3

Programs and Programming

In this chapter:

- Programming oversights: buffer overflows, off-by-one errors, incomplete mediation, time-of-check to time-of-use errors
- Malicious code: viruses, worms, Trojan horses
- Developer countermeasures: program development techniques, security principles
- Ineffective countermeasures

Programs are simple things but they can wield mighty power. Think about them for a minute: Programs are just strings of 0s and 1s, representing elementary machine commands such as move one data item, compare two data items, or branch to a different command. Those primitive machine commands implement higher-level programming language constructs such as conditionals, repeat loops, case selection, and arithmetic and string operations. And those programming language constructs give us pacemaker functions, satellite control, smart-home technology, traffic management, and digital photography, not to mention streaming video and social networks. The Intel 32- and 64-bit instruction set has about 30 basic primitives (such as move, compare, branch, increment and decrement, logical operations, arithmetic operations, trigger I/O, generate and service interrupts, push, pop, call, and return) and specialized instructions to improve performance on computations such as floating point operations or cryptography. These few machine commands are sufficient to implement the vast range of programs we know today.

Most programs are written in higher-level languages such as Java, C, C++, Perl, or Python; programmers often use libraries of code to build complex programs from pieces written by others. But most people are not programmers; instead, they use already written applications for word processing, web browsing, graphics design, accounting, and the like without knowing anything about the underlying program code. People do not expect to need to understand how power plants operate in order to turn on an electric

light. But if the light does not work, the problem could be anywhere from the power plant to the light bulb, and suddenly the user needs to trace potential problems from one end to the other. Although the user does not need to become a physicist or electrical engineer, a general understanding of electricity helps determine how to overcome the problem, or at least how to isolate faults under the user's control (burned out bulb, unplugged lamp).

In this chapter we describe security problems in programs and programming. As with the light, a problem can reside anywhere between the machine hardware and the user interface. Two or more problems may combine in negative ways, some problems can be intermittent or occur only when some other condition is present, and the impact of problems can range from annoying (perhaps not even perceptible) to catastrophic.

Security failures can result from intentional or nonmalicious causes; both can cause harm.

In Chapter 1 we introduce the notion of motive, observing that some security problems result from nonmalicious oversights or blunders, but others are intentional. A malicious attacker can exploit a nonmalicious flaw to cause real harm. Thus, we now study several common program failings to show how simple errors during programming can lead to large-scale problems during execution. Along the way we describe real attacks that have been caused by program flaws. (We use the term *flaw* because many security professionals use that term or the more evocative term *bug*. However, as you can see in Sidebar 3-1, the language for describing program problems is not universal.)

SIDEBAR 3-1 The Terminology of (Lack of) Quality

Thanks to Admiral Grace Murray Hopper, we casually call a software problem a "bug." [KID98] But that term can mean different things depending on context: a mistake in interpreting a requirement, a syntax error in a piece of code, or the (as-yet-unknown) cause of a system crash. The Institute of Electronics and Electrical Engineers (IEEE) suggests using a standard terminology (in IEEE Standard 729) for describing bugs in our software products [IEE83].

When a human makes a mistake, called an **error**, in performing some software activity, the error may lead to a **fault**, or an incorrect step, command, process, or data definition in a computer program, design, or documentation. For example, a designer may misunderstand a requirement and create a design that does not match the actual intent of the requirements analyst and the user. This design fault is an encoding of the error, and it can lead to other faults, such as incorrect code and an incorrect description in a user manual. Thus, a single error can generate many faults, and a fault can reside in any development or maintenance product.

A **failure** is a departure from the system's required behavior. It can be discovered before or after system delivery, during testing, or during operation and maintenance. Since the requirements documents can contain

faults, a failure indicates that the system is not performing as required, even though it may be performing as specified.

Thus, a fault is an inside view of the system, as seen by the eyes of the developers, whereas a failure is an outside view: a problem that the user sees. Every failure has at least one fault as its root cause. But not every fault corresponds to a failure; for example, if faulty code is never executed or a particular state is never entered, the fault will never cause the code to fail.

Although software engineers usually pay careful attention to the distinction between faults and failures, security engineers rarely do. Instead, security engineers use **flaw** to describe both faults and failures. In this book, we use the security terminology; we try to provide enough context so that you can understand whether we mean fault or failure.

3.1 UNINTENTIONAL (NONMALICIOUS) PROGRAMMING OVERSIGHTS

Programs and their computer code are the basis of computing. Without a program to guide its activity, a computer is pretty useless. Because the early days of computing offered few programs for general use, early computer users had to be programmers too—they wrote the code and then ran it to accomplish some task. Today's computer users sometimes write their own code, but more often they buy programs off the shelf; they even buy or share code components and then modify them for their own uses. And all users gladly run programs all the time: spreadsheets, music players, word processors, browsers, email handlers, games, simulators, and more. Indeed, code is initiated in myriad ways, from turning on a mobile phone to pressing "start" on a coffee-maker or microwave oven. But as the programs have become more numerous and complex, users are more frequently unable to know what the program is really doing or how.

More importantly, users seldom know whether the program they are using is producing correct results. If a program stops abruptly, text disappears from a document, or music suddenly skips passages, code may not be working properly. (Sometimes these interruptions are intentional, as when a CD player skips because the disk is damaged or a medical device program stops in order to prevent an injury.) But if a spreadsheet produces a result that is off by a small amount or an automated drawing package doesn't align objects exactly, you might not notice—or you notice but blame yourself instead of the program for the discrepancy.

These flaws, seen and unseen, can be cause for concern in several ways. As we all know, programs are written by fallible humans, and program flaws can range from insignificant to catastrophic. Despite significant testing, the flaws may appear regularly or sporadically, perhaps depending on many unknown and unanticipated conditions.

Program flaws can have two kinds of security implications: They can cause integrity problems leading to harmful output or action, and they offer an opportunity for exploitation by a malicious actor. We discuss each one in turn.

- A program flaw can be a fault affecting the correctness of the program's result—that is, a fault can lead to a failure. Incorrect operation is an integrity failing. As we saw in Chapter 1, integrity is one of the three fundamental security properties

of the C-I-A triad. Integrity involves not only correctness but also accuracy, precision, and consistency. A faulty program can also inappropriately modify previously correct data, sometimes by overwriting or deleting the original data. Even though the flaw may not have been inserted maliciously, the outcomes of a flawed program can lead to serious harm.

- On the other hand, even a flaw from a benign cause can be exploited by someone malicious. If an attacker learns of a flaw and can use it to manipulate the program's behavior, a simple and nonmalicious flaw can become part of a malicious attack.

Benign flaws can be—often are—exploited for malicious impact.

Thus, in both ways, program correctness becomes a security issue as well as a general quality problem. In this chapter we examine several programming flaws that have security implications. We also show what activities during program design, development, and deployment can improve program security.

Buffer Overflow

We start with a particularly well known flaw, the buffer overflow. Although the basic problem is easy to describe, locating and preventing such difficulties is challenging. Furthermore, the impact of an overflow can be subtle and disproportionate to the underlying oversight. This outsized effect is due in part to the exploits that people have achieved using overflows. Indeed, a buffer overflow is often the initial toehold for mounting a more damaging strike. Most buffer overflows are simple programming oversights, but they can be used for malicious ends. See Sidebar 3-2 for the story of a search for a buffer overflow.

Buffer overflows often come from innocent programmer oversights or failures to document and check for excessive data.

This example was not the first buffer overflow, and in the intervening time—approaching two decades—far more buffer overflows have been discovered. However, this example shows clearly the mind of an attacker. In this case, David was trying to improve security—he happened to be working for one of this book's authors at the time—but attackers work to defeat security for reasons such as those listed in Chapter 1. We now investigate sources of buffer overflow attacks, their consequences, and some countermeasures.

Anatomy of Buffer Overflows

A string overruns its assigned space or one extra element is shoved into an array; what's the big deal, you ask? To understand why buffer overflows are a major security issue, you need to understand how an operating system stores code and data.

As noted above, buffer overflows have existed almost as long as higher-level programming languages with arrays. Early overflows were simply a minor annoyance to

SIDEBAR 3-2 My Phone Number is 5656 4545 7890 1234 2929 2929 2929 . . .

In 1999, security analyst David Litchfield [LIT99] was intrigued by buffer overflows. He had both an uncanny sense for the kind of program that would contain overflows and the patience to search for them diligently. He happened onto the Microsoft Dialer program, dialer.exe.

Dialer was a program for dialing a telephone. Before cell phones, WiFi, broadband, and DSL, computers were equipped with modems by which they could connect to the land-based telephone network; a user would dial an Internet service provider and establish a connection across a standard voice telephone line. Many people shared one line between voice and computer (data) communication. You could look up a contact's phone number, reach for the telephone, dial the number, and converse; but the computer's modem could dial the same line, so you could feed the number to the modem from an electronic contacts list, let the modem dial your number, and pick up the receiver when your called party answered. Thus, Microsoft provided Dialer, a simple utility program to dial a number with the modem. (As of 2014, dialer.exe was still part of Windows 10, although the buffer overflow described here was patched shortly after David reported it.)

David reasoned that Dialer had to accept phone numbers of different lengths, given country variations, outgoing access codes, and remote signals (for example, to enter an extension number). But he also suspected there would be an upper limit. So he tried dialer.exe with a 20-digit phone number and everything worked fine. He tried 25 and 50, and the program still worked fine. When he tried a 100-digit phone number, the program crashed. The programmer had probably made an undocumented and untested decision that nobody would ever try to dial a 100-digit phone number . . . except David.

Having found a breaking point, David then began the interesting part of his work: Crashing a program demonstrates a fault, but exploiting that flaw shows how serious the fault is. By more experimentation, David found that the number to dial was written into the stack, the data structure that stores parameters and return addresses for embedded program calls. The dialer.exe program is treated as a program call by the operating system, so by controlling what dialer.exe overwrote, David could redirect execution to continue anywhere with any instructions he wanted. The full details of his exploitation are given in [LIT99].

programmers and users, a cause of errors and sometimes even system crashes. More recently, however, attackers have used them as vehicles to cause first a system crash and then a controlled failure with a serious security implication. The large number of security vulnerabilities based on buffer overflows shows that developers must pay more attention now to what had previously been thought to be just a minor annoyance.

Memory Allocation

Memory is a limited but flexible resource; any memory location can hold any piece of code or data. To make managing computer memory efficient, operating systems jam one data element next to another, without regard for data type, size, content, or purpose.[1] Users and programmers seldom know, much less have any need to know, precisely which memory location a code or data item occupies.

Computers use a pointer or register known as a **program counter** that indicates the next instruction. As long as program flow is sequential, hardware bumps up the value in the program counter to point just after the current instruction as part of performing that instruction. Conditional instructions such as IF(), branch instructions such as loops (WHILE, FOR) and unconditional transfers such as GOTO or CALL divert the flow of execution, causing the hardware to put a new destination address into the program counter. Changing the program counter causes execution to transfer from the bottom of a loop back to its top for another iteration. Hardware simply fetches the byte (or bytes) at the address pointed to by the program counter and executes it as an instruction.

Instructions and data are all binary strings; only the context of use says a byte, for example, 0x41 represents the letter A, the number 65, or the instruction to move the contents of register 1 to the stack pointer. If you happen to put the data string "A" in the path of execution, it will be executed as if it were an instruction. In Figure 3-1 we show a typical arrangement of the contents of memory, showing code, local data, the heap (storage for dynamically created data), and the stack (storage for subtask call and return data). As you can see, instructions move from the bottom (low addresses) of memory up; left unchecked, execution would proceed through the local data area and into the heap and stack. Of course, execution typically stays within the area assigned to program code.

Not all binary data items represent valid instructions. Some do not correspond to any defined operation, for example, operation 0x78 on a machine whose instructions are all numbers between 0x01 and 0x6f. Other invalid forms attempt to use nonexistent hardware features, such as a reference to register 9 on a machine with only eight hardware registers.

To help operating systems implement security, some hardware recognizes more than one mode of instruction: so-called privileged instructions that can be executed only when the processor is running in a protected mode. Trying to execute something that does not correspond to a valid instruction or trying to execute a privileged instruction when not in the proper mode will cause a **program fault**. When hardware generates a program fault, it stops the current thread of execution and transfers control to code that will take recovery action, such as halting the current process and returning control to the supervisor.

---

1. Some operating systems do separate executable code from nonexecutable data, and some hardware can provide different protection to memory addresses containing code as opposed to data. Unfortunately, however, for reasons including simple design and performance, most operating systems and hardware do not implement such separation. We ignore the few exceptions in this chapter because the security issue of buffer overflow applies even within a more constrained system. Designers and programmers need to be aware of buffer overflows, because a program designed for use in one environment is sometimes transported to another less protected one.

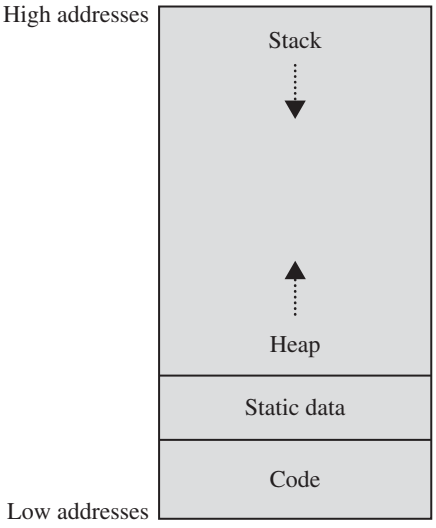


FIGURE 3-1 Typical Memory Organization

Code and Data

Before we can explain the real impact of buffer overflows, we need to clarify one point: Code, data, instructions, the operating system, complex data structures, user programs, strings, downloaded utility routines, hexadecimal data, decimal data, character strings, code libraries, photos, and everything else in memory are just strings of 0s and 1s; think of it all as bytes, each containing a number. The computer pays no attention to how the bytes were produced or where they came from. Each computer instruction determines how data values are interpreted: An Add instruction implies the data item is interpreted as a number, a Move instruction applies to any string of bits of arbitrary form, and a Jump instruction assumes the target is an instruction. But at the machine level, nothing prevents a Jump instruction from transferring into a data field or an Add command operating on an instruction, although the results may be unpleasant. Code and data are bit strings interpreted in a particular way.

In memory, code is indistinguishable from data. The origin of code (respected source or attacker) is also not visible.

You do not usually try to execute data values or perform arithmetic on instructions. But if 0x1C is the operation code for a Jump instruction, and the form of a Jump instruction is 1C *displ*, meaning execute the instruction at the address *displ* bytes ahead of this instruction, the string 0x1C0A is interpreted as jump forward 10 bytes. But, as shown in Figure 3-2, that same bit pattern represents the two-byte decimal integer 7178. So storing the number 7178 in a series of instructions is the same as having programmed a Jump. Most higher-level-language programmers do not care about the representation of instructions in memory, but curious investigators can readily find the correspondence. Manufacturers publish references specifying precisely the behavior of their chips, and

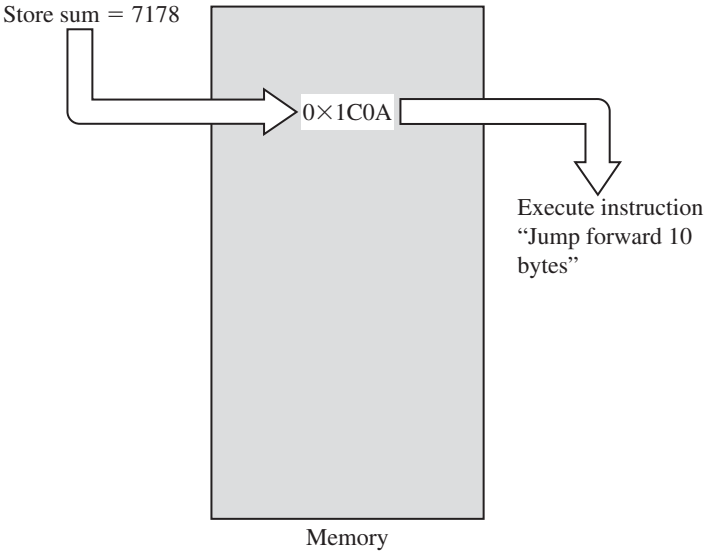


FIGURE 3-2 Bit Patterns Can Represent Data or Instructions

utility programs such as compilers, assemblers, and disassemblers help interested programmers develop and interpret machine instructions.

