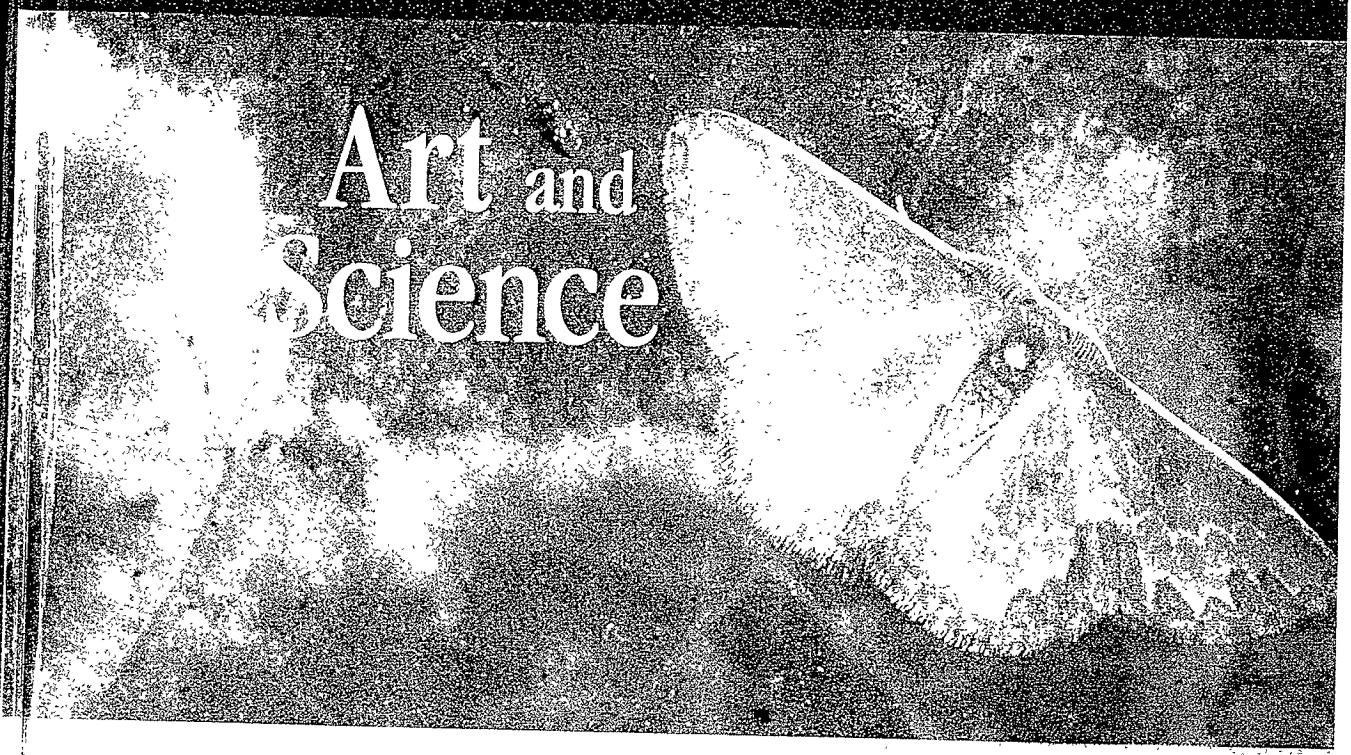
COMPUTER SECURITY

Art and Science



Matt Bishop

Computer Security

Art and Science

Matt Bishop

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Bishop, Matt.
Computer security : art and science / Matt Bishop.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-44099-7 (alk. paper)
1. Computer security. I. Title.

QA76.9.A25 B56 2002
005.8—dc21 2002026219

Text printed on recycled and acid-free paper

ISBN 0201440997

4 5 6 7 8 9 CRW 06 05 04 03

4th Printing November 2003

Chapter 22
Malicious Logic

TITUS ANDRONICUS: Ah!, wherefore dost thou urge the name of hands?
To bid Aeneas tell the tale twice o'er,
How Troy was burnt and he made miserable?
—*The Tragedy of Titus Andronicus*, III, ii, 26–28.

Computer viruses, worms, and Trojan horses are effective tools with which to attack computer systems. They assume an authorized user's identity. This makes most traditional access controls useless. This chapter presents several types of malicious logic, focusing on Trojan horses and computer viruses, and discusses defenses.

22.1 Introduction

Odysseus, of Trojan War fame, found the most effective way to breach a hitherto-impregnable fortress was to have people inside bring him in without knowing they were doing so [482, 1016]. The same approach works for computer systems.

Definition 22–1. *Malicious logic* is a set of instructions that cause a site's security policy to be violated.

EXAMPLE: The following UNIX script is named *ls* and is placed in a directory.

```
cp /bin/sh /tmp/.xxsh
chmod o+s,w+x /tmp/.xxsh
rm ./ls
ls $*
```

It creates a copy of the UNIX shell that is setuid to the user executing this program (see Section 14.3). This program is deleted, and then the correct *ls* command is executed. On most systems, it is against policy to trick someone into creating a shell that is setuid to themselves. If someone is tricked into executing this script, a violation of the (implicit) security policy occurs. This script is an example of malicious logic.

22.2 Trojan Horses

A critical observation is the notion of "tricked." Suppose the user *root* executed this script unintentionally (for example, by typing "ls" in the directory containing this file). This would be a violation of the security policy. However, if *root* deliberately typed

```
cp /bin/sh /tmp/.xxsh
chmod o+s,w+x /tmp/.xxsh
```

the security policy would not be violated. This illustrates a crucial component of the problems with malicious logic. The system cannot determine whether the instructions being executed by a process are known to the user or are a set of instructions that the user does not intend to execute. The next definition makes this distinction explicit.

Definition 22–2. A *Trojan horse* is a program with an overt (documented or known) effect and a *covert* (undocumented or unexpected) effect.

EXAMPLE: In the preceding example, the overt purpose is to list the files in a directory. The covert purpose is to create a shell that is setuid to the user executing the script. Hence, this program is a Trojan horse.

Dan Edwards was the first to use this term [26]. Trojan horses are often used in conjunction with other tools to attack systems.

EXAMPLE: The NetBus program allows an attacker to control a Windows NT workstation remotely. The attacker can intercept keystrokes or mouse motions, upload and download files, and act as a system administrator would act. In order for this program to work, the victim Windows NT system must have a server with which the NetBus program can communicate. This requires someone on the victim's system to load and execute a small program that runs the server.

This small program was placed in several small game programs as well as in some other "fun" programs, which could be distributed to Web sites where unsuspecting users would be likely to download them.

Trojan horses can make copies of themselves. One of the earliest Trojan horses was a version of the game *animal*. When this game was played, it created an extra copy of itself. These copies spread, taking up much room. The program was modified to delete one copy of the earlier version and create two copies of the modified program. Because it spread even more rapidly than the earlier version, the modified version of *animal* soon completely supplanted the earlier version. After a preset date, each copy of the later version deleted itself after it was played [290].

Definition 22–3. A *propagating Trojan horse* (also called a *replicating Trojan horse*) is a Trojan horse that creates a copy of itself.

Karger and Schell [552], and later Thompson [995], examined detection of Trojan horses. They constructed a Trojan horse that propagated itself slowly and in a manner that was difficult to detect. The central idea is that the Trojan horse modifies the compiler to insert itself into specific programs, including future versions of the compiler itself.

EXAMPLE: Thompson [995] added a Trojan horse to the *login* program. When a user logged in, the Trojan horse would accept a fixed password as well as the user's normal password. However, anyone reading the source code for the *login* program would instantly detect this Trojan horse. To obscure it, Thompson had the compiler check the program being compiled. If that program was *login*, the compiler added the code to use the fixed password. Now, no code needed to be added to the *login* program. Thus, an analyst inspecting the *login* program source code would see nothing amiss. If the analyst compiled the *login* program from that source, she would believe the executable to be uncorrupted.

The extra code is visible in the compiler source. To eliminate this problem, Thompson modified the compiler. This second version checked to see if the compiler (actually, the C preprocessor) was being recompiled. If so, the code to modify the compiler so as to include both this Trojan horse and the *login* Trojan horse code would be inserted. He compiled the second version of the compiler and installed the executable. He then replaced the corrupted source with the original version of the compiler. As with the *login* program, inspection of the source code would reveal nothing amiss, but compiling and installing the compiler would insert the two Trojan horses.

