Module: 06
System Security

Syllabus:

Lecture no	Content	Duration (Hr)	Self-Study (Hrs)
1	Software Vulnerabilities: Buffer Overflow	1	1
2	Malware	1	1
3	Viruses, Worms, Trojans	1	1
4	Logic Bomb, Bots, Rootkit	1	2
5	SQL injection	1	2
6	Cross-site scripting	1	2

Theoretical Background:
Types of Programming Flaws

The taxonomy of program flaws, divides them first into intentional and inadvertent flaws. They further divide intentional flaws into malicious and nonmalicious ones. In the taxonomy, the inadvertent flaws fall into six categories:

- validation error (incomplete or inconsistent): permission checks
- domain error: controlled access to data
- serialization and aliasing: program flow order
- inadequate identification and authentication: basis for authorization
- boundary condition violation: failure on first or last case
- other exploitable logic errors

Non-Malicious Programming Errors

Being human, programmers and other developers make many mistakes, most of which are unintentional and nonmalicious. Many such errors cause program malfunctions but do not lead to more serious security vulnerabilities. However, a few classes of errors have plagued programmers and security professionals for decades, and there is no reason to believe they will disappear. In this section we consider three classic error types that have enabled many recent security breaches. We explain each type, why it is relevant to security, and how it can be prevented or mitigated.

Buffer Overflows

A buffer overflow is the computing equivalent of trying to pour two liters of water into a one-liter pitcher: Some water is going to spill out and make a mess. And in computing, what a mess these errors have made!

Definition

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, `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. However, if the statement were

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

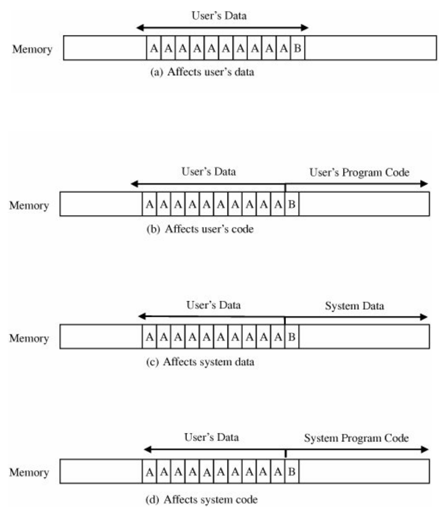
we could not identify the problem until `i` was set during execution to a too-big subscript. It would be useful if, during execution, the system produced an error message warning of a subscript out of bounds. Unfortunately, in some languages, buffer sizes do not have to be predefined, so there is no 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 the 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.

Let us examine this problem more closely. It is important 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 there are four cases to consider in deciding where the 'B' goes, as shown in Figure 3-1. 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.

Figure 3-1. Places Where a Buffer Can Overflow.



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 occur when the system owns the space immediately after the array that overflows. Spilling over into system data or code areas produces similar results to those for the user's space: computing with a faulty value or trying to execute an improper operation.

Incomplete Mediation

Incomplete mediation is another security problem that has been with us for decades. Attackers are exploiting it to cause security problems.

Definition

Consider the example of the previous section:

```
http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212
&parm2=2009Jan17
```

The two parameters look like a telephone number and a date. Probably the client's (user's) web browser enters those two values in their specified format for easy processing on the server's side. What would happen if `parm2` were submitted as 1800Jan01? Or 1800Feb30? Or 2048Min32? Or 1Aardvark2Many?

Security Implication

Incomplete mediation is easy to exploit, but it has been exercised less often than buffer overflows. Nevertheless, 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 the web site so that 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 passage 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; presumably Things knows how much its Objects cost, and they are unlikely to change dramatically since the time the price was quoted a few screens earlier.

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's web site and bought Objects at a fraction of their 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 found the error quickly.

Time-of-Check to Time-of-Use Errors

The third programming flaw we investigate 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.

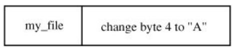
Definition

Access control is a fundamental part of computer security; we want to make sure that only those who should access an object are allowed that access. (We explore the access control mechanisms in operating systems in greater detail in Chapter 4.) 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. It is also known as a serialization or synchronization flaw.

To understand the nature of this flaw, consider a person's buying a sculpture that costs $100. The buyer removes five $20 bills from a wallet, 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 (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 3-2.

Figure 3-2. Data Structure for File Access.

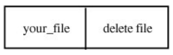


The data structure is essentially a "work ticket," requiring a stamp of authorization; once authorized, it is put on a queue of things to be done. Normally the access control mediator receives the data structure, determines whether the access should be allowed, and either rejects the access and stops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

It is at this point that 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 3-3. 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.

Figure 3-3. Modified Data.



The problem is called a time-of-check to time-of-use flaw because it exploits the delay between the two times. 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.

Kinds of Malicious Code

Malicious code or **rogue program** is the general name for unanticipated or undesired effects in programs or program parts, caused by an agent intent on damage. This definition excludes unintentional errors, although they can also have a serious negative effect. This definition also excludes coincidence, in which two benign programs combine for a negative effect. The **agent** is the writer of the program or the person who causes its distribution. By this definition, most faults found in software inspections, reviews, and testing do not qualify as malicious code, because we think of them as unintentional. However, keep in mind as you read this chapter that unintentional faults can in fact invoke the same responses as intentional malevolence; a benign cause can still lead to a disastrous effect.

You are likely to have been affected by a virus at one time or another, either because your computer was infected by one or because you could not access an infected system while its administrators were cleaning up the mess one made. In fact, your virus might actually have been a worm: The terminology of malicious code is sometimes used imprecisely. 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 it 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 all other connected systems.