Usually we do not treat code as data, or vice versa; attackers sometimes do, however, especially in memory overflow attacks. The attacker's trick is to cause data to spill over into executable code and then to select the data values such that they are interpreted as valid instructions to perform the attacker's goal. For some attackers this is a two-step goal: First cause the overflow and then experiment with the ensuing action to cause a desired, predictable result, just as David did.

Harm from an Overflow

Let us suppose a malicious person understands the damage that can be done by a buffer overflow; that is, we are dealing with more than simply a normal, bumbling programmer. The malicious programmer thinks deviously: What data values could I insert to cause mischief or damage, and what planned instruction codes could I force the system to execute? There are many possible answers, some of which are more malevolent than others. Here, we present two buffer overflow attacks that are used frequently. (See [ALE96] for more details.)

First, the attacker may replace code in the system space. As shown in Figure 3-3, memory organization is not as simple as shown in Figure 3-1. The operating system's code and data coexist with a user's code and data. The heavy line between system and user space is only to indicate a logical separation between those two areas; in practice, the distinction is not so solid.

Remember that every program is invoked by an operating system that may run with higher privileges than those of a regular program. Thus, if the attacker can gain control

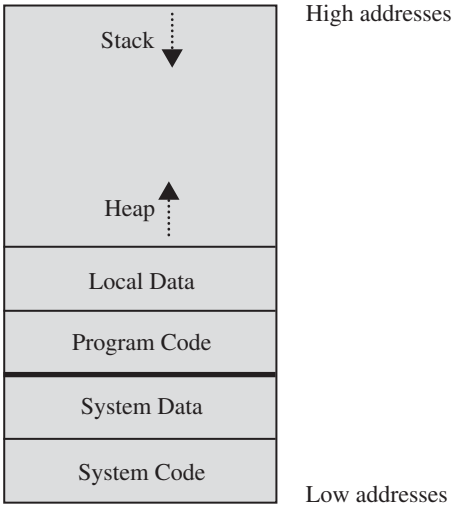


FIGURE 3-3 Memory Organization with User and System Areas

by masquerading as the operating system, the attacker can execute commands in a powerful role. Therefore, by replacing a few instructions right after returning from his or her own procedure, the attacker regains control from the operating system, possibly with raised privileges. This technique is called **privilege escalation**. If the buffer overflows into system code space, the attacker merely inserts overflow data that correspond to the machine code for instructions.

In the other kind of attack, the intruder may wander into an area called the stack and heap. Subprocedure calls are handled with a stack, a data structure in which the most recent item inserted is the next one removed (last arrived, first served). This structure works well because procedure calls can be nested, with each return causing control to transfer back to the immediately preceding routine at its point of execution. Each time a procedure is called, its parameters, the return address (the address immediately after its call), and other local values are pushed onto a stack. An old stack pointer is also pushed onto the stack, and a stack pointer register is reloaded with the address of these new values. Control is then transferred to the subprocedure.

As the subprocedure executes, it fetches parameters that it finds by using the address pointed to by the stack pointer. Typically, the stack pointer is a register in the processor. Therefore, by causing an overflow into the stack, the attacker can change either the old stack pointer (changing the context for the calling procedure) or the return address (causing control to transfer where the attacker intends when the subprocedure returns). Changing the context or return address allows the attacker to redirect execution to code written by the attacker.

In both these cases, the assailant must experiment a little to determine where the overflow is and how to control it. But the work to be done is relatively small—probably a day or two for a competent analyst. These buffer overflows are carefully explained in a paper by Mudge [MUD95] (real name, Pieter Zatko) of the famed l0pht computer

security group. Pincus and Baker [PIN04] reviewed buffer overflows ten years after Mudge and found that, far from being a minor aspect of attack, buffer overflows had been a significant attack vector and had spawned several other new attack types. That pattern continues today.

An alternative style of buffer overflow occurs when parameter values are passed into a routine, especially when the parameters are passed to a web server on the Internet. Parameters are passed in the URL line, with a syntax similar to

```
http://www.somesite.com/subpage/userinput.asp?
parm1=(808)555-1212
```

In this example, the application script userinput receives one parameter, parm1 with value (808)555-1212 (perhaps a U.S. telephone number). The web browser on the caller's machine will accept values from a user who probably completes fields on a form. The browser encodes those values and transmits them back to the server's web site.

The attacker might question what the server would do with a really long telephone number, say, one with 500 or 1000 digits. This is precisely the question David asked in the example we described in Sidebar 3-2. Passing a very long string to a web server is a slight variation on the classic buffer overflow, but no less effective.

Overwriting Memory

Now think about a buffer overflow. If you write an element past the end of an array or you store an 11-byte string in a 10-byte area, that extra data has to go somewhere; often it goes immediately after the last assigned space for the data.

A buffer (or array or string) is a space in which data can be held. A buffer resides in memory. Because memory is finite, a buffer's capacity is finite. For this reason, in many programming languages the programmer must declare the buffer's maximum size so that the compiler can set aside that amount of space.

Let us look at an example to see how buffer overflows can happen. Suppose a C language program contains the declaration

```
char sample[10];
```

The compiler sets aside 10 bytes to store this buffer, one byte for each of the 10 elements of the array, denoted sample[0] through sample[9]. Now we execute the statement

```
sample[10] = 'B';
```

The subscript is out of bounds (that is, it does not fall between 0 and 9), so we have a problem. The nicest outcome (from a security perspective) is for the compiler to detect the problem and mark the error during compilation, which the compiler could do in this case. However, if the statement were

```
sample[i] = 'B';
```

then the compiler could not identify the problem until i was set during execution either to a proper value (between 0 and 9) or to an out-of-bounds subscript (less than 0 or greater than 9). It would be useful if, during execution, the system produced an error message warning of a subscript exception. Unfortunately, in some languages, buffer

sizes do not have to be predefined, so there is no way to detect an out-of-bounds error. More importantly, the code needed to check each subscript against its potential maximum value takes time and space during execution, and resources are applied to catch a problem that occurs relatively infrequently. Even if the compiler were careful in analyzing the buffer declaration and use, this same problem can be caused with pointers, for which there is no reasonable way to define a proper limit. Thus, some compilers do not generate the code to check for exceeding bounds.

Implications of Overwriting Memory

Let us more closely examine the problem of overwriting memory. Be sure to recognize that the potential overflow causes a serious problem only in some instances. The problem's occurrence depends on what is adjacent to the array `sample`. For example, suppose each of the ten elements of the array `sample` is filled with the letter A and the erroneous reference uses the letter B, as follows:

```
for (i=0; i<=9; i++)
      sample[i] = 'A';
sample[10] = 'B'
```

All program and data elements are in memory during execution, sharing space with the operating system, other code, and resident routines. So four cases must be considered in deciding where the 'B' goes, as shown in Figure 3-4. If the extra character overflows into the user's data space, it simply overwrites an existing variable value (or it may be written into an as-yet unused location), perhaps affecting the program's result but affecting no other program or data.

In the second case, the 'B' goes into the user's program area. If it overlays an already executed instruction (which will not be executed again), the user should perceive no effect. If it overlays an instruction that is not yet executed, the machine will try to execute an instruction with operation code 0x42, the internal code for the character 'B'. If there is no instruction with operation code 0x42, the system will halt on an illegal instruction exception. Otherwise, the machine will use subsequent bytes as if they were the rest of the instruction, with success or failure depending on the meaning of the contents. Again, only the user is likely to experience an effect.

The most interesting cases (from a security perspective) occur when the system owns the space immediately after the array that overflows. Spilling over into system data or code areas produces results similar to those for the user's space: computing with a faulty value or trying to execute an operation.

Program procedures use both **local** data, data used strictly within one procedure, and **shared** or **common** or **global data**, which are shared between two or more procedures. Memory organization can be complicated, but we simplify the layout as in Figure 3-5. In that picture, local data are stored adjacent to the code of a procedure. Thus, as you can see, a data overflow either falls strictly within a data space or it spills over into an adjacent code area. The data end up on top of one of

- another piece of your data
- an instruction of yours

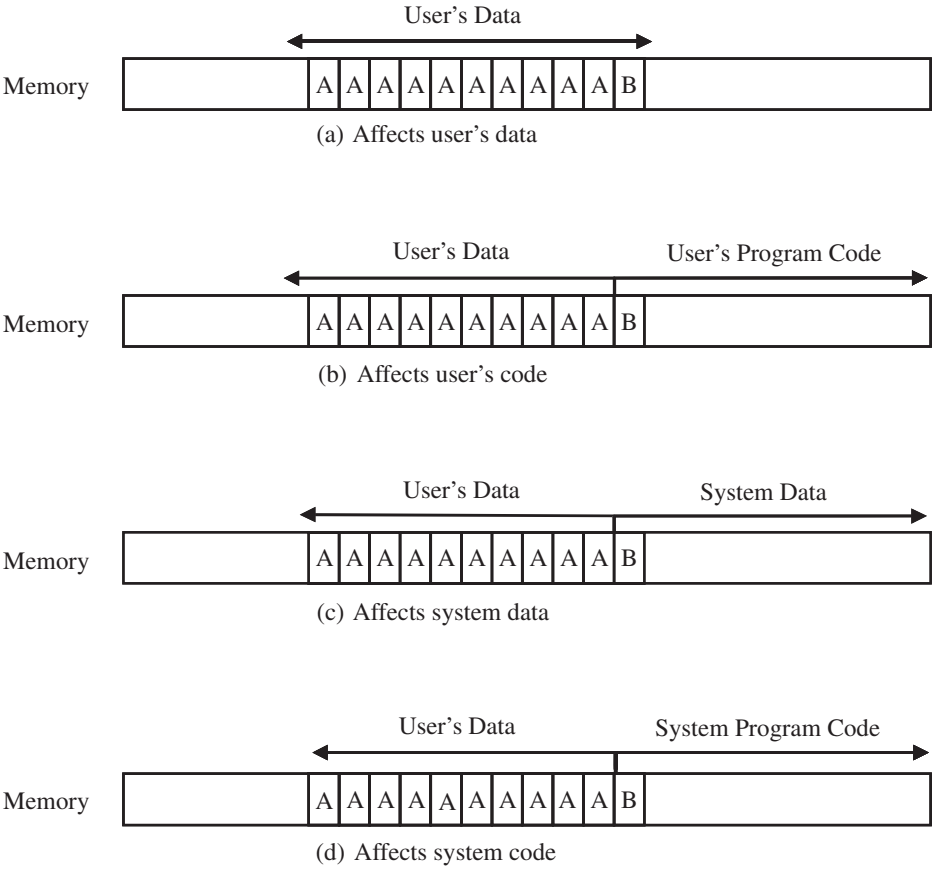


FIGURE 3-4 One-Character Overflow

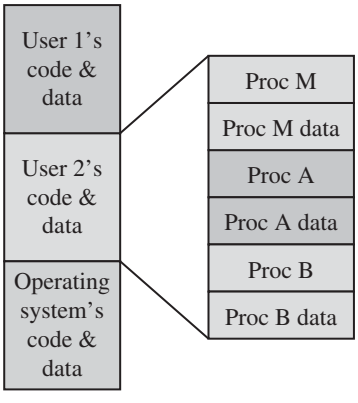


FIGURE 3-5 Memory of Different Procedures for Different Users

- data or code belonging to another program
- data or code belonging to the operating system

We consider each of these cases separately.

Affecting Your Own Data

Modifying your own data, especially with an unintended value, will obviously affect your computing. Perhaps a loop will repeat too many or too few times, a sum will be compromised, or a date will become garbled. You can imagine these possibilities for yourself. The error may be so egregious that you will easily recognize something is wrong, but a more subtle failure may escape your notice, perhaps forever.

From a security standpoint, few system controls protect you from this kind of error: You own your data space and anything you want to store there is your business. Some, but not all, programming languages generate checking code for things like arrays to ensure that you store elements only within the space allocated. For this reason, the defensive programming technique (discussed later in this chapter) recommends that you always check to ensure that array elements and strings are within their boundaries. As Sidebar 3-3 demonstrates, sometimes such an error lies dormant for a long time.

Affecting an Instruction of Yours

Again, the failure of one of your instructions affects you, and systems give wide latitude to what you can do to yourself. If you store a string that does not represent a valid or permitted instruction, your program may generate a fault and halt, returning control to

---

SIDEBAR 3-3 Too Many Computers

The ARPANET, precursor to today's Internet, began operation in 1969. Stephen Crocker and Mary Bernstein [CRO89] exhaustively studied the root causes of 17 catastrophic failures of the ARPANET, failures that brought down the entire network or a significant portion of it.

As you might expect, many of these failures occurred during the early 1970s as use of the network caused flaws to surface. The final one of their 17 causes appeared only in 1988, nearly 20 years after the inception of the network. This disruption was caused by an overflow.

The original ARPANET network comprised hosts that connected to specialized communications processors called IMPs. Each interface message processor (IMP) controlled an individual subnetwork, much like today's routers; the IMPs connected to other IMPs through dedicated communications lines. For reliability, each IMP had at least two distinct paths to each other IMP. The IMP connections were added to a table dynamically as communication between two IMPs was required by network traffic.

In 1988, one subnetwork added a connection to a 348th IMP. The table for IMP connections had been hard-coded in 1969 to only 347 entries, which seemed wildly excessive at the time, and in the intervening years

(*continues*)

SIDEBAR 3-3 *Continued*

people had forgotten that table size if, indeed, it had ever been publicized. (In 1967, 347 IMPs was far more than the designers ever envisioned the network would have.) Software handling the IMP's table detected this overflow but handled it by causing the IMP to reboot; upon rebooting, the IMP's table was cleared and would be repopulated as it discovered other reachable subnetworks. Apparently the authors of that software assumed such a table overflow would be a sporadic mistake from another cause, so clearing and rebooting would rid the table of the faulty data. Because the fault was due to a real situation, in 1989 the refreshed IMP ran for a while until its table refilled and then it failed and rebooted again.

It took some time to determine the source and remedy of this flaw, because twenty years had passed between coding and failing; everybody associated with the original design or implementation had moved on to other projects.

As this example shows, buffer overflows—like other program faults—can remain unexploited and undetected for some time, but they are still present.

---

the operating system. However, the system will try to execute a string that accidentally does represent a valid instruction, with effects depending on the actual value. Again, depending on the nature of the error, this faulty instruction may have no effect (if it is not in the path of execution or in a section that has already been executed), a null effect (if it happens not to affect code or data, such as an instruction to move the contents of register 1 to itself), or an unnoticed or readily noticed effect.

Destroying your own code or data is unpleasant, but at least you can say the harm was your own fault. Unless, of course, it wasn't your fault. One early flaw in Microsoft's Outlook involved the simple date field: A date is a few bytes long to represent a day, month, year, and time in GMT (Greenwich Mean Time) format. In a former version of Outlook, a message with a date of more than 1000 bytes exceeded the buffer space for message headers and ran into reserved space. Simply downloading such a message from a mail server would cause your system to crash, and each time you tried to restart, Outlook would try to reload the same message and crash again. In this case, you suffered harm from a buffer overflow involving only your memory area.

One program can accidentally modify code or data of another procedure that will not be executed until much later, so the delayed impact can be almost as difficult to diagnose as if the attack came from an unrelated, independent user. The most significant impact of a buffer overflow occurs when the excess data affect the operating system's code or data.

