeded at breaking German encrypted communications.

4. A program is written to compute the sum of the integers from 1 to 10. The programmer, well trained in reusability and maintainability, writes the program so that it computes the sum of the numbers from k to n. However, a team of security specialists scrutinizes the code. The team certifies that this program properly sets k to 1 and n to 10; therefore, the program is certified as being properly restricted in that it always operates on precisely the range 1 to 10. List different ways that this program can be sabotaged so that during execution it computes a different sum, such as 3 to 20.
5. One way to limit the effect of an untrusted program is confinement: controlling what processes have access to the untrusted program and what access the program has to other processes and data. Explain how confinement would apply to the earlier example of the program that computes the sum of the integers 1 to 10.
6. List three controls that could be applied to detect or prevent off-by-one errors.
7. The distinction between a covert storage channel and a covert timing channel is not clearcut. Every timing channel can be transformed into an equivalent storage channel. Explain how this transformation could be done.
8. List the limitations on the amount of information leaked per second through a covert channel in a multiaccess computing system.
9. An electronic mail system could be used to leak information. First, explain how the leakage could occur. Then, identify controls that could be applied to detect or prevent the leakage.
10. Modularity can have a negative as well as a positive effect. A program that is overmodularized performs its operations in very small modules, so a reader has trouble acquiring an overall perspective on what the system is trying to do. That is, although it may be easy to determine what individual modules do and what small groups of modules do, it is not easy to understand what they do in their entirety as a system. Suggest an approach that can be used during program development to provide this perspective.
11. You are given a program that purportedly manages a list of items through hash coding. The program is supposed to return the location of an item if the item is present or to return the location where the item should be inserted if the item is not in the list. Accompanying the program is a manual describing parameters such as the expected format of items in the table, the table size, and the specific calling sequence. You have only the object code of this program, not the source code. List the cases you would apply to test the correctness of the program's function.
12. You are writing a procedure to add a node to a doubly linked list. The system on which this procedure is to be run is subject to periodic hardware failures. The list your program is to maintain is of great importance. Your program must ensure the integrity of the list, even if the machine fails in the middle of executing your procedure. Supply the individual statements you would use in your procedure to update the list. (Your list should be fewer than a dozen statements long.) Explain the effect of a machine failure after each instruction. Describe how you would revise this procedure so that it would restore the integrity of the basic list after a machine failure.
13. Explain how information in an access log could be used to identify the true identity of an impostor who has acquired unauthorized access to a computing system. Describe several different pieces of information in the log that could be combined to identify the impostor.

14. Several proposals have been made for a processor that could decrypt encrypted data and machine instructions and then execute the instructions on the data. The processor would then encrypt the results. How would such a processor be useful? What are the design requirements for such a processor?

15. Explain in what circumstances penetrate-and-patch is a useful program maintenance strategy.

16. Describe a programming situation in which least privilege is a good strategy to improve security.

17. Explain why genetic diversity is a good principle for secure development. Cite an example of lack of diversity that has had a negative impact on security.

18. Describe how security testing differs from ordinary functionality testing. What are the criteria for passing a security test that differ from functional criteria?

19. (a) You receive an email message that purports to come from your bank. It asks you to click a link for some reasonable-sounding administrative purpose. How can you verify that the message actually did come from your bank?

(b) Now play the role of an attacker. How could you intercept the message described in part (a) and convert it to your purposes while still making both the bank and the customer think the message is authentic and trustworthy?

20. Open design would seem to favor the attacker, because it certainly opens the implementation and perhaps also the design for the attacker to study. Justify that open design overrides this seeming advantage and actually leads to solid security.

4

The Web—User Side

In this chapter:

- Attacks against browsers
- Attacks against and from web sites
- Attacks seeking sensitive data
- Attacks through email

In this chapter we move beyond the general programs of the previous chapter to more specific code that supports user interaction with the Internet. Certainly, Internet code has all the potential problems of general programs, and you should keep malicious code, buffer overflows, and trapdoors in mind as you read this chapter. However, in this chapter we look more specifically at the kinds of security threats and vulnerabilities that Internet access makes possible. Our focus here is on the user or client side: harm that can come to an individual user interacting with Internet locations. Then, in Chapter 6 we look at security networking issues largely outside the user's realm or control, problems such as interception of communications, replay attacks, and denial of service.