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

A **Trojan horse** is malicious code that, in addition to its primary effect, has a second, nonobvious malicious effect.[1] 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 evident reason to suspect that any other action took place.

[1] The name is a reference to the Greek legends of the Trojan war. Legend tells how the Greeks tricked the Trojans into breaking their defense wall to take a wooden horse, filled with the bravest of Greek soldiers, into their citadel. In the night, the soldiers descended and signaled their troops that the way in was now clear, and Troy was captured.

A **logic bomb** is a class of malicious code that "detonates" or goes off when a specified condition occurs. A **time bomb** is a logic bomb whose trigger is a time or date.

A **trapdoor** or **backdoor** is a feature in a program by which someone can access the program other than by the obvious, direct call, perhaps with special privileges. For instance, an automated bank teller program might allow anyone entering the number 990099 on the keypad to process the log of everyone's transactions at that machine. In this example, the trapdoor could be intentional, for maintenance purposes, or it could be an illicit way for the implementer to wipe out any record of a crime.

A **worm** is a program that spreads copies of itself through a network. Shock and Hupp [SHO82] are apparently the first to describe a worm, which, interestingly, was for nonmalicious purposes. 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 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.

White et al. [WHI89] also define a **rabbit** as a virus or worm that self-replicates without bound, with the intention of exhausting some computing resource. A rabbit might create copies of itself and store them on disk in an effort to completely fill the disk, for example.

These definitions match current careful usage. The distinctions among these terms are small, and often the terms are confused, especially in the popular press. The term "virus" is often used to refer to any piece of malicious code. Furthermore, two or more forms of malicious code can be combined to produce a third kind of problem. For instance, a virus can be a time bomb if the viral code that is spreading will trigger an event after a period of time has passed. The kinds of malicious code are summarized in Table 3-1.

Table 3-1. Types of Malicious Code.

Code Type	Characteristics
Virus	Attaches itself to program and propagates copies of itself to other programs
Trojan horse	Contains unexpected, additional functionality
Logic bomb	Triggers action when condition occurs
Time bomb	Triggers action when specified time occurs
Trapdoor	Allows unauthorized access to functionality
Worm	Propagates copies of itself through a network
Rabbit	Replicates itself without limit to exhaust resources

Because "virus" is the popular name given to all forms of malicious code and because fuzzy lines exist between different kinds of malicious code, we are not too restrictive in the following discussion. We want to look at how malicious code spreads, how it is activated, and what effect it can have. A virus is a convenient term for mobile malicious code, so in the following sections we use the term "virus" almost exclusively. The points made apply also to other forms of malicious code.

How Viruses Attach

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

For example, recall the SETUP program that you initiate 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; 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 initiates the execution of the program to which the virus is attached. (It is possible for execution to 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.

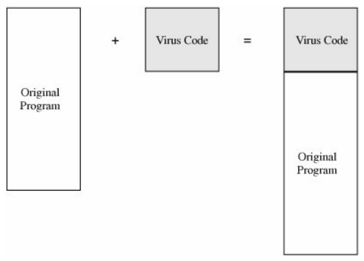
A more common means of virus activation is as an attachment to an e-mail message. In this attack, the virus writer tries to convince the victim (the recipient of the e-mail message) to open the attachment. Once the viral attachment is opened, the activated virus can do its work. Some modern e-mail handlers, in a drive to "help" the receiver (victim), automatically open attachments as soon as the receiver opens the body of the e-mail 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, it is safer to force users to open files on their own rather than automatically; it is a bad idea for programs to perform potentially security-relevant actions without a user's consent. However, ease-of-use often trumps security, so programs such as browsers, e-mail handlers, and viewers often "helpfully" open files without asking the user first.

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 program.

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 execu 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 3-4.

Figure 3-4. Virus Appended to a Program.

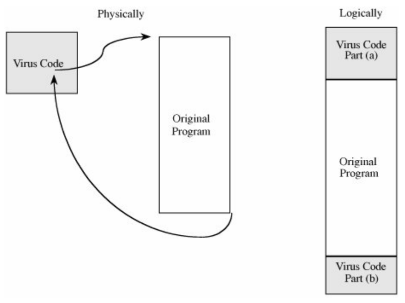


This kind of attachment is simple and usually effective. The virus writer does not need to know anything about the program to which the virus will attach, and c the attached program simply serves as a carrier for the virus. The virus performs its task and then transfers to the original program. Typically, the user is unaw 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 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 3-5.

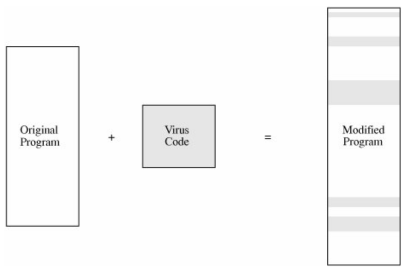
Figure 3-5. Virus Surrounding a Program.



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 3-6. Clearly, the virus writer has to know the exact structure of the original program to know where to insert which pieces of the virus.

Figure 3-6. Virus Integrated into a Program.



Finally, the virus can replace the entire target, either mimicking the effect of the target or ignoring the expected effect of the target and performing only the virus effect. In this case, the user is most likely to perceive the loss of the original program.

Document Viruses

Currently, the most popular virus type 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.