Thompson took special pains to ensure that the second version of the compiler was never released. It remained on the system for a considerable time before someone overwrote the executable with a new version from a different system [839]. Thompson's point[1] was that "no amount of source-level verification or scrutiny will protect you from using untrusted code," a point to be reiterated later.

22.3 Computer Viruses

This type of Trojan horse propagates itself only as specific programs (in the preceding example, the compiler and the *login* program). When the Trojan horse can propagate freely and insert a copy of itself into another file, it becomes a computer virus.

[1] See [995], p. 763.

Definition 22–4. A *computer virus* is a program that inserts itself into one or more files and then performs some (possibly null) action.

The first phase, in which the virus inserts itself into a file, is called the *insertion phase*. The second phase, in which it performs some action, is called the *execution phase*. The following pseudocode fragment shows how a simple computer virus works.

```
beginvirus:
   if spread-condition then begin
      for some set of target files do begin
         if target is not infected then begin
            determine where to place virus instructions
            copy instructions from beginvirus to endvirus
               into target
            alter target to execute added instructions
         end;
      end;
   end;
   perform some action(s)
   goto beginning of infected program
endvirus:
```

As this code indicates, the insertion phase must be present but need not always be executed. For example, the Lehigh virus [470] would check for an uninfected boot file (the *spread-condition* mentioned in the pseudocode) and, if one was found, would infect that file (the *set of target files*). Then it would increment a counter and test to see if the counter was at 4. If so, it would erase the disk. These operations were the *action(s)*.

Authorities differ on whether or not a computer virus is a type of Trojan horse. Most equate the purpose of the infected program with the overt action and consider the insertion and execution phases to be the covert action. To them, a computer virus is a Trojan horse [310, 516]. However, others argue that a computer virus has *no* covert purpose. Its overt purpose is to infect and execute. To these authorities, it is not a Trojan horse [204, 739]. In some sense this disagreement is semantic. In any case, defenses against a Trojan horse inhibit computer viruses.

According to Ferbrache [346], programmers wrote the first computer viruses on Apple II computers. A virus developed for research purposes in 1980 wrote itself to the disk boot sectors when the catalogue command was executed. Another one infected many copies of the game "Congo," which stopped working. Friends of its author had released it before it was fully debugged. The author rewrote it to replace existing copies of itself with the fully debugged version. Released into the wild, it rapidly supplanted the buggy copies.

In 1983, Fred Cohen was a graduate student at the University of Southern California. During a seminar on computer security, he described a type of Trojan horse that the teacher, Len Adleman, christened a computer virus [205]. To demonstrate the

effectiveness of the proposed attack, Cohen designed a computer virus to acquire privileges on a VAX-11/750 running the UNIX operating system. He obtained all system rights within half an hour on the average, the longest time being an hour and the shortest being less than 5 minutes. Because the virus did not degrade response time noticeably, most users never knew the system was under attack.

In 1984, an experiment involving a UNIVAC 1108 showed that viruses could spread throughout that system, too. Unlike the UNIX system, the UNIVAC partially implemented the Bell-LaPadula Model, using mandatory protection mechanisms.[2] Cohen's experiments indicated that the security mechanisms of systems that did not inhibit writing using mandatory access controls did little if anything to inhibit computer virus propagation [204, 205].

The Brain (or Pakistani) virus, written for IBM PCs, is thought to have been created in early 1986 [346] but was first reported in the United States in October 1987. It alters the boot sectors of floppy disks, possibly corrupting files in the process. It also spreads to any uninfected floppy disks inserted into the system. Since then, numerous variations of this virus have been reported [471].

In 1987, computer viruses infected Macintosh, Amiga, and other computers. The MacMag Peace virus would print a "universal message of peace" on March 2, 1988, and then delete itself [355]. This computer virus infected copies of the Aldus FreeHand program, which were recalled by their manufacturer [346].

In 1987, Tom Duff experimented on UNIX systems with a small virus that copied itself into executable files. The virus was not particularly virulent, but when Duff placed 48 infected programs on the most heavily used machine in the computing center, the virus spread to 46 different systems and infected 466 files, including at least one system program on each computer system, within 8 days. Duff did not violate the security mechanisms in any way when he seeded the original 48 programs [312]. He wrote another virus in a Bourne shell script. It could attach itself to any UNIX program. This demonstrated that computer viruses are not intrinsically machine-dependent and can spread to systems of varying architectures.

In 1989, Dr. Harold Joseph Highland developed a virus for Lotus 1-2-3 [471]. This virus, stored as a set of commands for that spreadsheet, was loaded automatically when a file was opened. Because the virus was intended for a demonstration only, it changed the value in a specific row and column and then spread to other files. This demonstrated that macros for office-type programs on personal computers could contain viruses.

Several types of computer viruses have been identified.

22.3.1 Boot Sector Infectors

The *boot sector* is the part of a disk used to bootstrap the system or mount a disk. Code in that sector is executed when the system "sees" the disk for the first time.

[2] Specifically, it implemented the simple security condition but not the *-property [516].

When the system boots, or the disk is mounted, any virus in that sector is executed. (The actual boot code is moved to another place, possibly another sector.)

Definition 22–5. A *boot sector infector* is a virus that inserts itself into the boot sector of a disk.

EXAMPLE: The Brain virus for the IBM PC is a boot sector infector. When the system boots from an infected disk, the virus is in the boot sector and is loaded. It moves the disk interrupt vector (location 13H or 19) to an alternative interrupt vector (location 6DH or 109) and sets the disk interrupt vector location to invoke the Brain virus now in memory. It then loads the original boot sector and continues the boot.

Whenever the user reads a floppy, the interrupt at location 13H is invoked. The Brain virus checks for the signature 1234H in the word at location 4. If the signature is present, control is transferred to the interrupt vector at location 6DH so that a normal read can proceed. Otherwise, the virus infects the disk.

To do this, it first allocates to itself three contiguous clusters (of two contiguous sectors each). The virus then copies the original boot sector to the first of the six contiguous sectors and puts copies of itself into the boot sector and the remaining five sectors.

If there are no unused clusters, the virus will not infect the disk. If it finds only one unused cluster, it will simply overwrite the next two. This accounts for the sometimes destructive nature of the Brain virus.

22.3.2 Executable Infectors

Definition 22–6. An *executable infector* is a virus that infects executable programs.

The PC variety of executable infectors are called COM or EXE viruses because they infect programs with those extensions. Figure 22–1 illustrates how infection can

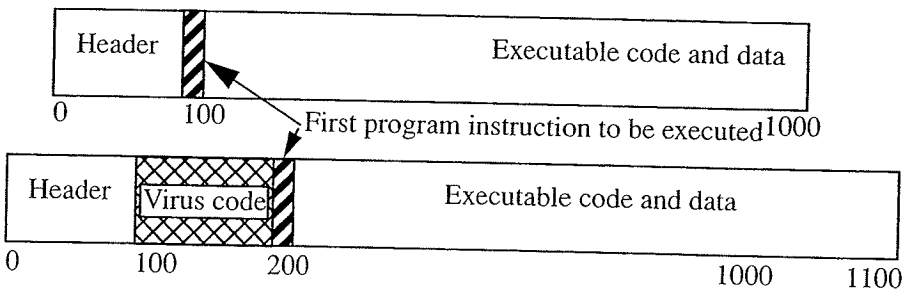


Figure 22–1 How an executable infector works. It inserts itself into the program so that the virus code will be executed before the application code. In this example, the virus is 100 words long and prepends itself to the executable code.

occur. The virus can prepend itself to the executable (as shown in the figure) or append itself.

EXAMPLE: The Jerusalem virus (also called the Israeli virus) is triggered when an infected program is executed. The virus first puts the value 0E0H into register *ax* and invokes the DOS service interrupt (21H). If on return the high eight bits of register *ax* contain 03H, the virus is already resident on the system and the executing version quits, invoking the original program. Otherwise, the virus sets itself up to respond to traps to the DOS service interrupt vector.

The Jerusalem virus then checks the date. If the year is 1987, it does nothing. Otherwise, if it is not a Friday and not the 13th (of any month), it sets itself up to respond to clock interrupts (but it will not infect on clock calls). It then loads and executes the file originally executed. When that file finishes, the virus puts itself in memory. It then responds to calls to the DOS service interrupt.

