9.5 ATTACKS FROM OUTSIDE THE SYSTEM

The threats discussed in the previous sections were largely caused from the inside, that is, perpetrated by users already logged in. However, for machines connected to the Internet or another network, there is a growing external threat. A networked computer can be attacked from a distant computer over the network. In nearly all cases, such an attack consists of some code being transmitted over the network to the target machine and executed there doing damage. As more and more computers join the Internet, the potential for damage keeps growing. In the following sections we will look at some of the operating systems aspects of these external threats, primarily focusing on viruses, worms, mobile code, and Java applets.

It is hard to open a newspaper these days without reading about another computer virus or worm attacking the world's computers. They are clearly a major security problem for individuals and companies alike. In the following sections we will examine how they work and what can be done about them.

I was somewhat hesitant to write this section in so much detail, lest it give some people bad ideas, but existing books give far more detail and even include real code (e.g., Ludwig, 1998). Also the Internet is full of information about viruses so the genie is already out of the bottle. In addition, it is hard for people to defend themselves against viruses if they do not know how they work. Finally, there are a lot of misconceptions about viruses floating around that need correction.

Unlike, say, game programmers, successful virus writers tend not to seek publicity after their products have made their debut. Based on the scanty evidence there is, it appears that most are high school or college students or recent graduates who wrote the virus as a technical challenge, not realizing (or caring) that a virus attack can cost the collective victims as much as a hurricane or earthquake. Let us call our antihero Virgil the virus writer. If Virgil is typical, his goals are to produce a virus that spreads quickly, is difficult to detect, and is hard to get rid of once detected.

What is a virus, anyway? To make a long story short, a **virus** is a program that can reproduce itself by attaching its code to another program, analogous to how biological viruses reproduce. In addition, the virus can also do other things in addition to reproducing itself. Worms are like viruses but are self replicating. That difference will not concern us here, so we will use the term ''virus'' to cover both for the moment. We will look at worms in Sec. 9.5.5.

9.5.1 Virus Damage Scenarios

Since a virus is just a program, it can do anything a program can do. For example, it can type a message, display an image on the screen, play music, or something else harmless. Unfortunately, it can also erase, modify, destroy, or steal files (by emailing them somewhere). Blackmail is also a possibility. Imagine a virus that encrypted all the files on the victim's hard disk, then displayed the following message:

GREETINGS FROM GENERAL ENCRYPTION!

TO PURCHASE A DECRYPTION KEY FOR YOUR HARD DISK, PLEASE SEND $100 IN SMALL, UNMARKED BILLS TO BOX 2154, PANAMA CITY, PANAMA. THANK YOU. WE APPRECIATE YOUR BUSINESS.

Another thing a virus can do is render the computer unusable as long as the virus is running. This is called a **denial of service attack**. The usual approach is consume resources wildly, such as the CPU, or filling up the disk with junk. Here is a one-line program that used to wipe out any UNIX system:

```
main( ) {while (1) fork( );}
```

This program creates processes until the process table is full, preventing any other processes from starting. Now imagine a virus that infected every program in the system with this code. To guard against this problem, many modern UNIX systems limit the number of children a process may have at once.

Even worse, a virus can permanently damage the computer's hardware. Many modern computers hold the BIOS in flash ROM, which can be rewritten under program control (to allow the manufacturer to distribute bug fixes electronically). A virus can write random junk in the flash ROM so that the computer will no longer boot. If the flash ROM chip is in a socket, fixing the problem requires opening up the computer and replacing the chip. If the flash ROM chip is soldered to the parentboard, probably the whole board has to be thrown out and a new one purchased. Definitely not a fun experience.

A virus can also be released with a specific target. A company could release a virus that checked if it was running at a competitor's factory and with no system administrator currently logged in. If the coast was clear, it would interfere with the production process, reducing product quality, thus causing trouble for the competitor. In all other cases it would do nothing, making it hard to detect.

Another example of a targeted virus is one that could be written by an ambitious corporate vice president and released onto the local LAN. The virus would check if it was running on the president's machine, and if so, go find a spreadsheet and swap two random cells. Sooner or later the president would make a bad decision based on the spreadsheet output and perhaps get fired as a result, opening up a position for you-know-who.

9.5.2 How Viruses Work

Enough for potential damage scenarios. Now let us see how viruses work. Virgil writes his virus, probably in assembly language, and then carefully inserts it into a program on his own machine using a tool called a **dropper**. That infected program is then distributed, perhaps by posting it to a bulletin board or a free software collection on the Internet. The program could be an exciting new game, a pirated version of some commercial software, or anything else likely to be considered desirable. People then begin to download the infected program.

Once installed on the victim's machine, the virus lies dormant until the infected program is executed. Once started, it usually begins by infecting other programs on the machine and then executing its **payload**. In many cases, the payload may do nothing until a certain date has passed to make sure that the virus is widespread before people begin noticing it. The date chosen might even send a political message (e.g., if it triggers on the 100th or 500th anniversary of some grave insult to the author's ethnic group).

In the discussion below, we will examine seven kinds of viruses based on what is infected. These are companion, executable program, memory, boot sector, device driver, macro, and source code viruses. No doubt new types will appear in the future.

Companion Viruses

A **companion virus** does not actually infect a program, but gets to run when the program is supposed to run. The concept is easiest to explain with an example. In MS-DOS, when a user types

```
prog
```

MS-DOS first looks for a program named *prog.com*. If it cannot find one, it looks for a program named *prog.exe*. In Windows, when the user clicks on Start and then Run, the same thing happens. Nowadays, most programs are *.exe* files; *.com* files are very rare.

Suppose that Virgil knows that many people run *prog.exe* from an MS-DOS prompt or from Run on Windows. He can then simply release a virus called *prog.com*, which will get executed when anyone tries to run *prog* (unless he actually types the full name: *prog.exe*). When *prog.com* has finished its work, it then just executes *prog.exe* and the user is none the wiser.

A somewhat related attack uses the Windows desktop, which contains shortcuts (symbolic links) to programs. A virus can change the target of a shortcut to make it point to the virus. When the user double clicks on an icon, the virus is executed. When it is done, the virus just runs the original target program.

Executable Program Viruses

One step up in complexity are viruses that infect executable programs. The simplest of these viruses just overwrites the executable program with itself. These are called **overwriting viruses**. The infection logic of such a virus is given in Fig. 9-1.

```
#include <sys/types.h>                     /* standard POSIX headers */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;                          /* for lstat call to see if file is sym link */

search(char *dir_name)
{                                          /* recursively search for executables */
     DIR *dirp;                            /* pointer to an open directory stream */
     struct dirent *dp;                    /* pointer to a directory entry */

     dirp = opendir(dir_name);             /* open this directory */
     if (dirp == NULL) return;             /* dir could not be opened; forget it */
     while (TRUE) {
          dp = readdir(dirp);              /* read next directory entry */
          if (dp == NULL) {                /* NULL means we are done */
          chdir ("..");                    /* go back to parent directory */
          break;                           /* exit loop */
     }
     if (dp->d_name[0] == '.') continue;   /* skip the . and .. directories */
     lstat(dp->d_name, &sbuf);             /* is entry a symbolic link? */
     if (S_ISLNK(sbuf.st_mode)) continue; /* skip symbolic links */
     if (chdir(dp->d_name) == 0) {         /* if chdir succeeds, it must be a dir */
          search(".");                     /* yes, enter and search it */
     } else {                                   /* no (file), infect it */
          if (access(dp->d_name,X_OK) == 0) /* if executable, infect it */
               infect(dp->d_name);
     }
     closedir(dirp);                       /* dir processed; close and return */
}
```

Figure 9-1. A recursive procedure that finds executable files on a UNIX system.

The main program of this virus would first copy its binary program into an array by opening *argv*[0] and reading it in for safe keeping. Then it would traverse the entire file system starting at the root directory by changing to the root directory and calling *search* with the root directory as parameter.

