Chapter 3

Operating Systems Security

Contents

3.1 Operating Systems Concepts

An ***operating system*** (OS) provides the interface between the users of a computer and that computer's hardware. In particular, an operating system manages the ways applications access the resources in a computer, including its disk drives, CPU, main memory, input devices, output devices, and network interfaces. It is the "glue" that allows users and applications to interact with the hardware of a computer. Operating systems allow application developers to write programs without having to handle low-level details such as how to deal with every possible hardware device, like the hundreds of different kinds of printers that a user could possibly connect to his or her computer. Thus, operating systems allow application programs to be run by users in a relatively simple and consistent way.

Operating systems handle a staggering number of complex tasks, many of which are directly related to fundamental security problems. For example, operating systems must allow for multiple users with potentially different levels of access to the same computer. For instance, a university lab typically allows multiple users to access computer resources, with some of these users, for instance, being students, some being faculty, and some being administrators that maintain these computers. Each different type of user has potentially unique needs and rights with respect to computational resources, and it is the operating system's job to make sure these rights and needs are respected while also avoiding malicious activities.

In addition to allowing for multiple users, operating systems also allow multiple application programs to run at the same time, which is a concept known as ***multitasking***. This technique is extremely useful, of course, and not just because we often like to simultaneously listen to music, read email, and surf the Web on the same machine. Nevertheless, this ability has an implied security need of protecting each running application from interference by other, potentially malicious, applications. Moreover, applications running on the same computer, even if they are not running at the same time, might have access to shared resources, like the filesystem. Thus, the operating system should have measures in place so that applications can't maliciously or mistakenly damage resources needed by other applications.

These fundamental issues have shaped the development of operating systems over the last decades. In this chapter, we explore the topic of operating system security, studying how operating systems work, how they are attacked, and how they are protected. We begin our study by discussing some of the fundamental concepts present in operating systems.

3.1.1 The Kernel and Input/Output

The ***kernel*** is the core component of the operating system. It handles the management of low-level hardware resources, including memory, processors, and input/output (I/O) devices, such as a keyboard, mouse, or video display. Most operating systems define the tasks associated with the kernel in terms of a ***layer*** metaphor, with the hardware components, such as the CPU, memory, and input/output devices being on the bottom, and users and applications being on the top.

The operating system sits in the middle, split between its kernel, which sits just above the computer hardware, and nonessential operating system services (like the program that prints the items in a folder as pretty icons), which interface with the kernel. The exact implementation details of the kernel vary among different operating systems, and the amount of responsibility that should be placed on the kernel as opposed to other layers of the operating system has been a subject of much debate among experts. In any case, the kernel creates the environment in which ordinary programs, called ***userland*** applications, can run. (See Figure 3.1.)

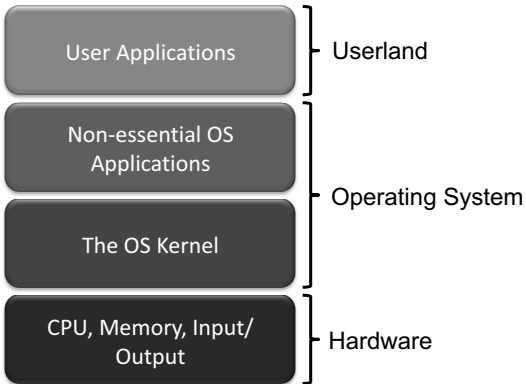


Figure 3.1: The layers of a computer system.

Input/Output Devices

The input/output devices of a computer include things like its keyboard, mouse, video display, and network card, as well as other more optional devices, like a scanner, Wi-Fi interface, video camera, USB ports, and other input/output ports. Each such device is represented in an operating system using a ***device driver***, which encapsulates the details of how interaction with that device should be done. The ***application programmer interface***

(*API*), which the device drivers present to application programs, allows those programs to interact with those devices at a fairly high level, while the operating system does the "heavy lifting" of performing the low-level interactions that make such devices actually work. We discuss some of the security issues related to input/output devices in the previous chapter (Section 2.4.2), including acoustic emissions and keyloggers, so we will instead focus here on the operating system calls that are needed to make input/output and other hardware interactions possible.

System Calls

Since user applications don't communicate directly with low-level hardware components, and instead delegate such tasks to the kernel, there must be a mechanism by which user applications can request the kernel to perform actions on their behalf. In fact, there are several such mechanisms, but one of the most common techniques is known as the ***system call***, or ***syscall*** for short. System calls are usually contained in a collection of programs, that is, a ***library*** such as the C library (libc), and they provide an interface that allows applications to use a predefined series of APIs that define the functions for communicating with the kernel. Examples of system calls include those for performing file I/O (open, close, read, write) and running application programs (exec). Specific implementation details for system calls depend on the processor architecture, but many systems implement system calls as ***software interrupts***—requests by the application for the processor to stop the current flow of execution and switch to a special handler for the interrupt. This process of switching to kernel mode as a result of an interrupt is commonly referred to as a ***trap***. System calls essentially create a bridge by which processes can safely facilitate communication between user and kernel space. Since moving into kernel space involves direct interaction with hardware, an operating system limits the ways and means that applications interact with its kernel, so as to provide both security and correctness.

3.1.2 Processes

The kernel defines the notion of a ***process***, which is an instance of a program that is currently executing. The actual contents of all programs are initially stored in persistent storage, such as a hard drive, but in order to actually be executed, the program must be loaded into random-access memory (RAM) and uniquely identified as a process. In this way, multiple copies of the same program can be run by having multiple processes initialized with

the same program code. For example, we could be running four different instances of a word processing program at the same time, each in a different window.

The kernel manages all running processes, giving each a fair share of the computer's CPU(s) so that the computer can execute the instructions for all currently running applications. This ***time slicing*** capability is, in fact, what makes multitasking possible. The operating system gives each running process a tiny slice of time to do some work, and then it moves on to the next process. Because each time slice is so small and the context switching between running processes happens so fast, all the active processes appear to be running at the same time to us humans (who process inputs at a much slower rate than computers).

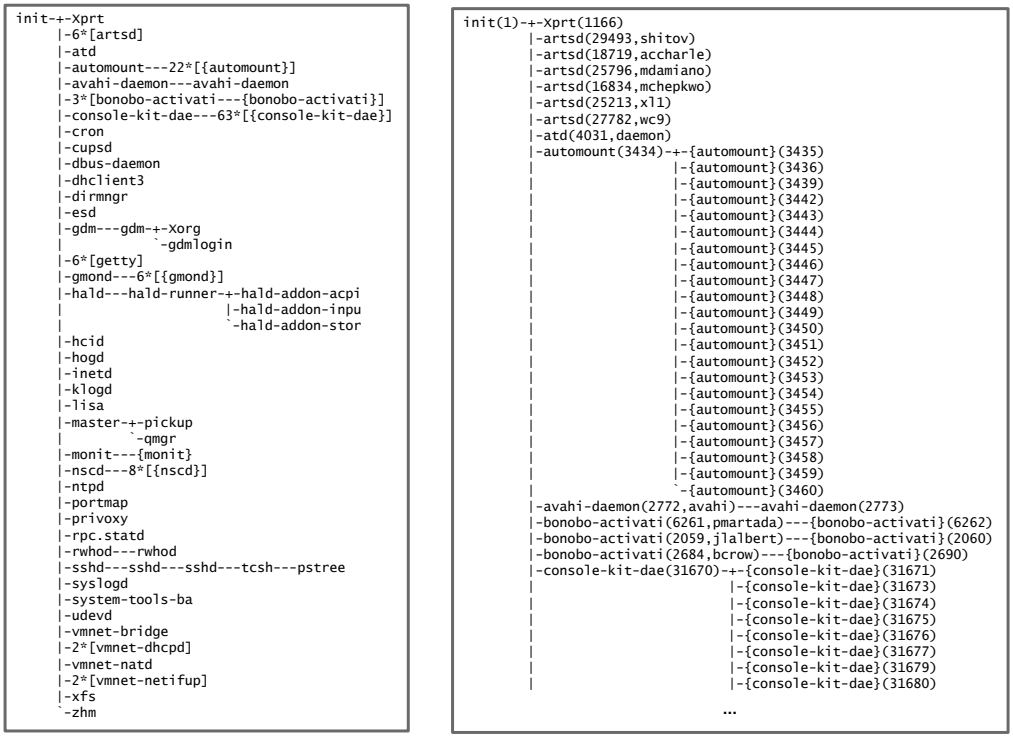
Users and the Process Tree

As mentioned above, most modern computer systems are designed to allow multiple users, each with potentially different privileges, to access the same computer and initiate processes. When a user creates a new process, say, by making a request to run some program, the kernel sees this as an existing process (such as a shell program or graphical user interface program) asking to create a new process. Thus, processes are created by a mechanism called ***forking***, where a new process is created (that is, ***forked***) by an existing process. The existing process in this action is known as the ***parent process*** and the one that that is being forked is known as the ***child process***.

On most systems, the new child process inherits the permissions of its parent, unless the parent deliberately forks a new child process with lower permissions than itself. Due to the forking mechanism for process creation, which defines parent-child relationships among processes, processes are organized in a rooted tree, known as the ***process tree***. In Linux, the root of this tree is the process `init`, which starts executing during the boot process right after the kernel is loaded and running. Process `init` forks off new processes for user login sessions and operating system tasks. Also, `init` becomes the parent of any "orphaned" process, whose parent has terminated.