If it is a Friday and the 13th (of any month), and the year is not 1987, the virus sets a flag in memory to be destructive. This flag means that the virus will delete files instead of infecting them.

Once in memory, the virus checks all calls to the DOS service interrupt, looking for those asking that files be executed (function 4B00H). When this happens, the virus checks the name of the file. If it is COMMAND.COM, the virus does nothing. If the memory flag is set to be destructive, the file is deleted. Otherwise, the virus checks the last five bytes of the file. If they are the string "MsDos," the file is infected.[3] If they are not, the virus checks the last character of the file name. If it is "M," the virus assumes that a .COM file is being executed and infects it; if it is "E," the virus assumes that a .EXE file is being executed and infects it. The file's attributes, especially the date and time of modification, are left unchanged.

22.3.3 Multipartite Viruses

Definition 22–7. A *multipartite virus* is one that can infect either boot sectors or applications.

Such a virus typically has two parts, one for each type. When it infects an executable, it acts as an executable infector; when it infects a boot sector, it works as a boot sector infector.

[3] According to Compulit, as cited in [471], "[t]he author of the virus apparently forgot to set the signature during .EXE file infection. This will cause multiple infections of .EXE files" (p. 47). Analysts at the Hebrew University of Jerusalem found that the size of a .COM file increased only one time, but the size of a .EXE file increased every time the file was executed.

22.3.4 TSR Viruses

Definition 22–8. A *terminate and stay resident* (TSR) virus is one that stays active (resident) in memory after the application (or bootstrapping, or disk mounting) has terminated.

TSR viruses can be boot sector infectors or executable infectors. Both the Brain and Jerusalem viruses are TSR viruses.

Viruses that are not TSR execute only when the host application is executed (or the disk containing the infected boot sector is mounted). An example is the Encroacher virus, which appends itself to the ends of executables.

22.3.5 Stealth Viruses

Definition 22–9. *Stealth* viruses are viruses that conceal the infection of files.

These viruses intercept calls to the operating system that access files. If the call is to obtain file attributes, the original attributes of the file are returned. If the call is to read the file, the file is disinfected as its data is returned. But if the call is to execute the file, the infected file is executed.

EXAMPLE: The Stealth virus (also called the IDF virus or the 4096 virus) is an executable infector. It modifies the DOS service interrupt handler (rather than the interrupt vector; this way, checking the values in the interrupt vector will not reveal the presence of the virus). If the request is for the length of the file, the length of the *uninfected* file is returned. If the request is to open the file, the file is temporarily disinfected; it is reinfected on closing. The Stealth virus also changes the time of last modification of the file in the file allocation table to indicate that the file is infected.

22.3.6 Encrypted Viruses

Computer virus detectors often look for known sequences of code to identify computer viruses (see Section 22.7.4). To conceal these sequences, some viruses encipher most of the virus code, leaving only a small decryption routine and a random cryptographic key in the clear. Figure 22–2 summarizes this technique.

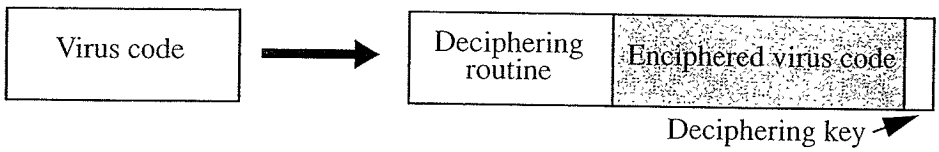


Figure 22–2 An encrypted virus. The ordinary virus code is at the left. The encrypted virus, plus encapsulating decryption information, is at the right.

Definition 22–10. An *encrypted* virus is one that enciphers all of the virus code except for a small decryption routine.

EXAMPLE: Ferbrache[4] cites the following as the decryption code in the 1260 virus. It uses two keys, stored in *k1* and *k2*. The virus code itself begins at the location *sov* and ends at the location *eov*. The pseudocode is as follows.

```
(* initialize the registers with the keys *)
rA ← k1;
rB ← k2;
(* initialize rC with the message *)
rC ← sov;
(* the encipherment loop *)
while (rC != eov) do begin
   (* encipher the byte of the message *)
   (*rC) ← (*rC) xor rA xor rB;
   (* advance all the counters *)
   rC ← rC + 1;
   rA ← rA + 1;
end
```

The dual keys and the shifting of the first key prevent a simple *xor*'ing from uncovering the deciphered virus.

22.3.7 Polymorphic Viruses

Definition 22–11. *A polymorphic* virus is a virus that changes its form each time it inserts itself into another program.

Consider an encrypted virus. The body of the virus varies depending on the key chosen, so detecting known sequences of instructions will not detect the virus. However, the decryption algorithm can be detected. Polymorphic viruses were designed to prevent this. They change the instructions in the virus to something equivalent but different. In particular, the deciphering code is the segment of the virus that is changed. In some sense, they are successors to the encrypting viruses and are often used in conjunction with them.

Consider polymorphism at the instruction level. All of the instructions

```
add 0 to operand
or 1 with operand
no operation
subtract 0 from operand
```

[4] See [346], p. 75.

have exactly the same effect, but they are represented as different bit patterns on most architectures. A polymorphic virus would insert these instructions into the deciphering segment of code.

EXAMPLE: A polymorphic version of the 1260 computer virus might look like the following. (The lines marked "random line" do nothing and are changed whenever the virus replicates.)

```
(* initialize the registers with the keys *)
rA ← k1;
rD ← rD + 1;(* random line *)
rB ← k2;
(* initialize rC with the message *)
rC ← sov;
rC ← rC + 1;(* random line *)
(* the encipherment loop *)
while (rC != eov) do begin
    rC ← rC - 1;(* random line X *)
    (* encipher the byte of the message *)
    (*rC) ← (*rC) xor rA xor rB;
    (* advance all the counters *)
    rC ← rC + 2;(* counter incremented ... *)
    (* to handle random line X *)
    rD ← rD + 1;(* random line *)
    rA ← rA + 1;
end
while (rC != sov) do begin(* random line *)
    rD ← rD - 1;(* random line *)
end(* random line *)
```

Examination shows that these instructions have the same effect as the four instructions listed above.

The production of polymorphic viruses at the instruction level has been automated. At least two tool kits, the Mutation Engine (MtE) and the Trident Polymorphic Engine (TPE), were available in 1992 [1064].

Polymorphism can exist at many levels. For example, a deciphering algorithm may have two completely different implementations, or two different algorithms may produce the same result. In these cases, the polymorphism is at a higher level and is more difficult to detect.

22.3.8 Macro Viruses

Definition 22–12. A *macro* virus is a virus composed of a sequence of instructions that is interpreted, rather than executed directly.

Conceptually, macro viruses are no different from ordinary computer viruses. Like Duff's *sh* computer virus, they can execute on any system that can interpret the instructions. For example, a spreadsheet virus executes when the spreadsheet interprets these instructions. If the macro language allows the macro to access files or other systems, the virus can access them, too.

EXAMPLE: The Melissa virus infected Word 97 and 98 documents on Windows and Macintosh systems. It is invoked when the program opens an infected file. It installs itself as the "open" macro and copies itself into the Normal template (so any files that are opened are infected). It then invokes a mail program and sends copies of itself to people in the user's address book associated with the program.

A macro virus can infect either executables or data files (the latter leads to the name *data virus*). If it infects executable files, it must arrange to be interpreted at some point. Duff's experiments did this by wrapping the executables with shell scripts. The resulting executables invoked the Bourne shell, which interpreted the virus code before invoking the usual executable.

Macro viruses are not bound by machine architecture. They use specific programs, and so, for example, a macro virus targeted at a Microsoft Word program will work on any system running Microsoft Word. The effects may differ. For example, most Macintosh users do not use the particular mail program that Melissa invoked, so although Macintosh Word files could have been infected, and the infection could have been spread, the virus did not mail itself to other users. On a Windows system, where most users did use that mail program, the infection was spread by mail.

22.4 Computer Worms

A computer virus infects other programs. A variant of the virus is a program that spreads from computer to computer, spawning copies of itself on each one.

Definition 22–13. A *computer worm* is a program that copies itself from one computer to another.

Research into computer worms began in the mid-1970s. Schoch and Hupp [889] developed distributed programs to do computer animations, broadcast messages, and perform other computations. These programs probed workstations. If the workstation was idle, the worm copied a *segment* onto the system. The segment was given data to process and communicated with the worm's controller. When any activity other than the segment's began on the workstation, the segment shut down.