We begin this chapter by looking at browsers, the software most users perceive as the gateway to the Internet. As you already know, a browser is software with a relatively simple role: connect to a particular web address, fetch and display content from that address, and transmit data from a user to that address. Security issues for browsers arise from several complications to that simple description, such as these:

- A browser often connects to more than the one address shown in the browser's address bar.
- Fetching data can entail accesses to numerous locations to obtain pictures, audio content, and other linked content.
- Browser software can be malicious or can be corrupted to acquire malicious functionality.
- Popular browsers support add-ins, extra code to add new features to the browser, but these add-ins themselves can include corrupting code.

- Data display involves a rich command set that controls rendering, positioning, motion, layering, and even invisibility.
- The browser can access any data on a user's computer (subject to access control restrictions); generally the browser runs with the same privileges as the user.
- Data transfers to and from the user are invisible, meaning they occur without the user's knowledge or explicit permission.

On a local computer you might constrain a spreadsheet program so it can access files in only certain directories. Photo-editing software can be run offline to ensure that photos are not released to the outside. Users can even inspect the binary or text content of word-processing files to at least partially confirm that a document does not contain certain text.

Browsers connect users to outside networks, but few users can monitor the actual data transmitted

Unfortunately, none of these limitations are applicable to browsers. By their very nature, browsers interact with the outside network, and for most users and uses, it is infeasible to monitor the destination or content of those network interactions. Many web interactions start at site A but then connect automatically to sites B, C, and D, often without the user's knowledge, much less permission. Worse, once data arrive at site A, the user has no control over what A does.

A browser's effect is immediate and transitory: pressing a key or clicking a link sends a signal, and there is seldom a complete log to show what a browser communicated. In short, browsers are standard, straightforward pieces of software that expose users to significantly greater security threats than most other kinds of software. Not surprisingly, attacking the browser is popular and effective. Not only are browsers a popular target, they present many vulnerabilities for attack, as shown in Figure 4-1, which shows the number of vulnerabilities discovered in the major browsers (Google Chrome, Mozilla Firefox, Microsoft Internet Explorer, Opera, and Safari), as reported by Secunia.

With this list of potential vulnerabilities involving web sites and browsers, it is no wonder attacks on web users happen with alarming frequency. Notice, also, that when major vendors release patches to code, browsers are often involved. In this chapter we

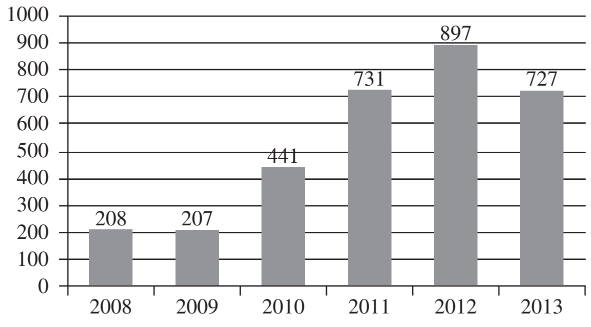


FIGURE 4-1 Number of Vulnerabilities Discovered in Browsers

look at security issues for end-users, usually involving browsers or web sites and usually directed maliciously against the user.

4.1 BROWSER ATTACKS

Assailants go after a browser to obtain sensitive information, such as account numbers or authentication passwords; to entice the user, for example, using pop-up ads; or to install malware. There are three attack vectors against a browser:

- Go after the operating system so it will impede the browser's correct and secure functioning.
- Tackle the browser or one of its components, add-ons, or plug-ins so its activity is altered.
- Intercept or modify communication to or from the browser.

We address operating system issues in Chapter 5 and network communications in Chapter 6. We begin this section by looking at vulnerabilities of browsers and ways to prevent such attacks.

Browser Attack Types

Because so many people (some of them relatively naïve or gullible) use them, browsers are inviting to attackers. A paper book is just what it appears; there is no hidden agent that can change the text on a page depending on who is reading. Telephone, television, and radio are pretty much the same: A signal from a central point to a user's device is usually uncorrupted or, if it is changed, the change is often major and easily detected, such as static or a fuzzy image. Thus, people naturally expect the same fidelity from a browser, even though browsers are programmable devices and signals are exposed to subtle modification during communication.

In this section we present several attacks passed through browsers.

