Chapter 7
Trojan Horses and Covert Channels

When people began thinking about making systems more secure, they naturally speculated about specific penetration techniques. At first, the approach to securing operating systems was directed toward closing the holes inadvertently left by designers. These holes typically allowed a penetrator to gain control of the operating system, or at least to bypass some particular access control mechanism. Some penetration techniques identified by "tiger teams" searching for holes were incredibly complex, as were the countermeasures.

The Trojan horse route to penetration, however, was not formally identified until surprisingly late in the history of computing.[1] This route was far easier to exploit than many of the highly sophisticated penetrations people were trying to thwart. Worse, this simple type of penetration was fundamentally impossible to prevent on nearly all systems. Only a complete change in the philosophy of protection and a complete restructuring of the system could come close to addressing the problem. The most insidious aspect of the Trojan horse attack is that it requires no discovery and exploitation of loopholes in the operating system. A successful Trojan horse attack can be mounted through the use of only the most well-documented and obviously desirable features of a flawless, bug-free system.

Do not assume that the Trojan horse problem is so esoteric that it only applies to computers entrusted with military secrets. Once you understand how easy it is to carry out a Trojan horse attack, you may wonder why anyone should have any confidence in the safety of any information in their system, why more systems are not constantly being penetrated, and why you should bother to close every small hole in your system while leaving gaping Trojan horse holes that are so easy to exploit.

One sentence can explain what a Trojan horse is, but chapters are needed to cover all the implications. Many who initially think they understand the Trojan horse are surprised when confronted with its ramifications. If you explain the Trojan horse problem to the management of a large computer installation, the likely response you will receive is "we don't have that problem here, because..." But if you ask about that installation's existing security controls, you will usually find multiple redundant measures strengthening "conventional" aspects of the system while leaving wide-open paths for a Trojan horse attack. After such a discussion, you might be able to convince the management that many of the controls in these conventional areas serve only to reinforce the iron links in a paper chain.

7.1 TROJAN HORSES AND VIRUSES

Most references define the Trojan horse in one or two sentences. A Trojan horse is a computer program that appears to the user to perform a legitimate function but in fact carries out some illicit

1. The term Trojan horse was first used by Dan Edwards (Anderson 1972).

function that the user of the program did not intend. The victim is the user of the program; the perpetrator is the program's developer.

We can identify several key requirements for launching a successful Trojan horse attack:

- You (the perpetrator) must write a program (or modify an existing program) to perform the illicit act in a way that does not arouse the suspicion of any future user of the program. The program should perform some interesting or useful function that will entice others to use it.
- You must have some way of making the program accessible to your victim–by allowing the victim access to the program, by installing it in a system library (which could require help from an honest but gullible system administrator), or by physically handing the victim a tape or disk.
- You must get the victim to run your program. This might happen incidentally (if your program replaces an existing program that the victim normally uses) or intentionally (if your program is directly invoked by the victim).
- You must have some way to reap the benefits of the illicit act. If the act is to copy private information (our primary concern), then you have to provide a repository for it that you can later access. This is normally quite easy if you have an account on the victim's system.

A special type of Trojan horse that propagates itself through a system or network of systems is the *virus* (Cohen 1984). "Infecting" a system with a virus usually requires a high level of skill on the part of the perpetrator, but once installed it can cause a great deal of harm and may be particularly difficult to eliminate.

7.1.1 Trojan Horse Examples

In section 3.4.1 we discussed some simple examples of the Trojan horse threat. Following are a few more sophisticated examples. of both Trojan horses and viruses:

- A program that plays the game *Adventure* uses idle time when the user is thinking to scan the user's directory and give "world" read access to all the victim's files. You (the perpetrator) later log in normally and read the files. The victim might eventually find out that the access rights were changed, but may still have a hard time figuring out which program did it and whether anyone read the files.
- A new improved `list_directory` program that everyone wants to use functions as advertised but never exits upon completion. Instead,. it pretends to exit, mimicking the response of the system command processor. The program reads and processes the victim's further commands normally (possibly by invoking the real command processor for each command) and never reveals the fact that it is still there. When the user finally types `logout`, the program simulates a genuine logout, but does not really log out. The next time any user walks up to the terminal and types `login`, the program reads the user's name and password and discreetly sends you (the perpetrator) a message containing the user's password. Then the program mimics a normal login procedure and finally does exit, returning the user to the command processor. The user never knows that all the prior input has been monitored, and you now have the user's password.