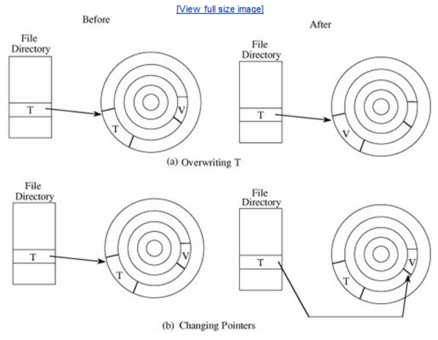
How Viruses Gain Control

The 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 3-7.

Figure 3-7. Virus Completely Replacing a Program.

[View full size image]



The virus can supplant T by altering the sequence that would have invoked T to now invoke the virus V; this invocation can be used to 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.

Homes for Viruses

The virus writer may find these qualities appealing in a virus:

- It is hard to detect.
- It is not easily destroyed or deactivated.
- It spreads infection widely.
- It can reinfect its home program or other programs.
- It is easy to create.
- It is machine independent and operating system independent.

One-Time Execution

The majority of viruses today execute only once, spreading their infection and causing their effect in that one execution. A virus often arrives as an e-mail attachment of a document virus. It is executed just by being opened.

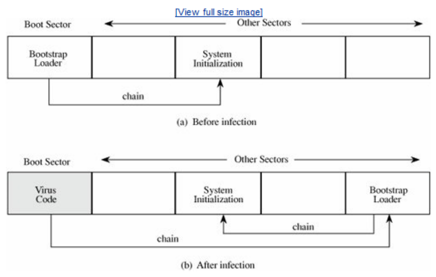
Boot Sector Viruses

A special case of virus attachment, but formerly a fairly popular one, is the so-called **boot sector virus.** 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.

The operating system is software stored on disk. Code copies the operating system from disk to memory and transfers control to it; this copying is called the **bootstrap** (often **boot**) load because the operating system figuratively pulls itself into memory by its bootstraps. The firmware does its control transfer by reading a fixed number of bytes from a fixed location on the disk (called the **boot sector**) to a fixed address in memory and then jumping to that address (which will turn out to contain the first instruction of the bootstrap loader). The bootstrap loader then reads into memory the rest of the operating system from disk. To run a different operating system, the user just inserts a disk with the new operating system and a bootstrap loader. When the user reboots from this new disk, the loader there brings in and runs another operating system. This same scheme is used for personal computers, workstations, and large mainframes.

To allow for change, expansion, and uncertainty, hardware designers reserve a large amount of space for the bootstrap load. The boot sector on a PC is slightly less than 512 bytes, but since the loader will be larger than that, the hardware designers support "chaining," in which each block of the bootstrap is chained to (contains the disk location of) the next block. This chaining allows big bootstraps but also simplifies the installation of a virus. The virus writer simply breaks the chain at any point, inserts a pointer to the virus code to be executed, and reconnects the chain after the virus has been installed. This situation is shown in Figure 3-8.

Figure 3-8. Boot Sector Virus Relocating Code.



The boot sector is an especially appealing place to house a virus. The virus gains control very 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, preventing their deletion. Thus, the virus code is not readily noticed by users.

Memory-Resident Viruses

Some parts of the operating system and most user programs execute, terminate, and disappear, with their space in memory being available for anything executed later. For very 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 was needed. 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. On a Windows machine the registry is the table of all critical system information, including programs to run at startup. If the virus gains control once, 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 virus will return on the next system restart.