Modification of code and data for one user or another is significant, but it is not a major computer security issue. However, as we show in the next section, buffer overflows perpetrated on the operating system can have serious consequences.

Affecting the Operating System or a Critical Application

The same basic scenarios occur for operating system code or data as for users, although again there are important variations. Exploring these differences also leads us to consider motive, and so we shift from thinking of what are essentially accidents to intentional malicious acts by an attacker.

Because the mix of programs changes continually on a computing system, there is little opportunity to affect any one particular use. We now consider the case in which an attacker who has already overtaken an ordinary user now wants to overtake the operating system. Such an attack can let the attacker plant permanent code that is reactivated each time a machine is restarted, for example. Or the attack may expose data, for example, passwords or cryptographic keys that the operating system is entrusted to safeguard. So now let us consider the impact a (compromised) user can have on the operating system.

Users' code and data are placed essentially at random: wherever there is free memory of an appropriate size. Only by tracing through system memory allocation tables can you learn where your program and data appear in memory. However, certain portions of the operating system are placed at particular fixed locations, and other data are located at places that can easily be determined during execution. Fixed or easily determined location distinguishes operating system routines, especially the most critical ones, from a user's code and data.

A second distinction between ordinary users and the operating system is that a user runs without operating system privileges. The operating system invokes a user's program as if it were a subprocedure, and the operating system receives control back when the user's program exits. If the user can alter what the operating system does when it regains control, the user can force the operating system to execute code the user wants to run, but with elevated privileges (those of the operating system). Being able to modify operating system code or data allows the user (that is, an attacker acting as the user) to obtain effective privileged status.

Privilege escalation, executing attack code with higher system permissions, is a bonus for the attacker.

The call and return sequence operates under a well-defined protocol using a data structure called the stack. Aleph One (Elias Levy) describes how to use buffer overflows to overwrite the call stack [ALE96]. In the next section we show how a programmer can use an overflow to compromise a computer's operation.

The Stack and the Heap

The **stack** is a key data structure necessary for interchange of data between procedures, as we described earlier in this chapter. Executable code resides at one end of memory, which we depict as the low end; above it are constants and data items whose size is known at compile time; above that is the heap for data items whose size can change during execution; and finally, the stack. Actually, as shown earlier in Figure 3-1, the heap and stack are at opposite ends of the memory left over after code and local data.

When procedure A calls procedure B, A pushes onto the stack its return address (that is, the current value of the program counter), the address at which execution should

resume when B exits, as well as calling parameter values. Such a sequence is shown in Figure 3-6.

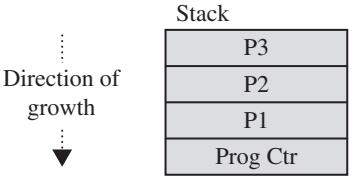


FIGURE 3-6 Parameters and Return Address

To help unwind stack data tangled because of a program that fails during execution, the stack also contains the pointer to the logical bottom of this program's section of the stack, that is, to the point just before where this procedure pushed values onto the stack. This data group of parameters, return address, and stack pointer is called a **stack frame**, as shown in Figure 3-7.

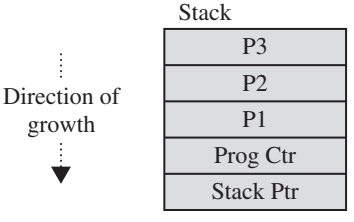


FIGURE 3-7 A Stack Frame

When one procedure calls another, the stack frame is pushed onto the stack to allow the two procedures to exchange data and transfer control; an example of the stack after procedure A calls B is shown in Figure 3-8.

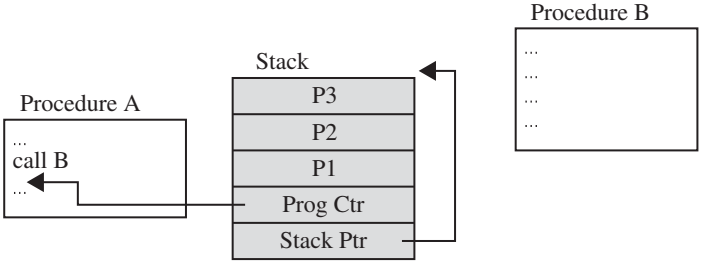


FIGURE 3-8 The Stack after a Procedure Call

Now let us consider a slightly deeper example: Suppose procedure A calls B that in turn calls C. After these two calls the stack will look as shown in Figure 3-7, with the return address to A on the bottom, then parameters from A to B, the return address from

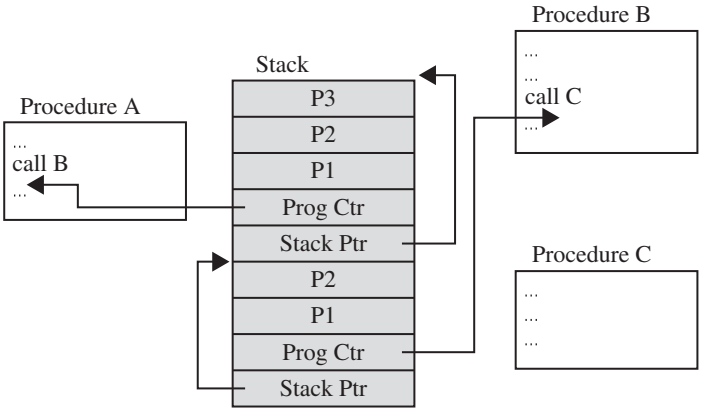


FIGURE 3-9 The Stack after Nested Procedure Calls

C to B, and parameters from B to C, in that order. After procedure C returns to B, the second stack frame is popped off the stack and it looks again like Figure 3-9.

The important thing to notice in these figures is the program counter: If the attacker can overwrite the program counter, doing so will redirect program execution after the procedure returns, and that redirection is, in fact, a frequently seen step in exploiting a buffer overflow.

Overflow into system space can redirect execution immediately or on exit from the current called procedure.

Refer again to Figure 3-1 and notice that the stack is at the top of memory, growing downward, and something else, called the heap, is at the bottom growing up. As you have just seen, the stack is mainly used for nested calls to procedures. The **heap** provides space for dynamic data, that is, data items whose size is not known when a program is compiled.

If you declare an array of ten elements in the source code of a routine, the compiler allocates enough space for those ten elements, as well as space for constants and individual variables. But suppose you are writing a general-purpose sort routine that works on any data, for example, tables with arbitrarily many rows and columns of any kind of data. You might process an array of 100 integers, a table of 20,000 telephone numbers, or a structure of 2,000 bibliographic references with names, titles, and sources. Even if the table itself is passed as a parameter so that you do not need space to store it within your program, you will need some temporary space, for example, for variables to hold the values of two rows as you compare them and perhaps exchange their positions. Because you cannot know when you write your code how large a row will be, modern programming languages let you defer declaring the size of these variables until the program executes. During execution, code inserted by the compiler into your program determines the sizes and asks the operating system to allocate dynamic memory, which the operating system gets from the heap. The heap grows and shrinks as memory is allocated and freed for dynamic data structures.

As you can see in Figure 3-1, the stack and heap grow toward each other, and you can predict that at some point they might collide. Ordinarily, the operating system monitors their sizes and prevents such a collision, except that the operating system cannot know that you will write 15,000 bytes into a dynamic heap space for which you requested only 15 bytes, or 8 bytes into a 4-byte parameter, or four return parameter values into three parameter spaces.

The attacker wants to overwrite stack memory, sometimes called **stack smashing**, in a purposeful manner: Arbitrary data in the wrong place causes strange behavior, but particular data in a predictable location causes a planned impact. Here are some ways the attacker can produce effects from an overflow attack:

- *Overwrite the program counter* stored in the stack so that when this routine exits, control transfers to the address pointed at by the modified program counter address.
- *Overwrite part of the code* in low memory, substituting the attacker's instructions for previous program statements.
- *Overwrite the program counter and data* in the stack so that the program counter now points into the stack, causing the data overwritten into the stack to be executed.

The common feature of these attack methods is that the attacker uses overflow data as code the victim will execute. Because this code runs under the authority of the victim, it carries the victim's privileges, and it can destroy the victim's data by overwriting it or can perform any actions the victim could, for example, sending email as if from the victim. If the overflow occurs during a system call, that is, when the system is running with elevated privileges, the attacker's code also executes with those privileges; thus, an attack that transfers control to the attacker by invoking one of the attacker's routines activates the attacker's code and leaves the attacker in control with privileges. And for many attackers the goal is not simply to destroy data by overwriting memory but also to gain control of the system as a first step in a more complex and empowering attack.

Buffer overflow attacks are interesting because they are the first example of a class of problems called data driven attacks. In a **data driven attack** the harm occurs by the data the attacker sends. Think of such an attack this way: A buffer overflows when someone stuffs too much into it. Most people accidentally put one more element in an array or append an additional character into a string. The data inserted relate to the application being computed. However, with a malicious buffer overflow the attacker, like David the nonmalicious researcher, carefully chooses data that will cause specific action, to make the program fail in a planned way. In this way, the selected data drive the impact of the attack.

Data driven attacks are directed by specially chosen data the attacker feeds a program as input.

Malicious exploitation of buffer overflows also exhibit one more important characteristic: They are examples of a multistep approach. Not only does the attacker overrun allocated space, but the attacker also uses the overrun to execute instructions to achieve the next step in the attack. The overflow is not a goal but a stepping stone to a larger purpose.

Buffer overflows can occur with many kinds of data, ranging from arrays to parameters to individual data items, and although some of them are easy to prevent (such as checking an array's dimension before storing), others are not so easy. Human mistakes will never be eliminated, which means overflow conditions are likely to remain. In the next section we present a selection of controls that can detect and block various kinds of overflow faults.

Overflow Countermeasures

It would seem as if the countermeasure for a buffer overflow is simple: Check before you write. Unfortunately, that is not quite so easy because some buffer overflow situations are not directly under the programmer's control, and an overflow can occur in several ways.

Although buffer overflows are easy to program, no single countermeasure will prevent them. However, because of the prevalence and seriousness of overflows, several kinds of protection have evolved.

The most obvious countermeasure to overwriting memory is to stay within bounds. Maintaining boundaries is a shared responsibility of the programmer, operating system, compiler, and hardware. All should do the following:

- Check lengths before writing.
- Confirm that array subscripts are within limits.
- Double-check boundary condition code to catch possible off-by-one errors.
- Monitor input and accept only as many characters as can be handled.
- Use string utilities that transfer only a bounded amount of data.
- Check procedures that might overrun their space.
- Limit programs' privileges, so if a piece of code is overtaken maliciously, the violator does not acquire elevated system privileges as part of the compromise.

Programming Controls

Later in this chapter we study programming controls in general. You may already have encountered these principles of software engineering in other places. Techniques such as code reviews (in which people other than the programmer inspect code for implementation oversights) and independent testing (in which dedicated testers hypothesize points at which a program could fail) can catch overflow situations before they become problems.

Language Features

Two features you may have noticed about attacks involving buffer overflows are that the attacker can write directly to particular memory addresses and that the language or compiler allows inappropriate operations on certain data types.

Anthony (C.A.R.) Hoare [HOA81] comments on the relationship between language and design:

> Programmers are always surrounded by complexity; we cannot avoid it. Our applications are complex because we are ambitious to use our computers in ever more sophisticated ways. Programming is complex because of the large number of conflicting

> objectives for each of our programming projects. If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.

Some programming languages have features that preclude overflows. For example, languages such as Java, .NET, Perl, and Python generate code to check bounds before storing data. The unchecked languages C, C++, and assembler language allow largely unlimited program access. To counter the openness of these languages, compiler writers have developed extensions and libraries that generate code to keep programs in check.

Code Analyzers

Software developers hope for a simple tool to find security errors in programs. Such a tool, called a **static code analyzer**, analyzes source code to detect unsafe conditions. Although such tools are not, and can never be, perfect, several good ones exist. Kendra Kratkiewicz and Richard Lippmann [KRA05] and the US-CERT Build Security In website at https://buildsecurityin.us-cert.gov/ contain lists of static code analyzers.

Separation

Another direction for protecting against buffer overflows is to enforce containment: separating sensitive areas from the running code and its buffers and data space. To a certain degree, hardware can separate code from data areas and the operating system.

Stumbling Blocks

Because overwriting the stack is such a common and powerful point of attack, protecting it becomes a priority.

Refer again to Figure 3-8, and notice that each procedure call adds a new stack frame that becomes a distinct slice of the stack. If our goal is to protect the stack, we can do that by wrapping each stack frame in a protective layer. Such a layer is sometimes called a **canary**, in reference to canary birds that were formerly taken into underground mines; the canary was more sensitive to limited oxygen, so the miners could notice the canary reacting before they were affected, giving the miners time to leave safely.

In this section we show how some manufacturers have developed cushions to guard against benign or malicious damage to the stack.

In a common buffer overflow stack modification, the program counter is reset to point into the stack to the attack code that has overwritten stack data. In Figure 3-10, the two parameters P1 and P2 have been overwritten with code to which the program counter has been redirected. (Two instructions is too short a set for many stack overflow attacks, so a real buffer overflow attack would involve more data in the stack, but the concept is easier to see with a small stack.)

StackGuard is an approach proposed by Crispin Cowan et al. [COW98] The attacker usually cannot tell exactly where the saved program counter is in the stack, only that there is one at an approximate address. Thus, the attacker has to rewrite not just the stack pointer but also some words around it to be sure of changing the true stack pointer, but this uncertainty to the attacker allows StackGuard to detect likely changes to the program counter. Each procedure includes a prolog code to push values on the stack, set the remainder of the stack frame, and pass control to the called return; then on return,

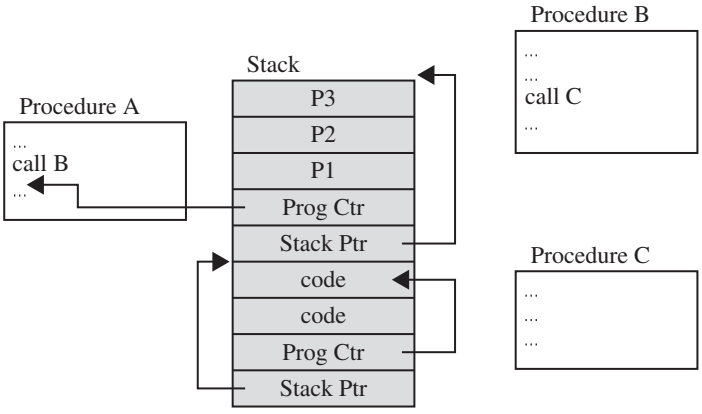


FIGURE 3-10 Compromised Stack

some termination code cleans up the stack, reloads registers, and returns. Just below the program counter, StackGuard inserts a canary value to signal modification; if the attacker rewrites the program counter and the added value, StackGuard augments the termination code to detect the modified added value and signal an error before returning. Thus, each canary value serves as a protective insert to protect the program counter. These protective inserts are shown in Figure 3-11. The idea of surrounding the return address with a tamper-detecting value is sound, as long as only the defender can generate and verify that value.