- An *Adventure* game, copied by the user from a public bulletin board where you have placed it for free distribution, modifies the user's command search list to cause a search of one of your own directories before searching the system libraries. In all subsequent sessions, every time the user types a system command, any one of a number of Trojan horse programs in your directory may be invoked instead of, or in addition to, the desired system command. Once the search list is modified, you can get the victim to run any of your programs practically at will. One of these Trojan horse programs might be an altered version of `show-search-list` that hides from the user the fact that your directory is on the user's list. You would probably also want to include a doctored version of `modify_search_list` to prevent your own directory from being deleted from the list. This example shows that, with a little planning on your part, a single mistake by a user can result in permanent compromise of the user's security.
- You quietly place your Trojan horse in a public user directory, and give it an interesting name like *Superspreadsheet*, hoping some user will find it and try it. Besides operating as a spreadsheet, the program scans the user's directories, looking for executable binary files (other programs) that the user owns and appending a section of Trojan horse code to each such file. It modifies the calling sequence in those files to transfer temporary control to the Trojan horse each time one of those programs is called. When one of those programs is later used–possibly by a different user–the Trojan horse scans that user's directories, looking for more files to append itself to. Of course, the operation of the programs modified by this Trojan horse is not visibly affected. On a system where many users share each other's programs, this virus will quickly infect most of the user software in the system. If system programmers or administrators ever use someone else's programs, the virus can infect system programs as well. Since nobody ever looks at object code to see if it matches compiled code, this virus is unlikely to be detected as long as it does no visible harm.

  Your hope is that someone on a compiler development team will use a program infected with your virus; your virus is designed to recognize when it is appended to the compiler, and it will thereafter cause the compiler to append the virus to all compiled programs automatically. In this way, recompiling a program will not eliminate the virus.

  This virus causes no functional harm to the operating system other than using up a little memory along with each executable program. You can use your imagination to decide what additional features an interesting virus might have.

  A primitive type of virus was installed as a penetration exercise on an early version of Honeywell’s Multics (Karger and Schell 1974).

As you can see, the illicit activity of the Trojan horse or virus need not hamper or frustrate the legitimate function of the command in which is embedded, although the simplest Trojan horses might just go after the information the particular command already uses. The best Trojan horses do their dirty work and leave no traces. Modifying the access rights to all the user's files can be very damaging, but it is also easily detected and potentially traceable to the program that caused it. Trojan horses that persist indefinitely (like the virus) can cause a great deal of harm while they exist, but a program that causes trouble has a chance of being detected eventually. A clever Trojan horse might even be programmed to delete itself if the user tries to do something that might reveal its presence. Because most systems keep track of logins, stealing and using a password is unlikely to work more than a few times before the penetration is detected (although password theft is probably the easiest route to computer crime and can certainly cause a great deal of damage).

The common goal in these examples is to allow you (the perpetrator) to read a user's information to which you have no access. The Trojan horse either copies the information into one of your files, or sets up access modes so that you can later read the information directly from the user's files. The success of the Trojan horse depends on the extent to which you can retrieve the information.

So far we have not directly talked about Trojan horses that delete, modify, or damage information. A Trojan horse or virus whose goal is to modify files can do its job without your having to log in. In fact, you do not need to have any access to the user's system at all (provided that you had some way of giving the program to the user in the first place). A write-only Trojan horse used unknowingly by a system administrator and acting to modify a system file can be particularly insidious. In keeping with the general philosophy of this book that computer security is primarily concerned with information disclosure, we will continue to think of the Trojan horse as a means of illicitly obtaining read access to information. Although the write-only Trojan horse attack is somewhat simpler to carry out, solutions to the Trojan horse information disclosure problem (to the extent that they are solutions) generally address the information modification problem, as well.

7.1.2 Limiting the Trojan Horse

Preventing a Trojan horse from doing its damage is fundamentally impossible without some mandatory controls, and keeping a Trojan horse out of your system is extremely difficult. While simple or special-purpose systems might be protected to a degree, no general-purpose system can be protected adequately. A few of the techniques discussed here can reduce the possibility of a successful Trojan horse attack; but these techniques are somewhat dangerous, in that they can give you a false sense of security. Before adopting any of them, therefore, be sure you understand their limitations.