Virus Signatures

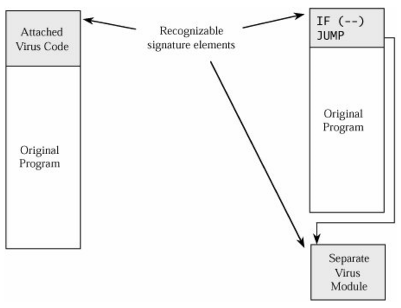
A virus cannot be completely invisible. Code must be stored somewhere, and the code must be in memory to execute. Moreover, the virus executes in a particular way, using certain methods to spread. Each of these characteristics yields a telltale pattern, called a **signature**, that can be found by a 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?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
%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
```

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 byte 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 obtain control of 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 3-9.

Figure 3-9. Recognizable Patterns in Viruses.



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.

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 3-2, 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.

Table 3-2. Virus Effects and Causes.

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 systems 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

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 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 disk 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 disks 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 47F0F00E08 (in hexadecimal) and has string 00113FFF located at word 12, it is unlikely that other programs or data files will 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. Many 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*, 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 constant code 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.

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 so that becomes the signature.

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.

The Source of Viruses

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.

Virus discovery could be aided by a procedure to determine if two programs are equivalent. However, theoretical results in computing are very discouraging when it comes to the complexity of the equivalence problem. The general question "Are these two programs equivalent?" is undecidable (although that question *can* be answered for many specific pairs of programs). Even ignoring the general undecidability problem, two modules may produce subtly different results that mayor may notbe 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 is dismaying, the particular is not. If we know that a particular virus may infect a computing system, we can check for it 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.

Prevention of Virus Infection

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 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-related files, and even media files can contain code or scripts that can be executed in various waysand 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 run or not! Against the principles of good security, e-mail handlers can be set to automatically open (without performing access control) attachments or embedded code for the recipient, so your e-mail 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. 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* (Power-Point) 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 is 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. 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 granular; if every sixteenth bit is part of a command string that can be executed, then the virus is very difficult to detect.

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, it is not feasible to cut off all contact with the outside world.

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. Governments, for military or diplomatic secrets, often run disconnected network communities. The trick seems to be in choosing one's community prudently. However, as use of the Internet and the World Wide Web increases, such separation is almost impossible to maintain.

Nevertheless, 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 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.

- *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 it 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 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 backupthat is, one taken before the infection.

- *Use virus detectors (often called virus scanners) regularly and update them daily.* Many of the available virus detectors can both detect and eliminate infection from viruses. Several scanners are better than one because one may detect the viruses that others miss. Because scanners search for virus signatures, they are constantly being revised as new viruses are discovered. New virus signature files or new versions of scanners are distributed frequently; often, you can request automatic downloads from the vendor's web site. Keep your detector's signature file up to date.

Truths and Misconceptions About Viruses

Because viruses often have a dramatic impact on the computer-using community, they are often highlighted in the press, particularly in the business section. However, there is much misinformation in circulation about viruses. Let us examine some of the popular claims about them.

- *Viruses can infect only Microsoft Windows systems. **False***. Among students and office workers, PCs running Windows are popular computers, and there may be more people writing software (and viruses) for them than for any other kind of processor. Thus, the PC is most frequently the target when someone decides to write a virus. However, the principles of virus attachment and infection apply equally to other processors, including Macintosh computers, Unix and Linux workstations, and mainframe computers. Cell phones and PDAs are now also virus targets. In fact, no writeable stored-program computer is immune to possible virus attack. As we noted in Chapter 1, this situation means that *all* devices containing computer code, including automobiles, airplanes, microwave ovens, radios, televisions, voting machines, and radiation therapy machines have the potential for being infected by a virus.

- *Viruses can modify "hidden" or "read-only" files. **True***. We may try to protect files by using two operating system mechanisms. First, we can make a file a hidden file so that a user or program listing all files on a storage device will not see the file's name. Second, we can apply a read-only protection to the file so that the user cannot change the file's contents. However, each of these protections is applied by software, and virus software can override the native software's protection. Moreover, software protection is layered, with the operating system providing the most elementary protection. If a secure operating system obtains control *before* a virus contaminator has executed, the operating system can prevent contamination as long as it blocks the attacks the virus will make.

- *Viruses can appear only in data files, or only in Word documents, or only in programs. **False***. What are data? What is an executable file? The distinction between these two concepts is not always clear, because a data file can control how a program executes and even cause a program to execute. Sometimes a data file lists steps to be taken by the program that reads the data, and these steps can include executing a program. For example, some applications contain a configuration file whose data are exactly such steps. Similarly, word-processing document files may contain startup commands to execute when the document is opened; these startup commands can contain malicious code. Although, strictly speaking, a virus can activate and spread only when a program executes, in fact, data files are acted on by programs. Clever virus writers have been able to make data control files that cause programs to do many things, including pass along copies of the virus to other data files.