The recursive procedure *search* processes a directory by opening it, then reading the entries one at a time using *readdir* until a *NULL* is returned, indicating that there are no more entries. If the entry is a directory, it is processed by changing to it and then calling *search* recursively; if it is an executable file, it is infected by calling *infect* with the name of the file to infect as parameter. Files starting with "." are skipped to avoid problems with the . and .. directories. Also, symbolic links are skipped because the program assumes that it can enter a directory using the chdir system call and then get back to where it was by going to .. , something that holds for hard links but not symbolic links. A fancier program could handle symbolic links, too.

The actual infection procedure, *infect* (not shown), merely has to open the file named in its parameter, copy the virus saved in the array over the file, and then close the file.

This virus could be ‘‘improved’’ in various ways. First, a test could be inserted into *infect* to generate a random number and just return in most cases without doing anything. In, say, one call out of 128, infection would take place, thereby reducing the chances of early detection, before the virus has had a good chance to spread. Biological viruses have the same property: those that kill their victims quickly do not spread nearly as fast as those that produce a slow, lingering death, giving the victims plenty of chance to spread the virus. An alternative design would be to have a higher infection rate (say, 25%) but a cutoff on the number of files infected at once to reduce disk activity and thus be less conspicuous.

Second, *infect* could check to see if the file is already infected. Infecting the same file twice just wastes time. Third, measures could be taken to keep the time of last modification and file size the same as it was to help hide the infection. For programs larger than the virus, the size will remain unchanged, but for programs smaller than the virus, the program will now be bigger. Since most viruses are smaller than most programs, this is not a serious problem.

Although this program is not very long (the full program is under one page of C and the text segment compiles to under 2 KB), an assembly code version of it can be even shorter. Ludwig (1998) gives an assembly code program for MS-DOS that infects all the files in its directory and is only 44 bytes when assembled.

Later in this chapter we will study antivirus programs, that is programs that track down and remove viruses. Nevertheless, it is interesting to note that the logic of Fig. 9-1, which a virus could use to find all the executable files to infect them could also be used by an antivirus program to track down all the infected programs in order to remove the virus. The technologies of infection and disinfection go hand in hand, which is why it is necessary to understand in detail how viruses work in order to be able to fight them effectively.

From Virgil’s point of view, the problem with an overwriting virus is that it is too easy to detect. After all, when an infected program executes, it may spread the virus some more, but it does not do what it is supposed to do, and the user will notice this instantly. Consequently, most viruses attach themselves to the program and do their dirty work, but allow the program to function normally afterward. Such viruses are called **parasitic viruses**.

Parasitic viruses can attach themselves to the front, the back, or the middle of the executable program. If a virus attaches itself to the front of a program, it has to first copy the program to RAM, write itself at the front of the file, and then copy the program back from RAM following itself, as shown in Fig. 9-2(b). Unfortunately, the program will not run at its new virtual address, so either the virus has to relocate the program as it is moved, or slide it back to virtual address 0 after finishing its own execution.

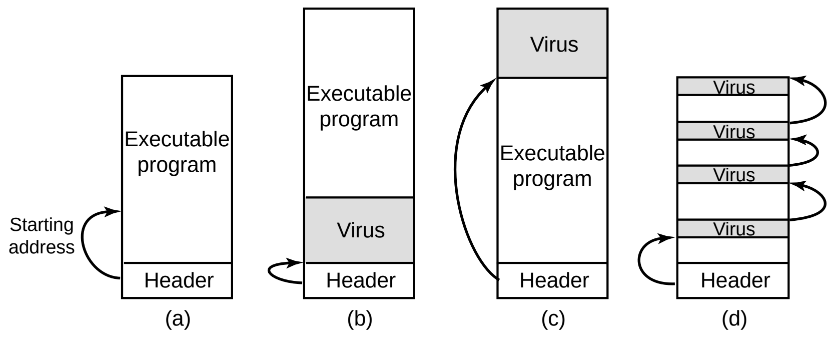


Figure 9-2. (a) An executable program. (b) With a virus at the front. (c) With a virus at the end. (d) With a virus spread over free space within the program.

To avoid either of the complex options required by these front loaders, most viruses are back loaders, attaching themselves to the end of the executable program instead of the front, changing the starting address field in the header to point to the start of the virus, as illustrated in Fig. 9-2(c). The virus will now execute at a different virtual address depending which infected program is running, but all this means is that Virgil has to make sure his virus is position independent, using relative instead of absolute addresses. That is not hard for an experienced programmer to do.

Complex executable program formats, such as *.exe* files on Windows and nearly all modern UNIX binary formats, allow a program to have multiple text and data segments, with the loader assembling them in memory and doing relocation on the fly. In some systems (Windows, for example), all segments (sections) are multiples of 512 bytes. If a segment is not full, the linker fills it out with 0s. A virus that understands this can try to hide itself in the holes. If it fits entirely, as in Fig. 9-2(d), the file size remains the same as that of the uninfected file, clearly a plus, since a hidden virus is a happy virus. Viruses that use this principle are called **cavity viruses**. Of course, if the loader does not load the cavity areas into memory, the virus will need another way of getting started.

Memory Resident Viruses

So far we have assumed that when an infected program is executed, the virus runs, passes control to the real program, and exits. In contrast, a **memory-resident virus** stays in memory all the time, either hiding at the very top of memory or perhaps down in the grass among the interrupt vectors, the last few hundred bytes of which are generally unused. A very smart virus can even modify the operating system's RAM bitmap to make the system think the virus' memory is occupied, to avoid the embarrassment of being overwritten.

A typical memory-resident virus captures one of the trap or interrupt vectors by copying the contents to a scratch variable and putting its own address there, thus directing that trap or interrupt to it. The best choice is the system call trap. In that way, the virus gets to run (in kernel mode) on every system call. When it is done, it just invokes the real system call by jumping to the saved trap address.

Why would a virus want to run on every system call? To infect programs, naturally. The virus can just wait until an exec system call comes along, and then, knowing that the file at hand is an executable binary (and probably a useful one at that), infect it. This process does not require the massive disk activity of Fig. 9-1 so it is far less conspicuous. Catching all system calls also gives the virus great potential for spying on data and performing all manner of mischief.

Boot Sector Viruses

As we discussed in Chap. 5, when most computers are turned on, the BIOS reads the master boot record from the start of the boot disk into RAM and executes it. This program determines which partition is active and reads in the first sector, the boot sector, from that partition and executes it. That program then either loads the operating system or brings in a loader to load the operating system. Unfortunately, many years ago one of Virgil's friends got the idea of creating a virus that could overwrite the master boot record or the boot sector, with devastating results. Such viruses, called **boot sector viruses**, are very common.

Normally, a boot sector virus, [which includes MBR (Master Boot Record) viruses], first copies the true boot sector to a safe place on the disk so it can boot the operating system when it is finished. The Microsoft disk formatting program, *fdisk*, skips the first track, so that is a good hiding place on Windows machines. Another option is to use any free disk sector and then update the bad sector list to mark the hideout as defective. In fact, if the virus is large, it can also disguise the rest of itself as bad sectors. If the root directory is large enough and in a fixed place, as it is in Windows 98, the end of the root directory is also a possibility. A really aggressive virus could even just allocate normal disk space for the true boot sector and itself and update the disk's bitmap or free list accordingly. Doing this requires an intimate knowledge of the operating system's internal data structures, but Virgil had a good professor for his operating systems course and studied hard.

When the computer is booted, the virus copies itself to RAM, either at the top or among the unused interrupt vectors. At this point the machine is in kernel mode, with the MMU off, no operating system, and no antivirus program running. Party time for viruses. When it is ready, it boots the operating system, usually staying memory resident.

One problem, however, is how to get control again later. The usual way is to exploit specific knowledge of how the operating system manages the interrupt vectors. For example, Windows does not overwrite all the interrupt vectors in one blow. Instead, it loads device drivers one at a time, and each one captures the interrupt vector it needs. This process can take a minute.