Restricting Access Control Flexibility

As was discussed in section 6.2.5, a Trojan horse can defeat any type of discretionary access control mechanism. As long as it is possible for the legitimate user to write a program that alters access control information for his or her own files, it is possible for a Trojan horse invoked by that user to do the same. Since the ability to write programs that alter access control information is a feature of most modern systems, it is difficult to imagine anyone being willing to eliminate this ability for the sake of security.

But suppose we do build a system that provides no unprivileged subroutine interface to the access control mechanism. In such a system, the only way for a user to specify access control information is by invoking a privileged system utility that sets the information based on input from the user’s terminal–not on input from another program. (This utility program would have to make sure it was really reading input from the terminal, and not from a command file, for example.)

Since we trust users not to give their own files away, it might seem that the Trojan horse threat to discretionary access control could thus be eliminated.

Notice, however, that several of the examples in section 7.1.1 do not require the Trojan horse to alter any access control information. For a Trojan horse to copy a user's files into the perpetrator's directory, the system need only allow the perpetrator to create a file manually that is writeable by the unsuspecting user. To avoid suspicion, the perpetrator might create a file that is writable by anyone, rather than solely by the specific user being targeted.

Let us then go further and mandate that the system not allow anyone to create a world-writable file (which is not a particularly useful feature anyway). In that. case the Trojan horse might use a mail utility or an interprocess message to communicate information. If these facilities do not exist either, the Trojan horse might find a world-readable file belonging to the user and store the information in it. No one could reasonably suggest that a system not allow a user to create world-readable files.

These examples should convince you that, except in very limited systems, it is usually not fruitful to try to prevent a Trojan horse attack by limiting the ways in which users can exchange information.

Procedural Controls

Within a general-purpose operating system, nobody has come up with a practical scheme for detecting a Trojan horse. If the system allows any user programming at all, there is no way to prevent a user from implementing a Trojan horse and convincing another person to use it. As used here, the term *programming* includes the ability to write command files, macros, and any other instructions that enable a user to cause things to happen outside the user's direct control.

Procedurally, however, users can be warned not to run any program: other than those in the system libraries, and they can be cautioned not to carry out any action that might accidentally invoke a “foreign” file in their directory as a command or program. Users need not be prevented from writing their own programs for their own use (because it would be pointless for a user to plant a Trojan horse in his or her own program), but users should be suspicious about any program that someone else has written. The effectiveness of such voluntary restrictions depends, of course, on the dedication of the users. The interesting aspect of such restrictions is that users are only protecting themselves (and information entrusted to them): one user's violating a voluntary restriction against using an outside program will not compromise any other user'; private information.

Unfortunately, voluntary restrictions are highly unreliable. Even sophisticated users may inadvertently violate the rules or be misled into doing so. In our earlier example where the search list was modified, one-time, possibly accidental use of a Trojan horse renders the user permanently vulnerable thereafter. The difficulties of the voluntary approach are exacerbated by the fact that those who would build a Trojan horse are not restricted. One can imagine an open system in which scores of users litter the system with Trojan horses in the hope that one of a handful of honest and careful users might one day make a mistake and type the wrong command name. In a multiuser system that allows data-sharing, there is no practical way to prevent program sharing.

In contrast to voluntary restrictions, enforced restrictions can be more nearly foolproof. In one approach (Karger 1987), a trusted mechanism in the system prevents programs called by a user from accessing files other than those intended by the user based on predefined usage pattern; of each program that the user calls. The Trojan horse can still damage the files it is legitimately given, but it cannot access additional file; without the user's knowledge. While such techniques are an interesting possibility, none has yet been implemented in practice.

System Controls: No Programming

Clearly the best restrictions are ones that the system automatically enforces. Limiting sharing is not practical, so the only restriction left involves programming.
Eliminating user programming might at first seem fairly easy: just get rid of all the compilers, assemblers, interpreters, and similar applications. In fact, many systems on which users do not need to write programs are operated this way. But if the system has a text editor and a command language, the ability to write command procedures (both batch and interactive) must also be eliminated, either by changing the command processor or by getting rid of all text editors. A DBMS that allows users to store complex queries as procedures for later access must be eliminated or restricted. Even without a command processor or DBMS, many text-processing tools such as editors and formatters are practically programming languages in their own right; these would have to be eliminated, too. (Remember that a successful Trojan horse might be as simple as a 1-line copy command embedded in an editor macro.) Even spreadsheet programs have features for user programmability.