Alas, the attack–countermeasure tennis match was played here, as we have seen in other situations such as password guessing: The attacker serves, the defender responds

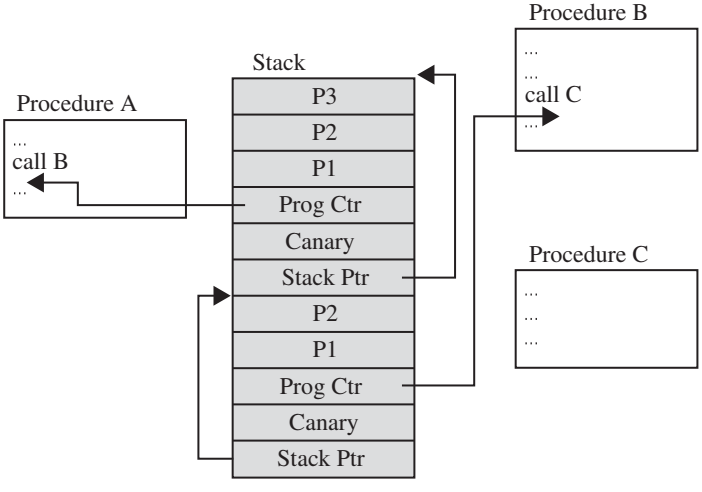


FIGURE 3-11 Canary Values to Signal Modification

with a countermeasure, the attacker returns the ball with an enhanced attack, and so on. The protective canary value has to be something to which the termination code can detect a change, for example, the recognizable pattern 0x0f1e2d3c, which is a number the attacker is unlikely to write naturally (although not impossible). As soon as the attacker discovers that a commercial product looks for a pad of exactly that value, we know what value the attacker is likely to write near the return address. Countering again, to add variety the defender picks random patterns that follow some sequence, such as 0x0f1e2d3c, 0x0f1e2d3d, and so on. In response, the attacker monitors the stack over time to try to predict the sequence pattern. The two sides continue to volley modifications until, as in tennis, one side fails.

Next we consider a programming flaw that is similar to an overflow: a failure to check and control access completely and consistently.

Incomplete Mediation

Mediation means checking: the process of intervening to confirm an actor's authorization before it takes an intended action. In the last chapter we discussed the steps and actors in the authentication process: the access control triple that describes what subject can perform what operation on what object. Verifying that the subject is authorized to perform the operation on an object is called **mediation**. Incomplete mediation is a security problem that has been with us for decades: Forgetting to ask "Who goes there?" before allowing the knight across the castle drawbridge is just asking for trouble. In the same way, attackers exploit incomplete mediation to cause security problems.

Definition

Consider the following URL. In addition to a web address, it contains two parameters, so you can think of it as input to a program:

```
http://www.somesite.com/subpage/userinput.asp?
parm1=(808)555-1212&parm2=2015Jan17
```

As a security professional trying to find and fix problems before they occur, you might examine the various parts of the URL to determine what they mean and how they might be exploited. For instance, the parameters parm1 and parm2 look like a telephone number and a date, respectively. Probably the client's (user's) web browser enters those two values in their specified format for easy processing on the server's side.

But what would happen if parm2 were submitted as 1800Jan01? Or 1800Feb30? Or 2048Min32? Or 1Aardvark2Many? Something in the program or the system with which it communicates would likely fail. As with other kinds of programming errors, one possibility is that the system would fail catastrophically, with a routine's failing on a data type error as it tried to handle a month named "Min" or even a year (like 1800) that was out of expected range. Another possibility is that the receiving program would continue to execute but would generate a very wrong result. (For example, imagine the amount of interest due today on a billing error with a start date of 1 Jan 1800.) Then again, the processing server might have a default condition, deciding to treat 1Aardvark2Many as 21 July 1951. The possibilities are endless.

A programmer typically dismisses considering bad input, asking why anyone would enter such numbers. Everybody knows there is no 30th of February and, for certain applications, a date in the 1800s is ridiculous. True. But ridiculousness does not alter human behavior. A person can type 1800 if fingers slip or the typist is momentarily distracted, or the number might have been corrupted during transmission. Worse, just because something is senseless, stupid, or wrong doesn't prevent people from doing it. And if a malicious person does it accidentally and finds a security weakness, other people may well hear of it. Security scoundrels maintain a robust exchange of findings. Thus, programmers should not assume data will be proper; instead, programs should validate that all data values are reasonable before using them.

Users make errors from ignorance, misunderstanding, distraction; user errors should not cause program failures.

Validate All Input

One way to address potential problems is to try to anticipate them. For instance, the programmer in the examples above may have written code to check for correctness on the *client*'s side (that is, the user's browser). The client program can search for and screen out errors. Or, to prevent the use of nonsense data, the program can restrict choices to valid ones only. For example, the program supplying the parameters might have solicited them by using a drop-down box or choice list from which only the twelve conventional months could have been selected. Similarly, the year could have been tested to ensure a reasonable value (for example, between 2000 and 2050, according to the application) and date numbers would have to be appropriate for the months in which they occur (no 30th of February, for example). Using such verification, the programmer may have felt well insulated from the possible problems a careless or malicious user could cause.

Guard Against Users' Fingers

However, the application is still vulnerable. By packing the result into the return URL, the programmer left these data fields in a place where the user can access (and modify) them. In particular, the user can edit the URL line, change any parameter values, and send the revised line. On the server side, the server has no way to tell if the response line came from the client's browser or as a result of the user's editing the URL directly. We say in this case that the data values are not completely mediated: The sensitive data (namely, the parameter values) are in an exposed, uncontrolled condition.

Unchecked data values represent a serious potential vulnerability. To demonstrate this flaw's security implications, we use a real example; only the name of the vendor has been changed to protect the guilty. Things, Inc., was a very large, international vendor of consumer products, called Objects. The company was ready to sell its Objects through a web site, using what appeared to be a standard e-commerce application. The management at Things decided to let some of its in-house developers produce a web site with which its customers could order Objects directly from the web.

To accompany the web site, Things developed a complete price list of its Objects, including pictures, descriptions, and drop-down menus for size, shape, color, scent, and

any other properties. For example, a customer on the web could choose to buy 20 of part number 555A Objects. If the price of one such part were $10, the web server would correctly compute the price of the 20 parts to be $200. Then the customer could decide whether to have the Objects shipped by boat, by ground transportation, or sent electronically. If the customer were to choose boat delivery, the customer's web browser would complete a form with parameters like these:

```
http://www.things.com/order.asp?custID=101&part=555A
&qy=20&price=10&ship=boat&shipcost=5&total=205
```

So far, so good; everything in the parameter passing looks correct. But this procedure leaves the parameter statement open for malicious tampering. Things should not need to pass the price of the items back to itself as an input parameter. Things presumably knows how much its Objects cost, and they are unlikely to change dramatically since the time the price was quoted a few screens earlier.

There is no reason to leave sensitive data under control of an untrusted user.

A malicious attacker may decide to exploit this peculiarity by supplying instead the following URL, where the price has been reduced from $205 to $25:

```
http://www.things.com/order.asp?custID=101&part=555A
&qy=20&price=1&ship=boat&shipcost=5&total=25
```

Surprise! It worked. The attacker could have ordered Objects from Things in any quantity at any price. And yes, this code was running on the web site for a while before the problem was detected.

From a security perspective, the most serious concern about this flaw was the length of time that it could have run undetected. Had the whole world suddenly made a rush to Things' web site and bought Objects at a fraction of their actual price, Things probably would have noticed. But Things is large enough that it would never have detected a few customers a day choosing prices that were similar to (but smaller than) the real price, say, 30 percent off. The e-commerce division would have shown a slightly smaller profit than other divisions, but the difference probably would not have been enough to raise anyone's eyebrows; the vulnerability could have gone unnoticed for years. Fortunately, Things hired a consultant to do a routine review of its code, and the consultant quickly found the error.

The vulnerability in this situation is that the customer (computer user) has unmediated access to sensitive data. An application running on the user's browser maintained the order details but allowed the user to change those details at will. In fact, few of these values should have been exposed in the URL sent from the client's browser to the server. The client's application should have specified part number and quantity, but an application on the server's side should have returned the price per unit and total price.

If data can be changed, assume they have been.

This web program design flaw is easy to imagine in other settings. Those of us interested in security must ask ourselves, How many similar problems are in running code today? And how will those vulnerabilities ever be found? And if found, by whom?

Complete Mediation

Because the problem here is incomplete mediation, the solution is complete mediation. Remember from Chapter 2 that one of our standard security tools is access control, sometimes implemented according to the reference monitor concept. The three properties of a reference monitor are (1) small and simple enough to give confidence of correctness, (2) unbypassable, and (3) always invoked. These three properties combine to give us solid, complete mediation.

Time-of-Check to Time-of-Use

The third programming flaw we describe also involves synchronization. To improve efficiency, modern processors and operating systems usually change the order in which instructions and procedures are executed. In particular, instructions that appear to be adjacent may not actually be executed immediately after each other, either because of intentionally changed order or because of the effects of other processes in concurrent execution.

Definition

Access control is a fundamental part of computer security; we want to make sure that only those subjects who should access an object are allowed that access. Every requested access must be governed by an access policy stating who is allowed access to what; then the request must be mediated by an access-policy-enforcement agent. But an incomplete mediation problem occurs when access is not checked universally. The **time-of-check to time-of-use** (**TOCTTOU**) flaw concerns mediation that is performed with a "bait and switch" in the middle.

Between access check and use, data must be protected against change.

To understand the nature of this flaw, consider a person's buying a sculpture that costs $100. The buyer takes out five $20 bills, carefully counts them in front of the seller, and lays them on the table. Then the seller turns around to write a receipt. While the seller's back is turned, the buyer takes back one $20 bill. When the seller turns around, the buyer hands over the stack of bills, takes the receipt, and leaves with the sculpture. Between the time the security was checked (counting the bills) and the access occurred (exchanging the sculpture for the bills), a condition changed: What was checked is no longer valid when the object (that is, the sculpture) is accessed.

A similar situation can occur with computing systems. Suppose a request to access a file were presented as a data structure, with the name of the file and the mode of access presented in the structure. An example of such a structure is shown in Figure 3-12.

File: my_file	Action: Change byte 4 to A

FIGURE 3-12 File Access Data Structure

The data structure is essentially a work ticket, requiring a stamp of authorization; once authorized, it is put on a queue of things to be done. Normally the access control mediator process receives the data structure, determines whether the access should be allowed, and either rejects the access and stops processing or allows the access and forwards the data structure to the file handler for processing.

To carry out this authorization sequence, the access control mediator would have to look up the file name (and the user identity and any other relevant parameters) in tables. The mediator could compare the names in the table to the file name in the data structure to determine whether access is appropriate. More likely, the mediator would copy the file name into its own local storage area and compare from there. Comparing from the copy leaves the data structure in the user's area, under the user's control.

At this point the incomplete mediation flaw can be exploited. While the mediator is checking access rights for the file my_file, the user could change the file name descriptor to your_file, the value shown in Figure 3-13. Having read the work ticket once, the mediator would not be expected to reread the ticket before approving it; the mediator would approve the access and send the now-modified descriptor to the file handler.

The problem is called a time-of-check to time-of-use flaw because it exploits the delay between the two actions: check and use. That is, between the time the access was checked and the time the result of the check was used, a change occurred, invalidating the result of the check.

Security Implication

The security implication here is clear: Checking one action and performing another is an example of ineffective access control, leading to confidentiality failure or integrity failure or both. We must be wary whenever a time lag or loss of control occurs, making sure that there is no way to corrupt the check's results during that interval.

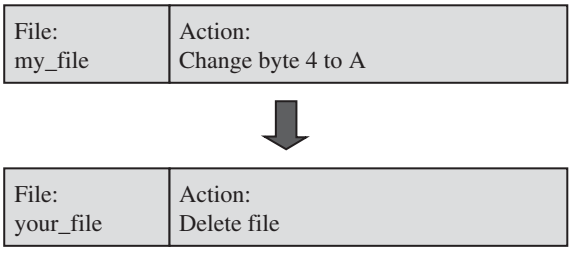


FIGURE 3-13 Unchecked Change to Work Descriptor

Countermeasures

Fortunately, there are ways to prevent exploitation of the time lag, again depending on our security tool, access control. Critical parameters are not exposed during any loss of control. The access-checking software must own the request data until the requested action is complete. Another protection technique is to ensure serial integrity, that is, to allow no interruption (loss of control) during the validation. Or the validation routine can initially copy data from the user's space to the routine's area—out of the user's reach—and perform validation checks on the copy. Finally, the validation routine can seal the request data to detect modification. Really, all these protection methods are expansions on the tamperproof criterion for a reference monitor: Data on which the access control decision is based and the result of the decision must be outside the domain of the program whose access is being controlled.

Undocumented Access Point

Next we describe a common programming situation. During program development and testing, the programmer needs a way to access the internals of a module. Perhaps a result is not being computed correctly so the programmer wants a way to interrogate data values during execution. Maybe flow of control is not proceeding as it should and the programmer needs to feed test values into a routine. It could be that the programmer wants a special debug mode to test conditions. For whatever reason the programmer creates an undocumented entry point or execution mode.

These situations are understandable during program development. Sometimes, however, the programmer forgets to remove these entry points when the program moves from development to product. Or the programmer decides to leave them in to facilitate program maintenance later; the programmer may believe that nobody will find the special entry. Programmers can be naïve, because if there is a hole, someone is likely to find it. See Sidebar 3-4 for a description of an especially intricate backdoor.

SIDEBAR 3-4 Oh Look: The Easter Bunny!

Microsoft's Excel spreadsheet program, in an old version, Excel 97, had the following feature.

- Open a new worksheet
- Press F5
- Type X97:L97 and press Enter
- Press Tab
- Hold <Ctrl-Shift> and click the Chart Wizard

A user who did that suddenly found that the spreadsheet disappeared and the screen filled with the image of an airplane cockpit! Using the arrow keys, the user could fly a simulated plane through space. With a few more

(*continues*)

SIDEBAR 3-4 *Continued*

keystrokes the user's screen seemed to follow down a corridor with panels on the sides, and on the panels were inscribed the names of the developers of that version of Excel.

Such a piece of code is called an **Easter egg**, for chocolate candy eggs filled with toys for children. This is not the only product with an Easter egg. An old version of Internet Explorer had something similar, and other examples can be found with an Internet search. Although most Easter eggs do not appear to be harmful, they raise a serious question: If such complex functionality can be embedded in commercial software products without being stopped by a company's quality control group, are there other holes, potentially with security vulnerabilities?

Backdoor

An undocumented access point is called a **backdoor** or **trapdoor**. Such an entry can transfer control to any point with any privileges the programmer wanted.

Few things remain secret on the web for long; someone finds an opening and exploits it. Thus, coding a supposedly secret entry point is an opening for unannounced visitors.

Secret backdoors are eventually found. Security cannot depend on such secrecy.

Another example of backdoors is used once an outsider has compromised a machine. In many cases an intruder who obtains access to a machine wants to return later, either to extend the raid on the one machine or to use the machine as a jumping-off point for strikes against other machines to which the first machine has access. Sometimes the first machine has privileged access to other machines so the intruder can get enhanced rights when exploring capabilities on these new machines. To facilitate return, the attacker can create a new account on the compromised machine, under a user name and password that only the attacker knows.

Protecting Against Unauthorized Entry

Undocumented entry points are a poor programming practice (but they will still be used). They should be found during rigorous code reviews in a software development process. Unfortunately, two factors work against that ideal.

First, being undocumented, these entry points will not be clearly labeled in source code or any of the development documentation. Thus, code reviewers might fail to recognize them during review.

Second, such backdoors are often added after ordinary code development, during testing or even maintenance, so even the scrutiny of skilled reviewers will not find them. Maintenance people who add such code are seldom security engineers, so they are not used to thinking of vulnerabilities and failure modes. For example, as reported

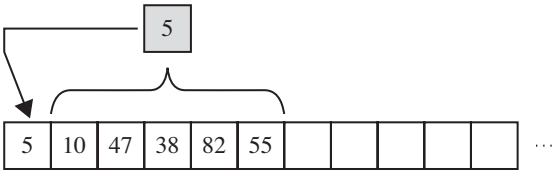
by security writer Brian Krebs in his blog *Krebs on Security*, 24 January 2013, security researcher Stefan Viehböck of SEC Consult Vulnerability Labs in Vienna, Austria found that some products from Barracuda Networks (maker of firewalls and other network devices) accepted remote (network) logins from user name "product" and no password. The engineer who inserted the backdoor probably thought the activity was protected by restricting the address range from which the logins would be accepted: Only logins from the range of addresses assigned to Barracuda would succeed. However, the engineer failed to consider (and a good security engineer would have caught) that the specified range also included hundreds of other companies.