This design gives the virus the handle it needs. It starts out by capturing all the interrupt vectors as shown in Fig. 9-3(a). As drivers load, some of the vectors are overwritten, but unless the clock driver is loaded first, there will be plenty of clock interrupts later that start the virus. Loss of the printer interrupt is shown in Fig. 9-3(b). As soon as the virus sees that one of its interrupt vectors has been overwritten, it can overwrite that vector again, knowing that it is now safe (actually, some interrupt vectors are overwritten several times during booting, but the pattern is deterministic and Virgil knows it by heart). Recapture of the printer is shown in Fig. 9-3(c). When everything is loaded, the virus restores all the interrupt vectors and keeps only the system call trap vector for itself. After all, getting control on every system call is much more fun than getting control after every floppy disk operation, but during booting, it cannot take the risk of losing control forever. At this point we have a memory-resident virus in control of system calls. In fact, this is how most memory-resident viruses get started in life.

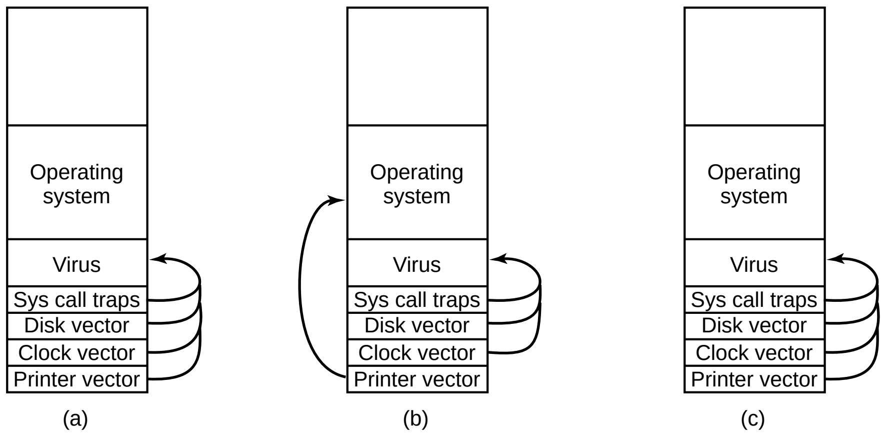


Figure 9-3. (a) After the virus has captured all the interrupt and trap vectors. (b) After the operating system has retaken the printer interrupt vector. (c) After the virus has noticed the loss of the printer interrupt vector and recaptured it.

Device Driver Viruses

Getting into memory like this is a little like spelunking (exploring caves)—you have to go through contortions and keep worrying about something falling down and landing on your head. It would be much simpler if the operating system would just kindly load the virus officially. With a little bit of work, that goal can be achieved. The trick is to infect a device driver, leading to a **device driver virus**. In Windows and some UNIX systems, device drivers are just executable programs that live on the disk and are loaded at boot time. If one of them can be infected using a parasitic virus, the virus will always be officially loaded at boot time. Even nicer, drivers run in kernel mode and after a driver is loaded, it is called, giving the virus a chance to capture the system call trap vector.

Macro Viruses

Many programs, such as *Word* and *Excel*, allow users to write macros to group several commands that can later be executed with a single keystroke. Macros can also be attached to menu items, so that when one of them is selected, the macro is executed. In Microsoft *Office*, macros can contain entire programs in Visual Basic, which is a complete programming language. The macros are interpreted rather than compiled, but that only affects execution speed, not what they can do. Since macros may be document specific, *Office* stores the macros for each document along with the document.

Now comes the problem. Virgil writes a document in *Word* and creates a macro that he attaches to the OPEN FILE function. The macro contains a **macro virus**. He then emails the document to the victim, who naturally opens it (assuming the email program has not already done this for him). Opening the document causes the OPEN FILE macro to execute. Since the macro can contain an arbitrary program, it can do anything, such as infect other *Word* documents, erase files, and more. In all fairness to Microsoft, *Word* does give a warning when opening a file with macros, but most users do not understand what this means and continue opening anyway. Besides, legitimate documents may also contain macros. And there are other programs that do not even give this warning, making it even harder to detect a virus.

With the growth of email, sending documents with viruses embedded in macros is an immense problem. Such viruses are much easier to write than concealing the true boot sector somewhere in the bad block list, hiding the virus among the interrupt vectors, and capturing the system call trap vector. This means that increasingly less skilled people can now write viruses, lowering the general quality of the product and giving virus writers a bad name.

Source Code Viruses

Parasitic and boot sector viruses are highly platform specific; document viruses are somewhat less so (*Word* runs on Windows and the Macintosh, but not on UNIX). The most portable viruses of all are **source code viruses**. Imagine the virus of Fig. 9-1, but with the modification that instead of looking for binary executable files, it looks for C programs, a change of only 1 line (the call to access). The *infect* procedure should be changed to insert the line

```
#include <virus.h>
```

at the top of each C source program. One other insertion is needed, the line

```
run_virus( );
```

to activate the virus. Deciding where to put this line requires some ability to parse C code, since it must be at a place that syntactically allows procedure calls and also not at a place where the code would be dead (e.g., following a return statement). Putting it in the middle of a comment does not work either, and putting it inside a loop might be too much of a good thing. Assuming the call can be placed properly (for example, just before the end of *main* or before the return statement if there is one), when the program is compiled, it now contains the virus, taken from *virus.h* (although *proj.h* might attract less attention should somebody see it).

When the program runs, the virus will be called. The virus can do anything it wants to, for example, look for other C programs to infect. If it finds one, it can include just the two lines given above, but this will only work on the local machine, where *virus.h* is assumed to be installed already. To have this work on a remote machine, the full source code of the virus must be included. This can be done by including the source code of the virus as an initialized character string, preferably as a list of 32-bit hexadecimal integers to prevent anyone from figuring out what it does. This string will probably be fairly long, but with today's multimegaline code, it might easily slip by.

To the uninitiated reader, all of these ways may look fairly complicated. One can legitimately wonder if they could be made to work in practice. They can be. Virgil is an excellent programmer and has a lot of free time on his hands. Check your local newspaper for proof.

9.5.3 How Viruses Spread

There are several scenarios for distribution. Let us start with the classical one. Virgil writes his virus, inserts it into some program he has written (or stolen), and starts distributing the program, for example, by putting it on a shareware Web site. Eventually, somebody downloads the program and runs it. At this point there are several options. To start with, the virus probably infects more files on the hard disk, just in case the victim decides to share some of these with a friend later. It can also try to infect the boot sector of the hard disk. Once the boot sector is infected, it is easy to start a kernel-mode memory-resident virus on subsequent boots.

In addition, the virus can check to see if there are any floppy disks in the drives, and if so, infect their files and boot sectors. Floppy disks are a good target because they get moved from machine to machine much more often than hard disks. If a floppy disk boot sector is infected and that disk is later used to boot a different machine, it can start infecting files and the hard disk boot sector on that machine. In the past, when floppy disks were the main transmission medium for programs, this mechanism was the main way viruses spread.

Nowadays, other options are available to Virgil. The virus can be written to check if the infected machine is on a LAN, something that is very likely on a machine belonging to a company or university. The virus can then start infecting unprotected files on the servers connected to this LAN. This infection will not extend to protected files, but that can be dealt with by making infected programs act strangely. A user who runs such a program will likely ask the system administrator for help. The administrator will then try out the strange program himself to see what is going on. If the administrator does this while logged in as superuser, the virus can now infect the system binaries, device drivers, operating system, and boot sectors. All it takes is one mistake like this and all the machines on the LAN are compromised.

Often machines on a LAN have authorization to log onto remote machines over the Internet or a private corporate, or even authorization to execute commands remotely without logging in. This ability provides more opportunity for viruses to spread. Thus one innocent mistake can infect the entire company. To prevent this scenario, all companies should have a general policy telling administrators never to make mistakes.

Another way to spread a virus is to post an infected program to a USENET newsgroup or bulletin board system to which programs are regularly posted. Also possible is to create a Web page that requires a special browser plug-in to view, and then make sure the plug-ins are infected.