By the time you eliminate all possibility of writing any type of program on a system, you have probably limited the use of the system to a few very specialized applications. Certainly no general-purpose system can be operated that way. But many large systems are in fact special purpose and need no kind of programmability. Large organizations such as airlines and banks use their operational computers solely for transaction processing, with separate computers for development. But even when the operational system has no need for programming, it is rare for designers to make more than half-hearted efforts to eliminate the ability to write programs. Usually such efforts are aimed at saving memory and storage rather than at increasing security.

It is frequently argued that even the best efforts at eliminating programming are doomed. After all, any system on a network is a potential recipient of a Trojan horse from another system that does allow programming. Moreover, Trojan horses need not always resemble a program. A list of financial transactions could contain a Trojan horse in the form of illicit transactions. But, while it is indeed very difficult (or perhaps impossible) to guarantee that no Trojan horse has entered the system, the guarantee need not be absolute. Through a systematic analysis of all possible paths into the system, it is possible to weigh the effort a penetrator must make to install a Trojan horse against the value of the information gained or damage done. A partial closing of such paths (which, to be of practical benefit, must still be relatively complete) is adequate in many cases.

Scrutinizing Vendor Software

One route to installing a Trojan horse that we have not considered is via the vendor of the software. Most organizations certainly trust their vendors not to plant Trojan horses (although rumors are not lacking about features such as time bombs that inactivate the software when the rental period expires). Indeed, prior to initial purchase of a software package, there is little reason for an organization to fear that there might be a Trojan horse in the software specifically targeted at that organization. Once the software is installed, however, a site with very sensitive data has good reason to fear updates to that software supplied by the vendor-not because the vendor is likely to be malicious, but because the vendor probably has no more control over the actions of its employees than the organization has over its. Imagining a scenario where a disgruntled employee quits an organization to work as a programmer for a vendor that supplies the organization with software is not difficult. Unless appropriate control is maintained over the acquisition of new or updated vendor software, the value of closing all other Trojan horse channels is limited.

Probably the only practical technique for screening vendor software—a method used by the government at certain highly secure installations—is to accept software updates from a vendor only in the form of source code, to be scrutinized manually for malicious code by site personnel and to be compiled locally. Programs that highlight only the differences between earlier and later versions of the source code are used as an aid. This technique, though laborious, is considered useful because of the assumption that a Trojan horse in source code is easy to spot. Nonetheless, a clever programmer might be able to hide a Trojan horse, especially within a complex program. Rather than providing 100 percent assurance, the technique of scrutinizing the source code probably only serves as a deterrent to penetrators by increasing the work required to hide a Trojan horse.

Mandatory Controls

As was stated in section 6.3, the only effective way to handle the Trojan horse threat is to use mandatory access controls. Under mandatory access controls, a Trojan horse is prevented from giving away information in a way that would violate the mandatory access restrictions. Consider, for example, the multilevel security model discussed in section 6.4.4 and illustrated in figure 6.3. The confinement property prevents a Trojan horse in a process running at the SECRET access class from writing SECRET information into an UNCLASSIFIED file. Everything writable by a SECRET process must have at least a SECRET access class.

It is important to remember that mandatory controls only thwart Trojan horse attacks that attempt to cross mandatory access class boundaries. The Trojan horse in our example can still bypass discretionary rules by copying information from the victim's SECRET file into another user's SECRET file. Since it is impractical to assign a different mandatory access class to each user, mandatory controls are only used to protect information that is more sensitive than information that is simply private to a single user.

For example, suppose that a corporation allows its competitors to buy time on its computer system. Corporate proprietary information in that system is assigned a mandatory access category, and only employees of the corporation are given access to that category. A Trojan horse used by one of those employees will not be able to pass information to competitors outside the category, but it will be free to transfer information among users within the category.