Process IDs

Each process running on a given computer is identified by a unique non-negative integer, called the ***process ID*** (***PID***). In Linux, the root of the process tree is `init`, with PID 0. In Figure 3.2, we show an example of the process tree for a Linux system, in both a compact form and an expanded form.



```
init-+-Xprt
     |-6*[artsd]
     |-atd
     |-automount---22*[{automount}]
     |-avahi-daemon---avahi-daemon
     |-3*[bonobo-activati---{bonobo-activati}]
     |-console-kit-dae---63*[{console-kit-dae}]
     |-cron
     |-cupsd
     |-dbus-daemon
     |-dhclient3
     |-dirmngr
     |-esd
     |-gdm---gdm-+-Xorg
     |           `-gdmlogin
     |-6*[getty]
     |-gmond---6*[{gmond}]
     |-hald---hald-runner-+-hald-addon-acpi
     |                    |-hald-addon-inpu
     |                    `-hald-addon-stor
     |-hcid
     |-hogd
     |-inetd
     |-klogd
     |-lisa
     |-master-+-pickup
     |        `-qmgr
     |-monit---{monit}
     |-nscd---8*[{nscd}]
     |-ntpd
     |-portmap
     |-privoxy
     |-rpc.statd
     |-rwhod---rwhod
     |-sshd---sshd---sshd---tcsh---pstree
     |-syslogd
     |-system-tools-ba
     |-udevd
     |-vmnet-bridge
     |-2*[vmnet-dhcpd]
     |-vmnet-natd
     |-2*[vmnet-netifup]
     |-xfs
     `-zhm
```

(a)

```
init(1)-+-Xprt(1166)
        |-artsd(29493,shitov)
        |-artsd(18719,accharle)
        |-artsd(25796,mdamiano)
        |-artsd(16834,mchepkwo)
        |-artsd(25213,xll)
        |-artsd(27782,wc9)
        |-atd(4031,daemon)
        |-automount(3434)-+-{automount}(3435)
        |                 |-{automount}(3436)
        |                 |-{automount}(3439)
        |                 |-{automount}(3442)
        |                 |-{automount}(3443)
        |                 |-{automount}(3444)
        |                 |-{automount}(3445)
        |                 |-{automount}(3446)
        |                 |-{automount}(3447)
        |                 |-{automount}(3448)
        |                 |-{automount}(3449)
        |                 |-{automount}(3450)
        |                 |-{automount}(3451)
        |                 |-{automount}(3452)
        |                 |-{automount}(3453)
        |                 |-{automount}(3454)
        |                 |-{automount}(3455)
        |                 |-{automount}(3456)
        |                 |-{automount}(3457)
        |                 |-{automount}(3458)
        |                 |-{automount}(3459)
        |                 `-{automount}(3460)
        |-avahi-daemon(2772,avahi)---avahi-daemon(2773)
        |-bonobo-activati(6261,pmartada)---{bonobo-activati}(6262)
        |-bonobo-activati(2059,jlalbert)---{bonobo-activati}(2060)
        |-bonobo-activati(2684,bcrow)---{bonobo-activati}(2690)
        |-console-kit-dae(31670)-+-{console-kit-dae}(31671)
        |                        |-{console-kit-dae}(31673)
        |                        |-{console-kit-dae}(31674)
        |                        |-{console-kit-dae}(31675)
        |                        |-{console-kit-dae}(31676)
        |                        |-{console-kit-dae}(31677)
        |                        |-{console-kit-dae}(31679)
        |                        |-{console-kit-dae}(31680)
                                 ...
```

(b)

Figure 3.2: The tree of processes in a Linux system produced by the pstree command. The process tree is visualized by showing the root on the upper left-hand corner, with children and their descendants to the right of it. (a) Compact visualization where children associated with the same command are merged into one node. For example, 6*[artsd] indicates that there are six children process associated with artsd, a service that manages access to audio devices. (b) Fragment of the full visualization, which also includes process PIDs and users.

Process Privileges

To grant appropriate privileges to processes, an operating system associates information about the user on whose behalf the process is being executed with each process. For example, Unix-based systems have an ID system where each process has a ***user ID*** (***uid***), which identifies the user associated with this process, as well as a ***group ID*** (***gid***), which identifies a group of users for this process. The uid is a number between 0 and 32,767 (0x7fff in

hexadecimal notation) that uniquely identifies each user. Typically, uid 0 is reserved for the root (administrator) account. The gid is a number within the same range that identifies a group the user belongs to. Each group has a unique identifier, and an administrator can add users to groups to give them varying levels of access. These identifiers are used to determine what resources each process is able to access. Also, processes automatically inherit the permissions of their parent processes.

In addition to the uid and gid, processes in Unix-based systems also have an ***effective user ID*** (***euid***). In most cases, the euid is the same as the uid—the ID of the user executing the process. However, certain designated processes are run with their euid set to the ID of the application's owner, who may have higher privileges than the user running the process (this mechanism is discussed in more detail in Section 3.3.3). In these cases, the euid generally takes precedence in terms of deciding a process's privileges.

Inter-Process Communication

In order to manage shared resources, it is often necessary for processes to communicate with each other. Thus, operating systems usually include mechanisms to facilitate ***inter-process communication*** (***IPC***). One simple technique processes can use to communicate is to pass messages by reading and writing files. Files are are readily accessible to multiple processes as a part of a big shared resource—the filesystem—so communicating this way is simple. Even so, this approach proves to be inefficient. What if a process wishes to communicate with another more privately, without leaving evidence on disk that can be accessed by other processes? In addition, file handling typically involves reading from or writing to an external hard drive, which is often much slower than using RAM.

Another solution that allows for processes to communicate with each other is to have them share the same region of physical memory. Processes can use this mechanism to communicate with each other by passing messages via this shared RAM memory. As long as the kernel manages the shared and private memory spaces appropriately, this technique can allow for fast and efficient process communication.

Two additional solutions for process communication are known as ***pipes*** and ***sockets***. Both of these mechanisms essentially act as tunnels from one process to another. Communication using these mechanisms involves the sending and receiving processes to share the pipe or socket as an in-memory object. This sharing allows for fast messages, which are produced at one end of the pipe and consumed at the other, while actually being in RAM memory the entire time.

Signals

Sometimes, rather than communicating via shared memory or a shared communication channel, it is more convenient to have a means by which processes can send direct messages to each other asynchronously. Unix-based systems incorporate ***signals***, which are essentially notifications sent from one process to another. When a process receives a signal from another process, the operating system interrupts the current flow of execution of that process, and checks whether that process has an appropriate signal handler (a routine designed to trigger when a particular signal is received). If a signal handler exists, then that routine is executed; if the process does not handle this particular signal, then it takes a default action. Terminating a nonresponsive process on a Unix system is typically performed via signals. Typing Ctrl-C in a command-line window sends the INT signal to the process, which by default results in termination.

Remote Procedure Calls

Windows supports signals in its low-level libraries, but does not make use of them in practice. Instead of using signals, Windows relies on the other previously mentioned techniques and additional mechanisms known as ***remote procedure calls*** (***RPC***), which essentially allow a process to call a subroutine from another process's program. To terminate a process, Windows makes use of a kernel-level API appropriately named TerminateProcess(), which can be called by any process, and will only execute if the calling process has permission to kill the specified target.

Daemons and Services

Computers today run dozens of processes that run without any user intervention. In Linux terminology, these background processes are known as ***daemons***, and are essentially indistinguishable from any other process. They are typically started by the init process and operate with varying levels of permissions. Because they are forked before the user is authenticated, they are able to run with higher permissions than any user, and survive the end of login sessions. Common examples of daemons are processes that control web servers, remote logins, and print servers.

Windows features an equivalent class of processes known as ***services***. Unlike daemons, services are easily distinguishable from other processes, and are differentiated in monitoring software such as the Task Manager.

3.1.3 The Filesystem

Another key component of an operating system is the ***filesystem***, which is an abstraction of how the external, nonvolatile memory of the computer is organized. Operating systems typically organize files hierarchically into ***folders***, also called ***directories***.

Each folder may contain files and/or subfolders. Thus, a volume, or drive, consists of a collection of nested folders that form a tree. The topmost folder is the root of this tree and is also called the root folder. Figure 3.3 shows a visualization of a file system as a tree.

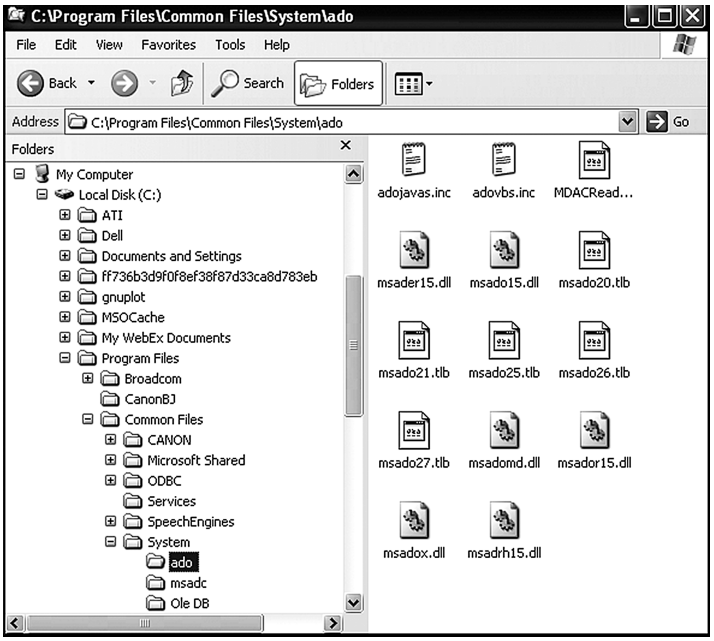


Figure 3.3: A filesystem as a tree, displayed by Windows Explorer.

File Access Control

One of the main concerns of operating system security is how to delineate which users can access which resources, that is, who can read files, write data, and execute programs. In most cases, this concept is encapsulated in the notion of file permissions, whose specific implementation depends on the operating system. Namely, each resource on disk, including both data files and programs, has a set of permissions associated with it.

File Permissions

File permissions are checked by the operating system to determine if a file is readable, writable, or executable by a user or group of users. This permission data is typically stored in the metadata of the file, along with attributes such as the type of file. When a process attempts to access a file, the operating system checks the identity of the process and determines whether or not access should be granted, based on the permissions of the file.

Several Unix-like operating systems have a simple mechanism for file permissions known as a ***file permission matrix***. This matrix is a representation of who is allowed to do what to the file, and contains permissions for three classes, each of which features a combination of bits. Files have an owner, which corresponds to the uid of some user, and a group, which corresponds to some group id.

First, there is the ***owner*** class, which determines permissions for the creator of the file. Next is the ***group*** class, which determines permissions for users in the same group as the file. Finally, the ***others*** class determines permissions for users who are neither the owner of the file nor in the same group as the file.

Each of these classes has a series of bits to determine what permissions apply. The first bit is the ***read bit***, which allows users to read the file. Second is the ***write bit***, which allows users to alter the contents of the file. Finally, there is the ***execute bit***, which allows users to run the file as a program or script, or, in the case of a directory, to change their current working directory to that one. An example of a file permission matrix for a set of files in a directory is shown in Figure 3.4.

```
rodan:~/java % ls -l
total 24
-rwxrwxrwx   1 goodrich faculty     2496 Jul 27 08:43 Floats.class
-rw-r--r--   1 goodrich faculty     2723 Jul 12  2006 Floats.java
-rw-------   1 goodrich faculty      460 Feb 25  2007 Test.java
rodan:~/java %
```

Figure 3.4: An example of the permission matrices for several files on a Unix system, using the ls -l command. The Floats.class file has read, write, and execute rights for its owner, goodrich, and nonowners alike. The Floats.java file, on the other hand, is readable by everyone, writeable only by its owner, and no one has execute rights. The file, Test.java, is only readable and writable by its owner—all others have no access rights.