A different attack is to infect a document and then email it to many people or broadcast it to a mailing list or USENET newsgroup, usually as an attachment. Even people who would never dream of running a program some stranger sent them might not realize that clicking on the attachment to open it can release a virus on their machine. To make matters worse, the virus can then look for the user's address book and then mail itself to everyone in the address book, usually with a Subject line that looks legitimate or interesting, like

```
Subject: Change of plans
Subject: Re: that last email
Subject: The dog died last night
Subject: I am seriously ill
Subject: I love you
```

When the email arrives, the receiver sees that the sender is a friend or colleague, and thus does not suspect trouble. Once the email has been opened, it is too late. The "I LOVE YOU" virus that spread around the world in June 2000 worked this way and did a billion dollars worth of damage.

Somewhat related to the actual spreading of active viruses is the spreading of virus technology. There are groups of virus writers who actively communicate over the Internet and help each other develop new technology, tools, and viruses. Most of these are probably hobbyists rather than career criminals, but the effects can be just as devastating. One other category of virus writers is the military, which sees viruses as a weapon of war potentially able to disable an enemy's computers.

Another issue related to spreading viruses is avoiding detection. Jails have notoriously bad computing facilities, so Virgil would prefer avoiding them. If he posts the initial virus from his home machine he is running a certain risk. If the attack is successful, the police might track him down by looking for the virus message with the youngest timestamp, since that is probably closest to the source of the attack.

To minimize his exposure, Virgil might go to an Internet cafe in a distant city and log in there. He can either bring the virus on a floppy disk and read it in himself, or if the machines do not all have floppy disk drives, ask the nice young lady at the desk to please read in the file *book.doc* so he can print it. Once it is on his hard disk, he renames the file *virus.exe* and executes it, infecting the entire LAN with a virus that triggers two weeks later, just in case the police decide to ask the airlines for a list of all people who flew in that week. An alternative is to forget the floppy disk and get the virus from a remote FTP site. Or bring a laptop and plug it in to an Ethernet or USB port that the Internet cafe has thoughtfully provided for laptop-toting tourists who want to read their email every day.

9.5.4 Antivirus and Anti-Antivirus Techniques

Viruses try to hide and users try to find them, which leads to a cat-and-mouse game. Let us now look at some of the issues here. To avoid showing up in directory listings, a companion virus, source code virus, or other file that should not be there can turn on the HIDDEN bit in Windows or use a file name beginning with the . character in UNIX. More sophisticated is to modify Windows' *explorer* or UNIX' *ls* to refrain from listing files whose names begin with *Virgil-*. Viruses can also hide in unusual and unsuspected places, such as the bad sector list on the disk or the Windows registry (an in-memory database available for programs to store uninterpreted strings). The flash ROM used to hold the BIOS and the CMOS memory are also possibilities although the former is hard to write and the latter is quite small. And, of course, the main workhorse of the virus world is infecting executable files and documents on the hard disk.

Virus Scanners

Clearly, the average garden-variety user is not going to find many viruses that do their best to hide, so a market has developed for antivirus software. Below we will discuss how this software works. Antivirus software companies have laboratories in which dedicated scientists work long hours tracking down and understanding new viruses. The first step is to have the virus infect a program that does nothing, often called a **goat file**, to get a copy of the virus in its purest form. The next step is to make an exact listing of the virus' code and enter it into the database of known viruses. Companies compete on the size of their databases. Inventing new viruses just to pump up your database is not considered sporting.

Once an antivirus program is installed on a customer's machine, the first thing it does is scan every executable file on the disk looking for any of the viruses in the database of known viruses. Most antivirus companies have a Web site from which customers can download the descriptions of newly-discovered viruses into their databases. If the user has 10,000 files and the database has 10,000 viruses, some clever programming is needed to make it go fast, of course.

Since minor variants of known viruses pop up all the time, a fuzzy search is needed, so a 3-byte change to a virus does not let it escape detection. However, fuzzy searches are not only slower than exact searches, but they may turn up false alarms, that is, warnings about legitimate files that happen to contain some code vaguely similar to a virus reported in Pakistan 7 years ago. What is the user supposed to do with the message:

```
WARNING! File xyz.exe may contain the lahore-9x virus. Delete?
```

The more viruses in the database and the broader the criteria for declaring a hit, the more false alarms there will be. If there are too many, the user will give up in disgust. But if the virus scanner insists on a very close match, it may miss some modified viruses. Getting it right is a delicate heuristic balance. Ideally, the lab should try to identify some core code in the virus that is not likely to change and use this as the virus signature to scan for.

Just because the disk was declared virus free last week does not mean that it still is, so the virus scanner has to be run frequently. Because scanning is slow, it is more efficient to check only those files that have been changed since the date of the last scan. The trouble is, a clever virus will reset the date of an infected file to its original date to avoid detection. The antivirus program's response to that is to check the date the enclosing directory was last changed. The virus' response to that is to reset the directory's date as well. This is the start of the cat-and-mouse game alluded to above.

Another way for the antivirus program to detect file infection is to record and store on the disk the lengths of all files. If a file has grown since the last check, it might be infected, as shown in Fig. 9-4(a-b). However, a clever virus can avoid detection by compressing the program and padding out the file to its original length. To make this scheme work, the virus must contain both compression and decompression procedures, as shown in Fig. 9-4(c). Another way for the virus to try to escape detection is to make sure its representation on the disk does not look at all like its representation in the antivirus software's database. One way to achieve this goal is to encrypt itself with a different key for each file infected. Before making a new copy, the virus generates a random 32-bit encryption key, for example by XORing the current time with the contents of, say, memory words 72,008 and 319,992. It then XORs its code with this key, word by word to produce the encrypted virus stored in the infected file, as illustrated in Fig. 9-4(d).

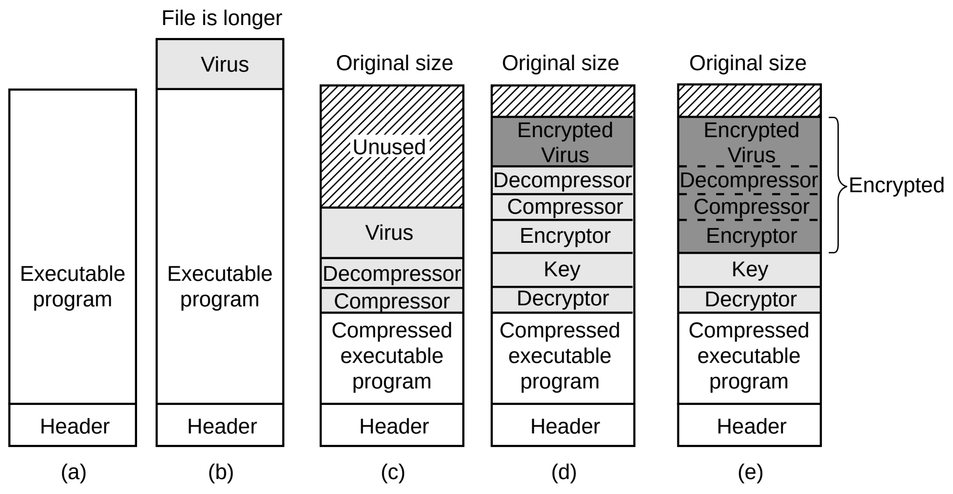


Figure 9-4. (a) A program. (b) An infected program. (c) A compressed infected program. (d) An encrypted virus. (e) A compressed virus with encrypted compression code.

The key is stored in the file. For secrecy purposes, putting the key in the file is not ideal, but the goal here is to foil the virus scanner, not prevent the dedicated scientists at the antivirus lab from reverse engineering the code. Of course, to run, the virus has to first decrypt itself, so it needs a decrypting procedure in the file as well.

This scheme is still not perfect because the compression, decompression, encryption, and decryption procedures are the same in all copies, so the antivirus program can just use them as the virus signature to scan for. Hiding the compression, decompression, and encryption procedures is easy: they are just encrypted along with the rest of the virus, as shown in Fig. 9-4(e). The decryption code cannot be encrypted, however. It has to actually execute on the hardware to decrypt the rest of the virus so it must be present in plaintext form. Antivirus programs know this, so they hunt for the decryption procedure.