Man-in-the-Browser

A **man-in-the-browser** attack is an example of malicious code that has infected a browser. Code inserted into the browser can read, copy, and redistribute anything the user enters in a browser. The threat here is that the attacker will intercept and reuse credentials to access financial accounts and other sensitive data.

Man-in-the-browser: Trojan horse that intercepts data passing through the browser

In January 2008, security researchers led by Liam Omurchu of Symantec detected a new Trojan horse, which they called SilentBanker. This code linked to a victim's browser as an add-on or browser helper object; in some versions it listed itself as a plug-in to display video. As a helper object, it set itself to intercept internal browser calls, including those to receive data from the keyboard, send data to a URL, generate or import a cryptographic key, read a file (including display that file on the screen), or connect to a site; this list includes pretty much everything a browser does.

SilentBanker started with a list of over 400 URLs of popular banks throughout the world. Whenever it saw a user going to one of those sites, it redirected the user's keystrokes through the Trojan horse and recorded customer details that it forwarded to remote computers (presumably controlled by the code's creators).

Banking and other financial transactions are ordinarily protected in transit by an encrypted session, using a protocol named SSL or HTTPS (which we explain in Chapter 6), and identified by a lock icon on the browser's screen. This protocol means that the user's communications are encrypted during transit. But remember that cryptography, although powerful, can protect only what it can control. Because SilentBanker was embedded within the browser, it intruded into the communication process as shown in Figure 4-2. When the user typed data, the operating system passed the characters to the browser. But before the browser could encrypt its data to transmit to the bank, SilentBanker intervened, acting as part of the browser. Notice that this timing vulnerability would not have been countered by any of the other security approaches banks use, such as an image that only the customer will recognize or two-factor authentication. Furthermore, the URL in the address bar looked and was authentic, because the browser actually did maintain a connection with the legitimate bank site.

SSL encryption is applied in the browser; data are vulnerable before being encrypted.

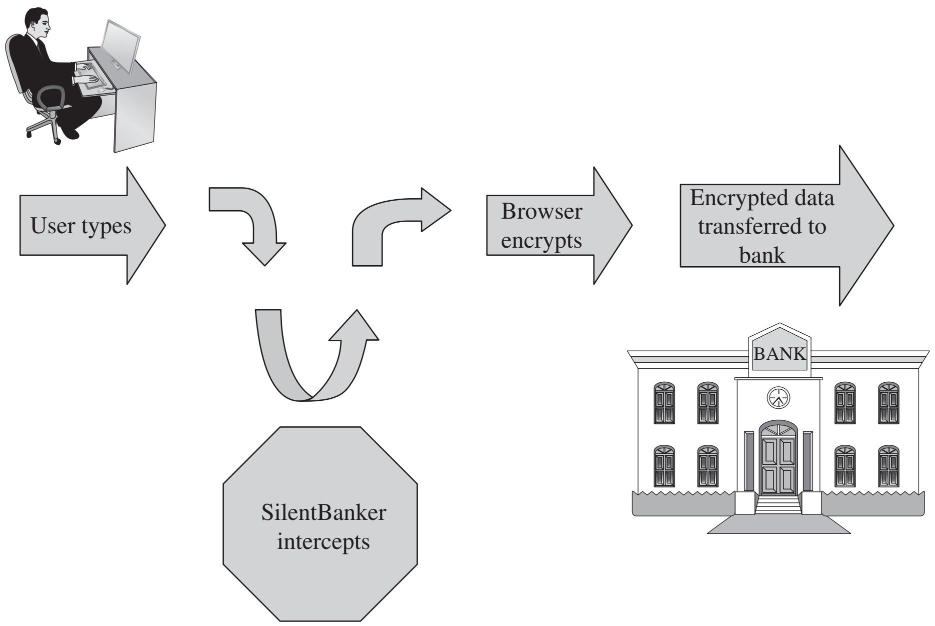
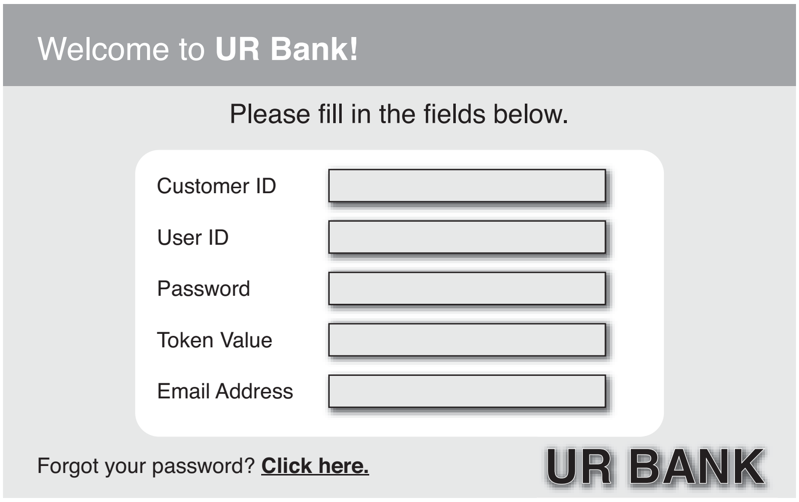