Unix File Permissions

The read, write, and execute bits are implemented in binary, but it is common to express them in decimal notation, as follows: the execute bit has weight 1, the write bit has weight 2, and read bit has weight 4. Thus, each combination of the 3 bits yields a unique number between 0 and 7, which summarizes the permissions for a class. For example, 3 denotes that both the execute and write bits are set, while 7 denotes that read, write, and execute are all set.

Using this decimal notation, the entire file permission matrix can be expressed as three decimal numbers. For example, consider a file with a permission matrix of 644. This denotes that the owner has permission to read and write the file (the owner class is set to 6), users in the same group can only read (the group class is set to 4), and other users can only read (the others class is set to 4). In Unix, file permissions can be changed using the chmod command to set the file permission matrix, and the chown command to change the owner or group of a file. A user must be the owner of a file to change its permissions.

Folders also have permissions. Having read permissions for a folder allows a user to list that folder's contents, and having write permissions for a folder allows a user to create new files in that folder. Unix-based systems employ a ***path-based approach*** for file access control. The operating system keeps track of the user's current ***working directory***. Access to a file or directory is requested by providing a path to it, which starts either at the ***root directory***, denoted with /, or at the current working directory. In order to get access, the user must have execute permissions for all the directories in the path. Namely, the path is traversed one directory at the time, beginning with the start directory, and for each such directory, the execute permission is checked.

As an example, suppose Bob is currently accessing directory /home/alice, the home directory of Alice (his boss), for which he has execute permission, and wants to read file

/home/alice/administration/memos/raises.txt.

When Bob issues the Unix command

cat administration/memos/raises.txt

to view the file, the operating system first checks if Bob has execute permission on the first folder in the path, administration. If so, the operating system checks next whether Bob has execute permissions on the next folder, memos. If so, the operating system finally checks whether Bob has read permission on file raises.txt. If Bob does not have execute permission on administration or memos, or does not have read permission on raises.txt, access is denied.

3.1.4 Memory Management

Another service that an operating system provides is ***memory management***, that is, the organization and allocation of the memory in a computer. When a process executes, it is allocated a region of memory known as its ***address space***. The address space stores the program code, data, and storage that a process needs during its execution. In the Unix memory model, which is used for most PCs, the address space is organized into five segments, which from low addresses to high, are as follows. (See Figure 3.5.)

1. ***Text***. This segment contains the actual machine code of the program, which was compiled from source code prior to execution.
2. ***Data***. This segment contains static program variables that have been initialized in the source code, prior to execution.
3. ***BSS***. This segment, which is named for an antiquated acronym for ***block started by symbol***, contains static variables that are uninitialized (or initialized to zero).
4. ***Heap***. This segment, which is also known as the ***dynamic*** segment, stores data generated during the execution of a process, such as objects created dynamically in an object-oriented program written in Java or C++.
5. ***Stack***. This segment houses a stack data structure that grows downwards and is used for keeping track of the call structure of subroutines (e.g., methods in Java and functions in C) and their arguments.

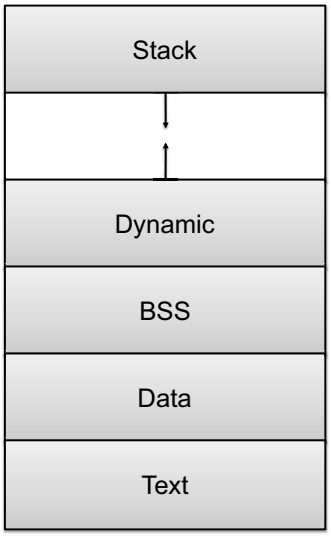


Figure 3.5: The Unix memory model.

Memory Access Permissions

Each of the five memory segments has its own set of access permissions (readable, writable, executable), and these permissions are enforced by the operating system. The text region is usually read-only, for instance, because it is generally not desirable to allow the alteration of a program's code during its execution. All other regions are writable, because their contents may be altered during a program's execution.

An essential rule of operating systems security is that processes are not allowed to access the address space of other processes, unless they have explicitly requested to share some of that address space with each other. If this rule were not enforced, then processes could alter the execution and data of other processes, unless some sort of process-based access control system were put in place. Enforcing address space boundaries avoids many serious security problems by protecting processes from changes by other processes.

In addition to the segmentation of address space in order to adhere to the Unix memory model, operating systems divide the address space into two broad regions: user space, where all user-level applications run, and kernel space, which is a special area reserved for core operating system functionality. Typically, the operating system reserves a set amount of space (one gigabyte, for example), at the bottom of each process's address space, for the kernel, which naturally has some of the most restrictive access privileges of the entire memory.

Contiguous Address Spaces

As described above, each process's address space is a contiguous block of memory. Arrays are indexed as contiguous memory blocks, for example, so if a program uses a large array, it needs an address space for its data that is contiguous. In fact, even the text portion of the address space, which is used for the computer code itself, should be contiguous, to allow for a program to include instructions such as "jump forward 10 instructions," which is a natural type of instruction in machine code.

Nevertheless, giving each executing process a contiguous slab of real memory would be highly inefficient and, in some cases, impossible. For example, if the total amount of contiguous address space is more than the amount of memory in the computer, then it is simply not possible for all executing processes to get a contiguous region of memory the size of its address space.

Virtual Memory

Even if all the processes had address spaces that could fit in memory, there would still be problems. Idle processes in such a scenario would still retain their respective chunks of memory, so if enough processes were running, memory would be needlessly scarce.

To solve these problems, most computer architectures incorporate a system of ***virtual memory***, where each process receives a virtual address space, and each virtual address is mapped to an address in real memory by the virtual memory system. When a virtual address is accessed, a hardware component known as the ***memory management unit*** looks up the real address that it is mapped to and facilitates access. Essentially, processes are allowed to act as if their memory is contiguous, when in reality it may be fragmented and spread across RAM, as depicted in Figure 3.6. Of course, this is useful, as it allows for several simplifications, such as supporting applications that index into large arrays as contiguous chunks of memory.

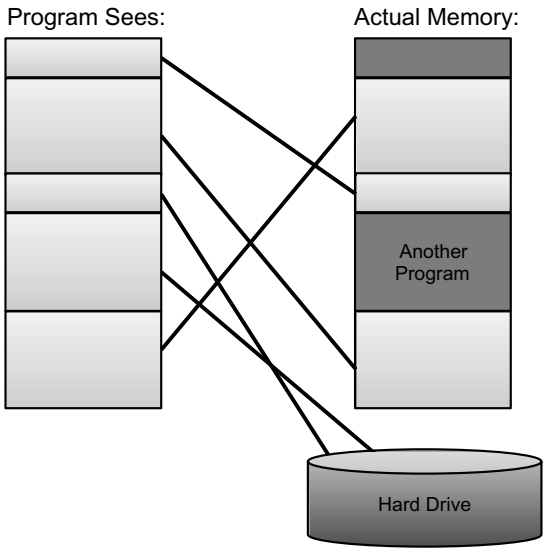


Figure 3.6: Mapping virtual addresses to real addresses.

An additional benefit of virtual memory systems is that they allow for the total size of the address spaces of executing processes to be larger than the actual main memory of the computer. This extension of memory is allowed because the virtual memory system can use a portion of the external drive to "park" blocks of memory when they are not being used by executing processes. This is a great benefit, since it allows for a computer to execute a set of processes that could not be multitasked if they all had to keep their entire address spaces in main memory all the time.

Page Faults

There is a slight time trade-off for benefit we get from virtual memory, however, since accessing the hard drive is much slower than RAM. Indeed, accessing a hard drive can be 10,000 times slower than accessing main memory.

So operating systems use the hard drive to store blocks of memory that are not currently needed, in order to have most memory accesses being in main memory, not the hard drive. If a block of the address space is not accessed for an extended period of time, it may be ***paged out*** and written to disk. When a process attempts to access a virtual address that resides in a paged out block, it triggers a ***page fault***.

When a page fault occurs, another portion of the virtual memory system known as the ***paging supervisor*** finds the desired memory block on the hard drive, reads it back into RAM, updates the mapping between the physical and virtual addresses, and possibly pages out a different unused memory block. This mechanism allows the operating system to manage scenarios where the total memory required by running processes is greater than the amount of RAM available. (See Figure 3.7.)

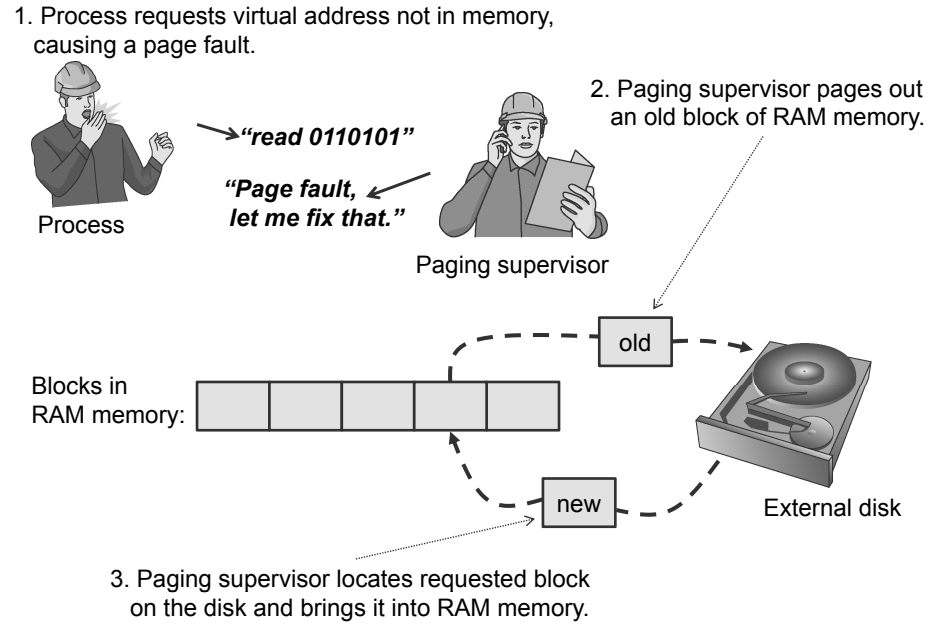


Figure 3.7: Actions resulting from a page fault.

3.1.5 Virtual Machines

Virtual machine technology is a rapidly emerging field that allows an operating system to run without direct contact with its underlying hardware. For instance, such systems may allow for substantial electrical power savings, by combining the activities of several computer systems into one, with the one simulating the operating systems of the others. The way this simulation is done is that an operating system is run inside a ***virtual machine*** (***VM***), software that creates a simulated environment the operating system can interact with. The software layer that provides this environment is known as a ***hypervisor*** or ***virtual machine monitor*** (***VMM***). The operating system running inside the VM is known as a ***guest***, and the native operating system is known as the ***host***. Alternately, the hypervisor can run directly in hardware without a host operating system, which is known as ***native virtualization***. To the guest OS, everything appears normal: it can interact with external devices, perform I/O, and so on. However, the operating system is in fact interacting with virtual devices, and the underlying virtual machine is bridging the gap between these virtual devices and the actual hardware, completely transparent to the guest operating system.