However, Virgil enjoys having the last word, so he proceeds as follows. Suppose that the decryption procedure needs to perform the calculation

```
X = (A + B + C − 4)
```

The straightforward assembly code for this calculation for a generic two-address computer is shown in Fig. 9-5(a). The first address is the source; the second is the destination, so MOV A,R1 moves the variable *A* to the register R1. The code in Fig. 9-5(b) does the same thing, only less efficiently due to the NOP (no operation) instructions interspersed with the real code.

But we are not done yet. It is also possible to disguise the decryption code.

```
MOV A,R1      MOV A,R1      MOV A,R1      MOV A,R1      MOV A,R1
ADD B,R1      NOP           ADD #0,R1     OR R1,R1      TST R1
ADD C,R1      ADD B,R1      ADD B,R1      ADD B,R1      ADD C,R1
SUB #4,R1     NOP           OR R1,R1      MOV R1,R5     MOV R1,R5
MOV R1,X      ADD C,R1      ADD C,R1      ADD C,R1      ADD B,R1
              NOP           SHL #0,R1     SHL R1,0      CMP R2,R5
              SUB #4,R1     SUB #4,R1     SUB #4,R1     SUB #4,R1
              NOP           JMP .+1       ADD R5,R5     JMP .+1
              MOV R1,X      MOV R1,X      MOV R1,X      MOV R1,X
                                          MOV R5,Y      MOV R5,Y
   (a)           (b)           (c)           (d)           (e)
```

Figure 9-5. Examples of a polymorphic virus.

There are many ways to represent NOP. For example, adding 0 to a register, ORing it with itself, shifting it left 0 bits, and jumping to the next instruction all do nothing. Thus the program of Fig. 9-5(c) is functionally the same as the one of Fig. 9-5(a). When copying itself, the virus could use Fig. 9-5(c) instead of Fig. 9-5(a) and still work later when executed. A virus that mutates on each copy is called a **polymorphic virus**.

Now suppose that R5 is not needed during this piece of the code. Then Fig. 9-5(d) is also equivalent to Fig. 9-5(a). Finally, in many cases it is possible to swap instructions without changing what the program does, so we end up with Fig. 9-5(e) as another code fragment that is logically equivalent to Fig. 9-5(a). A piece of code that can mutate a sequence of machine instructions without changing its functionality is called a **mutation engine**, and sophisticated viruses contain them to mutate the decryptor from copy to copy. The mutation engine itself can be hidden by encrypting it along with the body of the virus.

Asking the poor antivirus software to realize that Fig. 9-5(a) through Fig. 9-5(e) are all functionally equivalent is asking a lot, especially if the mutation engine has many tricks up its sleeve. The antivirus software can analyze the code to see what it does, and it can even try to simulate the operation of the code, but remember it may have thousands of viruses and thousands of files to analyze so it does not have much time per test or it will run horribly slowly.

As an aside, the store into the variable *Y* was thrown in just to make it harder to detect the fact that the code related to R5 is dead code, that is, does not do anything. If other code fragments read and write *Y*, the code will look perfectly legitimate. A well-written mutation engine that generates good polymorphic code can give antivirus software writers nightmares. The only bright side is that such an engine is hard to write, so Virgil's friends all use his code, which means there are not so many different ones in circulation—yet.

So far we have talked about just trying to recognize viruses in infected executable files. In addition, the antivirus scanner has to check the MBR, boot sectors, bad sector list, flash ROM, CMOS memory, etc., but what if there is a memory-resident virus currently running? That will not be detected. Worse yet, suppose the running virus is monitoring all system calls. It can easily detect that the antivirus program is reading the boot sector (to check for viruses). To thwart the antivirus program, the virus does not make the system call. Instead it just returns the true boot sector from its hiding place in the bad block list. It also makes a mental note to reinfect all the files when the virus scanner is finished.

To prevent being spoofed by a virus, the antivirus program could make hard reads to the disk, bypassing the operating system. However this requires having built-in device drivers for IDE, SCSI, and other common disks, making the antivirus program less portable and subject to failure on computers with unusual disks. Furthermore, since bypassing the operating system to read the boot sector is possible, but bypassing it to read all the executable files is not, there is also some danger that the virus can produce fraudulent data about executable files as well.

Integrity Checkers

A completely different approach to virus detection is **integrity checking**. An antivirus program that works this way first scans the hard disk for viruses. Once it is convinced that the disk is clean, it computes a checksum for each executable file and writes the list of checksums for all the relevant files in a directory to a file, *checksum*, in that directory. The next time it runs, it recomputes all the checksums and sees if they match what is in the file *checksum*. An infected file will show up immediately.

The trouble is Virgil is not going to take this lying down. He can write a virus that removes the checksum file. Worse yet, he can write a virus that computes the checksum of the infected file and replaces the old entry in the checksum file. To protect against this kind of behavior, the antivirus program can try to hide the checksum file, but that is not likely to work since Virgil can study the antivirus program carefully before writing the virus. A better idea is to encrypt it to make tampering easier to detect. Ideally, the encryption should involve use of a smart card with an externally stored key that programs cannot get at.

Behavioral Checkers

A third strategy used by antivirus software is **behavioral checking**. With this approach, the antivirus program lives in memory while the computer is running and catches all system calls itself. The idea is that it can then monitor all activity and try to catch anything that looks suspicious. For example, no normal program should attempt to overwrite the boot sector, so an attempt to do so is almost certainly due to a virus. Likewise, changing the flash ROM is highly suspicious.

But there are also cases that are less clear cut. For example, overwriting an executable file is a peculiar thing to do—unless you are a compiler. If the antivirus software detects such a write and issues a warning, hopefully the user knows whether overwriting an executable makes sense in the context of the current work. Similarly, *Word* overwriting a *.doc* file with a new document full of macros is not necessarily the work of a virus. In Windows, programs can detach from their executable file and go memory resident using a special system call. Again, this might be legitimate, but a warning might still be useful.

Viruses do not have to passively lie around waiting for an antivirus program to kill them, like cattle being led off to slaughter. They can fight back. A particularly interesting battle can occur if a memory-resident virus and a memory-resident antivirus meet up on the same computer. Years ago there was a game called Core Wars in which two programmers faced off by each dropping a program into an empty address space. The programs took turns probing memory, with the object of the game being to locate and wipe out your opponent before he wiped you out. The virus-antivirus confrontation looks a little like that, only the battlefield is the machine of some poor user who does not really want it to happen there. Worse yet, the virus has an advantage because its writer can find out a lot about the antivirus program by just buying a copy of it. Of course, once the virus is out there, the antivirus team can modify their program, forcing Virgil to go buy a new copy.

Virus Avoidance

Every good story needs a moral. The moral of this one is

Better safe than sorry.

Avoiding viruses in the first place is a lot easier than trying to track them down once they have infected a computer. Below are a few guidelines for individual users, but also some things that the industry as a whole can do to reduce the problem considerably.

What can users do to avoid a virus infection? First, choose an operating system that offers a high degree of security, with a strong kernel-user mode boundary and separate login passwords for each user and the system administrator. Under these conditions, a virus that somehow sneaks in cannot infect the system binaries.

Second, install only shrink-wrapped software bought from a reliable manufacturer. Even this is no guarantee since there have been cases where disgruntled employees have slipped viruses onto a commercial software product, but it helps a lot. Downloading software from Web sites and bulletin boards is risky behavior.

Third, buy a good antivirus software package and use it as directed. Be sure to get regular updates from the manufacturer's Web site.

Fourth, do not click on attachments to email and tell people not to send them to you. Email sent as plain ASCII text is always safe but attachments can start viruses when opened.

Fifth, make frequent backups of key files onto an external medium, such as floppy disk, CD-recordable, or tape. Keep several generations of each file on a series of backup media. That way, if you discover a virus, you may have a chance to restore files as they were before they were infected. Restoring yesterday's infected file does not help, but restoring last week's version might.

The industry should also take the virus threat seriously and change some dangerous practices. First, make simple operating systems. The more bells and whistles there are, the more security holes there are. That is a fact of life.

Second, forget active content. From a security point of view, it is a disaster. Viewing a document someone sends you should not require your running their program. JPEG files, for example, do not contain programs, and thus cannot contain viruses. All documents should work like that.