Thus, preventing or locking these vulnerable doorways is difficult, especially because the people who write them may not appreciate their security implications.

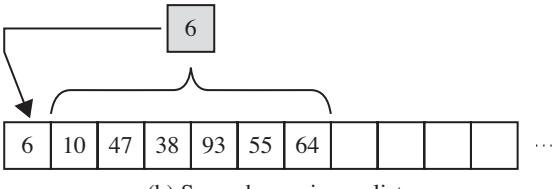
Off-by-One Error

When learning to program, neophytes can easily fail with the **off-by-one error**: miscalculating the condition to end a loop (repeat while i<=n or i<n? repeat until i=n or i>n?) or overlooking that an array of A[0] through A[n] contains n+1 elements.

Usually the programmer is at fault for failing to think correctly about when a loop should stop. Other times the problem is merging actual data with control data (sometimes called metadata or data about the data). For example, a program may manage a list that increases and decreases. Think of a list of unresolved problems in a customer service department: Today there are five open issues, numbered 10, 47, 38, 82, and 55; during the day, issue 82 is resolved but issues 93 and 64 are added to the list. A programmer may create a simple data structure, an array, to hold these issue numbers and may reasonably specify no more than 100 numbers. But to help with managing the numbers, the programmer may also reserve the first position in the array for the count of open issues. Thus, in the first case the array really holds six elements, 5 (the count), 10, 47, 38, 82, and 55; and in the second case there are seven, 6, 10, 47, 38, 93, 55, 64, as shown in Figure 3-14. A 100-element array will clearly not hold 100 data items plus one count.



(a) First open issues list



(b) Second open issues list

FIGURE 3-14 Both Data and Number of Used Cells in an Array

In this simple example, the program may run correctly for a long time, as long as no more than 99 issues are open at any time, but adding the 100th issue will cause the program to fail. A similar problem occurs when a procedure edits or reformats input, perhaps changing a one-character sequence into two or more characters (as for example, when the one-character ellipsis symbol "…" available in some fonts is converted by a word processor into three successive periods to account for more limited fonts.) These unanticipated changes in size can cause changed data to no longer fit in the space where it was originally stored. Worse, the error will appear to be sporadic, occurring only when the amount of data exceeds the size of the allocated space.

Alas, the only control against these errors is correct programming: always checking to ensure that a container is large enough for the amount of data it is to contain.

Integer Overflow

An integer overflow is a peculiar type of overflow, in that its outcome is somewhat different from that of the other types of overflows. An **integer overflow** occurs because a storage location is of fixed, finite size and therefore can contain only integers up to a certain limit. The overflow depends on whether the data values are signed (that is, whether one bit is reserved for indicating whether the number is positive or negative). Table 3-1 gives the range of signed and unsigned values for several memory location (word) sizes.

When a computation causes a value to exceed one of the limits in Table 3-1, the extra data does not spill over to affect adjacent data items. That's because the arithmetic is performed in a hardware register of the processor, not in memory. Instead, either a hardware program exception or fault condition is signaled, which causes transfer to an error handling routine, or the excess digits on the most significant end of the data item are lost. Thus, with 8-bit unsigned integers, $255 + 1 = 0$. If a program uses an 8-bit unsigned integer for a loop counter and the stopping condition for the loop is count = 256, then the condition will never be true.

Checking for this type of overflow is difficult, because only when a result overflows can the program determine an overflow occurs. Using 8-bit unsigned values, for example, a program could determine that the first operand was 147 and then check whether the second was greater than 108. Such a test requires double work: First determine the maximum second operand that will be in range and then compute the sum. Some compilers generate code to test for an integer overflow and raise an exception.

TABLE 3-1 Value Range by Word Size

Word Size	Signed Values	Unsigned Values
8 bits	−128 to +127	0 to 255 ($2^8 - 1$)
16 bits	−32,768 to +32,767	0 to 65,535 ($2^{16} - 1$)
32 bits	−2,147,483,648 to +2,147,483,647	0 to 4,294,967,296 ($2^{32} - 1$)

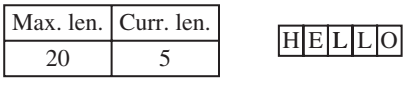
Unterminated Null-Terminated String

Long strings are the source of many buffer overflows. Sometimes an attacker intentionally feeds an overly long string into a processing program to see if and how the program will fail, as was true with the Dialer program. Other times the vulnerability has an accidental cause: A program mistakenly overwrites part of a string, causing the string to be interpreted as longer than it really is. How these errors actually occur depends on how the strings are stored, which is a function of the programming language, application program, and operating system involved.

Variable-length character (text) strings are delimited in three ways, as shown in Figure 3-15. The easiest way, used by Basic and Java, is to allocate space for the declared maximum string length and store the current length in a table separate from the string's data, as shown in Figure 3-15(a).

Some systems and languages, particularly Pascal, precede a string with an integer that tells the string's length, as shown in Figure 3-15(b). In this representation, the string "Hello" would be represented as 0x0548656c6c6f because 0x48, 0x65, 0x6c, and 0x6f are the internal representation of the characters "H," "e," "l," and "o," respectively. The length of the string is the first byte, 0x05. With this representation, string buffer overflows are uncommon because the processing program receives the length first and can verify that adequate space exists for the string. (This representation is vulnerable to the problem we described earlier of failing to include the length element when planning space for a string.) Even if the length field is accidentally overwritten, the application reading the string will read only as many characters as written into the length field. But the limit for a string's length thus becomes the maximum number that will fit in the length field, which can reach 255 for a 1-byte length and 65,535 for a 2-byte length.

The last mode of representing a string, typically used in C, is called **null terminated**, meaning that the end of the string is denoted by a null byte, or 0x00, as shown in Figure 3-15(c). In this form the string "Hello" would be 0x48656c6c6f00. Representing strings this way can lead to buffer overflows because the processing program determines the end of the string, and hence its length, only after having received the entire



(a) Separate length

(b) Length precedes string

(c) String ends with null

FIGURE 3-15 Variable-Length String Representations

string. This format is prone to misinterpretation. Suppose an erroneous process happens to overwrite the end of the string and its terminating null character; in that case, the application reading the string will continue reading memory until a null byte happens to appear (from some other data value), at any distance beyond the end of the string. Thus, the application can read 1, 100 to 100,000 extra bytes or more until it encounters a null.

The problem of buffer overflow arises in computation, as well. Functions to move and copy a string may cause overflows in the stack or heap as parameters are passed to these functions.

Parameter Length, Type, and Number

Another source of data-length errors is procedure parameters, from web or conventional applications. Among the sources of problems are these:

- *Too many parameters*. Even though an application receives only three incoming parameters, for example, that application can incorrectly write four outgoing result parameters by using stray data adjacent to the legitimate parameters passed in the calling stack frame. (The opposite problem, more inputs than the application expects, is less of a problem because the called applications' outputs will stay within the caller's allotted space.)
- *Wrong output type or size.* A calling and called procedure need to agree on the type and size of data values exchanged. If the caller provides space for a two-byte integer but the called routine produces a four-byte result, those extra two bytes will go somewhere. Or a caller may expect a date result as a number of days after 1 January 1970 but the result produced is a string of the form "dd-mmm-yyyy."
- *Too-long string.* A procedure can receive as input a string longer than it can handle, or it can produce a too-long string on output, each of which will also cause an overflow condition.

Procedures often have or allocate temporary space in which to manipulate parameters, so temporary space has to be large enough to contain the parameter's value. If the parameter being passed is a null-terminated string, the procedure cannot know how long the string will be until it finds the trailing null, so a very long string will exhaust the buffer.

Unsafe Utility Program

Programming languages, especially C, provide a library of utility routines to assist with common activities, such as moving and copying strings. In C the function `strcpy(dest, src)` copies a string from `src` to `dest`, stopping on a null, with the potential to overrun allocated memory. A safer function is `strncpy(dest, src, max)`, which copies up to the null delimiter or `max` characters, whichever comes first.

Although there are other sources of overflow problems, from these descriptions you can readily see why so many problems with buffer overflows occur. Next, we describe

several classic and significant exploits that have had a buffer overflow as a significant contributing cause. From these examples you can see the amount of harm that a seemingly insignificant program fault can produce.

Race Condition

As the name implies, a race condition means that two processes are competing within the same time interval, and the race affects the integrity or correctness of the computing tasks. For instance, two devices may submit competing requests to the operating system for a given chunk of memory at the same time. In the two-step request process, each device first asks if the size chunk is available, and if the answer is yes, then reserves that chunk for itself. Depending on the timing of the steps, the first device could ask for the chunk, get a "yes" answer, but then not get the chunk because it has already been assigned to the second device. In cases like this, the two requesters "race" to obtain a resource. A race condition occurs most often in an operating system, but it can also occur in multithreaded or cooperating processes.

Unsynchronized Activity

In a **race condition** or **serialization flaw** two processes execute concurrently, and the outcome of the computation depends on the order in which instructions of the processes execute.

Race condition: situation in which program behavior depends on the order in which two procedures execute

Imagine an airline reservation system. Each of two agents, A and B, simultaneously tries to book a seat for a passenger on flight 45 on 10 January, for which there is exactly one seat available. If agent A completes the booking before that for B begins, A gets the seat and B is informed that no seats are available. In Figure 3-16 we show a timeline for this situation.

However, you can imagine a situation in which A asks if a seat is available, is told yes, and proceeds to complete the purchase of that seat. Meanwhile, between the time A asks and then tries to complete the purchase, agent B asks if a seat is available. The system designers knew that sometimes agents inquire about seats but never complete the booking; their clients often choose different itineraries once they explore their options. For later reference, however, the booking software gives each agent a reference number to make it easy for the server to associate a booking with a particular flight. Because A has not completed the transaction before the system gets a request from B, the system tells B that the seat is available. If the system is not designed properly, both agents can complete their transactions, and two passengers will be confirmed for that one seat (which will be uncomfortable, to say the least). We show this timeline in Figure 3-17.

A race condition is difficult to detect because it depends on the order in which two processes execute. But the execution order of the processes can depend on many other things, such as the total load on the system, the amount of available memory space, the priority of each process, or the number and timing of system interrupts to the processes. During testing, and even for a long period of execution, conditions may never cause

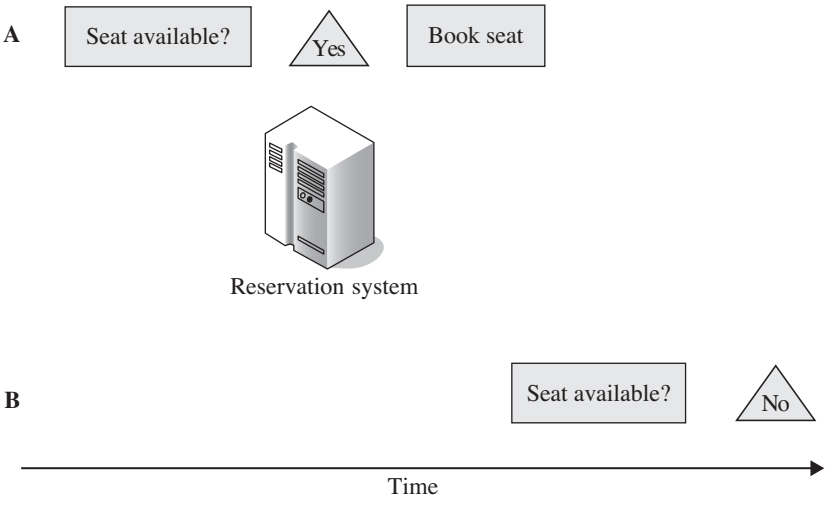


FIGURE 3-16 Seat Request and Reservation Example

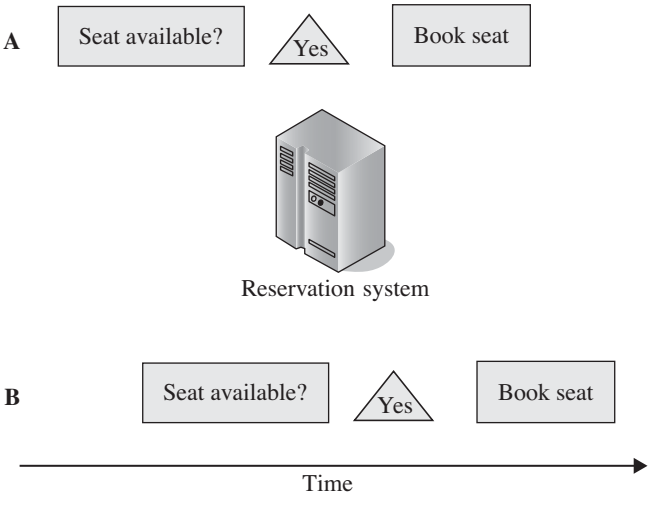


FIGURE 3-17 Overbooking Example

this particular overload condition to occur. Given these difficulties, programmers can have trouble devising test cases for all the possible conditions under which races can occur. Indeed, the problem may occur with two independent programs that happen to access certain shared resources, something the programmers of each program never envisioned.

Most of today's computers are configured with applications selected by their owners, tailored specifically for the owner's activities and needs. These applications, as well as the operating system and device drivers, are likely to be produced by different vendors

with different design strategies, development philosophies, and testing protocols. The likelihood of a race condition increases with this increasing system heterogeneity.

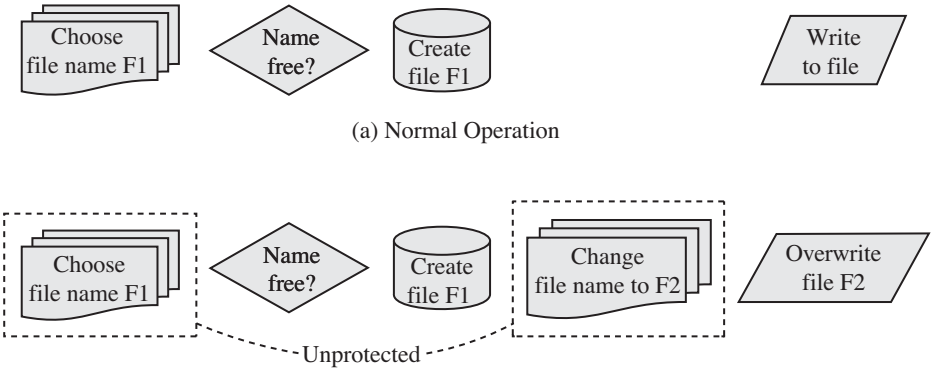
Security Implication

The security implication of race conditions is evident from the airline reservation example. A race condition between two processes can cause inconsistent, undesired and therefore wrong, outcomes—a failure of integrity.

A race condition also raised another security issue when it occurred in an old version of the Tripwire program. Tripwire is a utility for preserving the integrity of files, introduced in Chapter 2. As part of its operation it creates a temporary file to which it writes a log of its activity. In the old version, Tripwire (1) chose a name for the temporary file, (2) checked the file system to ensure that no file of that name already existed, (3) created a file by that name, and (4) later opened the file and wrote results. Wheeler [WHE04] describes how a malicious process can subvert Tripwire's steps by changing the newly created temporary file to a pointer to any other system file the process wants Tripwire to destroy by overwriting.

In this example, the security implication is clear: Any file can be compromised by a carefully timed use of the inherent race condition between steps 2 and 3, as shown in Figure 3-18. Overwriting a file may seem rather futile or self-destructive, but an attacker gains a strong benefit. Suppose, for example, the attacker wants to conceal which other processes were active when an attack occurred (so a security analyst will not know what program caused the attack). A great gift to the attacker is that of allowing an innocent but privileged utility program to obliterate the system log file of process activations. Usually that file is well protected by the system, but in this case, all the attacker has to do is point to it and let the Tripwire program do the dirty work.