Implementing Virtual Machines

There are two main implementations of VMs. The first is ***emulation***, where the host operating system simulates virtual interfaces that the guest operating system interacts with. Communications through these interfaces are translated on the host system and eventually passed to the hardware. The benefit of emulation is that it allows more hardware flexibility. For example, one can emulate a virtual environment that supports one processor on a machine running an entirely different processor. The downside of emulation is that it typically has decreased performance due to the conversion process associated with the communication between the virtual and real hardware.

The second VM implementation is known simply as ***virtualization***, and removes the above conversion process. As a result, the virtual interfaces within the VM must be matched with the actual hardware on the host machine, so communications are passed from one to the other seamlessly. This reduces the possibilities for running exotic guest operating systems, but results in a significant performance boost.

Advantages of Virtualization

Virtualization has several advantages:

- ***Hardware Efficiency***. Virtualization allows system administrators to host multiple operating systems on the same machine, ensuring an efficient allocation of hardware resources. In these scenarios, the hypervisor is responsible for effectively managing the interactions between each operating system and the underlying hardware, and for ensuring that these concurrent operations are both efficient and safe. This management may be very complex—one set of hardware may be forced to manage many operating systems simultaneously.

- ***Portability***. VMs provide portability, that is, the ability to run a program on multiple different machines. This portability comes from the fact that the entire guest operating system is running as software virtually, so it is possible to save the entire state of the guest operating system as a snapshot and transfer it to another machine. This portability also allows easy restoration in the event of a problem. For example, malware researchers frequently employ VM technology to study malware samples in an environment that can easily be restored to a clean state should anything go awry.

- ***Security***. In addition to maximizing available resources and providing portable computing solutions, virtual machines provide several benefits from a security standpoint. By containing the operating system in a virtual environment, the VM functions as a strict ***sandbox*** that protects the rest of the machine in the event that the guest operating system is compromised. In the event of a breach, it is a simple matter to disconnect a virtual machine from the Internet without interrupting the operations of other services on the host machine.

- ***Management Convenience***. Finally, the ability to take snapshots of the entire virtual machine state can prove very convenient. Suppose Bob, a user on a company network, is running a virtualized version of Windows that boots automatically when he turns on his machine. If Bob's operating system becomes infected with malware, then a system administrator could just log in to the host operating system, disconnect Bob from the company network, and create a snapshot of Bob's virtual machine state. After reviewing the snapshot on another machine, the administrator might decide to revert Bob's machine to a clean state taken previously. The whole process would be reasonably time consuming and resource intensive on ordinary machines, but VM technology makes it relatively simple.

3.2 Process Security

To protect a computer while it is running, it is essential to monitor and protect the processes that are running on that computer.

3.2.1 Inductive Trust from Start to Finish

The trust that we place on the processes running on a computer is an inductive belief based on the integrity of the processes that are loaded when the computer is turned on, and that this state is maintained even if the computer is shut down or put into a hibernation state.

The Boot Sequence

The action of loading an operating system into memory from a powered-off state is known as ***booting***, originally ***bootstrapping***. This task seems like a difficult challenge—initially, all of the operating system's code is stored in persistent storage, typically the hard drive. However, in order for the operating system to execute, it must be loaded into memory. When a computer is turned on, it first executes code stored in a firmware component known as the ***BIOS*** (***basic input/output system***). On modern systems, the BIOS loads into memory the ***second-stage boot loader***, which handles loading the rest of the operating system into memory and then passes control of execution to the operating system. (See Figure 3.8.)

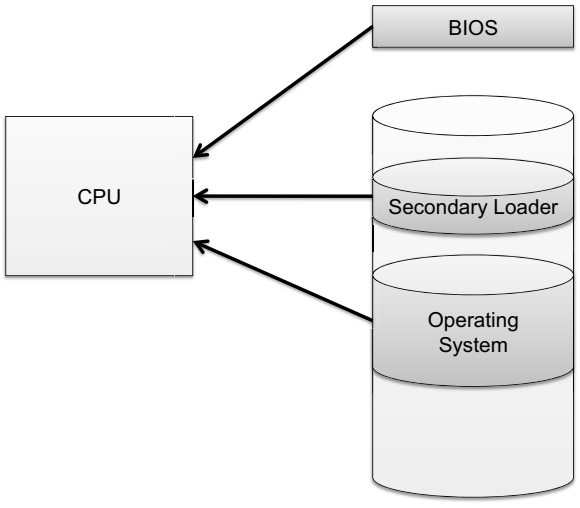


Figure 3.8: Operation of the BIOS.

A malicious user could potentially seize execution of a computer at several points in the boot process. To prevent an attacker from initiating the first stages of booting, many computers feature a BIOS password that does not allow a second-stage boot loader to be executed without proper authentication, which is a topic we discuss, with respect to BIOS-related security issues, in Section 2.4.4.

The Boot Device Hierarchy

There are some other security issues related to the boot sequence, however. Most second-stage boot loaders allow the user to specify which device should be used to load the rest of the operating system. In most cases, this option defaults to booting from the hard drive, or in the event of a new installation, from external media such as a DVD drive. Thus, one should make sure that the operating system is always booted from trusted media.

There is a customizable hierarchy that determines the order of precedence of booting devices: the first available device in the list is used for booting. This flexibility is important for installation and troubleshooting purposes, but as discussed in Section 2.4.4, it could allow an attacker with physical access to boot another operating system from an external media, bypassing the security mechanisms built into the operating system intended to be run on the computer. To prevent these attacks, many computers utilize second-stage boot loaders that feature password protections that only allow authorized users to boot from external storage media.

Hibernation

Modern machines have the ability to go into a powered-off state known as ***hibernation***. While going into hibernation, the operating system stores the entire contents of the machine's memory into a ***hibernation file*** on disk so that the state of the computer can be quickly restored when the system is powered back on. Without additional security precautions, hibernation exposes a machine to potentially invasive forensic investigation.

Since the entire contents of memory are stored into the hibernation file, any passwords or sensitive information that were stored in memory at the time of hibernation are preserved. A live CD attack can be performed to gain access to the hibernation file. (See Section 2.4.4.) Windows stores the hibernation file as `C:\hiberfil.sys`. Security researchers have shown the feasibility of reversing the compression algorithm used in this file, so as to extract a viewable snapshot of RAM at the time of hibernation, which opens the possibility of the attack shown in Figure 3.9.

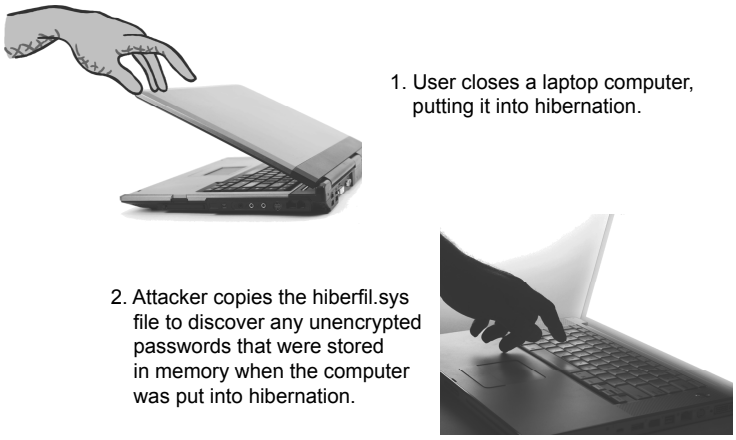


Figure 3.9: The hibernation attack.

Attacks that modify the hiberfil.sys file have also been demonstrated, so that the execution of programs on the machine is altered when the machine is powered on. Interestingly, Windows does not delete the hibernation file after resuming execution, so it may persist even after the computer is rebooted several times. A related attack on virtual memory page files, or swap files, is discussed in Section 3.3.1. To defend against these attacks, hard disk encryption should be used to protect hibernation files and swap files.

3.2.2 Monitoring, Management, and Logging

One of the most important aspects of operating systems security is something military people call "situational awareness." Keeping track of what processes are running, what other machines have interacted with the system via the Internet, and if the operating system has experienced any unexpected or suspicious behavior can often leave important clues not only for troubleshooting ordinary problems, but also for determining the cause of a security breach. For example, noticing log entries of repeated failed attempts to log in may warn of a brute-force attack, and prompt a system administrator to change passwords to ensure safety.

Event Logging

Operating systems therefore feature built-in systems for managing event logging. For example, as depicted in Figure 3.10, Windows includes an event logging system known simply as the Windows Event Log.

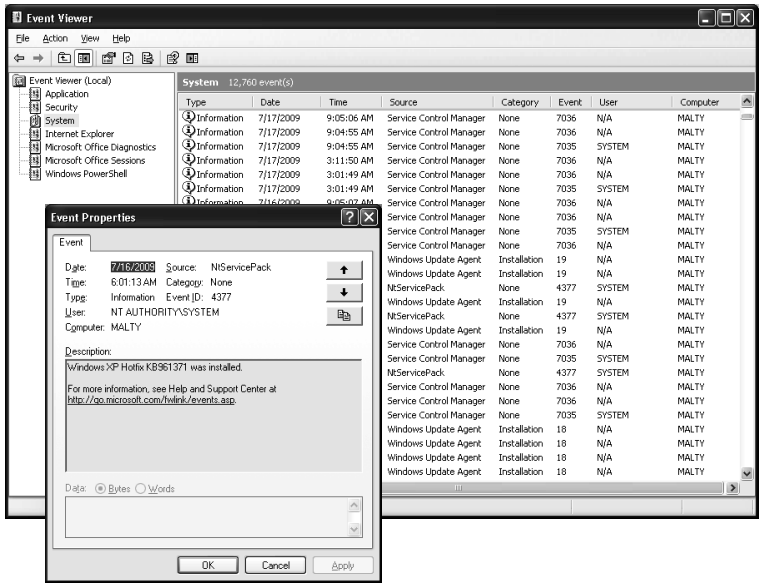


Figure 3.10: The Windows Event Log.

Windows defines three possible sources of logs, "System," "Application," and "Security." The System log can only be written to by the operating system itself, while the Application log may be written to by ordinary applications. Finally, the Security log can only be written to by a special Windows service known as the Local Security Authority Subsystem Service, visible in Process Explorer as lsass.exe. This service is responsible for enforcing security policies such as access control and user authentication. In addition to these three predefined sources, users can define their own log sources. Each log entry is known as an ***event***. Events are given unique identifiers, which correspond to any of the potential occurrences on a Windows machine that might prompt logging. Examples include applications exiting unexpectedly, users failing to properly authenticate, network connections being made, and so on.