Third, there should be a way to selectively write protect specified disk cylinders to prevent viruses from infecting the programs on them. This protection could be implemented by having a bitmap inside the controller listing the write protected cylinders. The map should only be alterable when the user has flipped a mechanical toggle switch on the computer's front panel.

Fourth, flash ROM is a nice idea, but it should only be modifiable when an external toggle switch has been flipped, something that will only happen when the user is consciously installing a BIOS update. Of course, none of this will be taken seriously until a really big virus hits. For example, one that hit the financial world and reset all bank accounts to 0. Of course, by then it would be too late.

Recovery from a Virus Attack

When a virus is detected, the computer should be halted immediately since a memory-resident virus may still be running. The computer should be rebooted from a CD-ROM or floppy disk that has always been write protected, and which contains the full operating system to bypass the boot sector, hard disk copy of the operating system, and disk drivers, all of which may now be infected. Then an antivirus program should be run from its original CD-ROM, since the hard disk version may also be infected.

The antivirus program may detect some viruses and may even be able to eliminate them, but there is no guarantee that it will get them all. Probably the safest course of action at this point is to save all files that cannot contain viruses (like ASCII and JPEG files). Those files that might contain viruses (like *Word* files) should be converted to another format that cannot contain viruses, such as flat ASCII text (or at least the macros should be removed). All the saved files should be saved on an external medium. Then the hard disk should be reformatted using a format program taken from a write-protected floppy disk or a CD-ROM to insure that it itself is not infected. It is especially important that the MBR and boot sectors are also fully erased. Then the operating system should be reinstalled from the original CD-ROM. When dealing with virus infections, paranoia is your best friend.

9.5.5 The Internet Worm

The first large-scale Internet computer security violation began in the evening of Nov. 2, 1988 when a Cornell graduate student, Robert Tappan Morris, released a worm program into the Internet. This action brought down thousands of computers at universities, corporations, and government laboratories all over the world before it was tracked down and removed. It also started a controversy that has not yet died down. We will discuss the highlights of this event below. For more technical information see (Spafford, 1989). For the story viewed as a police thriller, see (Hafner and Markoff, 1991).

The story began sometime in 1988 when Morris discovered two bugs in Berkeley UNIX that made it possible to gain unauthorized access to machines all over the Internet. Working alone, he wrote a self replicating program, called a **worm**, that would exploit these errors and replicate itself in seconds on every machine it could gain access to. He worked on the program for months, carefully tuning it and having it try to hide its tracks.

It is not known whether the release on Nov. 2, 1988 was intended as a test, or was the real thing. In any event, it did bring most of the Sun and VAX systems on the Internet to their knees within a few hours of its release. Morris' motivation is unknown, but it is possible that he intended the whole idea as a high-tech practical joke, but which due to a programming error got completely out of hand.

Technically, the worm consisted of two programs, the bootstrap and the worm proper. The bootstrap was 99 lines of C called *l1.c*. It was compiled and executed on the system under attack. Once running, it connected to the machine from which it came, uploaded the main worm, and executed it. After going to some trouble to hide its existence, the worm then looked through its new host's routing tables to see what machines that host was connected to and attempted to spread the bootstrap to those machines.

Three methods were tried to infect new machines. Method 1 was to try to run a remote shell using the *rsh* command. Some machines trust other machines, and just run *rsh* without any further authentication. If this worked, the remote shell uploaded the worm program and continued infecting new machines from there.

Method 2 made use of a program present on all BSD systems called *finger* that allows a user anywhere on the Internet to type

```
finger name@site
```

to display information about a person at a particular installation. This information usually includes the person's real name, login, home and work addresses and telephone numbers, secretary's name and telephone number, FAX number, and similar information. It is the electronic equivalent of the phone book.

Finger works as follows. At every BSD site a background process called the **finger daemon** runs all the time fielding and answering queries from all over the Internet. What the worm did was call *finger* with a specially handcrafted 536-

byte string as parameter. This long string overflowed the daemon's buffer and overwrote its stack, the way shown in Fig. 9-0(c). The bug exploited here was the daemon's failure to check for overflow. When the daemon returned from the procedure it was in at the time it got the request, it returned not to *main*, but to a procedure inside the 536-byte string on the stack. This procedure tried to execute *sh*. If it worked, the worm now had a shell running on the machine under attack.

Method 3 depended on a bug in the mail system, *sendmail*, which allowed the worm to mail a copy of the bootstrap and get it executed.

Once established, the worm tried to break user passwords. Morris did not have to do much research on how to accomplish this. All he had to do was ask his father, a security expert at the National Security Agency, the U.S. government's code breaking agency, for a reprint of a classic paper on the subject that Morris, Sr. and Ken Thompson wrote a decade earlier at Bell Labs (Morris and Thompson, 1979). Each broken password allowed the worm to log in on any machines the password's owner had accounts on.

Every time the worm gained access to a new machine, it checked to see if any other copies of the worm were already active there. If so, the new copy exited, except one time in seven it kept going, possibly in an attempt to keep the worm propagating even if the system administrator there started up his own version of the worm to fool the real worm. The use of 1 in 7 created far too many worms, and was the reason all the infected machines ground to a halt: they were infested with worms. If Morris had left this out and just exited whenever another worm was sighted, the worm would probably have gone undetected.

Morris was caught when one of his friends spoke with the *New York Times* computer reporter, John Markoff, and tried to convince Markoff that the incident was an accident, the worm was harmless, and the author was sorry. The friend inadvertently let slip that the perpetrator's login was *rtm*. Converting *rtm* into the owner's name was easy—all that Markoff had to do was to run *finger*. The next day the story was the lead on page one, even upstaging the presidential election three days later.

Morris was tried and convicted in federal court. He was sentenced to a fine of $10,000, 3 years probation, and 400 hours of community service. His legal costs probably exceeded $150,000. This sentence generated a great deal of controversy. Many in the computer community felt that he was a bright graduate student whose harmless prank had gotten out of control. Nothing in the worm suggested that Morris was trying to steal or damage anything. Others felt he was a serious criminal and should have gone to jail.

One permanent effect of this incident was the establishment of **CERT** (**Computer Emergency Response Team**), which provides a central place to report break-in attempts, and a group of experts to analyze security problems and design fixes. While this action was certainly a step forward, it also has its downside. CERT collects information about system flaws that can be attacked and how to fix them. Of necessity, it circulates this information widely to thousands of system administrators on the Internet. Unfortunately, the bad guys (possibly posing as system administrators) may also be able to get bug reports and exploit the loopholes in the hours (or even days) before they are closed.

9.5.6 Mobile Code

Viruses and worms are programs that get onto a computer without the owner's knowledge and against the owner's will. Sometimes, however, people more-or-less intentionally import and run foreign code on their machines. It usually happens like this. In the distant past (which, in the Internet world, means last year), most Web pages were just static HTML files with a few associated images. Nowadays, increasingly many Web pages contain small programs called **applets**. When a Web page containing applets is downloaded, the applets are fetched and executed. For example, an applet might contain a form to be filled out, plus interactive help in filling it out. When the form is filled out, it could be sent somewhere over the Internet for processing. Tax forms, customized product order forms, and many other kinds of forms could benefit from this approach.

Another example in which programs are shipped from one machine to another for execution on the destination machine are **agents**. These are programs that are launched by a user to perform some task and then report back. For example, an agent could be asked to check out some travel Web sites to find the cheapest flight from Amsterdam to San Francisco. Upon arriving at each site, the agent would run there, get the information it needs, then move on to the next Web site. When it was all done, it could come back home and report what it had learned.

A third example of mobile code is a PostScript file that is to be printed on a PostScript printer. A PostScript file is actually a program in the PostScript programming language that is executed inside the printer. It normally tells the printer to draw certain curves and then fill them in, but it can do anything else it wants to as well. Applets, agents, and PostScript files are just three examples of **mobile code**, but there are many others.