7.2 COVERT CHANNELS

A key notion behind the Trojan horse attack is illicit communication through a legitimate information channel intended for interprocess communication: a file, an interprocess message, or shared memory. Mandatory access controls can prevent such communication across access classes. But a system usually allows processes to communicate in numerous other ways. that are not normally used for communication and are not normally protected by mandatory controls. We call these other paths covert information channels, or simply covert channels (Lampson 1973; Lipner 1975).

Covert channels have also been called leakage paths because information can escape unintentionally. People worry about leakage paths because it is impossible to predict how much information an errant program might leak through such a channel. The practical impact of unintentional leakage, however, is usually minor and not a primary concern to us; much more serious is the intentional leakage caused by a Trojan horse.

Systems abound with covert channels. Every bit of information in the system (that is, every object) that can be modified by one process and read by another–directly or indirectly–is potentially a covert channel. Where mandatory controls prevent a Trojan horse from communicating information through files and other conventional objects, any bit of information not protected by mandatory controls is potentially an alternate path.

A covert channel's most important parameter is its bandwidth–the rate, in bits per second, at which information can be communicated between processes. This bandwidth is a function of the number of bits in the object and of performance characteristics of the system that determine the rate at which the object can be changed or modulated.

There are two types of covert channels: a storage channel is any communication path that results when one process causes an object to be written and another process observes the effect; a timing channel is any communication path that results when a process produces some effect on system performance that is observable by another process and is measurable with a timing base such as a real-time clock.

7.2.1 Covert Storage Channels

Covert storage channels use three types of information:

- Object attributes
- Object existence
- Shared resources

Object Attributes

The easiest-to-use and most common storage channels in systems are usually file names. A 32-character file name can be changed by one process and read by another process, resulting in a 32-character message transfer between the processes even if the file itself is not readable or writable by the processes. This channel can usually be eliminated by designing the access controls so that file names are objects protected by mandatory access controls in the same manner as the files are.

The use of file names is one example of the use of file attributes as storage channels. File attributes are items of information about a file that the operating system maintains in addition to the data in the file. Examples of other file attributes include length, format, date modified, and discretionary access control lists. The file attributes may be directly readable (as are file names), or their values may be indirectly inferred. Unlike file names, however, the values of most attributes are not directly modifiable by a process, and communicating via the attributes requires encoding the message to be sent in a form that uses the legal range of values of those attributes. For a process to change the file length, for example, the process may have to rewrite part of the file. This file length channel is limited to communicating a relatively small number of bits at a time, depending on the range of possible lengths. Changing the file format might be easy and direct, but the formats possible might be very few, leading to a rather narrow channel. Surprisingly, the access control list often provides one of the largest covert storage channels, since the list may be quite long and there might be few restrictions on the format of the user names on the list (see section 6.2 and figure 6.1). The values of the date and time when a file was last modified are usually difficult to control with any precision. The operating system usually updates the date and time at relatively long intervals, and the value may be no more accurate than to the nearest second. The bandwidth of such a channel can be no greater than one bit every 2 seconds; nonetheless, over a long period of time, an undetected Trojan horse can patiently transmit a significant amount of information by modifying a file at specific intervals.

Object Existence

File attributes are storage objects that are indirectly writable. Storage channels also include any items of information about the file that can be deduced by a process. For example, the fact that a given file exists is a bit of information; and even if you have no access to any of a file's attributes, you may still be able to infer whether a particular file exists. A simple way to do so would be to try to access the file and check the returned status condition. Some systems obligingly tell you whether your problem is `file does not exist` or `you have no access to the file`. If the system can support ten file creations or deletions per second, the Trojan horse can communicate ten bits of information per second.

If the system does not tell you directly whether a file inaccessible to you exists, you might try to create a new file with the same name as that file. If the system gives you a `nameduplication` or other error, you will have confirmed that the file already exists. If the system allows you to create and use the new file, you will have established that the file did not previously exist.

The single bit of information about existence of a file may not seem like much information, but some systems strive to provide high-speed file creation and deletion. Thus, though the information channel is narrow, its bandwidth can be high, especially if multiple files are used.