Unix-based systems, including Linux, have differing logging mechanisms depending on the specific distribution. Typically, log files are stored in /var/log or some similar location and are simple text files with descriptive names. For example, auth.log contains records of user authentication, while kern.log keeps track of unexpected kernel behavior. Like Windows logs, entries contain a timestamp along with a description of the event. Typically, writing to these log files can only be done by a special syslog daemon. While Windows log files may allow easier handling when using Microsoft's event logging tools, the simple text format of Unix logs, containing one event per line, allows quick and easy perusing.

Process Monitoring

There are several scenarios where we would like to find out exactly which processes are currently running on our computer. For example, our computer might be sluggish and we want to identify an application using up lots of CPU cycles or memory. Or we may suspect that our computer has been compromised by a virus and we want to check for suspicious processes. Of course, we would like to terminate the execution of such a misbehaving or malicious process, but doing so requires that we identify it first. Every operating system therefore provides tools that allow users to monitor and manage currently running processes. Examples include the ***task manager*** application in Windows and the ps, top, pstree, and kill commands in Linux.

Process Explorer

Process monitoring tools might seem like they are aimed at expert users or administrators, since they present a detailed listing of running processes and associated execution statistics, but they are useful tools for ordinary users too. In Figure 3.11, we show a screen shot of just such a tool—***Process Explorer***—which is a highly customizable and useful tool for monitoring processes in the Microsoft Windows operating system.

Process Explorer is a good example of the kind of functionality that can be provided by a good process monitoring tool. The tool bar of Process Explorer contains various buttons, including one for terminating processes. The mini graphs show the usage histories of CPU time, main memory, and I/O, which are useful for identifying malicious or misbehaving processes. The processes tree pane shows the processes currently running and has a tabular format.

The components of Process Explorer provide a large amount of information for process monitoring and managing. The left column (Process) displays the tree of processes, that is, the processes and their parent-child relationship, by means of a standard outline view. Note, for example, in our screen shot shown in Figure 3.11, that process explorer.exe is the parent of many processes, including the ***Firefox*** web browser and the ***Thunderbird*** email client. Next to the process name is the icon of the associated program, which helps to facilitate visual identification. The remaining columns display, from left to right, the process ID (PID), percentage of CPU time used (CPU), size (in KB) of the process address space (Virtual Size), and description of the process (Description).

Large usage of CPU time and/or address space often indicate problematic processes that may need to be terminated. A customization window for the background color of processes is also shown in this example. In

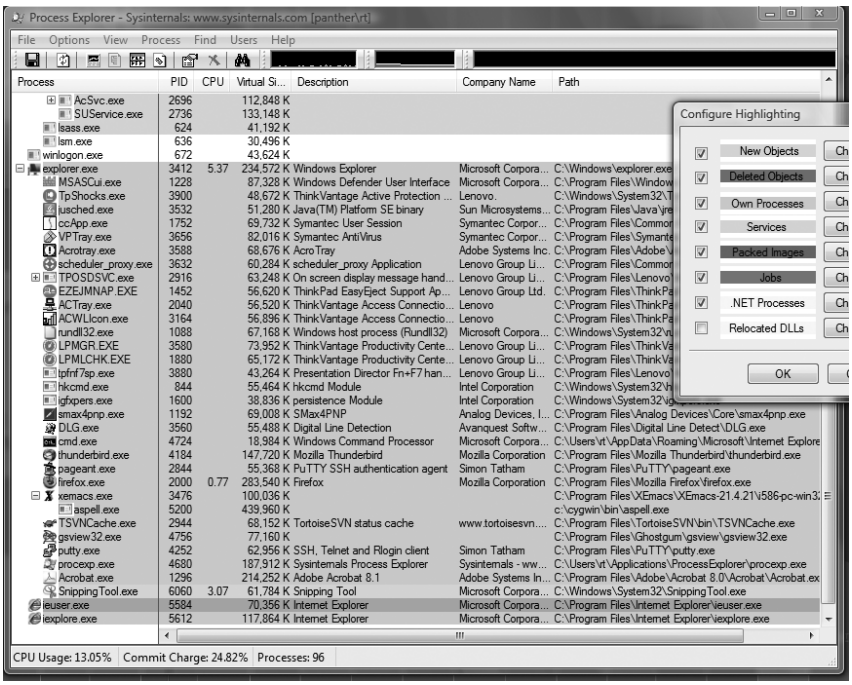


Figure 3.11: Screen shot of the ***Process Explorer*** utility for Microsoft Windows, by Mark Russinovich, configured with three components: a menu bar (top), a tool bar and three mini graphs (middle), and a process tree pane (bottom).

particular, different colors are used to highlight newly started processes, processes being terminated, user processes (started by the same user running Process Explorer), and system processes, such as services. All of these features provide a useful graphical user interface for identifying malicious and misbehaving processes, as well as giving a simple means to kill them once they are identified.

In addition to monitoring performance, it is important to gather detailed information about the ***process image***, that is, the executable program associated with the process. In our example of Figure 3.11, Process Explorer provides the name of the entity that has developed the program (Company) and the location on disk of the image (Path). The location of the image may allow the detection of a virus whose file name is the same as that of a legitimate application but is located in a nonstandard directory.

An attacker may also try to replace the image of a legitimate program with a modified version that performs malicious actions. To counter this attack, the software developer can digitally sign the image (see Section 1.3.2) and Process Explorer can be used to verify the signature and display the name of the entity who has signed the image (Verified Signer).

3.3 Memory and Filesystem Security

The contents of a computer are encapsulated in its memory and filesystem. Thus, protection of a computer's content has to start with the protection of its memory and its filesystem.

3.3.1 Virtual Memory Security

As we observed in Section 3.1.4, virtual memory is a useful tool for operating systems. It allows for multiple processes with a total address space larger than our RAM memory to run effectively, and it supports these multiple processes to each view its address spaces as being contiguous. Even so, these features come with some security concerns.

Windows and Linux Swap Files

On Windows, virtual memory pages that have been written to the hard disk are actually contained in what is known as the ***page file***, located at C:\pagefile.sys. Linux, on the other hand, typically requires users to set up an entire partition of their hard disk, known as the ***swap partition***, to contain these memory pages. In addition to the swap partition, Linux alternately supports a ***swap file***, which functions similarly to the Windows page file. In all cases, each operating system enforce rules preventing users from viewing the contents of virtual memory files while the OS is running, and it may be configured such that they are deleted when the machine is shut down.

Attacks on Virtual Memory

However, if an attacker suddenly powered off the machine without properly shutting down and booted to another operating system via external media, it may be possible to view these files and reconstruct portions of memory, potentially exposing sensitive information. To mitigate these risks, hard disk encryption should be used in all cases where potentially untrusted parties have physical access to a machine. Such encryption does not stop such an attacker from reading a swap file, of course, since he would have physical access to the computer. But it does prevent such an attacker from learning anything useful from the contents of these files, provided he is not able to get the decryption keys.

3.3.2 Password-Based Authentication

The question of who is allowed access to the resources in a computer system begins with a central question of operating systems security:

How does the operating system securely identify its users?

The answer to this question is encapsulated in the ***authentication*** concept, that is, the determination of the identity or role that someone has (in this case, with respect to the resources the operating system controls).

A standard authentication mechanism used by most operating systems is for users to log in by entering a ***username*** and ***password***. If the entered password matches the stored password associated with the entered username, then the system accepts this authentication and logs the users into the system.

Instead of storing the passwords as clear text, operating systems typically keep cryptographic one-way hashes of the passwords in a password file or database instead. Thanks to the one-way property of cryptographic hash functions (see Section 1.3.4), an attacker who gets hold of the password file cannot efficiently derive from it the actual passwords and has to resort to a guessing attack. That is, the basic approach to guessing passwords from the password file is to conduct a ***dictionary attack*** (Section 1.4.2), where each word in a dictionary is hashed and the resulting value is compared with the hashed passwords stored in the password file. If users of a system use weak passwords, such as English names and words, the dictionary attack can often succeed with a dictionary of only 500,000 words, as opposed to the search space of over 5 quadrillion words that could be formed from eight characters that can be typed on a standard keyboard.

Password Salt

One way to make the dictionary attack more difficult to launch is to use ***salt***, which is a cryptographic technique of using random bits as part of the input to a hash function or encryption algorithm so as to increase the randomness in the output. In the case of password authentication, salt would be introduced by associating a random number with each userid. Then, rather than comparing the hash of an entered password with a stored hash of a password, the system compares the hash of an entered password and the salt for the associated userid with a stored hash of the password and salt. Let U be a userid and P be the corresponding password. When using salt, the password file stores the triplet $(U, S, h(S||P))$, where S is the salt for U and h is a cryptographic hash function. (See Figure 3.12.)

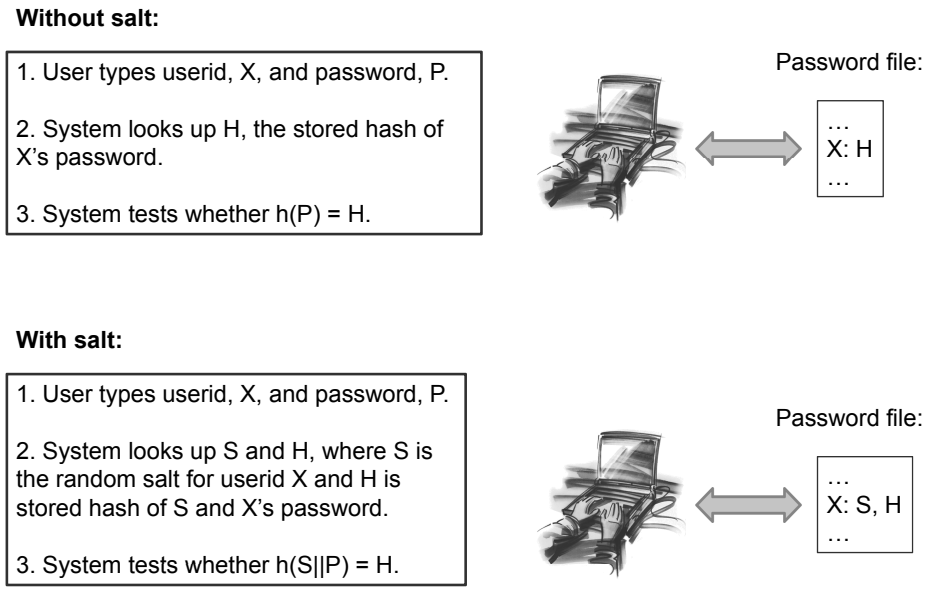


Figure 3.12: Password salt. We use $||$ to denote string concatenation and h to denote a cryptographic hash function.