Race conditions depend on the order and timing of two different processes, making these errors hard to find (and test for).



(a) Normal Operation

(b) Overwriting Filename Other Than One Checked

FIGURE 3-18 File Name Race Condition

If the malicious programmer acts too early, no temporary file has yet been created, and if the programmer acts too late, the file has been created and is already in use. But if the programmer's timing is between too early and too late, Tripwire will innocently write its temporary data over whatever file is pointed at. Although this timing may seem to be a serious constraint, the attacker has an advantage: If the attacker is too early, the attacker can try again and again until either the attack succeeds or is too late.

Thus, race conditions can be hard to detect; testers are challenged to set up exactly the necessary conditions of system load and timing. For the same reason, race condition threats are hard for the attacker to execute. Nevertheless, if race condition vulnerabilities exist, they can also be exploited.

The vulnerabilities we have presented here—incomplete mediation, race conditions, time-of-check to time-of-use, and undocumented access points—are flaws that can be exploited to cause a failure of security. Throughout this book we describe other sources of failures because programmers have many process points to exploit and opportunities to create program flaws. Most of these flaws may have been created because the programmer failed to think clearly and carefully: simple human errors. Occasionally, however, the programmer maliciously planted an intentional flaw. Or, more likely, the assailant found one of these innocent program errors and exploited it for malicious purpose. In the descriptions of program flaws we have pointed out how an attacker could capitalize on the error. In the next section we explain in more detail the harm that malicious code can cause.

3.2 MALICIOUS CODE—MALWARE

In May 2010, researcher Roger Thompson of the antivirus firm AVG detected malicious code at the web site of the U.S. Bureau of Engraving and Printing, a part of the Treasury Department [MCM10]. The site has two particularly popular sections: a description of the design of the newly redesigned U.S. $100 bill and a set of steps for identifying counterfeit currency.

The altered web site contained a hidden call to a web site in the Ukraine, which then attempted to exploit known vulnerabilities in the web site to lodge malicious code on unsuspecting users' machines. Visitors to the site would download pictures and text, as expected; what visitors couldn't see, and probably did not expect, was that they also downloaded an additional web code script that invoked code at the Ukrainian site.

The source of the exploit is unknown; some researchers think it was slipped into the site's tracking tool that tallies and displays the number of visits to a web page. Other researchers think it was introduced in a configuration flaw from the company acting as the Treasury Department's web site provider.

Two features of this attack are significant. First, U.S. government sites are seldom unwitting propagators of code attacks because administrators strongly defend the sites and make them resistant to attackers. But precisely those characteristics make users more willing to trust these sites to be free of malicious code, so users readily open their windows and download their content, which makes such sites attractive to attackers.

Second, this attack seems to have used the Eleonore attack toolkit [FIS10]. The kit is a package of attacks against known vulnerabilities, some from as long ago as

2005, combined into a ready-to-run package. A kind of "click and run" application, the $2000 kit has been around in different versions since 2009. Each kit sold is preconfigured for use against only one web site address (although customers can buy additional addresses), so the attacker who bought the kit intended to dispatch the attack specifically through the Treasury web site, perhaps because of its high credibility with users.

As malicious code attacks go, this one was not the most sophisticated, complicated, or devastating, but it illustrates several important features we explore as we analyze malicious code, the topic of this chapter. We also describe some other malicious code attacks that have had a far more serious impact.

Malicious code comes in many forms under many names. In this chapter we explore three of the most popular forms: viruses, Trojan horses, and worms. The distinctions among them are small, and we do not need to classify any piece of code precisely. More important is to learn about the nature of attacks from these three: how they can spread, what harm they can cause, and how they can be controlled. We can then apply this knowledge to other types of malicious code, including code forms that do not yet have popular names.

Malware—Viruses, Trojan Horses, and Worms

Malicious code or **rogue programs** or **malware** (short for MALicious softWARE) is the general name for programs or program parts planted by an agent with malicious intent to cause unanticipated or undesired effects. The agent is the program's writer or distributor. Malicious intent distinguishes this type of code from unintentional errors, even though both kinds can certainly have similar and serious negative effects. This definition also excludes coincidence, in which minor flaws in two benign programs combine for a negative effect. Most faults found in software inspections, reviews, and testing do not qualify as malicious code; their cause is usually unintentional. However, unintentional faults can in fact invoke the same responses as intentional malevolence; a benign cause can still lead to a disastrous effect.

Malicious code can be directed at a specific user or class of users, or it can be for anyone.

You may have been affected by malware at one time or another, either because your computer was infected or because you could not access an infected system while its administrators were cleaning up the mess caused by the infection. The malware may have been caused by a worm or a virus or neither; the infection metaphor often seems apt, but the terminology of malicious code is sometimes used imprecisely. Here we distinguish names applied to certain types of malware, but you should focus on methods and impacts, instead of names. That which we call a virus by any other name would smell as vile.

A **virus** is a program that can replicate itself and pass on malicious code to other nonmalicious programs by modifying them. The term "virus" was coined because the affected program acts like a biological virus: It infects other healthy subjects by attaching itself to the program and either destroying the program or coexisting with it.

Because viruses are insidious, we cannot assume that a clean program yesterday is still clean today. Moreover, a good program can be modified to include a copy of the virus program, so the infected good program itself begins to act as a virus, infecting other programs. The infection usually spreads at a geometric rate, eventually overtaking an entire computing system and spreading to other connected systems.

Virus: code with malicious purpose; intended to spread

A virus can be either transient or resident. A **transient virus** has a life span that depends on the life of its host; the virus runs when the program to which it is attached executes, and it terminates when the attached program ends. (During its execution, the transient virus may spread its infection to other programs.) A **resident virus** locates itself in memory; it can then remain active or be activated as a stand-alone program, even after its attached program ends.

The terms worm and virus are often used interchangeably, but they actually refer to different things. A **worm** is a program that spreads copies of itself through a network. (John Shoch and Jon Hupp [SHO82] are apparently the first to describe a worm, which, interestingly, was created for nonmalicious purposes. Researchers at the Xerox Palo Alto Research Center, Shoch and Hupp wrote the first program as an experiment in distributed computing.) The primary difference between a worm and a virus is that a worm operates through networks, and a virus can spread through any medium (but usually uses a copied program or data files). Additionally, the worm spreads copies of itself as a stand-alone program, whereas the virus spreads copies of itself as a program that attaches to or embeds in other programs.

Worm: program that spreads copies of itself through a network

Spreading copies of yourself seems boring and perhaps narcissistic. But worms do have a common, useful purpose. How big is the Internet? What addresses are in use? Worm programs, sometimes called "crawlers" seek out machines on which they can install small pieces of code to gather such data. The code items report back to collection points, telling what connectivity they have found. As we describe in Chapter 6, this kind of reconnaissance can also have a negative security purpose; the worms that travel and collect data do not have to be evil.

As a slightly different example of this type of worm, consider how search engines know about all the pages on the web. A **bot** (short for robot), is a kind of worm used in vast numbers by search engine hosts like Bing and Google. Armies of these agents run on any computers on which they can install themselves. Their purpose is to scan accessible web content continuously and report back to their controller any new content they have found. In this way, the agents find pages that their controllers then catalog, enabling the search engines to return these results in response to individuals' queries. Thus, when you post a new web page (or modify an old one) with results of your research on why people like peanut butter, a crawler soon notices that page and informs its controller of the contents and whereabouts of your new page.

A **Trojan horse** is malicious code that, in addition to its primary effect, has a second, nonobvious, malicious effect. The name is derived from a reference to the Trojan war. Legends tell how the Greeks tricked the Trojans by leaving a great wooden horse outside the Trojans' defensive wall. The Trojans, thinking the horse a gift, took it inside and gave it pride of place. But unknown to the naïve Trojans, the wooden horse was filled with the bravest of Greek soldiers. In the night, the Greek soldiers descended from the horse, opened the gates, and signaled their troops that the way in was now clear to capture Troy. In the same way, Trojan horse malware slips inside a program undetected and produces unwelcome effects later on.

As an example of a computer Trojan horse, consider a login script that solicits a user's identification and password, passes the identification information on to the rest of the system for login processing, but also retains a copy of the information for later, malicious use. In this example, the user sees only the login occurring as expected, so there is no reason to suspect that any other, unwelcome action took place.

Trojan horse: program with benign apparent effect but second, hidden, malicious effect

To remember the differences among these three types of malware, understand that a Trojan horse is on the surface a useful program with extra, undocumented (malicious) features. It does not necessarily try to propagate. By contrast, a virus is a malicious program that attempts to spread to other computers, as well as perhaps performing unpleasant action on its current host. The virus does not necessarily spread by using a network's properties; it can be spread instead by traveling on a document transferred by a portable device (that memory stick you just inserted in your laptop!) or triggered to spread to other, similar file types when a file is opened. However, a worm requires a network for its attempts to spread itself elsewhere.

Beyond this basic terminology, there is much similarity in types of malicious code. Many other types of malicious code are shown in Table 3-2. As you can see, types of malware differ widely in their operation, transmission, and objective. Any of these terms is used popularly to describe malware, and you will encounter imprecise and overlapping definitions. Indeed, people sometimes use virus as a convenient general term for malicious code. Again, let us remind you that nomenclature is not critical; impact and effect are. Battling over whether something is a virus or worm is beside the point; instead, we concentrate on understanding the mechanisms by which malware perpetrates its evil.

In this chapter we explore viruses in particular, because their ability to replicate and cause harm gives us insight into two aspects of malicious code. Throughout the rest of this chapter we may also use the general term **malware** for any type of malicious code. You should recognize that, although we are interested primarily in the malicious aspects of these code forms so that we can recognize and address them, not all activities listed here are always malicious.

Every month the security firm Kaspersky reports the top 20 infections detected on users' computers by its products. (See http://www.securelist.com/en/analysis.) In April

TABLE 3-2 Types of Malicious Code

Code Type	Characteristics
Virus	Code that causes malicious behavior and propagates copies of itself to other programs
Trojan horse	Code that contains unexpected, undocumented, additional functionality
Worm	Code that propagates copies of itself through a network; impact is usually degraded performance
Rabbit	Code that replicates itself without limit to exhaust resources
Logic bomb	Code that triggers action when a predetermined condition occurs
Time bomb	Code that triggers action when a predetermined time occurs
Dropper	Transfer agent code only to drop other malicious code, such as virus or Trojan horse
Hostile mobile code agent	Code communicated semi-autonomously by programs transmitted through the web
Script attack, JavaScript, Active code attack	Malicious code communicated in JavaScript, ActiveX, or another scripting language, downloaded as part of displaying a web page
RAT (remote access Trojan)	Trojan horse that, once planted, gives access from remote location
Spyware	Program that intercepts and covertly communicates data on the user or the user's activity
Bot	Semi-autonomous agent, under control of a (usually remote) controller or "herder"; not necessarily malicious
Zombie	Code or entire computer under control of a (usually remote) program
Browser hijacker	Code that changes browser settings, disallows access to certain sites, or redirects browser to others
Rootkit	Code installed in "root" or most privileged section of operating system; hard to detect
Trapdoor or backdoor	Code feature that allows unauthorized access to a machine or program; bypasses normal access control and authentication
Tool or toolkit	Program containing a set of tests for vulnerabilities; not dangerous itself, but each successful test identifies a vulnerable host that can be attacked
Scareware	Not code; false warning of malicious code attack

2014, for example, there were eight adware attacks (ads offering useless or malicious programs for sale), and nine Trojan horses or Trojan horse transmitters in the top 20, and two exploit script attacks, which we also describe in this chapter. But the top attack type, comprising 81.73 percent of attacks, was malicious URLs, described in the next chapter. A different measure counts the number of pieces of malicious code Kaspersky products found on protected computers (that is, malware not blocked by Kaspersky's email and Internet activity screens). Among the top 20 types of malware were five

SIDEBAR 3-5 The Real Impact of Malware

Measuring the real impact of malware, especially in financial terms, is challenging if not impossible. Organizations are loath to report breaches except when required by law, for fear of damage to reputation, credit rating, and more. Many surveys report number of incidents, financial impact, and types of attacks, but by and large they are convenience surveys that do not necessarily represent the real situation. Shari Lawrence Pfleeger [PFL08], Rachel Rue [RUE09], and Ian Cook [COO10] describe in more detail why these reports are interesting but not necessarily trustworthy.

For the last several years, Verizon has been studying breaches experienced by many customers willing to collaborate and provide data; the Verizon reports are among the few credible and comparable studies available today. Although you should remember that the results are particular to the type of customer Verizon supports, the results are nonetheless interesting for illustrating that malware has had severe impacts in a wide variety of situations.

The 2014 Verizon Breach Report [VER14] shows that, from 2010 to 2013, the percentage of data breaches motivated by financial gain fell from about 90 percent to 55 percent, while the number of breaches for purpose of espionage rose from near zero percent to almost 25 percent. Although the figures show some swings from year to year, the overall trend is downward for financial gain and upward for espionage. (Verizon acknowledges part of the increase is no doubt due to more comprehensive reporting from a larger number of its reporting partners; thus the data may reflect better data collection from more sources.)

Do not be misled, however. Espionage certainly has a financial aspect as well. The cost of a data breach at a point of sale (fraud at the checkout desk) is much easier to calculate than the value of an invention or a pricing strategy. Knowing these things, however, can help a competitor win sales away from the target of the espionage.

Trojan horses, one Trojan horse transmitter, eight varieties of adware, two viruses, two worms, and one JavaScript attack. So all attack types are important, and, as Sidebar 3-5 illustrates, general malicious code has a significant impact on computing.

We preface our discussion of the details of these types of malware with a brief report on the long history of malicious code. Over time, malicious code types have evolved as the mode of computing itself has changed from multiuser mainframes to single-user personal computers to networked systems to the Internet. From this background you will be able to understand not only where today's malicious code came from but also how it might evolve.

History of Malicious Code

The popular literature and press continue to highlight the effects of malicious code as if it were a relatively recent phenomenon. It is not. Fred Cohen [COH87] is sometimes

credited with the discovery of viruses, but Cohen only gave a name to a phenomenon known long before. For example, Shoch and Hupp [SHO82] published a paper on worms, and Ken Thompson, in his 1984 Turing Award lecture, "Reflections on Trusting Trust" [THO84], described malicious code that can be passed by a compiler. In that lecture, he refers to an earlier Air Force document, the Multics security evaluation by Paul Karger and Roger Schell [KAR74, KAR02]. In fact, references to malicious code go back at least to 1970. Willis Ware's 1970 study (publicly released in 1979 [WAR70]) and James P. Anderson's planning study for the U.S. Air Force [AND72] *still,* decades later, accurately describe threats, vulnerabilities, and program security flaws, especially intentional ones.

Perhaps the progenitor of today's malicious code is the game Darwin, developed by Vic Vyssotsky, Doug McIlroy, and Robert Morris of AT&T Bell Labs in 1962 (described in [ALE72]). This program was not necessarily malicious but it certainly was malevolent: It represented a battle among computer programs, the objective of which was to kill opponents' programs. The battling programs had a number of interesting properties, including the ability to reproduce and propagate, as well as hide to evade detection and extermination, all of which sound like properties of current malicious code.

Malicious code dates certainly to the 1970s, and likely earlier. Its growth has been explosive, but it is certainly not a recent phenomenon.

Through the 1980s and early 1990s, malicious code was communicated largely person-to-person by means of infected media (such as removable disks) or documents (such as macros attached to documents and spreadsheets) transmitted through email. The principal exception to individual communication was the Morris worm [ROC89, SPA89, ORM03], which spread through the young and small Internet, then known as the ARPANET. (We discuss the Morris worm in more detail later in this chapter.)