Given the long discussion about viruses and worms earlier, it should be clear that allowing foreign code to run on your machine is more than a wee bit risky. Nevertheless, some people do want to run these foreign programs, thus the question arises: ‘‘Can mobile code be run safely’’? The short answer is: ‘‘Yes, but not easily.’’ The fundamental problem is that when a process imports an applet or other mobile code into its address space and runs it, that code is running as part of a valid user process and has all the power that user has, including the ability to read, write, erase or encrypt the user's disk files, email data to far-away countries, and much more.

Long ago, operating systems developed the process concept to build walls between users. The idea is that each process has its own protected address space and own UID, allowing it to touch files and other resources belonging to it, but not to other users. For providing protection against one part of the process (the applet) and the rest, the process concept does not help. Threads allow multiple threads of control within a process, but do nothing to protect one thread against another one.

In theory, running each applet as a separate process helps a little, but is often infeasible. For example, a Web page may contain two or more applets that interact with each other and with the data on the Web page. The Web browser may also need to interact with the applets, starting and stopping them, feeding them data, and so on. If each applet is put in its own process, the whole thing will not work. Furthermore, putting an applet in its own address space does not make it any harder for the applet to steal or damage data. If anything, it is easier since nobody is watching in there.

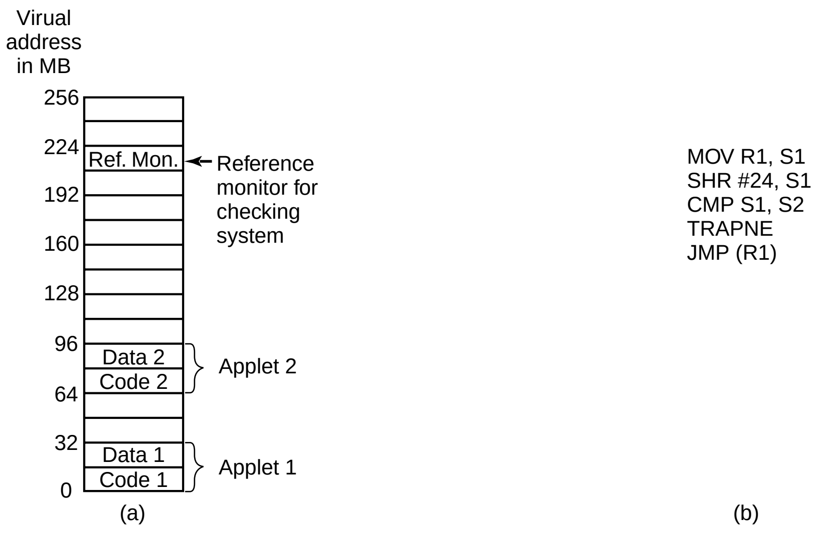
Various new methods of dealing with applets (and mobile code in general) have been proposed and implemented. Below we will look at three of these methods: sandboxing, interpretation, and code signing. Each one has its own strengths and weaknesses.

Sandboxing

The first method, called **sandboxing**, attempts to confine each applet to a limited range of virtual addresses enforced at run time (Wahbe et al., 1993). It works by dividing the virtual address space up into equal-size regions, which we will call sandboxes. Each sandbox must have the property that all of its addresses share some string of high-order bits. For a 32-bit address space, we could divide it up into 256 sandboxes on 16-MB boundaries so all addresses within a sandbox had a common upper 8 bits. Equally well, we could have 512 sandboxes on 8-MB boundaries, with each sandbox having a 9-bit address prefix. The sandbox size should be chosen to be large enough to hold the largest applet without wasting too much virtual address space. Physical memory is not an issue if demand paging is present, as it usually is. Each applet is given two sandboxes, one for the code and one for the data, as illustrated in for the case of 16 sandboxes of 16 MB each. Fig. 9-6(a),

The basic idea behind a sandbox is to guarantee that an applet cannot jump to code outside its code sandbox or reference data outside its data sandbox. The reason for having two sandboxes is to prevent an applet from modifying its code during execution to get around these restrictions. By preventing all stores into the code sandbox, we eliminate the danger of self-modifying code. As long as an applet is confined this way, it cannot damage the browser or other applets, plant viruses in memory, or otherwise do any damage to memory.

As soon as an applet is loaded, it is relocated to begin at the start of its sandbox. Then checks are made to see if code and data references are confined to the appropriate sandbox. In the discussion below, we will just look at code references (i.e., JMP and CALL instructions), but the same story holds for data references as well. Static JMP instructions that use direct addressing are easy to check: does



```
MOV R1, S1
SHR #24, S1
CMP S1, S2
TRAPNE
JMP (R1)
```

Figure 9-6. (a) Memory divided into 16-MB sandboxes. (b) One way of checking an instruction for validity.

the target address land within the boundaries of the code sandbox? Similarly, relative JMPs are also easy to check. If the applet has code that tries to leave the code sandbox, it is rejected and not executed. Similarly, attempts to touch data outside the data sandbox cause the applet to be rejected.

The hard part is dynamic JMPs. Most machines have an instruction in which the address to jump to is computed at run time, put in a register, and then jumped to indirectly, for example by JMP (R1) to jump to the address held in register 1. The validity of such instructions must be checked at run time. This is done by inserting code directly before the indirect jump to test the target address. An example of such a test is shown in Fig. 9-6(b). Remember that all valid addresses have the same upper *k* bits, so this prefix can be stored in a scratch register, say S2. Such a register cannot be used by the applet itself, which may require rewriting it to avoid this register.

The code works as follows: First the target address under inspection is copied to a scratch register, S1. Then this register is shifted right precisely the correct number of bits to isolate the common prefix in S1. Next the isolated prefix is compared to the correct prefix initially loaded into S2. If they do not match, a trap occurs and the applet is killed. This code sequence requires four instructions and two scratch registers.

Patching the binary program during execution requires some work, but it is doable. It would be simpler if the applet were presented in source form and then compiled locally using a trusted compiler that automatically checked the static addresses and inserted code to verify the dynamic ones during execution. Either way, there is some run-time overhead associated with the dynamic checks. Wahbe et al. (1993) have measured this as about 4%, which is generally acceptable.

A second problem that must be solved is what happens when an applet tries to make a system call? The solution here is straightforward. The system call instruction is replaced by a call to a special module called a **reference monitor** on the same pass that the dynamic address checks are inserted (or, if the source code is available, by linking with a special library that calls the reference monitor instead of making system calls). Either way, the reference monitor examines each attempted call and decides if it is safe to perform. If the call is deemed acceptable, such as writing a temporary file in a designated scratch directory, the call is allowed to proceed. If the call is known to be dangerous or the reference monitor cannot tell, the applet is killed. If the reference monitor can tell which applet called it, a single reference monitor somewhere in memory can handle the requests from all applets. The reference monitor normally learns about the permissions from a configuration file.

Interpretation

The second way to run untrusted applets is to run them interpretively and not let them get actual control of the hardware. This is the approach used by Web browsers. Web page applets are commonly written in Java, which is a normal programming language, or in a high-level scripting language such as safe-TCL or Javascript. Java applets are first compiled to a virtual stack-oriented machine language called **JVM** (**Java Virtual Machine**). It is these JVM applets that are put on the Web page. When they are downloaded, they are inserted into a JVM interpreter inside the browser as illustrated in Fig. 9-7.

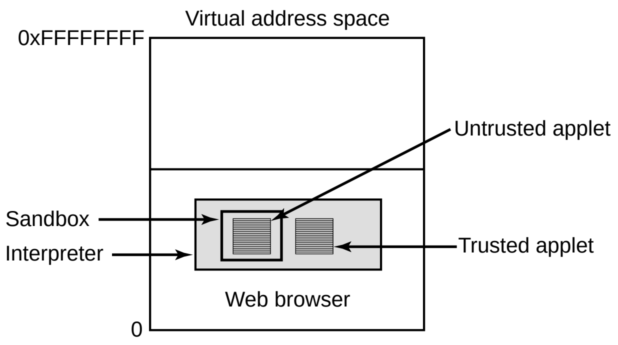


Figure 9-7. Applets can be interpreted by a Web browser.