FIGURE 4-2 SilentBanker Operates in the Middle of the Browser



Welcome to **UR Bank!**

Please fill in the fields below.

Customer ID	
User ID	
Password	
Token Value	
Email Address	

Forgot your password? **Click here.**

FIGURE 4.3 Additional Data Obtained by Man in the Browser

As if intercepting details such as name, account number, and authentication data were not enough, SilentBanker also changed the effect of customer actions. So, for example, if a customer instructed the bank to transfer money to an account at bank A, SilentBanker converted that request to make the transfer go to its own account at bank B, which the customer's bank duly accepted as if it had come from the customer. When the bank returned its confirmation, SilentBanker changed the details before displaying them on the screen. Thus, the customer found out about the switch only after the funds failed to show up at bank A as expected.

A variant of SilentBanker intercepted other sensitive user data, using a display like the details shown in Figure 4-3. Users see many data request boxes, and this one looks authentic. The request for token value might strike some users as odd, but many users would see the bank's URL on the address bar and dutifully enter private data.

As you can see, man-in-the-browser attacks can be devastating because they represent a valid, authenticated user. The Trojan horse could slip neatly between the user and the bank's web site, so all the bank's content still looked authentic. SilentBanker had little impact on users, but only because it was discovered relatively quickly, and virus detectors were able to eradicate it promptly. Nevertheless, this piece of code demonstrates how powerful such an attack can be.

Keystroke Logger

We introduce another attack approach that is similar to a man in the browser. A **keystroke logger** (or **key logger**) is either hardware or software that records all keystrokes entered. The logger either retains these keystrokes for future use by the attacker or sends them to the attacker across a network connection.

As a hardware device, a keystroke logger is a small object that plugs into a USB port, resembling a plug-in wireless adapter or flash memory stick. Of course, to compromise a computer you have to have physical access to install (and later retrieve) the device. You also need to conceal the device so the user will not notice the logger (for example, installing it on the back of a desktop machine). In software, the logger is just a program installed like any malicious code. Such devices can capture passwords, login identities, and all other data typed on the keyboard. Although not limited to browser interactions, a keystroke logger could certainly record all keyboard input to the browser.

Page-in-the-Middle

A **page-in-the-middle** attack is another type of browser attack in which a user is redirected to another page. Similar to the man-in-the-browser attack, a page attack might wait until a user has gone to a particular web site and present a fictitious page for the user. As an example, when the user clicks "login" to go to the login page of any site, the attack might redirect the user to the attacker's page, where the attacker can also capture the user's credentials.

The admittedly slight difference between these two browser attacks is that the man-in-the-browser action is an example of an infected browser that may never alter the sites visited by the user but works behind the scenes to capture information. In a page-in-the-middle action, the attacker redirects the user, presenting different web pages for the user to see.

Program Download Substitution

Coupled with a page-in-the-middle attack is a download substitution. In a **download substitution**, the attacker presents a page with a desirable and seemingly innocuous program for the user to download, for example, a browser toolbar or a photo organizer utility. What the user does not know is that instead of or in addition to the intended program, the attacker downloads and installs malicious code.

A user agreeing to install a program has no way to know what that program will actually do.

The advantage for the attacker of a program download substitution is that users have been conditioned to be wary of program downloads, precisely for fear of downloading malicious code. In this attack, the user knows of and agrees to a download, not realizing what code is actually being installed. (Then again, users seldom know what really installs after they click [Yes].) This attack also defeats users' access controls that would normally block software downloads and installations, because the user intentionally accepts this software.