EXAMPLE: On November 2, 1988, a program targeting Berkeley and Sun UNIX-based computers entered the Internet; within hours, it had rendered several thousand

computers unusable [322, 323, 845, 900, 901, 952, 953, 974]. Among other techniques, this program used a virus-like attack to spread: it inserted some instructions into a running process on the target machine and arranged for those instructions to be executed. To recover, these machines had to be disconnected from the network and rebooted, and several critical programs had to be changed and recompiled to prevent reinfection. Worse, the only way to determine if the program had suffered other malicious side effects (such as deletion of files) was to disassemble it. Fortunately, the only purpose of this virus turned out to be self-propagation. Infected sites were extremely lucky that the worm[5] did not infect a system program with a virus designed to delete files and did not attempt to damage attacked systems.

Since then, there have been several incidents involving worms. The Father Christmas worm was interesting because it was a form of macro worm.

EXAMPLE: Slightly before the Internet worm, an electronic "Christmas card" passed around several IBM-based networks. This card was an electronic letter instructing the recipient to save the message and run it as a program. The program drew a Christmas tree (complete with blinking lights) and printed "Merry Christmas!" It then checked the recipient's list of previously received mail and the recipient's address book to create a new list of e-mail addresses. It then sent copies of itself to all these addresses. The worm quickly overwhelmed the IBM networks and forced the networks and systems to be shut down [422].

This worm had the characteristics of a macro worm. It was written in a high-level job control language, which the IBM systems interpreted. Like the Melissa virus, which was written in the Visual Basic programming language, the Father Christmas worm was never directly executed—but its effects (spreading from system to system) were just as serious.

22.5 Other Forms of Malicious Logic

Malicious logic can have other effects, alone or in combination with the effects discussed in Sections 22.2 to 22.4.

22.5.1 Rabbits and Bacteria

Some malicious logic multiplies so rapidly that resources become exhausted. This creates a denial of service attack.

[5] We use the conventional terminology of calling this program a "computer worm" because its dominant method of propagation was from computer system to computer system. Others, notably Eichin and Rochlis [322], have labeled it a "computer virus."

Definition 22–14. A *bacterium* or a *rabbit* is a program that absorbs all of some class of resource.

A bacterium is not required to use all resources on the system. Resources of a specific class, such as file descriptors or process table entry slots, may not affect currently running processes. They will affect new processes.

EXAMPLE: Dennis Ritchie [840] presented the following shell script as something that would quickly exhaust either disk space or inode tables on a UNIX Version 7 system.

```
while true
do
    mkdir x
    chdir x
done
```

He pointed out, however, that the user who caused a crash using this program would be immediately identified when the system was rebooted.

22.5.2 Logic Bombs

Some malicious logic triggers on an external event, such as a user logging in or the arrival of midnight, Friday the 13th.

Definition 22–15. A *logic bomb* is a program that performs an action that violates the security policy when some external event occurs.

Disaffected employees who plant Trojan horses in systems use logic bombs. The events that cause problems are related to the troubles the employees have, such as deleting the payroll roster when that user's name is deleted.

EXAMPLE: In the early 1980s, a program posted to the USENET news network promised to make administering systems easier. The directions stated that the *shar* archive containing the program had to be unpacked, and the program compiled and installed, as *root*. Midway down the *shar* archive were the lines

```
cd /
rm -rf *
```

Anyone who followed the instructions caused these lines to be executed. These commands deleted all files in the system. Some system administrators executed the program with unlimited privileges, thereby damaging their systems.

22.6 Theory of Malicious Logic

The types of malicious logic discussed so far are not distinct. Computer viruses are a form of Trojan horses. Computer viruses may contain logic bombs, as might computer worms. Some worms and viruses are bacteria because they absorb all the resources of some type.

EXAMPLE: The Internet worm was a bacterium on many systems. During its infection, the worm opened a port on the network. When another worm tried to infect the system, it first checked the port. If the port was open, the infecting worm knew that another worm was resident on the computer. The author apparently feared that this check would lead to a defense of system administrators opening the port with a small program. So, once out of every six times, the check was ignored and the worm reinfected the infected system. Because the worm was so prolific, infected machines quickly had many different copies of the worm and were overwhelmed. The worms consumed the CPU.

EXAMPLE: The Father Christmas worm created so much network traffic that the networks became unusable and had to be shut down until all instances of the worm were purged from the mail queues. Hence, it was a bacterium also.

An obvious question is whether a universal detector can be written to detect malicious logic. We consider the narrower question of whether there is an algorithm that can determine if an arbitrary program contains replicating code (this covers both replicating Trojan horses and computer viruses).

22.6.1 Theory of Computer Viruses

Cohen [207] asked if a single algorithm could detect computer viruses precisely. He demonstrated that the virus detection problem, like the safety problem (see Theorem 3–2), is undecidable.

> **Definition 22–16.** [207] Let T be a Turing machine and let V be a set of sequences of symbols on the machine tape. Let s_v be a distinguished state of T. For every $v \in V$, when T lies at the beginning of v in tape square k, suppose that after some number of instructions are executed, a sequence $v' \in V$ lies on the tape beginning at location k', where either $k + |v| \leq k'$ or $k' + |v| \leq k$. Then (T, V) is a *viral set* and the elements of V are *computer viruses*.

Figure 22–3 illustrates this definition. The virus v may copy another element of V either before or after itself but may not overwrite itself. Both possibilities are shown. If v' precedes v, then $k' + |v| \leq k$; otherwise, v precedes v', and $k + |v| \leq k'$. Definition 22–16 is a formal version of Definition 22–4. It focuses on the replication

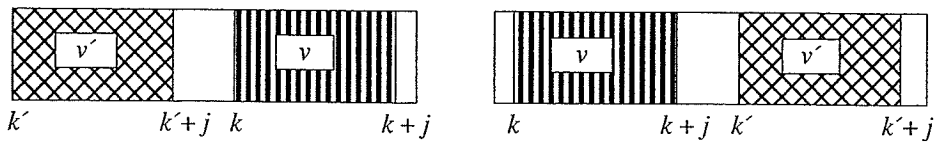


Figure 22–3 Illustration of Cohen's definition of a viral set. Here, *v*, *v´*, *k*, and *k´* are as in Definition 22–16, and |*v*| = *j*. The Turing machine can make copies of *v* either before or after the tape squares containing *v* but does not overwrite any part of *v*. Each diagram shows a possible position for *v´* with respect to *v* on the tape.

(copying) aspect of computer viruses but includes the execution phase as a component of v that need not be copied. In this case, v' would be the infection part of v, and the actions other than infection would be the remainder of v.

Cohen established the undecidability of detecting generic computer viruses by showing that, if such a decision procedure existed, it would solve the halting problem. Consider an arbitrary Turing machine T and an arbitrary sequence S of symbols on tape. Construct a second Turing machine T' and tape V such that, when T halts on S, V and T' create a copy of S on the tape. Then T' replicates S if and only if T halts on S. By Definition 22–16, a replicating program is a computer virus. So, there is a procedure that decides if (T', V) is a viral set if and only if there is a procedure that determines if T halts on S—that is, if there is a procedure that will solve the halting problem. Because the latter does not exist, neither can the former.

Theorem 22–1. [207] It is undecidable whether an arbitrary program contains a computer virus.

Proof Let T and V define a Turing machine and sequence of tape symbols, respectively. We construct a second Turing machine T' and sequence V' such that T' reproduces V if and only if running T on V halts.

Let A and B be tape symbols, so $A, B \in M$. Let q_i, $i \geq 1$ be states of the Turing machine, so $q_i \in K$ for $i \geq 1$. Let a, b, i, and j be non-negative integers. We also redefine the function δ as δ: $K \times M \rightarrow K \times M \times \{L, R, -\}$, where "–" refers to no motion. This function is equivalent to the δ function in Section 3.2 (see Exercise 13).

We will find it convenient to abbreviate arguments and values of δ as follows. Let x, y, z, u, and s_i, $i \geq 1$, represent values drawn from the set of tape symbols M. We can then write

$$\delta(q_a, y) = (q_a, y, L) \text{ when } y \neq A$$

to represent all definitions of δ where the first argument to δ is q_a and the second argument to δ is an element of M other than A.