During the late 1990s, as the Internet exploded in popularity, so too did its use for communicating malicious code. Network transmission became widespread, leading to Melissa (1999), ILoveYou (2000), and Code Red and NIMDA (2001), all programs that infected hundreds of thousands—and possibly millions—of systems.

Malware continues to become more sophisticated. For example, one characteristic of Code Red, its successors SoBig and Slammer (2003), as well as most other malware that followed, was exploitation of known system vulnerabilities, for which patches had long been distributed but for which system owners had failed to apply the protective patches. In 2012 security firm Solutionary looked at 26 popular toolkits used by hackers and found that 58 percent of vulnerabilities exploited were over two years old, with some dating back to 2004.

A more recent phenomenon is called a **zero-day attack**, meaning use of malware that exploits a previously unknown vulnerability or a known vulnerability for which no countermeasure has yet been

Zero day attack: Active malware exploiting a product vulnerability for which the manufacturer has no countermeasure available.

distributed. The moniker refers to the number of days (zero) during which a known vulnerability has gone without being exploited. The exploit window is diminishing rapidly, as shown in Sidebar 3-6.

SIDEBAR 3-6 Rapidly Approaching Zero

Y2K or the year 2000 problem, when dire consequences were forecast for computer clocks with 2-digit year fields that would turn from 99 to 00, was an ideal problem: The threat was easy to define, time of impact was easily predicted, and plenty of advance warning was given. Perhaps as a consequence, very few computer systems and people experienced significant harm early in the morning of 1 January 2000. Another countdown clock has computer security researchers much more concerned.

The time between general knowledge of a product vulnerability and appearance of code to exploit that vulnerability is shrinking. The general exploit timeline follows this sequence:

- An attacker discovers a previously unknown vulnerability.
- The manufacturer becomes aware of the vulnerability.
- Someone develops code (called proof of concept) to demonstrate the vulnerability in a controlled setting.
- The manufacturer develops and distributes a patch or workaround that counters the vulnerability.
- Users implement the control.
- Someone extends the proof of concept, or the original vulnerability definition, to an actual attack.

As long as users receive and implement the control before the actual attack, no harm occurs. An attack before availability of the control is called a **zero-day exploit**. Time between proof of concept and actual attack has been shrinking. Code Red, one of the most virulent pieces of malicious code, in 2001 exploited vulnerabilities for which the patches had been distributed more than a month before the attack. But more recently, the time between vulnerability and exploit has steadily declined. On 18 August 2005, Microsoft issued a security advisory to address a vulnerability of which the proof of concept code was posted to the French SIRT (Security Incident Response Team) web site frsirt.org. A Microsoft patch was distributed a week later. On 27 December 2005, a vulnerability was discovered in Windows metafile (.WMF) files. Within hours hundreds of sites began to exploit the vulnerability to distribute malicious code, and within six days a malicious code toolkit appeared, by which anyone could easily create an exploit. Microsoft released a patch in nine days.

Security firm Symantec in its Global Internet Security Threat Report [SYM14b] found 23 zero-day vulnerabilities in 2013, an increase from 14 the previous year and 8 for 2011. Although these seem like small numbers the important observation is the upward trend and the rate of increase. Also,

(*continues*)

SIDEBAR 3-6 *Continued*

software under such attack is executed by millions of users in thousands of applications. Because a zero-day attack is a surprise to the maintenance staff of the affected software, the vulnerability remains exposed until the staff can find a repair. Symantec reports vendors take an average of four days to prepare and distribute a patch for the top five zero-day attacks; users will actually apply the patch at some even later time.

But what exactly is a zero-day exploit? It depends on who is counting. If the vendor knows of the vulnerability but has not yet released a control, does that count as zero day, or does the exploit have to surprise the vendor? David Litchfield of Next Generation Software in the U.K. identified vulnerabilities and informed Oracle. He claims Oracle took an astonishing 800 days to fix two of them and others were not fixed for 650 days. Other customers are disturbed by the slow patch cycle—Oracle released no patches between January 2005 and March 2006 [GRE06]. Distressed by the lack of response, Litchfield finally went public with the vulnerabilities to force Oracle to improve its customer support. Obviously, there is no way to determine if a flaw is known only to the security community or to attackers as well unless an attack occurs.

Shrinking time between knowledge of vulnerability and exploit puts pressure on vendors and users both, and time pressure is not conducive to good software development or system management.

The worse problem cannot be controlled: vulnerabilities known to attackers but not to the security community.

Today's malware often stays dormant until needed, or until it targets specific types of software to debilitate some larger (sometimes hardware) system. For instance, Conficker (2008) is a general name for an infection that leaves its targets under the control of a master agent. The effect of the infection is not immediate; the malware is latent until the master agent causes the infected agents to download specific code and perform a group attack.

Malware doesn't attack just individual users and single computers. Major applications and industries are also at risk.

For example, Stuxnet (2010) received a great deal of media coverage in 2010. A very sophisticated piece of code, Stuxnet exploits a vulnerability in Siemens' industrial control systems software. This type of software is especially popular for use in supervisory control and data acquisition (SCADA) systems, which control processes in chemical manufacturing, oil refining and distribution, and nuclear power plants—all processes whose failure can have catastrophic consequences. Table 3-3 gives a timeline of some of the more notable malicious code infections.

With this historical background we now explore more generally the many types of malicious code.

TABLE 3-3 Notable Malicious Code Infections

Year	Name	Characteristics
1982	Elk Cloner	First virus; targets Apple II computers
1985	Brain	First virus to attack IBM PC
1988	Morris worm	Allegedly accidental infection disabled large portion of the ARPANET, precursor to today's Internet
1989	Ghostballs	First multipartite (has more than one executable piece) virus
1990	Chameleon	First polymorphic (changes form to avoid detection) virus
1995	Concept	First virus spread via Microsoft Word document macro
1998	Back Orifice	Tool allows remote execution and monitoring of infected computer
1999	Melissa	Virus spreads through email address book
2000	IloveYou	Worm propagates by email containing malicious script. Retrieves victim's address book to expand infection. Estimated 50 million computers affected.
2000	Timofonica	First virus targeting mobile phones (through SMS text messaging)
2001	Code Red	Virus propagates from 1st to 20th of month, attacks whitehouse.gov web site from 20th to 28th, rests until end of month, and restarts at beginning of next month; resides only in memory, making it undetected by file-searching antivirus products
2001	Code Red II	Like Code Red, but also installing code to permit remote access to compromised machines
2001	Nimda	Exploits known vulnerabilities; reported to have spread through 2 million machines in a 24-hour period
2003	Slammer worm	Attacks SQL database servers; has unintended denial-of-service impact due to massive amount of traffic it generates
2003	SoBig worm	Propagates by sending itself to all email addresses it finds; can fake From: field; can retrieve stored passwords
2004	MyDoom worm	Mass-mailing worm with remote-access capability
2004	Bagle or Beagle worm	Gathers email addresses to be used for subsequent spam mailings; SoBig, MyDoom, and Bagle seemed to enter a war to determine who could capture the most email addresses
2008	Rustock.C	Spam bot and rootkit virus
2008	Conficker	Virus believed to have infected as many as 10 million machines; has gone through five major code versions
2010	Stuxnet	Worm attacks SCADA automated processing systems; zero-day attack
2011	Duqu	Believed to be variant on Stuxnet
2013	CryptoLocker	Ransomware Trojan that encrypts victim's data storage and demands a ransom for the decryption key

Technical Details: Malicious Code

The number of strains of malicious code is unknown. According to a testing service [AVC10], malicious code detectors (such as familiar antivirus tools) that look for malware "signatures" cover over 1 million definitions, although because of mutation, one strain may involve several definitions. Infection vectors include operating systems, document applications (primarily word processors and spreadsheets), media players, browsers, document-rendering engines (such as Adobe PDF reader) and photo-editing programs. Transmission media include documents, photographs, and music files, on networks, disks, flash media (such as USB memory devices), and even digital photo frames. Infections involving other programmable devices with embedded computers, such as mobile phones, automobiles, digital video recorders, and cash registers, are becoming targets for malicious code.

In this section we explore four aspects of malicious code infections:

- *harm*—how they affect users and systems
- *transmission and propagation*—how they are transmitted and replicate, and how they cause further transmission
- *activation*—how they gain control and install themselves so that they can reactivate
- *stealth*—how they hide to avoid detection

We begin our study of malware by looking at some aspects of harm caused by malicious code.

Harm from Malicious Code

Viruses and other malicious code can cause essentially unlimited harm. Because malware runs under the authority of the user, it can do anything the user can do. In this section we give some examples of harm malware can cause. Some examples are trivial, more in the vein of a comical prank. But other examples are deadly serious with obvious critical consequences.

We can divide the payload from malicious code into three categories:

- *Nondestructive.* Examples of behavior are sending a funny message or flashing an image on the screen, often simply to show the author's capability. This category would also include **virus hoaxes**, messages falsely warning of a piece of malicious code, apparently to cause receivers to panic and forward the message to contacts, thus spreading the panic.
- *Destructive.* This type of code corrupts files, deletes files, damages software, or executes commands to cause hardware stress or breakage with no apparent motive other than to harm the recipient.
- *Commercial or criminal intent*. An infection of this type tries to take over the recipient's computer, installing code to allow a remote agent to cause the computer to perform actions on the agent's signal or to forward sensitive data to the agent. Examples of actions include collecting personal data, for example, login credentials to a banking web site, collecting proprietary data, such as corporate

plans (as was reported for an infection of computers of five petroleum industry companies in February 2011), or serving as a compromised agent for sending spam email or mounting a denial-of-service attack, as described in Chapter 6.

As we point out in Chapter 1, without our knowing the mind of the attacker, motive can be hard to determine. However, this third category has an obvious commercial motive. Organized crime has taken an interest in using malicious code to raise money [WIL01, BRA06, MEN10].

Harm to Users

Most malicious code harm occurs to the infected computer's data. Here are some real-world examples of malice.

- Hiding the cursor.
- Displaying text or an image on the screen.
- Opening a browser window to web sites related to current activity (for example, opening an airline web page when the current site is a foreign city's tourist board).
- Sending email to some or all entries in the user's contacts or alias list. Note that the email would be delivered as having come from the user, leading the recipient to think it authentic. The Melissa virus did this, sending copies of itself as an attachment that unsuspecting recipients would open, which then infected the recipients and allowed the infection to spread to their contacts.
- Opening text documents and changing some instances of "is" to "is not," and vice versa. Thus, "Raul is my friend" becomes "Raul is not my friend." The malware changed only a few instances in random locations, so the change would not be readily apparent. Imagine the effect these changes would have on a term paper, proposal, contract, or news story.
- Deleting all files. The Jerusalem virus did this every Friday that was a 13th day of the month.
- Modifying system program files. Many strains of malware do this to ensure subsequent reactivation and avoid detection.
- Modifying system information, such as the Windows registry (the table of all critical system information).
- Stealing and forwarding sensitive information such as passwords and login details.

In addition to these direct forms of harm, the user can be harmed indirectly. For example, a company's public image can be harmed if the company's web site is hijacked to spread malicious code. Or if the attack makes some web files or functions unavailable, people may switch to a competitor's site permanently (or until the competitor's site is attacked).

Although the user is most directly harmed by malware, there is secondary harm as the user tries to clean up a system after infection. Next we consider the impact on the user's system.

Harm to the User's System

Malware writers usually intend that their code persist, so they write the code in a way that resists attempts to eradicate it. Few writers are so obvious as to plant a file named "malware" at the top-level directory of a user's disk. Here are some maneuvers by which malware writers conceal their infection; these techniques also complicate detection and eradication.

- Hide the file in a lower-level directory, often a subdirectory created or used by another legitimate program. For example, the Windows operating system maintains subdirectories for some installed programs in a folder named "registered packages." Inside that folder are subfolders with unintelligible names such as {982FB688-E76B-4246-987B-9218318B90A}. Could you tell to what package that directory belongs or what files properly belong there?
- Attach, using the techniques described earlier in this chapter, to a critical system file, especially one that is invoked during system startup (to ensure the malware is reactivated).
- Replace (retaining the name of) a noncritical system file. Some system functionality will be lost, but a cursory look at the system files will not highlight any names that do not belong.
- Hide copies of the executable code in more than one location.
- Hide copies of the executable in different locations on different systems so no single eradication procedure can work.
- Modify the system registry so that the malware is always executed or malware detection is disabled.

As these examples show, ridding a system of malware can be difficult because the infection can be in the system area, installed programs, the user's data or undocumented free space. Copies can move back and forth between memory and a disk drive so that after one location is cleaned, the infection is reinserted from the other location.

For straightforward infections, simply removing the offending file eradicates the problem. Viruses sometimes have a **multipartite** form, meaning they install themselves in several pieces in distinct locations, sometimes to carry out different objectives. In these cases, if only one piece is removed, the remaining pieces can reconstitute and reinstall the deleted piece; eradication requires destroying all pieces of the infection. But for more deeply established infections, users may have to erase and reformat an entire disk, and then reinstall the operating system, applications, and user data. (Of course, users can reinstall these things only if they have intact copies from which to begin.)

Thus, the harm to the user is not just in the time and effort of replacing data directly lost or damaged but also in handling the secondary effects to the system and in cleaning up any resulting corruption.

Harm to the World

An essential character of most malicious code is its spread to other systems. Except for specifically targeted attacks, malware writers usually want their code to infect many people, and they employ techniques that enable the infection to spread at a geometric rate.

The Morris worm of 1988 infected only 3,000 computers, but those computers constituted a significant proportion, perhaps as much as half, of what was then the Internet. The IloveYou worm (transmitted in an email message with the alluring subject line "I Love You") is estimated to have infected 100,000 servers; the security firm Message Labs estimated that, at the attack's height, 1 email of every 28 transmitted worldwide was an infection from the worm. Code Red is believed to have affected close to 3 million hosts. By some estimates, the Conficker worms (several strains) control a network of 1.5 million compromised and unrepaired hosts under the worms' author's control [MAR09]. Costs of recovery from major infections like these typically exceed $1 million US. Thus, computer users and society in general bear a heavy cost for dealing with malware.

Damage Estimates

How do you determine the cost or damage of any computer security incident? The problem is similar to the question of determining the cost of a complex disaster such as a building collapse, earthquake, oil spill, or personal injury. Unfortunately, translating harm into money is difficult, in computer security and other domains.

The first step is to enumerate the losses. Some will be tangibles, such as damaged equipment. Other losses include lost or damaged data that must be re-created or repaired, and degradation of service in which it takes an employee twice as long to perform a task. Costs also arise in investigating the extent of damage. (Which programs and data are affected and which archived versions are safe to reload?) Then there are intangibles and unmeasurables such as loss of customers or damage to reputation.

Estimating the cost of an incident is hard. That does not mean the cost is zero or insignificant, just hard to determine.

You must determine a fair value for each thing lost. Damaged hardware or software is easy if there is a price to obtain a replacement. For damaged data, you must estimate the cost of staff time to recover, re-create, or repair the data, including the time to determine what is and is not damaged. Loss of customers can be estimated from the difference between number of customers before and after an incident; you can price the loss from the average profit per customer. Harm to reputation is a real loss, but extremely difficult to price fairly. As we saw when exploring risk management, people's perceptions of risk affect the way they estimate the impact of an attack. So their estimates will vary for the value of loss of a human's life or damage to reputation.

Knowing the losses and their approximate cost, you can compute the total cost of an incident. But as you can easily see, determining what to include as losses and valuing them fairly can be subjective and imprecise. Subjective and imprecise do not mean invalid; they just indicate significant room for variation. You can understand, therefore, why there can be orders of magnitude differences in damage estimates for recovering from a security incident. For example, estimates of damage from Code Red range from $500 million to $2.6 billion, and one estimate of the damage from Conficker, for which 9 to 15 million systems were repaired (plus 1.5 million not yet cleaned of the infection), was $9.2 billion, or roughly $1,000 per system [DAN09].