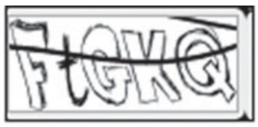
User-in-the-Middle

A different form of attack puts a human between two automated processes so that the human unwittingly helps spammers register automatically for free email accounts.

A **CAPTCHA** is a puzzle that supposedly only a human can solve, so a server application can distinguish between a human who makes a request and an automated program generating the same request repeatedly. Think of web sites that request votes to determine the popularity of television programs. To avoid being fooled by bogus votes from automated program scripts, the voting sites sometimes ensure interaction with an active human by using CAPTCHAs (an acronym for Completely Automated Public Turing test to tell Computers and Humans Apart—sometimes finding words to match a clever acronym is harder than doing the project itself).

The puzzle is a string of numbers and letters displayed in a crooked shape against a grainy background, perhaps with extraneous lines, like the images in Figure 4-4; the user has to recognize the string and type it into an input box. Distortions are intended to defeat optical character recognition software that might be able to extract the characters. (Figure 4-5 shows an amusing spoof of CAPTCHA puzzles.) The line is fine between what a human can still interpret and what is too distorted for pattern recognizers to handle, as described in Sidebar 4-1.

Sites offering free email accounts, such as Yahoo mail and Hotmail, use CAPTCHAs in their account creation phase to ensure that only individual humans obtain accounts. The mail services do not want their accounts to be used by spam senders who use thousands of new account names that are not yet recognized by spam filters; after using the account for a flood of spam, the senders will abandon those account names and move



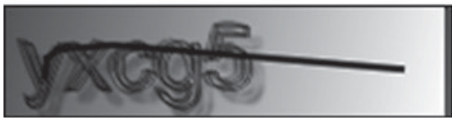
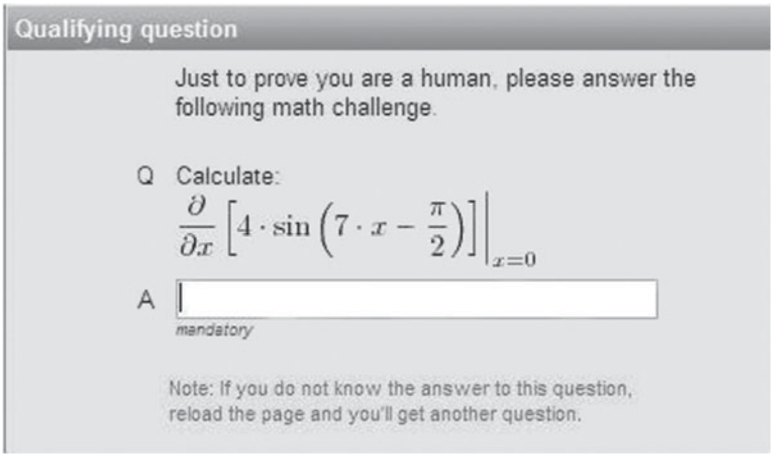


FIGURE 4-4 CAPTCHA Example



Qualifying question

Just to prove you are a human, please answer the following math challenge.

Q Calculate:

$$\frac{\partial}{\partial x}\left[4 \cdot \sin\left(7 \cdot x - \frac{\pi}{2}\right)\right]\Big|_{x=0}$$

A

mandatory

Note: If you do not know the answer to this question, reload the page and you'll get another question.

FIGURE 4-5 CAPTCHA Spoof

SIDEBAR 4-1 CAPTCHA? Gotcha!

We have seen how CAPTCHAs were designed to take advantage of how humans are much better at pattern recognition than are computers. But CAPTCHAs, too, have their vulnerabilities, and they can be defeated with the kinds of security engineering techniques we present in this book. As we have seen in every chapter, a wily attacker looks for a vulnerability to exploit and then designs an attack to take advantage of it.