Shared Resources

The use of file existence as a one-bit covert storage channel is an example of a more general single-bit channel involving shared or global resources. Almost every system contains certain resources that are pooled among a number of active processes or users. Such resources include disk blocks, physical memory, I/O buffers, allocated I/O devices, and various queues for shared devices such as printers and plotters. Without per-process quotas, these types of shared resources can be consumed by a single process. For example, one process could submit so many print jobs that the printer queue fills up. When that happens, other processes on the system simply receive some kind of error condition when they try to submit a job. A one-bit channel exists between the sending process that fills the queue and the receiving process that gets the error message. The sending process can transmit multiple bits in a serial fashion by alternately submitting and then canceling the last job on the queue. Some systems tell a process how many total jobs there are on a printer queue; communication via the queue is then easy and does not require filling the whole queue, and the information about the total number of jobs provides a channel that is wider than a single bit.

One way to minimize the queue overflow channel (or any shared resource exhaustion channel) is to use a per-process quota. In our printer queue example, a limit could be imposed on the number of jobs that any one process might place on the queue. If the system guarantees that a process will always be able to submit jobs up to its quota, then for all practical purposes the queue appears to each process as a private, queue, revealing no information about other processes' jobs on the queue. But a queue structured in this way is not actually a shared queue, and all of the benefits of resource sharing are eliminated when resources are statically allocated to each process. Nonetheless, static allocation is often necessary to ensure complete. closure of certain high-bandwidth shared-resource covert channels.

A way to reduce the bandwidth of resource exhaustion channels is to limit the rate at which a process can discover that the resource is y exhausted. Usually a process cannot directly ask how much of a shared resource is available. The only way it can determine how much space is on a printer queue is to see how many jobs it can submit to the queue. When the process reaches the end of the queue, the system can delay the process for a certain amount of time before allowing it to attempt to put additional jobs on the queue. Since it is highly abnormal for a process to constantly bang away at the end of a queue, delaying a process trying to do so–even for several seconds-is unlikely to affect the performance of any legitimate operation.

One problem with such a bandwidth-limiting scheme is that the process may have access to many different shared resources. Therefore the limit must be based on the total number of resource exhaustion conditions that a process may be able to detect, not just on each resource individually. We also have to worry about the possibility that several processes can work in collusion, thereby multiplying the bandwidth by the number of processes.

Probably the simplest way to address the shared resource channel is to audit each case of resource exhaustion, in order to detect an excessive number of such cases within a given time interval. The threat of audit and detection might well suffice to deter a penetrator from using this technique. While auditing is usually not a reliable method of distinguishing between legitimate and illegitimate actions, resource exhaustion happens rarely enough that establishing a relatively low audit threshold (minimum number of incidents to trigger an audit) could be a valuable security measure.

7.2.2 Covert Timing Channels

Because the usefulness of covert storage channels is measured as a bandwidth, in bits per second, people often mistake certain types of storage channels for timing channels. In order for a covert channel to be classified as a timing channel, a real-time clock, interval timer, or the equivalent must be involved. The clock allows the receiving process to calculate relative amounts of real time between successive events. A channel that does not require a clock or timer is a storage channel. The distinction is important because, without any way for a process to determine the passage of time, a timing channel disappears. Storage channels, on the other hand, are not affected when access to a clock is eliminated.

A simple example of a timing channel is the percentage of CPU time available to a process. A Trojan horse in one process transmits 1’s and 0’s by using up varying fractions of CPU time at 1-second intervals in a busy loop. The receiving process reads the bits by counting the number of its own loops that it is able to perform in each interval. If these two processes are the only ones running on the machine, the receiving process's loop count in each second is a direct function of the sending process’s CPU utilization. The bandwidth of this channel depends on the range of values for the loop count that can be predictably communicated.

Timing channels tend to be noisy because they are affected by processes on the system other than the ones actually communicating. The noisier a channel is, the lower the effective bandwidth becomes; however, it is usually possible to minimize the noise caused by other processes by running late at night, when few other processes are running. An effective Trojan horse can choose the times it runs.

It is often suggested that timing channels be eliminated by removing the ability for a process to read a clock. Our example above does not work if the receiving process has no time reference. But even if the receiving process has no direct access to a clock, there are ways for it to determine passage of time. For example, the process can measure 0.1-second intervals by counting characters received from a terminal while the user (who is the penetrator on the receiving end) holds down a repeat key that enters characters at the fixed rate of 10 per second. The process may even be able to manufacture its own clock by counting the number of disk accesses it can make or the number of characters it can write to a terminal between specific events to be timed. On multiprocessor systems, one process can use program loops to determine time intervals on behalf of another process. Even if none of these techniques works, the user can always operate a stopwatch at his or her terminal and count the seconds between events.