Three actions recur in our construction of T'. We define abbreviations to simplify the description of that Turing machine. For any symbol $x \in M$, $LS(q_a, x, q_b)$ represents the sequence

$\delta(q_a, x) = (q_b, x, -)$
$\delta(q_a, y) = (q_a, y, L)$ when $y \neq x$

This sequence takes effect when the Turing machine is in state q_a. It moves the head to the left, skipping over take squares, until the machine encounters a square with the symbol x. At that point, the Turing machine enters state q_b, and the head remains over the square with the X symbol.

The abbreviation $RS(q_a, x, q_b)$ is defined similarly, but for motion to the right:

$\delta(q_a, x) = (q_b, x, -)$
$\delta(q_a, y) = (q_a, y, R)$ when $y \neq x$

This sequence moves the head to the right until a square containing x is found. The head stops at that square.

The third abbreviation, $COPY(q_a, x, y, z, q_b)$, means that the Turing machine's head moves right to the next square containing the symbol x and copies the symbols on the tape until the next square with the symbol y is encountered. The copy is placed after the first symbol z following the symbol y. Once the copying is completed, the Turing machine enters state q_b.

The following sequence captures this. The part of each line following the semicolon is a comment, for exposition purposes only. We assume that the symbols A and B do not occur on the tape. If necessary, we augment the set M with two symbols and use them for A and B.

$RS(q_a, x, q_{a+i})$	; move the head over the next x
$\delta(q_{a+i}, x) = (q_{a+i+1}, A, -)$	; replace x with symbol A
$RS(q_{a+i+1}, y, q_{a+i+2})$	; skip to the end of the segment to copy
$RS(q_{a+i+2}, z, q_{a+i+3})$	; skip to the location to copy it to (which
$\delta(q_{a+i+3}, z) = (q_{a+i+4}, z, R)$	; is the square after the one containing z)
$\delta(q_{a+i+4}, u) = (q_{a+i+5}, B, -)$ for any $u \in M$	; mark it with B
$LS(q_{a+i+5}, A, q_{a+i+6})$	; move the head back to where x was
$\delta(q_{a+i+6}, A) = (q_{a+i+7}, x, -)$	; put x back
$\delta(q_{a+i+7}, s_j) = (q_{a+i+5j+10}, A, R)$ for $s_j \neq y$	; overwrite the first uncopied symbol
$\delta(q_{a+i+7}, y) = (q_{a+i+8}, y, R)$	; for the terminal one, go to cleanup
$RS(q_{a+i+5j+10}, B, q_{a+i+5j+11})$	; move to location to copy symbol to
$\delta(q_{a+i+5j+11}, B) = (q_{a+i+5j+12}, s_j, R)$	; put it down

$\delta(q_{a+i+5j+12}, u) = (q_{a+i+5j+13}, B, -)$	; mark where the next symbol goes
$LS(q_{a+i+5j+13}, A, q_{a+i+5j+14})$	; go back to where the original was
$\delta(q_{a+i+5j+14}, A) = (q_{a+i+7}, s_j, R)$	; copy it back
$RS(q_{a+i+8}, B, q_{a+i+9})$	; last symbol—move to where it goes
$\delta(q_{a+i+9}, B) = (q_b, y, -)$	; write it and enter terminal state

We proceed to construct T' and V'. Define the set of symbols in T' to be

$$M' = \{ A, B, C, D \} \cup M$$

where $A, B, C, D \notin M$, and the set of states to be

$$K' = \{ q_a, q_b, q_c, q_d, q_e, q_f, q_g, q_h, q_H \} \cup K$$

where $q_a, q_b, q_c, q_d, q_e, q_f, q_g, q_h, q_H \notin K$. The initial state of T' is q_a, and the halting state of T' is q_H. The initial state of T is q_f, and we simulate the halting state of T by the state q_h. We abbreviate the execution of T on the tape with the head at the current position as $SIMULATE(q_f, T, q_h)$, where q_f is the state of T' corresponding to the initial state of T and q_h is the state of T' corresponding to the final (terminal) state of T.

Let $V' = (A, B, V, C, D)$. Then the transition function δ for T' is:

$\delta(q_a, A) = (q_b, A, -)$	; check beginning of tape
$\delta(q_a, y) = (q_H, y, -)$ for $y \neq A$	; halting state
$COPY(q_b, B, C, D, q_c)$	; copy V after D
$LS(q_c, A, q_d)$	; head moves to D (T executes the copy
$RS(q_d, D, q_e)$	; of V, not the original one)
$\delta(q_e, D) = (q_e, D, R)$	; move over the D
$\delta(q_e, B) = (q_f, B, R)$	; enter the initial state of T with the
	; head at the beginning of V
$SIMULATE(q_f, T, q_h)$	; simulate T running on V
$LS(q_h, A, q_g)$	; T terminated—go to beginning
	; of T''s tape
$COPY(q_g, A, D, D, q_H)$	; copy initial contents over results of
	; running T on V (reproduction)

The Turing machine T' first makes a copy of V. It then simulates T running on the copy of V. The original V is to the left of the copy (see Figure 22–4), so the simulation of T cannot alter it. If the simulation halts, T' enters state q_h, in which the original copy of V is recopied. This satisfies Definition 22–16. On

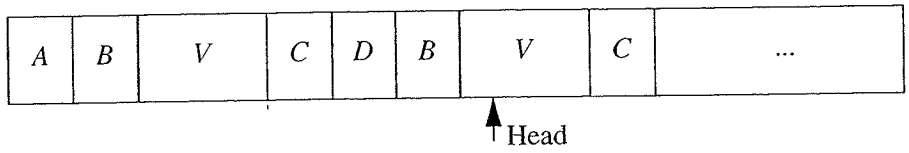


Figure 22–4 The tape V' at state q_f. The head is positioned over the tape for T. Note that, when T is being simulated, the head can never move left over B because T cannot move to the left of the (simulated) tape.

the other hand, if the simulation never halts, V is never recopied, and Definition 22–16 is never satisfied. This establishes the desired result.

Adelman used a completely different approach to obtain a generalization of this result, which we state without proof.

Theorem 22–2. [9] It is undecidable whether an arbitrary program contains malicious logic.

These results mean that there is no generic technique for detecting all malicious logic, or even all computer viruses. Hence, defenses must focus on particular aspects of malicious logic. Furthermore, multiple defenses are needed. We turn to these defenses now.

22.7 Defenses

Defending against malicious logic takes advantage of several different characteristics of malicious logic to detect, or to block, its execution. The defenses inhibit the suspect behavior. The mechanisms are imprecise. They may allow malicious logic that does not exhibit the given characteristic to proceed, and they may prevent programs that are not malicious but do exhibit the given characteristic from proceeding.

22.7.1 Malicious Logic Acting as Both Data and Instructions

Some malicious logic acts as both data and instructions. A computer virus inserts code into another program. During this writing, the object being written into the file (the set of virus instructions) is data. The virus then executes itself. The instructions it executes are the same as what it has just written. Here, the object is treated as an executable set of instructions. Protection mechanisms based on this property treat all programs as type "data" until some certifying authority changes the type to "executable" (instructions). Both new systems designed to meet strong security policies and enhancements of existing systems use these methods (see Section 15.3.1).

EXAMPLE: Boebert, Young, Kain, and Hansohn [127] propose labeling of subjects and objects in the Logical Coprocessor Kernel or LOCK (formerly the Secure Ada Target or SAT) [126, 434, 881, 882], a system designed to meet the highest level of security under the U.S. Department of Defense TCSEC (see Section 21.2). Once compiled, programs have the label "data" and cannot be executed until a sequence of specific, auditable events changes the label to "executable." After that, the program cannot be modified. This scheme recognizes that viruses treat programs as data (when they infect them by changing the file's contents) and as instructions (when the program executes and spreads the virus) and rigidly separates the two.

EXAMPLE: Duff [312] has suggested a variant for UNIX-based systems. Noting that users with execute permission for a file usually also have read permission, he proposes that files with execute permission be of type "executable" and that those without it be of type "data." Unlike the LOCK, "executable" files could be modified, but doing so would change those files' types to "data." If the certifying authority were the omnipotent user, the virus could spread only if run as that user. Libraries and other system components of programs must also be certified before use to prevent infection from nonexecutable files.

Both the LOCK scheme and Duff's proposal trust that the administrators will never certify a program containing malicious logic (either by accident or deliberately) and that the tools used in the certification process are not themselves corrupt.