The advantage of running interpreted code over compiled code, is that every instruction is examined by the interpreter before being executed. This gives the interpreter the opportunity to check if the address is valid. In addition, system calls are also caught and interpreted. How these calls are handled is a matter of the security policy. For example, if an applet is trusted (e.g., it came from the local disk), its system calls could be carried out without question. However, if an applet is not trusted (e.g., it came in over the Internet), it could be put in what is effectively a sandbox to restrict its behavior.

High-level scripting languages can also be interpreted. Here no machine addresses are used, so there is no danger of a script trying to access memory in an impermissible way. The downside of interpretation in general is that it is very slow compared to running native compiled code.

Code

Yet another way to deal with applet security is to know where they came from and only accept applets from trusted sources. With this approach, a user can maintain a list of trusted applet vendors and only run applets from those vendors. Applets from all other sources are rejected as too dicey. In this approach, no actual security mechanisms are present at run time. Applets from trustworthy vendors are run as is and code from other vendors is not run at all or in a restricted way (sandboxed or interpreted with little or no access to user files and other system resources).

To make this scheme work, as a minimum, there has to be a way for a user to determine that an applet was written by a trustworthy vendor and not modified by anyone after being produced. This is done using a digital signature, which allows the vendor to sign the applet in such a way that future modifications can be detected.

Code signing is based on public-key cryptography. An applet vendor, typically a software company, generates a (public key, private key) pair, making the former public and zealously guarding the latter. To sign an applet, the vendor first computes a hash function of the applet to get a 128-bit or 160-bit number, depending on whether MD5 or SHA is used. It then signs the hash value by encrypting it with its private key (actually, decrypting it using the notation of Fig. 9-0). This signature accompanies the applet wherever it goes.

When the user gets the applet, the browser computes the hash function itself. It then decrypts the accompanying signature using the vendor's public key and compares what the vendor claims the hash function is with what the browser itself computed. If they agree, the applet is accepted as genuine. Otherwise it is rejected as a forgery. The mathematics involved makes it exceedingly difficult for anyone to tamper with the applet in such a way as its hash function will match the hash function that is obtained by decrypting the genuine signature. It is equally difficult to generate a new false signature that matches without having the private key. The process of signing and verifying is illustrated in Fig. 9-8.

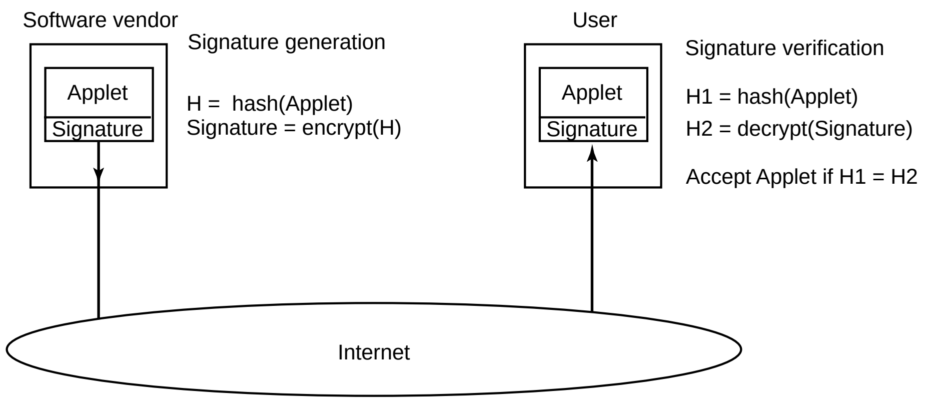


Figure 9-8. How code signing works.

9.5.7 Java Security

The Java programming language and accompanying run-time system were designed to allow a program to be written and compiled once and then shipped over the Internet in binary form and run on any machine supporting Java. Security was a part of the Java design from the beginning. In this section we will describe how it works.

Java is a type-safe language, meaning that the compiler will reject any attempt to use a variable in a way not compatible with its type. In contrast, consider the following C code:

```
naughty_func()
{
    char *p;
    p = rand();
    *p = 0;
}
```

It generates a random number and stores it in the pointer *p*. Then it stores a 0 byte at the address contained in *p*, overwriting whatever was there, code or data. In Java, constructions that mix types like this are forbidden by the grammar. In addition, Java has no pointer variables, casts, user-controlled storage allocation (such as *malloc* and *free*) and all array references are checked at run time.

Java programs are compiled to an intermediate binary code called **JVM** (**Java Virtual Machine**) **byte code**. JVM has about 100 instructions, most of which push objects of a specific type onto the stack, pop them from the stack, or combine two items on the stack arithmetically. These JVM programs are typically interpreted, although in some cases they can be compiled into machine language for faster execution. In the Java model, applets sent over the Internet for remote execution are JVM programs.

When an applet arrives, it is run through a JVM byte code verifier that checks if the applet obeys certain rules. A properly compiled applet will automatically obey them, but there is nothing to prevent a malicious user from writing a JVM applet in JVM assembly language. The checks include

1. Does the applet attempt to forge pointers?
2. Does it violate access restrictions on private class members?
3. Does it try to use a variable of one type as another type?
4. Does it generate stack overflows or underflows?
5. Does it illegally convert variables of one type to another?

If the applet passes all the tests, it can be safely run without fear that it will access memory other than its own.

However, applets can still make system calls by calling Java methods (procedures) provided for that purpose. The way Java deals with that has evolved over time. In the first version of Java, **JDK** (**Java Development Kit**) **1.0**. applets were divided into two classes: trusted and untrusted. Applets fetched from the local disk were trusted and allowed to make any system calls they wanted. In contrast, applets fetched over the Internet were untrusted. They were run in a sandbox, as shown in Fig. 9-7, and allowed to do practically nothing.

After some experience with this model, Sun decided that it was too restrictive. In JDK 1.1, code signing was employed. When an applet arrived over the Internet, a check was made to see if it was signed by a person or organization the user trusted (as defined by the user's list of trusted signers). If so, the applet was allowed to do whatever it wanted. If not, it was run in a sandbox and severely restricted.

After more experience, this proved unsatisfactory as well, so the security model was changed again. JDK 1.2 provides a configurable fine-grain security policy that applies to all applets, both local and remote. The security model is complicated enough that an entire book can be written describing it (Gong, 1999), so we will just briefly summarize some of the highlights.

Each applet is characterized by two things: where it came from and who signed it. Where it came from is its URL; who signed it is which private key was used for the signature. Each user can create a security policy consisting of a list of rules. A rule may list a URL, a signer, an object, and an action that the applet may perform on the object if the applet's URL and signer match the rule. Conceptually, the information provided is shown in the table of Fig. 9-9, although the actual formatting is different and is related to the Java class hierarchy.

One kind of action permits file access. The action can specify a specific file or directory, the set of all files in a given directory, or the set of all files and directories recursively contained in a given directory. The three lines of Fig. 9-9

URL	Signer	Object	Action
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/–	Read, Write, Delete

Figure 9-9. Some examples of protection that can be specified with JDK 1.2.

correspond to these three cases. In the first line, the user, Susan, has set up her permissions file so that applets originating at her tax preparer's machine, *www.taxprep.com* and signed by the company, have read access to her tax data located in the file *1040.xls*. This is the only file they can read and no other applets can read this file. In addition, all applets from all sources, whether signed or not, can read and write files in */usr/tmp*.

Furthermore, Susan also trusts Microsoft enough to allow applets originating at its site and signed by Microsoft to read, write, and delete all the files below the *Office* directory in the directory tree, for example, to fix bugs and install new versions of the software. To verify the signatures, Susan must either have the necessary public keys on her disk or must acquire them dynamically, for example in the form of a certificate signed by a company she trusts and whose public key she already has.

Files are not the only resources that can be protected. Network access can also be protected. The objects here are specific ports on specific computers. A computer is specified by an IP address or DNS name; ports on that machine are specified by a range of numbers. The possible actions include asking to connect to the remote computer and accepting connections originated by the remote computer. In this way, an applet can be given network access, but restricted to talking only to computers explicitly named in the permissions list. Applets may dynamically load additional code (classes) as needed, but user-supplied class loaders can precisely control on which machines such classes may originate. Numerous other security features are also present.