In the same way, Jeff Yan and Ahmad Salah El Ahmad [YAN11] defeated CAPTCHAs by focusing on invariants—things that do not change even when the CAPTCHAs distort them. They investigated CAPTCHAs produced by major web services, including Google, Microsoft, and Yahoo for their free email services such as Hotmail. A now-defunct service called CAPTCHAservice.org provided CAPTCHAs to commercial web sites for a fee. Each of the characters in that service's CAPTCHAs had a different number of pixels, but the number of pixels for a given character remained constant when the character was distorted—an invariant that allowed Yan and El Ahmad to differentiate one character from another without having to recognize the character. Yahoo's CAPTCHAs used a fixed angle for image transformation. Yan and El Ahmad pointed out that "Exploiting invariants is a classic cryptanalysis strategy. For example, differential cryptanalysis works by observing that a subset of pairs of plaintexts has an invariant relationship preserved through numerous cipher rounds. Our work demonstrates that exploiting invariants is also effective for studying CAPTCHA robustness."

Yan and Ahmad successfully used simple techniques to defeat the CAPTCHAs, such as pixel counts, color-filling segmentation, and histogram analysis. And they defeated two kinds of invariants: pixel level and string level. A pixel-level invariant can be exploited by processing the CAPTCHA images at the pixel level, based on what does not change (such as number of pixels or angle of character). String-level invariants do not change across the entire length of the string. For example, Microsoft in 2007 used a CAPTCHA with a constant length of text in the challenge string; this invariant enabled Yan and El Ahmad to identify and segment connected characters. Reliance on dictionary words is another string-level invariant; as we saw with dictionary-based passwords, the dictionary limits the number of possible choices.

So how can these vulnerabilities be eliminated? By introducing some degree of randomness, such as an unpredictable number of characters in a string of text. Yan and El Ahmad recommend "introduc[ing] more types of global shape patterns and have them occur in random order, thus making it harder for computers to differentiate each type." Google's CAPTCHAs allow the characters to run together; it may be possible to remove the white space between characters, as long as readability does not suffer. Yan and El Ahmad point out that this kind of security engineering analysis leads to more robust CAPTCHAs, a process that mirrors what we have already seen in other security techniques, such as cryptography and software development.

on to another bunch. Thus, spammers need a constant source of new accounts, and they would like to automate the process of obtaining new ones.

Petmail (http://petmail.lothar.com) is a proposed anti-spam email system. In the description the author hypothesizes the following man-in-the-middle attack against CAPTCHAs from free email account vendors. First, the spam sender creates a site that will attract visitors; the author suggests a site with pornographic photos. Second, the spammer requires people to solve a CAPTCHA in order to enter the site and see the photos. At the moment a user requests access, the spam originator automatically generates a request to create a new email account (Hotmail, for example). Hotmail presents a CAPTCHA, which the spammer then presents to the pornography requester. When the requester enters the solution, the spammer forwards that solution back to Hotmail. If the solution succeeds, the spammer has a new account and allows the user to see the photos; if the solution fails, the spammer presents a new CAPTCHA challenge to the user. In this way, the attacker in the middle splices together two interactions by inserting a small amount of the account creation thread into the middle of the photo access thread. The user is unaware of the interaction in the middle.

How Browser Attacks Succeed: Failed Identification and Authentication

The central failure of these in-the-middle attacks is faulty authentication. If A cannot be assured that the sender of a message is really B, A cannot trust the authenticity of anything in the message. In this section we consider authentication in different contexts.

Human Authentication

As we first stated in Chapter 2, authentication is based on something you know, are, or possess. People use these qualities all the time in developing face-to-face authentication. Examples of human authentication techniques include a driver's license or identity card, a letter of introduction from a mutual acquaintance or trusted third party, a picture (for recognition of a face), a shared secret, or a word. (The original use of "password" was a word said to a guard to allow the speaker to pass a checkpoint.) Because we humans exercise judgment, we develop a sense for when an authentication is adequate and when something just doesn't seem right. Of course, humans can also be fooled, as described in Sidebar 4-2.

SIDEBAR 4-2 Colombian Hostages Freed by Man-in-the-Middle Trick

Colombian guerrillas captured presidential candidate Ingrid Betancourt in 2002, along with other political prisoners. The guerillas, part of the FARC movement, had considered Betancourt and three U.S. contractors to be their most valuable prisoners. The captives were liberated in 2008 through a scheme involving two infiltrations: one infiltration of the local group that held the hostages, and the other of the central FARC command structure.

Having infiltrated the guerillas' central command organization, Colombian defense officials tricked the local FARC commander, known as Cesar,