22.7.2 Malicious Logic Assuming the Identity of a User

Because a user (unknowingly) executes malicious logic, that code can access and affect objects within the user's protection domain. So, limiting the objects accessible to a given process run by the user is an obvious protection technique. This draws on the mechanisms for confining information (see Chapter 17, "Confinement Problem").

22.7.2.1 Information Flow Metrics

Cohen suggests an approach [206]. This approach is to limit the distance a virus can spread.

> **Definition 22–17.** Define the *flow distance metric* $fd(x)$ for some information x as follows. Initially, all information has $fd(x) = 0$. Whenever x is shared, $fd(x)$ increases by 1. Whenever x is used as input to a computation, the flow distance of the output is the maximum of the flow distance of the input.

Information is accessible only while its flow distance is less than some particular value.

EXAMPLE: Anne, Bill, and Cathy work on the same computer. The system uses the flow distance metric to limit the flow of information. Anne can access information with a flow distance less than 3, and Bill and Cathy can access information with a flow distance less than 2. Anne creates a program *dovirus* containing a computer virus. Bill executes it. Because the contents of the program have a flow distance of 0, when the virus infects Bill's file *safefile*, the flow distance of the virus is 1, and so Bill can access it. Hence, the copying succeeds. Now, if Cathy executes *safefile*, when the virus tries to spread to her files, its flow distance increases to 2. Hence, the infection is not permitted (because Cathy can only access information with a flow distance of 0 or 1).

This example also shows the problem with the flow distance policy (which constrains sharing based on the flow distance metric). Although Cathy cannot be infected by viruses that Bill has acquired, she can be infected by viruses that Bill has written. (For example, had Cathy run Anne's *dovirus* program, she would have had her files infected.) The bounding constant limits the transitivity of trust. This number should therefore be low. If it is 1, only the people from whom Cathy copies files are trusted. Cathy does not trust anyone that they trust.

This mechanism raises interesting implementation issues. The metric is associated with *information* and not *objects*. Rather than tagging specific information in files, systems implementing this policy would most likely tag objects, treating the composition of different information as having the maximum flow distance of the information. This will inhibit sharing.

Ultimately, the only way to use this policy is to make the bounding constant 0. This isolates each user into his or her own protection domain and allows no sharing. Cohen points out that this defeats the main purpose of scientific or development environments, in which users build on the work of others.

22.7.2.2 Reducing the Rights

The user can reduce her associated protection domain when running a suspect program. This follows from the principle of least privilege (see Section 13.2.1). Wiseman discusses one approach [1055], and Juni and Ponto present another idea in the context of a medical database [532].

EXAMPLE: Smith [939] combines ACLs and C-Lists to achieve this end. Suppose s_1 owns a file o_1 and s_2 owns a program o_2 and a file o_3. The union of discretionary ACLs is

$$B_{ACL} = \{ (s_1, o_1, r), (s_1, o_1, w), (s_1, o_2, x), (s_1, o_3, w), (s_2, o_2, r), (s_2, o_2, w), (s_2, o_2, x), (s_2, o_3, r) \}$$

Program o_2 contains a Trojan horse. If s_1 wants to execute o_2, he must ensure that it does not write to o_3. Ideally, s_1's protection domain will be reduced to $\{ (s_1, o_2, x) \}$.

Then if p_{12}, the process (subject) created when s_1 executes o_2, tries to access o_3, the access will be denied. In fact, p_{12} inherits the access rights of s_1. So, the default protection domain for p_{12} will be

$$PD(p_{12}) = PD(s_1) = \{ (p_{12}, o_1, r), (p_{12}, o_1, w), (p_{12}, o_2, x), (p_{12}, o_3, w) \}$$

Now, because s_1 can write to o_3, so can p_{12}. Moreover, s_1 cannot constrain this behavior because s_1 does not own o_3 and so cannot delete its access rights over o_3.

Smith's solution is to require each user s_i to define an *authorization denial subset* $R(s_i)$ to contain those ACL entries that it will not allow others to exercise over the objects that s_i owns. In this example, if $R(s_2) = \{ (s_1, o_3, w) \}$, then

$$PD(p_{12}) = PD(s_1) \cap \neg (\cup_{j \neq 1} R(s_j)) = \{ (p_{12}, o_1, r), (p_{12}, o_1, w), (p_{12}, o_2, x) \}$$

where "$\neg$" means set complement. Now p_{12} cannot write to o_3.

Although effective, this approach begs the question of how to determine which entries should be in the authorization denial subsets. Karger suggests basing access on the program being executed and some characteristic of the file being accessed.

EXAMPLE: Karger proposes a knowledge-based subsystem to determine if a program makes reasonable file accesses [550]. The subsystem sits between the kernel open routine and the application. The subsystem contains information about the names of the files that each program is expected to access. For example, a UNIX C compiler reads from C source files (the names of which end in ".c" and ".h") and writes to temporary files (the names of which begin with "/tmp/ctm") and assembly files (whose names end in ".s"). It executes the assembler, which reads from assembly files and writes to object files (with names ending in ".o"). The compiler then invokes the linking loader, which reads from object files and library files (whose names end in ".a") and writes to executable files (with names ending in ".out" unless the user supplies an alternative name). So, Karger's subsystem has the following associations.

Program	*Reads*	*Writes*	*Executes*
Compiler	*.c, *.h	*.s, /tmp/ctm*	Assembler, loader
Assembler	*.s	*.o	
(Linking) loader	*.o, *.a	*.out	

(The "*" means zero or more characters.)

When the subsystem is invoked, it checks that the access is allowed. If not, it either denies the access or asks the user whether to permit the access.

A related approach is to base access to files on some characteristic of the command or program [206], possibly including subject authorizations as well [204].

EXAMPLE: Lai and Gray [603] have implemented a modified version of Karger's scheme on a UNIX system. Unlike Karger, they combine knowledge about each command with the command-line arguments of the current invocation. Their idea is to use this information to determine the user's intent to access files and the type of access. They do not protect these files, but instead prevent other files not named on the command line from being accessed (with two exceptions).

Processes are divided into two groups. File accesses by trusted processes are not checked. Associated with each untrusted process is a *valid access list* (VAL) consisting of the arguments of the process plus any temporary files created. When an untrusted process tries to access a file, the kernel executes the following sequence of steps.

1. If the process is requesting access to a file on the VAL, the access is allowed if the effective UID and GID of the process allow the access.
2. If the process is opening the file for reading and the file is world-readable, the open is allowed.
3. If the process is creating a file, the creation is allowed if the effective UID and GID of the process allow the creation. The file is entered into the VAL of the process and is marked as a *new nonargument* (NNA) *file*. The file's protection modes are set so that no other user may access the file.
4. Otherwise, an entry in the system log reflects the request, and the user is asked if the access is to be allowed. If the user agrees, the access is allowed if the effective UID and GID of the process allow it. Otherwise, the access is denied.

VALs are created whenever a trusted process spawns an untrusted process, and are inherited.

Files marked NNA have permissions such that only the creating user can access them. They are in the VAL of the creating process, and no others, so only that process and its descendents can access the NNA file. However, neither the creating process nor its descendants may change the protection modes of that file. When the file is deleted, its entry is removed from the VAL. When the process terminates, the user is notified of any existing NNA files.

The trusted processes in a 4.3BSD UNIX environment are UNIX command interpreters (*csh* and *sh*), the programs that spawn them on login (*getty* and *login*), programs that access the file system recursively (*ar*, *chgrp*, *chown*, *diff*, *du*, *dump*, *find*, *ls*, *rcp*, *restore*, and *tar*), programs that often access files not in their argument lists (*binmail*, *cpp*, *dbx*, *mail*, *make*, *script*, and *vi*), and various network daemons (*fingerd*, *ftpd*, *ntalkd*, *rlogind*, *rshd*, *sendmail*, *talkd*, *telnetd*, *tftpd*, and *uucpd*). Furthermore, a program called *trust* enables *root* to spawn trusted processes other than those listed above.