How Salt Increases Search Space Size

Using password salt significantly increases the search space needed for a dictionary attack. Assuming that an attacker cannot find the salt associated with a userid he is trying to compromise, then the search space for a dictionary attack on a salted password is of size

$$2^B \times D,$$

where B is the number of bits of the random salt and D is the size of the list of words for the dictionary attack. For example, if a system uses a 32-bit salt for each userid and its users pick the kinds of passwords that would be in a 500,000 word dictionary, then the search space for attacking salted passwords would be

$$2^{32} \times 500{,}000 = 2{,}147{,}483{,}648{,}000{,}000,$$

which is over 2 quadrillion. Also, even if an attacker can find the salt associated with each userid (which the system should store in encrypted form), by employing salted passwords, an operating system can limit his dictionary attack to one userid at a time (since he would have to use a different salt value for each one).

Password Authentication in Windows and Unix-based Systems

In Microsoft Windows systems, password hashes are stored in a file called the ***Security Accounts Manager*** (***SAM***) file, which is not accessible to regular users while the operating system is running. Older versions of Windows stored hashed passwords in this file using an algorithm based on DES known as ***LAN Manager hash***, or ***LM hash***, which has some security weaknesses. This password-hashing algorithm pads a user's password to 14 characters, converts all lowercase letters to uppercase, and uses each of the 7-byte halves to generate a DES key. These two DES keys are used to encrypt a stored string (such as "KGS!@#$%"), resulting in two 8-byte ciphertexts, which are concatenated to form the final hash. Because each half of the user's password is treated separately, the task of performing a dictionary attack on an LM hash is actually made easier, since each half has a maximum of seven characters. In addition, converting all letters to uppercase significantly reduces the search space. Finally, the LM hash algorithm does not include a salt, so using tables of precomputed information is especially effective.

Windows improved these weaknesses by introducing the NTLM algorithm. NTLM is a challenge-response protocol used for authentication by several Windows components. The protocol involves a server, in this case the operating system, and a client, in this case a service attempting to authenticate a user. The operating system sends an 8-byte random number as a challenge to the client. Next, the client computes two 24-byte responses using two secrets, the LM hash of the password and the MD4 hash of the password. For each secret, the client pads the 16-byte hash to 21 bytes with null characters, splits the 21 bytes into three groups of 7 bytes, and uses each 7-byte segment as a key to DES encrypt the 8-byte challenge. Finally, the three 8-byte ciphertexts (for each secret) are concatenated, resulting in two 24-byte responses (one using the MD4 hash, and the other using the LM hash). These two responses are sent to the server, which has performed the same computations using its stored hashes, and authenticates the user. While NTLM has not been completely broken, some weaknesses have been identified. Specifically, both the MD4 and LM hashes are unsalted and as such are vulnerable to precomputation attacks.

Unix-based systems feature a similar password mechanism, and store authentication information at /etc/passwd, possibly in conjunction with /etc/shadow. However, most Unix variants use salt and are not as restricted in the choice of hash algorithm, allowing administrators to chose their preference. At the time of this writing, most systems use a salted MD5 algorithm or a DES variant, but many are able to use other hash algorithms such as Blowfish.

3.3.3 Access Control and Advanced File Permissions

Once a user is authenticated to a system, the next question that must be addressed is that of ***access control***:

> ***How does the operating system determine what users have permission to do?***

To address in detail this question with respect to files, we need to develop some terminology. A ***principal*** is either a user or a group of users. A principal can be explicitly defined as a set of users, such as a group, friends, consisting of users peter and paul, or it can be one of the principals predefined by the operating system. For example, in Unix-based systems, the following users and groups are defined for each file (or folder). User owner refers to the user owning the file. Group group, called the ***owning group***, is the default group associated with the file. Also, group all includes all the users in the system and group other consists of all but owner, i.e., of all users except the owner of the file.

A ***permission*** is a specific action on a file or folder. For example, file permissions include read and write and program files may additionally have an execute permission. A folder may also have a list permission, which refers to being able to inspect (list) the contents of the folder, and execute, which allows for setting the current directory as that folder. The execute permission of folders is the basis for the path-based access control mechanism in Unix-based systems. (See Section 3.1.3.)

Access Control Entries and Lists

An ***access control entry*** (***ACE***) for a given file or folder consists of a triplet (***principal***, ***type***, ***permission***), where type is either ***allow*** or ***deny***. An ***access control list*** (***ACL***) is an ordered list of ACEs (Section 1.2.2).

There are a number of specific implementation details that must be considered when designing an operating system permissions scheme. For one, how do permissions interact with the file organization of the system? Specifically, is there a hierarchy of inheritance? If a file resides in a folder, does it inherit the permissions of its parent, or override them with its own permissions? What happens if a user has permission to write to a file but not to the directory that the file resides in? The meaning of read, write, and execute permissions seems intuitive for files, but how do these permissions affect folders? Finally, if permissions aren't specifically granted or denied, are they implied by default? Interestingly, even between two of the most popular operating system flavors, Linux and Windows, the answers to these questions can vary dramatically.

Linux Permissions

Linux inherits most of its access control systems from the early Unix systems discussed previously. Linux features file permission matrices, which determine the privileges various users have in regards to a file. All permissions that are not specifically granted are implicitly denied, so there is no mechanism (or need) to explicitly deny permissions. According to the path-based access control principle, in order to access a file, each ancestor folder (in the filesystem tree) must have execute permission and the file itself must have read permission. Finally, owners of files are given the power to change the permissions on those files—this is known as ***discretionary access control*** (***DAC***).

In addition to the three basic permissions (read, write, and execute), Linux allows users to set extended attributes for files, which are applied to all users attempting to access these files. Example extended attributes include making a file append-only (so a user may only write to the end of the file) and marking a file as "immutable," at which point not even the root user can delete or modify the file (unless he or she removes the attribute first). These attributes can be set and viewed with the chattr and lsattr commands, respectively.

More recently, Linux has begun supporting an optional ACL-based permissions scheme. ACLs on Linux can be checked with the getfacl command, and set with the setfacl command. Within this scheme, each file has basic ACEs for the owner, group, and other principals and additional ACEs for specific users or groups, called ***named users*** and ***named groups***, can be created. There is also a ***mask*** ACE, which specifies the maximum allowable permissions for the owning group and any named users and groups. Let U be the euid of the process attempting access to the file or folder with certain requested permissions. To determine whether to grant access, the operating system tries to match the following conditions and selects the ACE associated with the first matching condition:

- U is the userid of the file owner: the ACE for owner;
- U is one of the named users: the ACE for U;
- one of the groups of U is the owning group and the ACE for group contains the requested permissions: the ACE for group;
- one of the groups of U is a named group G and its ACE contains the requested permissions: the ACE for G;
- for each group G of U that is the owning group or a named group, the ACE for G does not contain the requested permissions: the empty ACE;
- otherwise: the ACE for other.

If the ACE for owner or other or the empty ACE has been selected, then its permissions determine access. Else, the selected ACE is "ANDed" with the mask ACE and the permissions of the resulting ACE determine access. Note that the although multiple ACEs could be selected in the fourht condition, the access decision does not depend on the specific ACE selected. At the time of this writing, Linux's ACL scheme is not very widely used, despite the fact that it allows for more flexibility in access control.

Some Linux distributions have even more advanced access control mechanisms. ***Security-Enhanced Linux*** (***SELinux***), developed primarily by the United States National Security Agency, is a series of security enhancements designed to be applied to Unix-like systems. SELinux features strictly enforced ***mandatory access control***, which defines virtually every allowable action on a machine. Each rule consists of a ***subject***, referring to the process attempting to gain access, an ***object***, referring to the resource being accessed, and a series of permissions, which are checked by the operating system appropriately. SELinux embodies the principle of ***least privilege***: limiting every process to the bare minimum permissions needed to function properly, which significantly minimizes the effects of a security breach. In addition, unlike DAC, users are not given the power to decide security attributes of their own files. Instead, this is delegated to a central security policy administrator. These enhancements allow SELinux to create a much more restrictive security environment.

Windows Permissions

Windows uses an ACL model that allows users to create sets of rules for each user or group. These rules either ***allow*** or ***deny*** various permissions for the corresponding principal. If there is no applicable allow rule, access is denied by default. The basic permissions are known as ***standard permissions***, which for files consist of modify, read and execute, read, write, and finally, full control, which grants all permissions. Figure 3.13 depicts the graphical interface for editing permissions in Windows XP.

To finely tune permissions, there are also ***advanced permissions***, which the standard permissions are composed of. These are also shown in Figure 3.13. For example, the standard read permission encompasses several advanced permissions: read data, read attributes, read extended attributes, and read permissions. Setting read to allow for a particular principal automatically allows each of these advanced permissions, but it is also possible to set only the desired advanced permissions.

As in Linux, folders have permissions too: read is synonymous with the ability to list the contents of a folder, and write allows a user to create new files within a folder. However, while Linux checks each folder in the path to

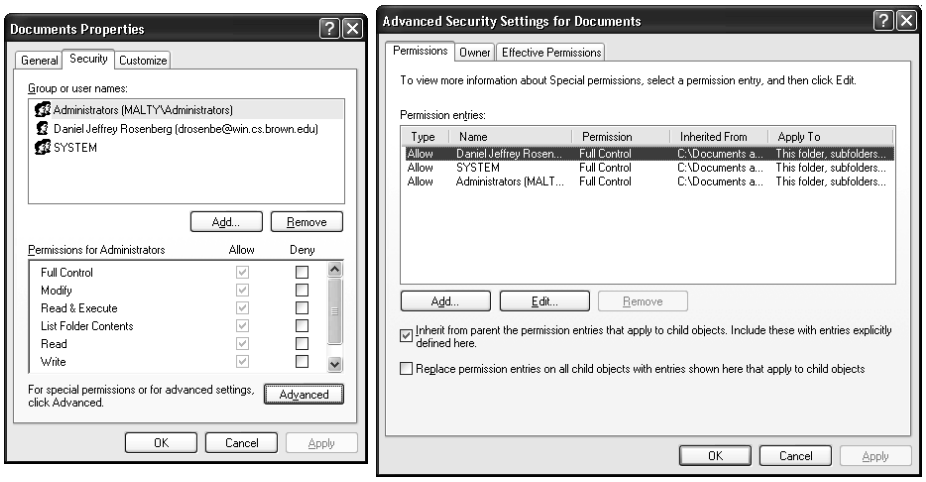


Figure 3.13: Customizing file permissions in Windows XP.

a file before allowing access, Windows has a different scheme. In Windows, the path to a file is simply an identifier that has no bearing on permissions. Only the ACL of the file in question is inspected before granting access. This allows administrators to deny a user access to a folder, but allow access to a file within that folder, which would not be possible in Linux.