- *Viruses spread only on disks or only through e-mail*. **False**. File-sharing is often done as one user provides a copy of a file to another user by writing the file on a transportable disk. However, any means of electronic file transfer will work. A file can be placed in a network's library or posted on a bulletin board. It can be attached to an e-mail message or made available for download from a web site. Any mechanism for sharing filesof programs, data, documents, and so forthcan be used to transfer a virus.

- *Viruses cannot remain in memory after a complete power off/power on reboot. **True, but . . .*** If a virus is resident in memory, the virus is lost when the memory loses power. That is, computer memory (RAM) is volatile, so all contents are deleted when power is lost.[2] However, viruses written to disk certainly can remain through a reboot cycle. Thus, you can receive a virus infection, the virus can be written to disk (or to network storage), you can turn the machine off and back on, and the virus can be reactivated during the reboot. Boot sector viruses gain control when a machine reboots (whether it is a hardware or software reboot), so a boot sector virus may remain through a reboot cycle because it activates immediately when a reboot has completed.

  [2] Some very low-evel hardware settings (for example, the size of disk installed) are retained in memory called "nonvolatile RAM," but these locations are not directly accessible by programs and are written only by programs run from read-only memory (ROM) during hardware initialization. Thus, they are highly immune to virus attack.

- *Viruses cannot infect hardware. **True***. Viruses can infect only things they can modify; memory, executable files, and data are the primary targets. If hardware contains writeable storage (so-called firmware) that can be accessed under program control, that storage *is* subject to virus attack. There have been a few instances of firmware viruses. Because a virus can control hardware that is subject to program control, it may seem as if a hardware device has been infected by a virus, but it is really the software driving the hardware that has been infected. Viruses can also exercise hardware in any way a program can. Thus, for example, a virus could cause a disk to loop incessantly, moving to the innermost track then the outermost and back again to the innermost.

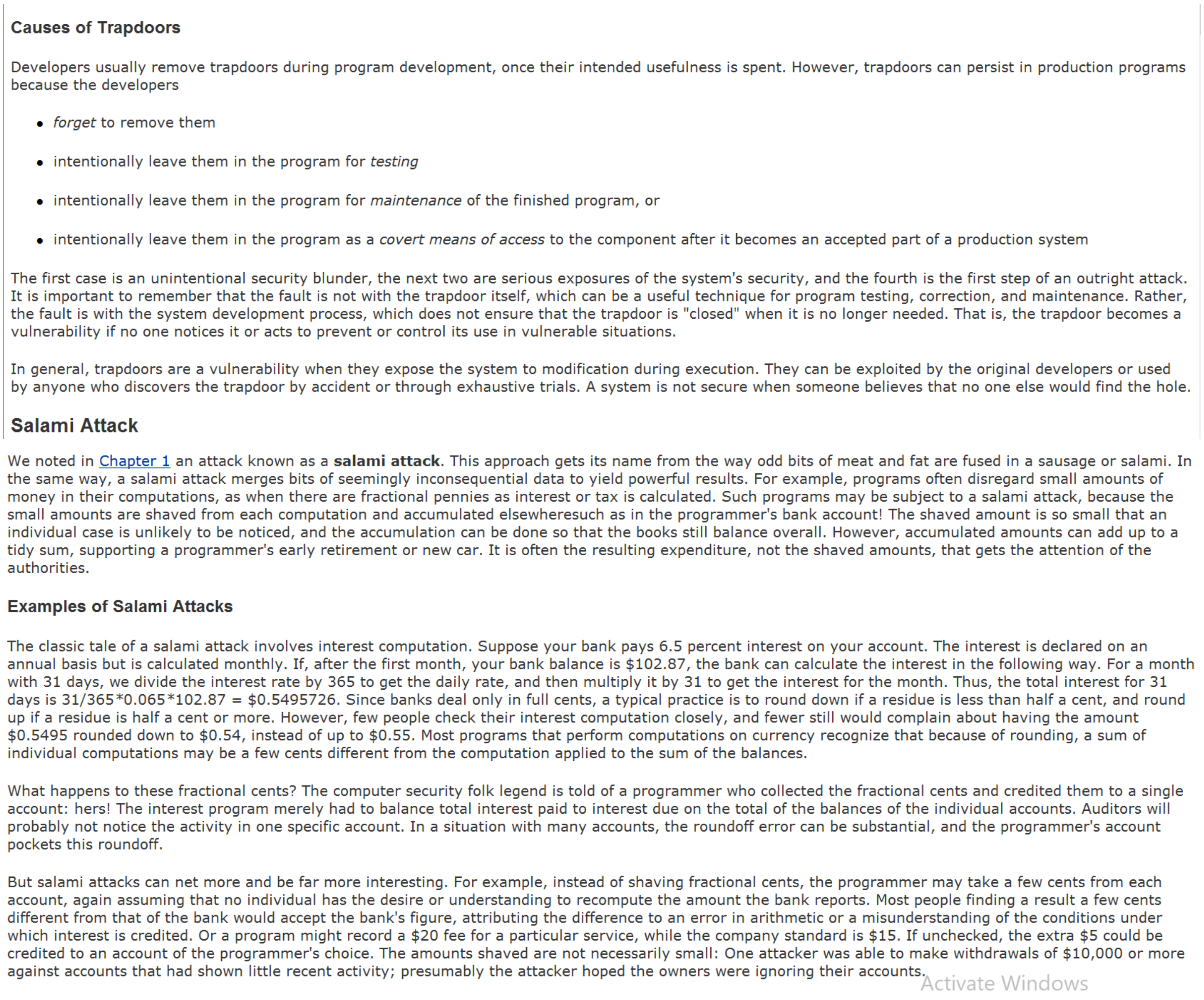
- *Viruses can be malevolent, benign, or benevolent. **True.*** Not all viruses are bad. For example, a virus might locate uninfected programs, compress them so that they occupy less memory, and insert a copy of a routine that decompresses the program when its execution begins. At the same time, the virus is spreading the compression function to other programs. This virus could substantially reduce the amount of storage required for stored programs, possibly by up to 50 percent. However, the compression would be done at the request of the virus, not at the request, or even knowledge, of the program owner.

Trapdoors

A **trapdoor** is an undocumented entry point to a module. Developers insert trapdoors during code development, perhaps to test the module, to provide "hooks" by which to connect future modifications or enhancements, or to allow access if the module should fail in the future. In addition to these legitimate uses, trapdoors can allow a programmer access to a program once it is placed in production.

Examples of Trapdoors

Because computing systems are complex structures, programmers usually develop and test systems in a methodical, organized, modular manner, taking advantage of the way the system is composed of modules or components. Often, programmers first test each small component of the system separate from the other components, in a step called **unit testing**, to ensure that the component works correctly by itself. Then, developers test components together during **integration testing**, to see how they function as they send messages and data from one to the other. Rather than paste all the components together in a "big bang" approach, the testers group logical clusters of a few components, and each cluster is tested in a way that allows testers to control and understand what might make a component or its interface fail. (For a more detailed look at testing, see Pfleeger and Atlee [PFL06a].)



Causes of Trapdoors

Developers usually remove trapdoors during program development, once their intended usefulness is spent. However, trapdoors can persist in production programs because the developers

- *forget* to remove them
- intentionally leave them in the program for *testing*
- intentionally leave them in the program for *maintenance* of the finished program, or
- intentionally leave them in the program as a *covert means of access* to the component after it becomes an accepted part of a production system

The first case is an unintentional security blunder, the next two are serious exposures of the system's security, and the fourth is the first step of an outright attack. It is important to remember that the fault is not with the trapdoor itself, which can be a useful technique for program testing, correction, and maintenance. Rather, the fault is with the system development process, which does not ensure that the trapdoor is "closed" when it is no longer needed. That is, the trapdoor becomes a vulnerability if no one notices it or acts to prevent or control its use in vulnerable situations.

In general, trapdoors are a vulnerability when they expose the system to modification during execution. They can be exploited by the original developers or used by anyone who discovers the trapdoor by accident or through exhaustive trials. A system is not secure when someone believes that no one else would find the hole.

Salami Attack

We noted in Chapter 1 an attack known as a **salami attack**. This approach gets its name from the way odd bits of meat and fat are fused in a sausage or salami. In the same way, a salami attack merges bits of seemingly inconsequential data to yield powerful results. For example, programs often disregard small amounts of money in their computations, as when there are fractional pennies as interest or tax is calculated. Such programs may be subject to a salami attack, because the small amounts are shaved from each computation and accumulated elsewheresuch as in the programmer's bank account! The shaved amount is so small that an individual case is unlikely to be noticed, and the accumulation can be done so that the books still balance overall. However, accumulated amounts can add up to a tidy sum, supporting a programmer's early retirement or new car. It is often the resulting expenditure, not the shaved amounts, that gets the attention of the authorities.

Examples of Salami Attacks

The classic tale of a salami attack involves interest computation. Suppose your bank pays 6.5 percent interest on your account. The interest is declared on an annual basis but is calculated monthly. If, after the first month, your bank balance is $102.87, the bank can calculate the interest in the following way. For a month with 31 days, we divide the interest rate by 365 to get the daily rate, and then multiply it by 31 to get the interest for the month. Thus, the total interest for 31 days is 31/365*0.065*102.87 = $0.5495726. Since banks deal only in full cents, a typical practice is to round down if a residue is less than half a cent, and round up if a residue is half a cent or more. However, few people check their interest computation closely, and fewer still would complain about having the amount $0.5495 rounded down to $0.54, instead of up to $0.55. Most programs that perform computations on currency recognize that because of rounding, a sum of individual computations may be a few cents different from the computation applied to the sum of the balances.