As an example, consider the assembler when invoked from the *cc* program. The assembler is called as

```
as x.s /tmp/cc2345
```

and the assembler creates the file */tmp/as1111* during the assembly. The VAL is

```
x.s /tmp/cc2345 /tmp/as1111
```

with the first file being read-only and the next two being readable and writable (the first because *cc* created it and the second because *as* created it). In *cc*'s VAL, the temporary file */tmp/cc2345* is marked NNA; in *as*'s VAL, it is not (because it is a command-line argument to *as*). The loader is invoked as

```
ld /lib/crt0.o /tmp/cc2345 -lc -o x
```

The loader's VAL is

```
/lib/crt0.o /tmp/cc2345 /lib/libc.a x
```

The first three files are read-only and the last file is readable and writable.

Now, suppose a Trojan horse assembler is to copy the program to another user's area. When it attempts to create the target file, rule 3 forces the target to be readable only by the originator. Hence, the attacker cannot read the newly created file. If the attacker creates the file with privileges to allow him to read it, the victim is asked if write access to the file should be allowed. This alerts the user to the presence of the Trojan horse.

An alternative mechanism is interception of requests to open files. The "watchdog" or "guardian" then performs a check to determine if the access is to be allowed. This effectively redefines the system calls involved. The issues of determining how to write watchdogs to meet the desired goals and allowing users to specify semantics for file accesses [88, 259] may prove useful in some contexts—for example, in protecting a limited set of files.

All such mechanisms (1) trust the users to take explicit actions to limit their protection domains sufficiently, (2) trust tables to describe the programs' expected actions sufficiently for the mechanisms to apply those descriptions and to handle commands with no corresponding table entries effectively, or (3) trust specific programs and the kernel when they would be the first programs malicious logic would attack.

22.7.2.3 Sandboxing

Sandboxes and virtual machines (see Section 17.2) implicitly restrict process rights. A common implementation of this approach is to restrict the program by modifying it. Usually, special instructions inserted into the object code cause traps whenever an instruction violates the security policy. If the executable dynamically loads libraries, special libraries with the desired restrictions replace the standard libraries.

EXAMPLE: Bishop and Dilger [117] propose a modification to UNIX system calls to detect race conditions in file accesses. A race condition occurs when successive system calls operate on an object identified by name, and the name can be rebounded to a different object between the first and second system calls. The augmentation involved would record the inode number (unique identifier) of the object identified in the first system call. When the object named in the second system call differed from the object named in the first system call, the mechanism would take appropriate action.

22.7.3 Malicious Logic Crossing Protection Domain Boundaries by Sharing

Inhibiting users in different protection domains from sharing programs or data will inhibit malicious logic from spreading among those domains. This takes advantage of the separation implicit in integrity policies (see Chapter 6).

EXAMPLE: When users share procedures, the LOCK system (see Section 22.7.1) keeps only one copy of the procedure in memory. A master directory, accessible only to a trusted hardware controller, associates with each procedure a unique owner and with each user a list of others whom that user trusts. Before executing any procedure, the dynamic linker checks that the user executing the procedure trusts the procedure's owner [125]. This scheme assumes that users' trust in one another is always well-placed.

A more general proposal [1066] suggests that programs to be protected be placed at the lowest possible level of an implementation of a multilevel security policy. Because the mandatory access controls will prevent those processes from writing to objects at lower levels, any process can read the programs but no process can write to them. Such a scheme would have to be combined with an integrity model to provide protection against viruses to prevent both disclosure and file corruption.

EXAMPLE: The Data General model (see Figure 5–3, on page 129) places the executables below the user region in the hierarchy of layers. The site-specific executables are highest, followed by the trusted data, and the Data General executables are at the lowest level. This prevents alteration of the Data General executables and trusted data by site executables and alteration of all executables and trusted data by user applications.

Carrying this idea to its extreme would result in isolation of each domain. Because sharing would not be possible, no viruses could propagate. Unfortunately, the usefulness of such systems would be minimal.

22.7.4 Malicious Logic Altering Files

Mechanisms using *manipulation detection codes* (or *MDCs*) apply some function to a file to obtain a set of bits called the *signature block* and then protect that block. If, after recomputing the signature block, the result differs from the stored signature block, the file has changed, possibly as a result of malicious logic altering the file. This mechanism relies on selection of good cryptographic checksums (see Section 9.4).

EXAMPLE: Tripwire [568, 569] is an integrity checker that targets the UNIX environment. This program computes a signature block for each file and stores it in a database. The signature of each file consists of file attributes (such as size, owner, protection mode, and inode number) and various cryptographic checksums (such as MD-4, MD-5, HAVAL, SHS, and various CRCs). The system administrator selects the components that make up the signature.

When Tripwire is executed, it recomputes each signature block and compares the recomputed blocks with those in the database. If any of them differ, the change is reported as indicating a possibly corrupted file.

An assumption is that the signed file does not contain malicious logic before it is signed. Page [793] has suggested expansion of Boebert and Kain's model [126] to include the software development process (in effect, limiting execution domains for each development tool and user) to ensure that software is not contaminated during development.

EXAMPLE: Pozzo and Grey [817, 818] have implemented Biba's integrity model on the distributed operating system LOCUS [811] to make the level of trust in the above-mentioned assumption explicit. They have different classes of signed executable programs. *Credibility ratings* (Biba's "integrity levels") assign a measure of trustworthiness on a scale of 0 (unsigned) to N (signed and formally verified), based on the origin of the software. Trusted file systems contain only signed executable files with the same credibility level. Associated with each user (subject) is a *risk level* that starts out as the highest credibility level. Users may execute programs with credibility levels no less than their risk levels.When the credibility level is lower than the risk level, a special "run-untrusted" command must be used.

All integrity-based schemes rely on software that if infected may fail to report tampering. Performance will be affected because encrypting the file or computing the signature block may take a significant amount of time. The encrypting key must also be secret because if it is not, then malicious logic can easily alter a signed file without the change being detected.

Antivirus scanners check files for specific viruses and, if a virus is present, either warn the user or attempt to "cure" the infection by removing the virus. Many such agents exist for personal computers, but because each agent must look for a

particular virus or set of viruses, they are very specific tools and, because of the undecidability results stated earlier, cannot deal with viruses not yet analyzed.

22.7.5 Malicious Logic Performing Actions Beyond Specification

Fault-tolerant techniques keep systems functioning correctly when the software or hardware fails to perform to specifications. Joseph and Avižienis have suggested treating the infection and execution phases of a virus as errors. The first such proposal [529, 530] breaks programs into sequences of nonbranching instructions and checksums each sequence, storing the results in encrypted form. When the program is run, the processor recomputes checksums, and at each branch a coprocessor compares the computed checksum with the encrypted checksum; if they differ, an error (which may be an infection) has occurred. Later proposals advocate checking of each instruction [260]. These schemes raise issues of key management and protection as well as the degree to which the software managing keys, which transmit the control flow graph to the coprocessor and implement the recovery mechanism, can be trusted.

A proposal based on *N-version programming* [48] requires implementation of several different versions of an algorithm, running them concurrently and periodically checking their intermediate results against each other. If they disagree, the value assumed to be correct is the intermediate value that a majority of the programs have obtained, and the programs with different values are malfunctioning (possibly owing to malicious logic). This requires that a majority of the programs are not infected and that the underlying operating system is secure. Also, Knight and Leveson [574] question the efficacy of N-version programming. Detecting the spread of a virus would require voting on each file system access. To achieve this level of comparison, the programs would all have to implement the same algorithm, which would defeat the purpose of using N-version programming [575].

22.7.5.1 Proof-Carrying Code

Necula has proposed a technique that combines specification and integrity checking [762]. His method, called *proof-carrying code* (PCC), requires a "code consumer" (user) to specify a safety requirement. The "code producer" (author) generates a proof that the code meets the desired safety property and integrates that proof with the executable code. This produces a PCC binary. The binary is delivered (through the network or other means) to the consumer. The consumer then validates the safety proof and, if it is correct, can execute the code knowing that it honors that policy. The key idea is that the proof consists of elements drawn from the native code. If the native code is changed in a way that violates the safety policy, the proof is invalidated and will be rejected.

EXAMPLE: Necula and Lee [763] tested their method on UNIX-based network packet filters as supported by the Berkeley Packet Filter (BPF) [669, 724]. These filters were written in an interpreted language. The kernel performed the interpretations

and prevented the filter from looping and from writing to any location except the packet's data or a small scratch memory. The filters were rewritten in assembly language and augmented with proofs that showed that they met the safety policy that the kernel enforced. The proofs ranged from 300 to 900 bytes, and the validation times ranged from 0.3 to 1.3 ms. As expected, the start-up cost was higher (because the proofs had to be validated *before* the filters were run), but the runtimes were considerably shorter. In their experiments, in which 1,000 packets were received per second (on the average), the total cost of using the BPF exceeded the PCC after 1,200 packets. The method also compared favorably with implementations using a restrictive subset of Modula-3 (after 10,500 packets) [89, 496] and software fault isolation (after 28,000 packets; see Section 17.2.2).