In Windows, any ACEs applied to a folder may be set to apply not to just the selected folder, but also to the subfolders and files within it. The ACEs automatically generated in this way are called ***inherited ACEs***, as opposed to ACEs that are specifically set, which are called ***explicit ACEs***. Note that administrators may stop the propagation of inheritance at a particular folder, ensuring that the children of that folder do not inherit ACEs from ancestor folders.

This scheme of inheritance raises the question of how ACEs should take precedence. In fact, there is a simple hierarchy that the operating system uses when making access control decision. At any level of the hierarchy, deny ACEs take precedence over allow ACEs. Also, explicit ACEs take precedence over inherited ACEs, and inherited ACEs take precedence in order of the distance between the ancestor and the object in question—the parent's ACEs take precedent over the grandparent's ACEs, and so on. With this algorithm in place, resolving permissions is a simple matter of enumerating the entries of the ACL in the appropriate order until an applicable rule is found. This hierarchy, along with the finely granulated control of Windows permissions, provides administrators with substantial flexibility, but also may create the potential for security holes due to its complexity—if rules are not carefully applied, sensitive resources may be exposed.

The SetUID Bit

A related access-control question of operating systems security is how to give certain programs permission to perform tasks that the users running them should not otherwise be allowed to do. For example, consider the password mechanism in early Unix systems, where user login information is stored in /etc/passwd. Clearly, ordinary users should not be able to edit this file, or a user could simply change the password of another user and assume their identity. However, users should be allowed to change their own passwords.

In other words, a program is needed that can be run by an ordinary user, allowing changes to a file that ordinary users cannot alter. In the existing architecture, however, this doesn't seem possible. Since processes inherit the permissions of their parent process, a password-changing program run by an ordinary user would be restricted to the permissions of that user, and would be unable to write to the /etc/passwd file.

To solve this problem, Unix systems have an additional bit in the file permission matrix known as a ***setuid bit***. If this bit is set, then that program runs with the effective user ID of its owner, rather than the process that executed it. For example, the utility used to change passwords in Unix is passwd. This program is owned by the root account, has the execute bit set for the others class, and has the setuid bit set. When a user runs passwd, the program runs with the permissions of the root user, allowing it to alter the /etc/passwd file, which can only be written by the root user. Setuid programs can also drop their higher privileges by making calls to the setuid family of functions.

Although it is less commonly used, it is possible to set a ***setgid bit***, which functions similarly to setuid, but for groups. When the setgid bit is set, the effective group ID of the running process is equal to the ID of the group that owns the file, as opposed to the group id of the parent process.

The setuid mechanism is effective in that it solves the access-without-privileges problem, but it also raises some security concerns. In particular, it requires that setuid programs are created using safe programming practices. If an attacker can force a setuid program to execute arbitrary code, as we discuss later with respect to buffer overflow attacks, then the attacker can exploit the setuid mechanism to assume the permissions of the program's owner, creating a ***privilege escalation*** scenario.

An Example SetUID Program

An example setuid program can be found in Code Fragment 3.1. In this example, the application calls seteuid() to drop and restore its permissions.

Note that this program runs with the permissions of the user for most of its execution, but briefly raises its permissions to that of its owner in order to write to a log file that ordinary users presumably cannot access.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

static uid_t euid, uid;

int main(int argc, char * argv[])
{
  FILE *file;
  /* Store real and effective user IDs */
  uid = getuid();
  euid = geteuid();
   /* Drop priviliges */
  seteuid(uid);
  /* Do something useful */
  /* ... */
  /* Raise privileges */
  seteuid(euid);
  /* Open the file */
  file = fopen("/home/admin/log", "a");
  /* Drop privileges again */
  seteuid(uid);
  /* Write to the file */
  fprintf(file, "Someone used this program.\n");
  /* Close the file stream and return */
  fclose(file);
  return 0;
}
```

Code Fragment 3.1: A simple C program that uses seteuid() to change its permissions. The fprintf action is done using the permissions of the owner of this program, not the user running this program.

3.3.4 File Descriptors

In order for processes to work with files, they need a shorthand way to refer to those files, other than always going to the filesystem and specifying a path to the files in question. In order to efficiently read and write files stored on disk, modern operating systems rely on a mechanism known as ***file descriptors***. File descriptors are essentially index values stored in a table, aptly known as the ***file descriptor table***. When a program needs to access a file, a call is made to the open system call, which results in the kernel creating a new entry in the file descriptor table which maps to the file's location on the disk. This new file descriptor is returned to the program, which can now issue read or write commands using that file descriptor. When receiving a read or write system call, the kernel looks up the file descriptor in the table and performs the read or write at the appropriate location on disk. Finally, when finished, the program should issue the close system call to remove the open file descriptor.

Reading and Writing with File Descriptors

Several security checks occur during the process of performing a read or write on a file, given its file descriptor. When the open system call is issued, the kernel checks that the calling process has permission to access the file in the manner requested—for example, if a process requests to open a file for writing, the kernel ensures that the file has the write permission set for that process before proceeding. Next, whenever a call to read or write is issued, the kernel checks that the file descriptor being written to or read from has the appropriate permissions set. If not, the read or write fails and the program typically halts.

On most modern systems, it is possible to pass open file descriptors from one process to another using ordinary IPC mechanisms. For example, on Unix-based systems (including Linux) it is possible to open a file descriptor in one process and send a copy of the file descriptor to another process via a local socket.

File Descriptor Leaks

A common programming error that can lead to serious security problems is known as a ***file descriptor leak***. A bit of additional background is required to understand this type of vulnerability. First, it is important to note that when a process creates a child process (using a fork command), that child process inherits copies of all of the file descriptors that are open in the parent. Second, the operating system only checks whether a process

has permissions to read or write to a file at the moment of creating a file descriptor entry; checks performed at the time of actually reading or writing to a file only confirm that the requested action is allowed according to the permissions the file descriptor was opened with. Because of these two behaviors, a dangerous scenario can arise when a program with high privileges opens a file descriptor to a protected file, fails to close it, and then creates a process with lower permissions. Since the new process inherits the file descriptors of its parent, it will be able to read or write to the file, depending on how the parent process issued the open system call, regardless of the fact that the child process might not have permission to open that file in other circumstances.

An Example Vulnerability

An example of this scenario can be found in Code Fragment 3.2. Notice in this example how there is no call to close the file descriptor before executing a new process. As a result, the child is able to read the file. In a situation such as this one, the child could access the open file descriptor via a number of mechanisms, most commonly using the fcntl() family of functions. To fix this vulnerability, a call to fclose(), which would close the file descriptor, should be made before executing the new program.

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[])
{

  /* Open the password file for reading */
  FILE *passwords;
  passwords = fopen("/home/admin/passwords", "r");

  /* Read the passwords and do something useful */
  /* ... */

  /* Fork and execute Joe's shell without closing the file */
  execl("/home/joe/shell", "shell", NULL);

}
```

Code Fragment 3.2: A simple C program vulnerable to a file descriptor leak.

3.3.5 Symbolic Links and Shortcuts

It is often useful for users to be able to create links or shortcuts to other files on the system, without copying the entire file to a new location. For example, it might be convenient for a user to have a link to a program on their desktop while keeping the actual program at another location. In this way, if the user updates the underlying file, all links to it will automatically be referring to the updated version.

In Linux and other Unix-based systems, this can be accomplished through the use of ***symbolic links***, also known as ***symlinks*** or ***soft links***, which can be created using the `ln` command. To the user, symlinks appear to reside on the disk like any other file, but rather than containing information, they simply point to another file or folder on disk.

This linking is completely transparent to applications, as well. If a program attempts to open and read from a symlink, the operating system follows the link so that the program actually interacts with the file the symlink is pointing to. Symlinks can be chained together, so that one symlink points to another, and so on, as long as the final link points to an actual file. In these cases, programs attempting to access a symlink follow the chain of links until reaching the file.

Symlinks can often provide a means by which malicious parties can trick applications into performing undesired behavior, however. As an example, consider a program that opens and reads a file specified by the user. Suppose that this program is designed specifically to prohibit the reading of one particular file, say, `/home/admin/passwords`, for example. An unsafe version of this program would simply check that the filename specified by the user is not `/home/admin/passwords`. However, an attacker could trick this program by creating a symlink to the passwords file and specifying the path of the symlink instead. To solve this ***aliasing*** problem, the program should either check if the provided filename refers to a symlink, or confirm the actual filename being opened by using a `stat` system call, which retrieves information on files.

More recent versions of Windows support symlinks similar to those on Unix, but much more common is the use of ***shortcuts***. A shortcut is similar to a symlink in that it is simply a pointer to another file on disk. However, while symlinks are automatically resolved by the operating system so that their use is transparent, Windows shortcuts appear as regular files, and only programs that specifically identify them as shortcuts can follow them to the referenced files. This prevents most of the symlink attacks that are possible on Unix-based systems, but also limits their power and flexibility.

3.4 Application Program Security

Many attacks don't directly exploit weaknesses in the OS kernel, but rather attack insecure programs. These programs, operating at the applications layer, could even be nonkernel operating system programs, such as the program to change passwords, which runs with higher privileges than those granted to common users. So these programs should be protected against privilege escalation attacks. But before we can describe such protections, we need to first discuss some details about program creation.

3.4.1 Compiling and Linking

The process of converting source code, which is written in a programming language, such as Java or C++, to the machine code instructions that a processor can execute is known as ***compiling***. A program may be compiled to be either ***statically linked*** or ***dynamically linked***. With static linking, all shared libraries, such as operating system functions, that a program needs during its execution are essentially copied into the compiled program on disk. This may prove to be safer from a security perspective, but is inconvenient in that it uses additional space for duplicate code that might be used by many programs, and it may limit debugging options.

The alternative is dynamic linking, where shared libraries are loaded when the program is actually run. When the program is executed, the ***loader*** determines which shared libraries are needed for the program, finds them on the disk, and imports them into the process's address space. In Microsoft Windows, each of these external libraries is known as a ***dynamic linking library*** (***DLL***), while in many Unix systems, they are simply referred to as ***shared objects***. Dynamic linking is an optimization that saves space on the hard disk, and allows developers to modularize their code. That is, instead of recompiling an entire application, it may be possible to alter just one DLL, for instance, to fix a bug since DLL that could potentially affect many other programs. The process of injecting arbitrary code into programs via shared libraries is known as ***DLL injection***. DLL injection can be incredibly useful for the purposes of debugging, in that programmers can easily change functions in their applications without recompiling their code. However, this technique poses a potential security risk because it may allow malicious parties to inject their own code into legitimate programs. Imagine the consequences if a guest user redefined a function called by a system administrator program; hence, the need for administrative privileges.