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

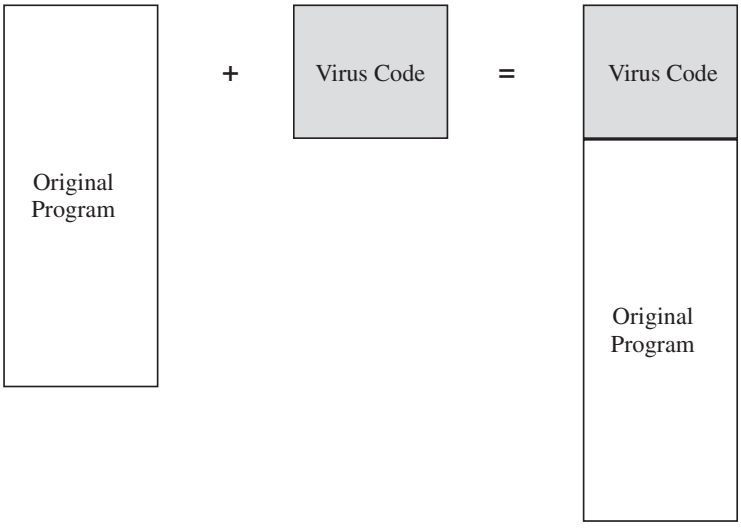


FIGURE 3-19 Virus Attachment

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.

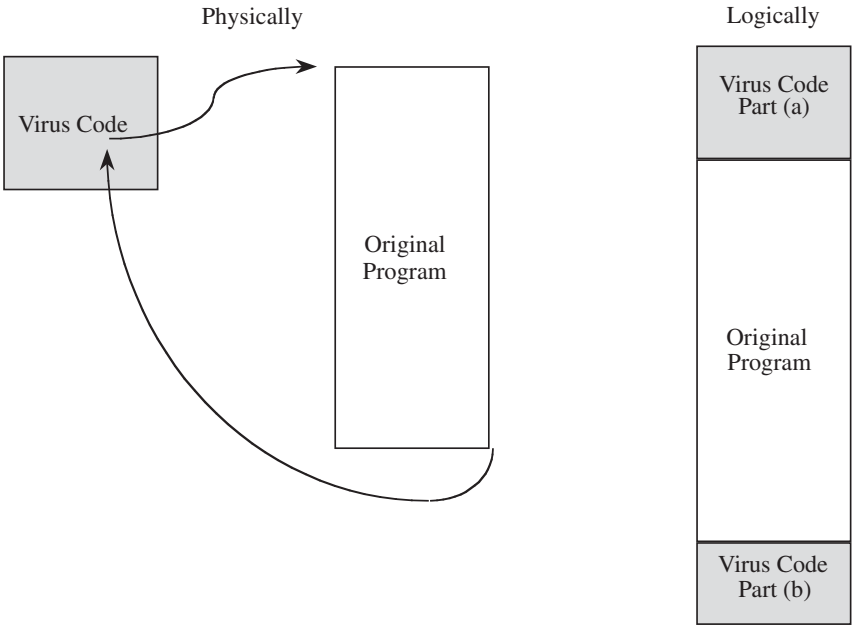


FIGURE 3-20 Surrounding Virus

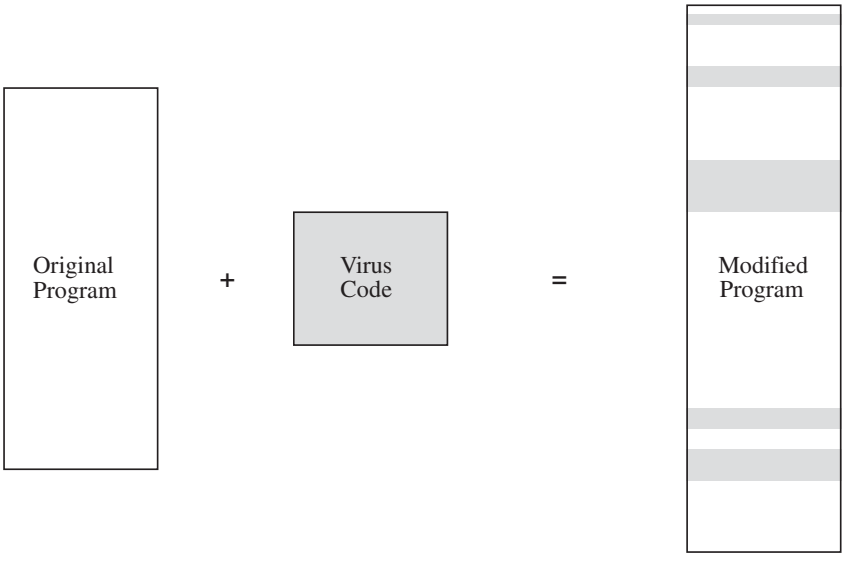


FIGURE 3-21 Virus Insertion

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.

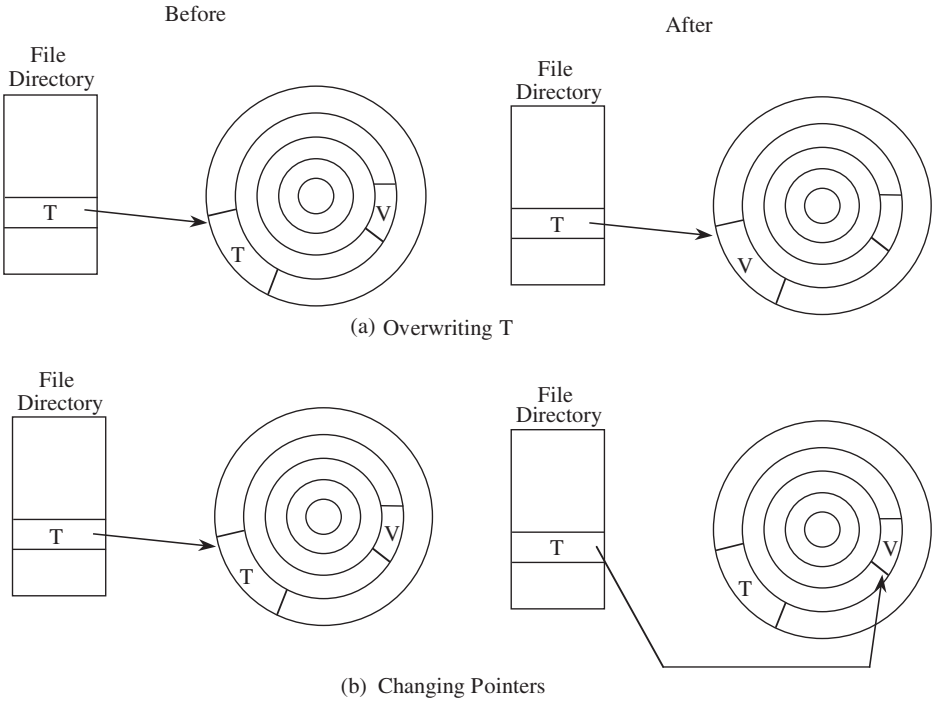


FIGURE 3-22 Virus V Replacing Target T

- 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.

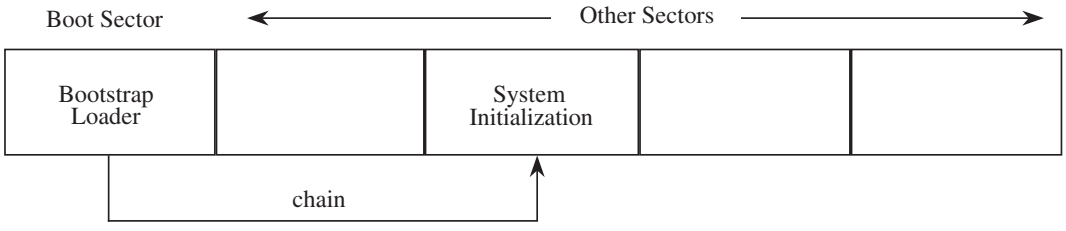
When a computer is started, control begins with firmware that determines which hardware components are present, tests them, and transfers control to an operating system. A given hardware platform can run many different operating systems, so the operating system is not coded in firmware but is instead 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.

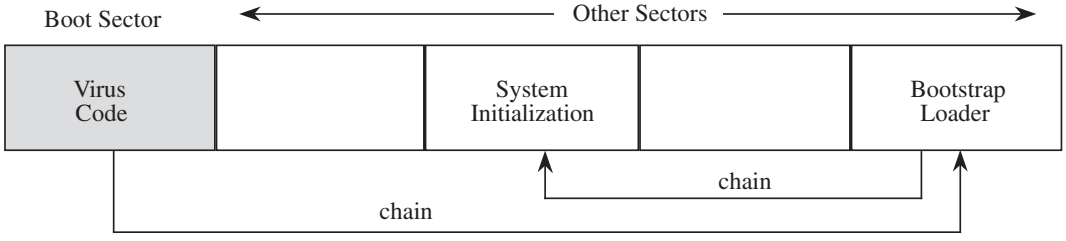
Malicious code can intrude in this bootstrap sequence in several ways. An assault can revise or add to the list of modules to be loaded, or substitute an infected module for a good one by changing the address of the module to be loaded or by substituting a modified routine of the same name. With boot 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.



(a) Before infection



(b) After infection

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

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

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

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?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
%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

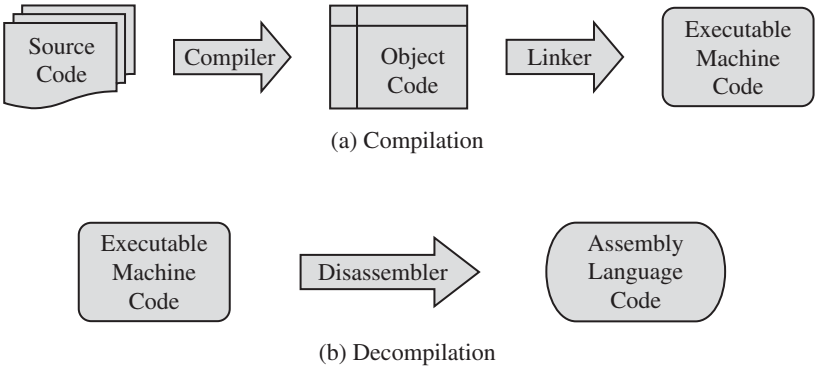


FIGURE 3-24 The Compilation Process: (a) Compilation. (b) Decompilation

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.

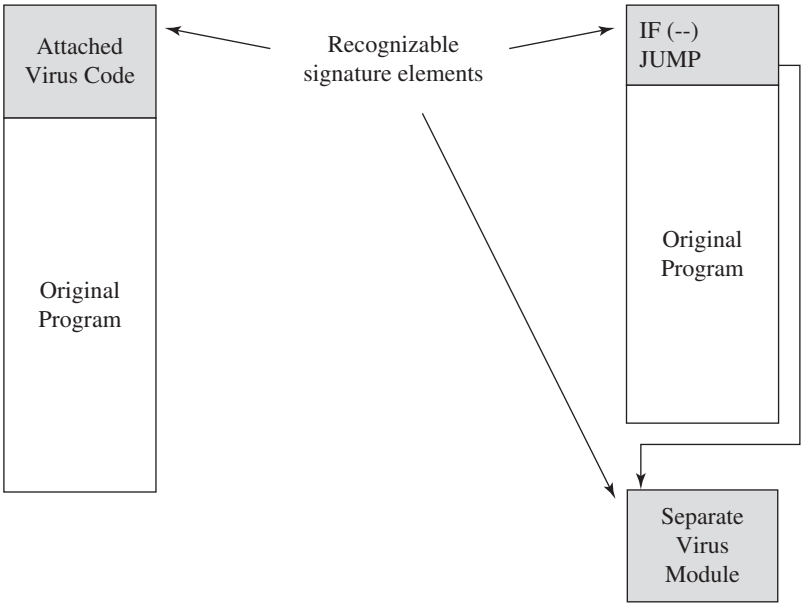


FIGURE 3-25 Recognizable Patterns in Viruses

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

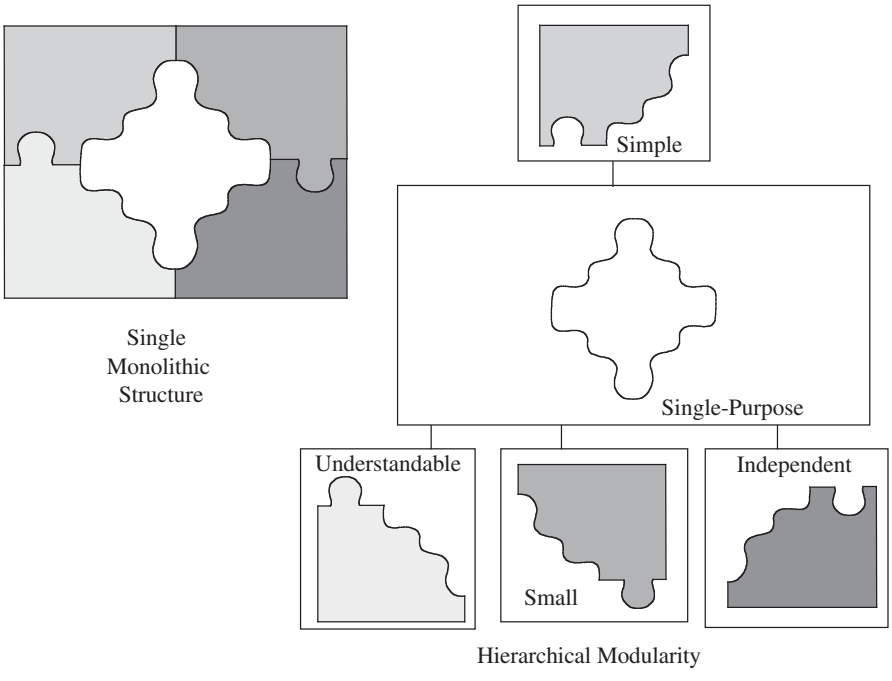


FIGURE 3-26 Modularity

- *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.

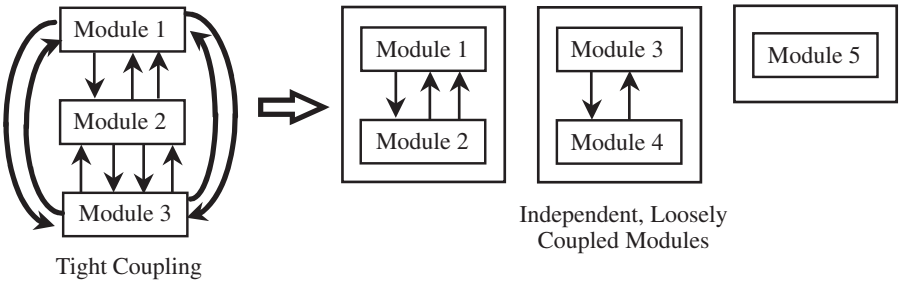


FIGURE 3-27 Types of Coupling

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.

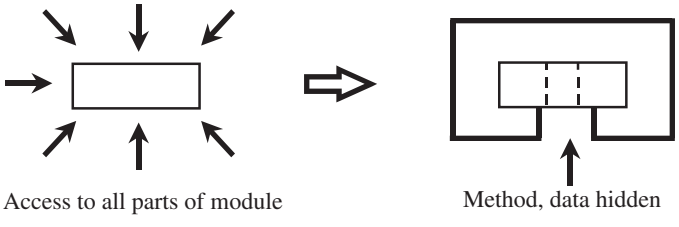


FIGURE 3-28 Information Hiding

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.