22.7.6 Malicious Logic Altering Statistical Characteristics

Like human languages, programs have specific statistical characteristics that malicious logic might alter. Detection of such changes may lead to detection of malicious logic.

EXAMPLE: Malicious logic might be present if a program appears to have more programmers than were known to have worked on it or if one particular programmer appears to have worked on many different and unrelated programs [1066]. Programmers have their own individual styles of writing programs. At the source code level, features such as language, formatting, and comment styles can distinguish coding styles. However, adherence to organizational coding standards obscures these features [598]. At the object code level, features such as choice of data structures and algorithms may distinguish programmers [957].

Comparison of object and source may reveal that the object file contains conditionals not corresponding to any in the source. In this case, the object may be infected [385]. Similar proposals suggest examination of the appearance of programs for identical sequences of instructions or byte patterns [516, 1066]. The disadvantage of such comparisons is that they require large numbers of comparisons and need to take into account the reuse of common library routines or of code [564].

Another proposal suggests that a filter be designed to detect, analyze, and classify all modifications that a program makes as ordinary or suspicious [247]. Along the same lines, Dorothy Denning suggests the use of an intrusion-detection expert system[6] to detect viruses by looking for increases in file size, increases in the frequency of writing to executable files, or alterations in the frequency of execution of a specific program in ways that do not match the profiles of users who are spreading the infection [270].

[6] Chapter 25, "Intrusion Detection," discusses this system in more detail.

22.7.7 The Notion of Trust

The effectiveness of any security mechanism depends on the security of the underlying base on which the mechanism is implemented and the correctness of the implementation. If the trust in the base or in the implementation is misplaced, the mechanism will not be secure. Thus, "secure," like "trust," is a relative notion, and the design of any mechanism for enhancing computer security must attempt to balance the cost of the mechanism against the level of security desired and the degree of trust in the base that the site accepts as reasonable. Research dealing with malicious logic assumes that the interface, software, and/or hardware used to implement the proposed scheme will perform exactly as desired, meaning that the trust is in the underlying computing base, the implementation, and (if done) the verification.

22.8 Summary

Malicious logic is a perplexing problem. It highlights the impotence of standard access controls, because authorized users are requesting authorized actions. The security controls cannot determine if the user knows about such actions.

The most exciting idea is the separation of data from instructions. It unites notions of strong typing with security. In addition to blocking much malicious logic, it has applications for security in general (see Chapter 23, "Vulnerability Analysis," for examples).

Currently, file scanners are the most popular defensive mechanism. Both integrity scanners and antivirus scanners look for changes in files. Antivirus scanners (which also check for some nonvirus Trojan horses) use a database of virus signatures. New dictionaries of these signatures are released periodically, or in the event of a major virus attack. For example, updated virus dictionaries were released within hours after Melissa's discovery.

Integrity scanners check for changes in files, but without determining their causes. If the contents of a file have changed since the last scan, the integrity checker reports this fact, but another agency (user, program) must determine the reason for the change.

22.9 Research Issues

Malicious logic is a fertile ground for study, because the problem is simple but defies easy solution. The key observation is that any solution must distinguish between the actions that users knowingly perform and those same actions when users unknowingly perform them. Humans have a difficult time determining if the actions of others

are deliberate, and so how can computers be endowed with such powers of discrimination? This raises three issues for research: human interaction, integrity checking, and analysis of actions.

Effective procedural mechanisms will prevent users from downloading suspect programs, but how can users be persuaded to abide by these rules, and how can the effects of violating these rules be ameliorated? The notion of "sandboxing," or restriction of privileges (as discussed in Section 22.7.2.3), is intuitively appealing but difficult to put into practice. One issue is how to define the sandbox. The system on which the program is to be run can define the domain of execution (as some Web browsers do) or can be constrained through a combination of the system and of the program itself. In the latter case, the program carries credentials and the receiving system checks them. Both the credentials and the way in which they are checked influence the effectiveness of the reduced domain of execution.

Integrity checking is another area of active research. Cryptographic checksums have been discussed in Section 9.4, and integrity models in Chapter 6. The application of integrity models and the protection and updating of checksums are central to system security. Networks complicate the problem.

Analysis of actions for anomalies is the basis for one form of intrusion detection. Among the issues are characterization of the expected behavior of a program to such a degree that the anomalies that viruses introduce can be distinguished from normal behavior. Because computer viruses typically increase the number of writes (during the infection phase and possibly during the execution phase), examining this number may be fruitful, but other behaviors, such as transitions between localities within the program, are also affected. Could these behaviors be detected?

22.10 Further Reading

Fites, Johnston, and Kratz [355], Hruska [495], and Levin [624] present overviews of computer viruses and their effects. The National Institute of Standards and Technology Special Publication 500-166 [1025] discusses management techniques for minimizing the threats of computer viruses. Spafford, Heaphy, and Ferbrache's book [956] is well written and gives a good exposition of the state of the art in the late 1980s. Arnold [39] and Ludwig [645] describe how to write computer viruses; Arnold's book includes sample code for UNIX systems. Cohen's short course on computer viruses [208] is an excellent technical survey. McIlroy's essay [679] presents a wonderful overview of computer viruses.

Denning's essay [281] presents the nomenclature for malicious logic used in this chapter. His anthology [282], and that of Hoffman [476], collect many of the seminal, and most interesting, papers in the study of malicious logic. Parker [799], Whiteside [1041], and others describe attacks on systems using various forms of malicious logic in a more informal (and enjoyable) manner.

Appel and Felty [35] discuss a semantic model for proof-carrying code.

22.11 Exercises

1. Tripwire does not encipher the signature blocks. What precautions must installers take to ensure the integrity of the database?
2. Consider how a system with capabilities as its access control mechanism could deal with Trojan horses.
   a. In general, do capabilities offer more or less protection against Trojan horses than do access control lists? Justify your answer in light of the theoretical equivalence of ACLs and C-Lists.
   b. Consider now the inheritance properties of new processes. If the creator controls which capabilities the created process is given initially, how could the creator limit the damage that a Trojan horse could do?
   c. Can capabilities protect against all Trojan horses? Either show that they can or describe a Trojan horse process that C-Lists cannot protect against.
3. Describe in detail how an executable infecting computer virus might append itself to an executable. What changes must it make to the executable, and why?
4. A computer system provides protection using the Bell-LaPadula policy. How would a virus spread if:
   a. the virus were placed on the system at system low (the compartment that all other compartments dominate)?
   b. the virus were placed on the system at system high (the compartment that dominates all other compartments)?
5. A computer system provides protection using the Biba integrity model. How would a virus spread if:
   a. the virus were placed on the system at system low (the compartment that all other compartments dominate)?
   b. the virus were placed on the system at system high (the compartment that dominates all other compartments)?
6. A computer system provides protection using the Chinese Wall model. How would a virus spread throughout the system if it were placed within a company dataset? Assume that it is a macro virus.
7. Discuss controls that would prevent Dennis Ritchie's bacterium (see Section 22.5.1) from absorbing all system resources and causing a system crash.
8. How could Thompson's rigged compiler be detected?

9. Place the SAT/LOCK mechanism of treating instructions and data as separate types into the framework of the Clark-Wilson model. In particular, what are the constrained data objects, the transaction procedures, and the certification and enforcement rules?
10. Critique Lai and Gray's virus prevention mechanism described in Section 22.7.2.2. In particular, how realistic is its assessment of the set of programs to be trusted? Are there programs that they omitted or that they should have omitted?
11. Design a signature detection scheme to detect polymorphic viruses, assuming that no encipherment of virus code was used.
12. Assume that the Clark-Wilson model is implemented on a computer system. Could a computer virus that scrambled constrained data items be introduced into the system? Why or why not? Specifically, if not, identify the precise control that would prevent the virus from being introduced, and explain why it would prevent the virus from being introduced; if yes, identify the specific control or controls that would allow the virus to be introduced and explain why they fail to keep it out.
13. Prove that the δ function defined in Section 22.6.1 is equivalent to the δ function in Section 3.2.