What happens to these fractional cents? The computer security folk legend is told of a programmer who collected the fractional cents and credited them to a single account: hers! The interest program merely had to balance total interest paid to interest due on the total of the balances of the individual accounts. Auditors will probably not notice the activity in one specific account. In a situation with many accounts, the roundoff error can be substantial, and the programmer's account pockets this roundoff.

But salami attacks can net more and be far more interesting. For example, instead of shaving fractional cents, the programmer may take a few cents from each account, again assuming that no individual has the desire or understanding to recompute the amount the bank reports. Most people finding a result a few cents different from that of the bank would accept the bank's figure, attributing the difference to an error in arithmetic or a misunderstanding of the conditions under which interest is credited. Or a program might record a $20 fee for a particular service, while the company standard is $15. If unchecked, the extra $5 could be credited to an account of the programmer's choice. The amounts shaved are not necessarily small: One attacker was able to make withdrawals of $10,000 or more against accounts that had shown little recent activity; presumably the attacker hoped the owners were ignoring their accounts.

Rootkits:

A **rootkit** is a malicious software that allows an unauthorized user to have privileged access to a computer and to restricted areas of its software. A **rootkit** may contain a number of malicious tools such as keyloggers, banking credential stealers, password stealers, antivirus disablers, and bots for DDoS attacks.

A rootkit is a clandestine computer program designed to provide continued privileged access to a computer while actively hiding its presence. The term rootkit is a connection of the two words "root" and "kit." Originally, a rootkit was a collection of tools that enabled administrator-level access to a computer or network. Root refers to the Admin account on Unix and Linux systems, and kit refers to the software components that implement the tool. Today rootkits are generally associated with malware – such as Trojans, worms, viruses – that conceal their existence and actions from users and other system processes.

What Can a Rootkit Do?

A rootkit allows someone to maintain command and control over a computer without the computer user/owner knowing about it. Once a rootkit has been installed, the controller of the rootkit has the ability to remotely execute files and change system configurations on the host machine. A rootkit on an infected computer can also access log files and spy on the legitimate computer owner's usage.

Rootkit Detection
It is difficult to detect rootkits. There are no commercial products available that can find and remove all known and unknown rootkits. There are various ways to look for a rootkit on an infected machine. Detection methods include behavioral-based methods (e.g., looking for strange behavior on a computer system), signature scanning and memory dump analysis. Often, the only option to remove a rootkit is to completely rebuild the compromised system.

Rootkit Protection
Many rootkits penetrate computer systems by piggybacking with software you trust or with a virus. You can safeguard your system from rootkits by ensuring it is kept patched against known vulnerabilities. This includes patches of your OS, applications and up-to-date virus definitions. Don't accept files or open email file attachments from unknown sources. Be careful when installing software and carefully read the end-user license agreements.

Bots:

A bot -- short for "robot" and also called an internet bot -- is a computer program that operates as an agent for a user or other program, or to simulate a human activity. Bots are normally used to automate certain tasks, meaning they can run without specific instructions from humans.

An organization or individual can use a bot to replace a repetitive task that a human would otherwise have to perform. Bots are also much faster at these tasks than humans.

How do bots work?

Normally, bots will operate over a network. Bots that can communicate with one another will use internet-based services to do so -- such as instant messaging, In general, more than half of internet traffic is bots that interact with web pages, talk with users, scan for content and perform other tasks.

Bots are made from sets of algorithms which aid them in their designated tasks. Tasks bots can normally handle include conversing with a human -- which attempts to mimic human behaviors -- or gathering content from other websites. There are plenty of different types of bots designed differently to accomplish a wide variety of tasks.

As an example, a chatbot will operate on one of multiple methods of operation. A rule-based chatbot will interact with people by giving pre-defined prompts for the individual to select. An intellectually independent chatbot will make use of machine learning to learn from human inputs as well as watching out for known keywords. AI chatbots are a combination of rule-based and intellectually independent chatbots. Chatbots may also use pattern matching, natural language processing (NLP) and natural language generation (NLG) tools.

Organizations or individuals that make use of bots can also use bot management software, which includes software tools that aid in managing bots and protecting against malicious bots. Bot managers can be included as part of a web app security platform. A bot manager can be used to allow the use of some bots and block the use of others that might cause harm to a system. To do this, a bot manager will classify any incoming requests by humans and good bots and known malicious and unknown bots. Any suspect bot traffic is then directed away from a site by the bot manager. Some basic bot management feature sets include IP rate limiting and CAPTCHAs. IP rate limiting will limit the number of same-address-requests, while CAPTCHAs are used as a sort of puzzle to differentiate bots from humans.

Buffer Overflows

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, 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. However, if the statement were

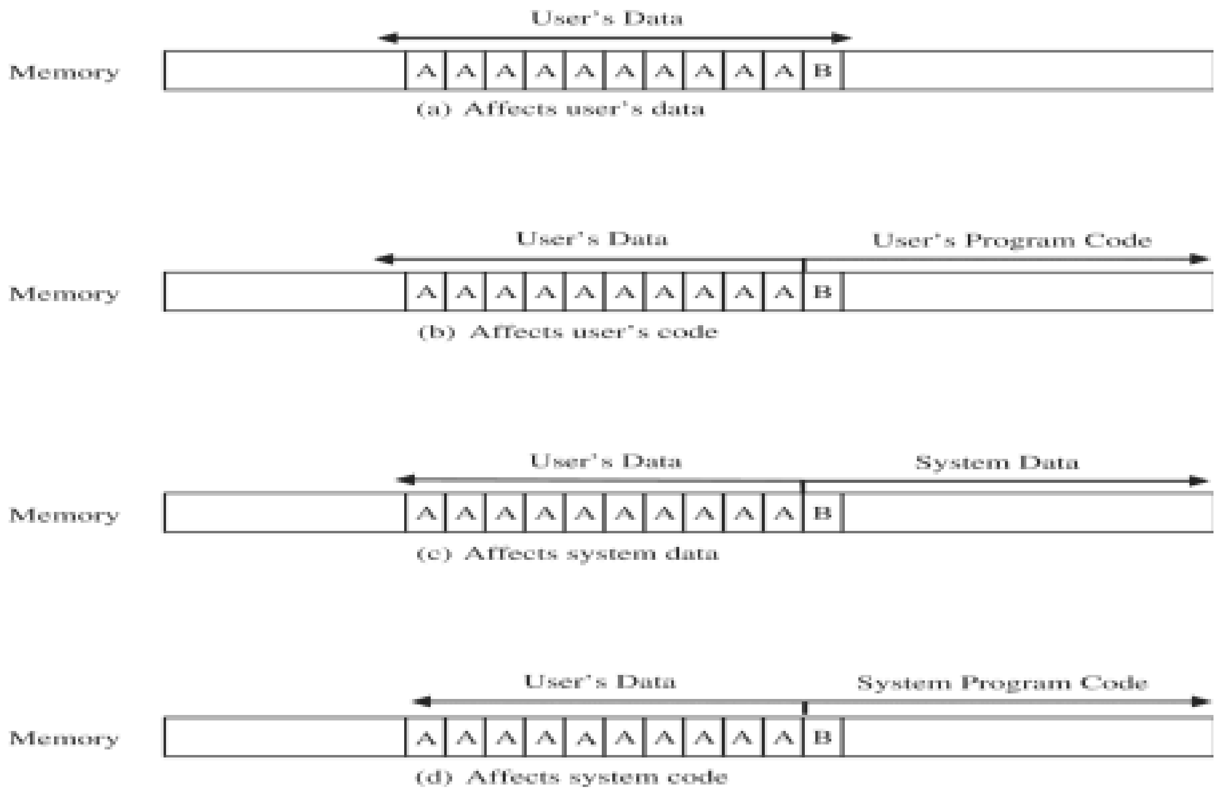
```
sample[i] = 'B';
```

we could not identify the problem until i was set during execution to a too-big subscript. It would be useful if, during execution, the system produced an error message warning of a subscript out of bounds. 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 the 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.

Let us examine this problem more closely. It is important 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, there are four cases to consider in deciding where the 'B' goes, as shown in the figure below 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.



Format String Vulnerability and Prevention with Example:

A format string is an ASCII string that contains text and format parameters.

Example:

```
// A statement with format string
printf("my name is : %s\n", "CSS");


// Output
// My name is : CSS
```

Format string vulnerabilities are a class of bug that take advantage of an easily avoidable programmer error. If the programmer passes an attacker-controlled buffer as an argument to a printf (or any of the related functions, including sprintf, fprintf, etc), the attacker can perform writes to arbitrary memory addresses. The following program contains such an error:

filter_none

edit

play_arrow

brightness_4

```
// A simple C program with format
// string vulnerability
#include<stdio.h>

int main(int argc, char** argv)
{
char buffer[100];
strncpy(buffer, argv[1], 100);

// We are passing command line
// argument to printf
printf(buffer);

return 0;
}
```

Since printf has a variable number of arguments, it must use the format string to determine the number of arguments. In the case above, the attacker can pass the string "%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p" and fool the printf into thinking it has 15 arguments. It will naively print the next 15 addresses on the stack, thinking they are its arguments:

```
$ ./a.out "%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"
0xffffdddd 0x64 0xf7ec1289 0xffffdbdf 0xffffdbde (nil) 0xffffdcc4 0xffffdc64 (nil)
0x25207025 0x70252070 0x20702520 0x25207025 0x70252070 0x20702520
```

At about 10 arguments up the stack, we can see a repeating pattern of 0x252070 – those are our %ps on the stack! We start our string with AAAA to see this more explicitly:

```
$ ./a.out "AAAA%p %p %p %p %p %p %p %p %p %p"
AAAA0xffffdde8 0x64 0xf7ec1289 0xffffdbef 0xffffdbee (nil) 0xffffdcd4 0xffffdc74 (nil)
0x41414141
```

The 0x41414141 is the hex representation of AAAA. We now have a way to pass an arbitrary value (in this case, we're passing 0x41414141) as an argument to printf. At this point we will take advantage of another format string feature: in a format specifier, we can also select a specific argument. For example, printf("%2$x", 1, 2, 3) will print 2. In general, we can do

printf("%$x") to select an arbitrary argument to printf. In our case, we see that 0x41414141 is the 10th argument to printf, so we can simplify our string1:

```
$ ./a.out 'AAAA%10$p'
AAAA0x41414141
```

Preventing Format String Vulnerabilities:

Always specify a format string as part of program, not as an input. Most format string vulnerabilities are solved by specifying "%s" as format string and not using the data string as format string.

If possible, make the format string a constant. Extract all the variable parts as other arguments to the call. Difficult to do with some internationalization libraries

SQL INJECTION

SQL injection, also known as SQLI, is a common attack vector that uses malicious SQL code for backend database manipulation to access information that was not intended to be displayed. This information may include any number of items, including sensitive company data, user lists or private customer details.

A successful SQL attack may result in the unauthorized viewing of user lists, the deletion of entire tables and, in certain cases, the attacker gaining administrative rights to a database, all of which are highly detrimental to a business.

SQL is a standardized language used to access and manipulate databases to build customizable data views for each user. SQL queries are used to execute commands, such as data retrieval, updates and record removal. Different SQL elements implement these tasks, e.g., queries using the SELECT statement to retrieve data, based on user-provided parameters.

A typical eStore's SQL database query may look like the following:

```
SELECT ItemName, ItemDescription
FROM Item
WHERE ItemNumber = ItemNumber
```

From this, the web application builds a string query that is sent to the database as a single SQL statement:

```
sql_query= "
SELECT ItemName, ItemDescription
FROM Item
WHERE ItemNumber = " & Request.QueryString("ItemID")
```

A user-provided input http://www.estore.com/items/items.asp?itemid=999 can then generates the following SQL query:

```
SELECT ItemName, ItemDescription
FROM Item
WHERE ItemNumber = 999
```

As you can gather from the syntax, this query provides the name and description for item number 999.

SQL INJECTION EXAMPLE

An attacker wishing to execute SQL injection manipulates a standard SQL query to exploit non-validated input vulnerabilities in a database. There are many ways that this attack vector can be executed, several of which will be shown here to provide you with a general idea about how SQLI works.

For example, the above-mentioned input, which pulls information for a specific product, can be altered to read http://www.estore.com/items/items.asp?itemid=999 or 1=1.

As a result, the corresponding SQL query looks like this:

SELECT ItemName, ItemDescription

FROM Items

WHERE ItemNumber = 999 OR 1=1

And since the statement 1 = 1 is always true, the query returns all of the product names and descriptions in the database, even those they you may not be eligible to access.

Attackers are also able to take advantage of incorrectly filtered characters to alter SQL commands, including using a semicolon to separate two fields.

For example, this input http://www.estore.com/items/iteams.asp?itemid=999; DROP TABLE Userswould generate the following SQL query:

SELECT ItemName, ItemDescription

FROM Items

WHERE ItemNumber = 999; DROP TABLE USERS

As a result, the entire user database could be deleted.

Another way SQL queries can be manipulated is with a UNION SELECT statement. This combines two unrelated SELECT queries to retrieve data from different database tables.

For example, the input http://www.estore.com/items/items.asp?itemid=999 UNION SELECT user-name, password FROM USERS produces the following SQL query:

SELECT ItemName, ItemDescription

FROM Items

WHERE ItemID = '999' UNION SELECT Username, Password FROM Users;

Using the UNION SELECT statement, this query combines the request for item 999’s name and description with another that pulls names and passwords for every user in the database.

CROSS-SITE SCRIPTING (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user’s browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

References:

1. Behrouz Forouzan, “Cryptography & Network Security”
2. Charles P. Pfleeger,” Security in Computing “, Pearson Education
3. https://www.veracode.com/security/rootkit
4. https://whatis.techtarget.com/definition/bot-robot#:~:text=A%20bot%20%2D%2D%20short%20for,without%20specific%20instructions%20from%20humans.

Short Answer Type Questions:

1. What is SQL Injection?
2. What do you mean by Computer Virus?
3. Compare Computer Virus & Worm.
4. Define Trojans, Logic Bombs, Bots & Rootkits.
5. What are Trapdoors?
6. What are the different types of Computer Virus?
7. What is Virus Signature?

Long Answer Type Questions:

1. What is Malware?
2. What do you mean by Malicious Codes?
3. What are the kinds of Malicious Codes?
4. What do you mean by non-malicious programming errors?
5. What is a Boot Sector Virus?
6. How can Computer Programs be made secure?
7. How can SQL Injection attack be prevented, mitigated & controlled?
8. What are the various Software Vulnerabilities? Explain in detail.
9. How are Bots different from Rootkits?
10. Explain non-malicious programming errors in detail.
11. What is Buffer-Overflow Attack. Explain in detail.
12. How can a computer system be protected against Computer Virus?
13. How do Computer Viruses propagate & attach themselves to programs in the computing system?

--***---