Timing channels are insidious for two reasons: there are no formal techniques for finding them in a system; and there is usually no way to detect their use and hence to audit them. Whereas storage channels can often be countered by controlling the rate at which specific, identifiable objects in the system are modified, timing channels do not involve observation of any identifiable objects.

Computer security technology has little to offer those who wish to find and block timing channels. Computer security projects to date have failed, by and large, to address the problem in a systematic way. The best advice for planners designing a new system would be to understand the timing channel problem from the start of the system design and to be constantly aware of the threat. Most obvious channels are, uncovered during the design and development process. You cannot'; completely close many of the channels you find, but at least you will have a good idea of where they are and can deal with them on an individual basis.

At the current state of the art in secure operating systems, the timing channel is far more difficult for a penetrator to exploit than many other avenues. Perhaps someday, when these other routes are closed, we will have better solutions to the timing channel problem.

7.3 TRAP DOORS

The trap door (Karger and Schell 1974) is an illicit piece of software in an operating system that provides a way for a penetrator to break into the operating system reliably and without detection. The trap door is activated by a special command or unlikely sequence of events that the penetrator can cause at will and that no one else is likely to discover by accident. A trap door is only useful in software that runs with privileges that the penetrator does not otherwise have; otherwise, the trap door does not give the penetrator anything not already obtainable. For this reason, we usually think of trap doors in operating systems and not in applications.

A trap door is much like a bug in an operating system that permits a penetration. Indeed, a penetration might be necessary to install the trap door in the first place. A trap door may also be installed by a dishonest employee of the vendor of the operating system. The techniques for inserting trap doors are much like those for inserting Trojan horses, but they are more difficult to carry out in an operating system.

Unlike Trojan horses and covert channels, trap doors can only be installed by exploiting flaws in the operating system or by infiltrating the system's development team. Hence, trap doors can be avoided by employing the usual techniques for developing reliable trusted software: no special techniques are required.

REFERENCES

Anderson, J. P. 1972. “Computer Security Technology Planning Study.” ESD-TR-73-51, vols. 1 and 2. Hanscom AFB, Mass.: Air Force Electronic Systems Division. (Also available through Defense Technical Information Center, Alexandria, Va., DTIC AD-758206.)

The first study to document the government's computer security problem and the proposed solutions in the form of the reference monitor and the security kernel; now no longer useful as a primary technical reference, but historically significant.

Cohen, F. 1984. “Computer Viruses: Theory and Experiments.” In *Proceedings of the 7th National Computer Security Conference*, pp. 24063. Gaithersburg, Md.: National Bureau of Standards.

The term virus *was first introduced in this paper.*

Karger, P. A. 1987. “Limiting the Potential Damage of Discretionary Trojan Horses.” In *Proceedings of the 1987 Symposium on Security and Privacy*, pp. 32-37. Washington, D.C.: IEEE Computer Society.

Discusses a technique to limit discretionary Trojan horses on the basis of built-in knowledge of usage patterns, and provides a good overview of the problem and helpful references to related techniques.

Karger, P. A., and Schell, R. R. 1974. “Multics Security Evaluation: Vulnerability Analysis.” ESD-TR-74-193, vol. 2. Hanscom AFB, Mass.: Air Force Electronic Systems Division. (Also available through National Technical Information Service, Springfield, Va., NTIS AD-A001120.)

A discussion of penetrations of Multics, pointing out several classic types of flaws in various areas; useful as a guide to detecting flaws in other operating systems.

Lampson, B. W. 1973. “A Note on the Confinement Problem.” *Communications of the ACM* 16(10):613-15.

One of the first papers to discuss covert channels (called confinement *or* leakage paths*) and techniques for closing them.*

Lipner, S. B. 1975. “A Comment on the Confinement Problem.” *ACM Operating Systems Review* 9(5):192-96.

In response to Lampson's paper, this paper discusses some fundamental problems involved in attempting to close covert channels in a system with shared resources.