3.4.2 Simple Buffer Overflow Attacks

A classic example of such an application program attack, which allows for privilege escalation, is known as a ***buffer overflow*** attack. In any situation where a program allocates a fixed-size buffer in memory in which to store information, care must be taken to ensure that copying user-supplied data to this buffer is done securely and with boundary checks. If this is not the case, then it may be possible for an attacker to provide input that exceeds the length of the buffer, which the program will then dutifully attempt to copy to the allotted buffer. However, because the provided input is larger than the buffer, this copying may overwrite data beyond the location of the buffer in memory, and potentially allow the attacker to gain control of the entire process and execute arbitrary code on the machine (recall that the address space for a process includes both the data and the code for that process).

Arithmetic Overflow

The simplest kind of overflow condition is actually a limitation having to do with the representation of integers in memory. In most 32-bit architectures, signed integers (those that can be either positive or negative) are expressed in what is known as ***two's compliment*** notation. In hex notation, signed integers 0x00000000 to 0x7fffffff (equivalent to $2^{31} - 1$) are positive numbers, and 0x80000000 to 0xffffffff are negative numbers. The threshold between these two ranges allows for overflow or underflow conditions. For example, if a program continually adds very large numbers and eventually exceeds the maximum value for a signed integer, 0x7fffffff, the representation of the sum overflows and becomes negative rather than positive. Similarly, if a program adds many negative numbers, eventually the sum will underflow and become positive. This condition also applies to unsigned integers, which consist of only positive numbers from 0x00000000 to 0xffffffff. Once the highest number is reached, the next sequential integer wraps around to zero.

An Example Vulnerability

This numerical overflow behavior can sometimes be exploited to trick an application to perform undesirable behavior. As an example, suppose a network service keeps track of the number of connections it has received since it has started, and only grants access to the first five users. An unsafe implementation can be found in Code Fragment 3.3.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // ...
  // ...
  // Does nothing to check overflow conditions
  connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
  return 1;
}
```

Code Fragment 3.3: A C program vulnerable to an arithmetic overflow.

An attacker could compromise the above system by making a huge number of connections until the connections counter overflows and wraps around to zero. At this point, the attacker will be authenticated to the system, which is clearly an undesirable outcome. To prevent these types of attacks, safe programming practices must be used to ensure that integers are not incremented or decremented indefinitely and that integer upper bounds or lower bounds are respected. An example of a safe version of the program above can be found in Code Fragment 3.4.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // ...
  // ...
  // Prevents overflow conditions
  if(connections < 5)
    connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
   return 1;
}
```

Code Fragment 3.4: A variation of the program in Code Fragment 3.3, protected against arithmetic overflow.

3.4.3 Stack-Based Buffer Overflow

Another type of buffer overflow attack exploits the special structure of the memory stack. Recall from Section 3.1.4, that the stack is the component of the memory address space of a process that contains data associated with function (or method) calls. The stack consists of frames, each associated with an active call. A frame stores the local variables and arguments of the call and the return address for the parent call, i.e., the memory address where execution will resume once the current call terminates. At the base of the stack is the frame of the main() call. At the end of the stack is the frame of the currently running call. This organizational structure allows for the CPU to know where to return to when a method terminates, and it also automatically allocates and deallocates the space local variables require.

In a buffer overflow attack, an attacker provides input that the program blindly copies to a buffer that is smaller than the input. This commonly occurs with the use of unchecked C library functions, such as strcpy() and gets(), which copy user input without checking its length.

A buffer overflow involving a local variable can cause a program to overwrite memory beyond the buffer's allocated space in the stack, which can have dangerous consequences. An example of a program that has a stack buffer overflow vulnerability is shown in Code Fragment 3.5.

In a stack-based buffer overflow, an attacker could overwrite local variables adjacent in memory to the buffer, which could result in unexpected behavior. Consider an example where a local variable stores the name of a command that will be eventually executed by a call to system(). If a buffer adjacent to this variable is overflowed by a malicious user, that user could replace the original command with one of his or her choice, altering the execution of the program.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  // Create a buffer on the stack
  char buf[256];
  // Does not check length of buffer before copying argument
  strcpy(buf,argv[1]);
  // Print the contents of the buffer
  printf("%s\n",buf);
  return 1;
}
```

Code Fragment 3.5: A C program vulnerable to a stack buffer overflow.

Although this example is somewhat contrived, buffer overflows are actually quite common (and dangerous). A buffer overflow attack is especially dangerous when the buffer is a local variable or argument within a stack frame, since the user's input may overwrite the return address and change the execution of the program. In a ***stack smashing*** attack, the attacker exploits a stack buffer vulnerability to inject malicious code into the stack and overwrite the return address of the current routine so that when it terminates, execution is passed to the attacker's malicious code instead of the calling routine. Thus, when this context switch occurs, the malicious code will be executed by the process on behalf of the attacker. An idealized version of a stack smashing attack, which assumes that the attacker knows the exact position of the return address, is illustrated in Figure 3.14.

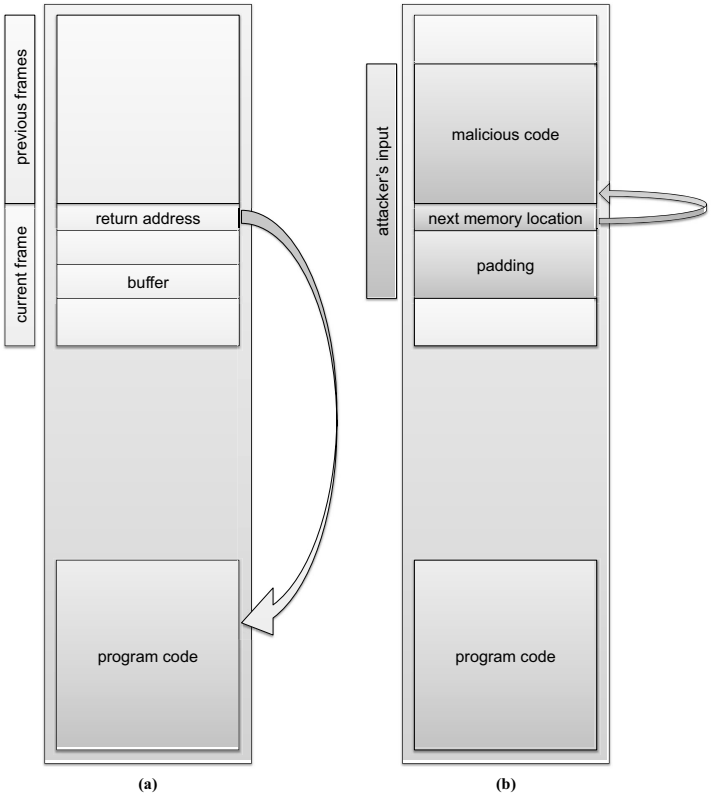


Figure 3.14: A stack smashing attack under the assumption that the attacker knows the position of the return address. (a) Before the attack, the return address points to a location in the program code. (b) Exploiting the unprotected buffer, the attacker injects into the address space input consisting of padding up to the return address location, a modified return address that points to the next memory location, and malicious code. After completing execution of the current routine, control is passed to the malicious code.

Seizing Control of Execution

In a realistic situation of a stack-based buffer overflow attack, the first problem for the attacker is to guess the location of the return address with respect to the buffer and to determine what address to use for overwriting the return address so that execution is passed to the attacker's code. The nature of operating system design makes this challenging for two reasons.

First, processes cannot access the address spaces of other processes, so the malicious code must reside within the address space of the exploited process. Because of this, the malicious code is often kept in the buffer itself, as an argument to the process provided when it is started, or in the user's shell environment, which is typically imported into the address space of processes.

Second, the address space of a given process is unpredictable and may change when a program is run on different machines. Since all programs on a given architecture start the stack at the same relative address for each process, it is simple to determine where the stack starts, but even with this knowledge, knowing exactly where the buffer resides on the stack is difficult and subject to guesswork.

Several techniques have been developed by attackers to overcome these challenges, including ***NOP sledding***, ***return-to-libc***, and the ***jump-to-register*** or ***trampolining*** techniques.

NOP Sledding

NOP sledding is a method that makes it more likely for the attacker to successfully guess the location of the code in memory by increasing the size of the target. A ***NOP*** or ***No-op*** is a CPU instruction that does not actually do anything except tell the processor to proceed to the next instruction. To use this technique, the attacker crafts a payload that contains an appropriate amount of data to overrun the buffer, a guess for a reasonable return address in the process's address space, a very large number of NOP instructions, and finally, the malicious code. When this payload is provided to a vulnerable program, it copies the payload into memory, overwriting the return address with the attacker's guess. In a successful attack, the process will jump to the guessed return address, which is likely to be somewhere in the high number of NOPs (known as the NOP sled). The processor will then "sled through" all of the NOPs until it finally reaches the malicious code, which will then be executed. NOP sledding is illustrated in Figure 3.15.

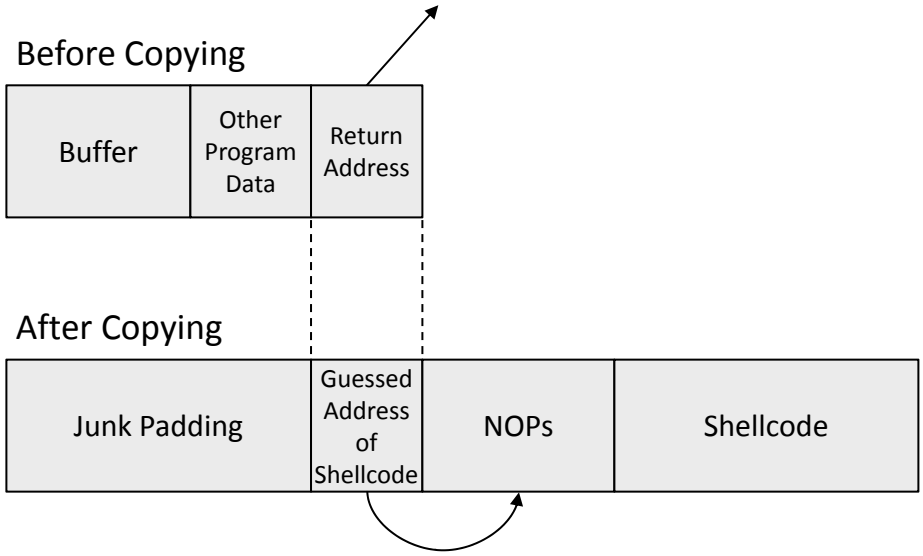


Figure 3.15: The NOP sledding technique for stack smashing attacks.

Trampolining

Despite the fact that NOP sledding makes stack-based buffer overflows much more likely to succeed, they still require a good deal of guesswork and are not extremely reliable. Another technique, known as ***jump-to-register*** or ***trampolining***, is considered more precise. As mentioned above, on initialization, most processes load the contents of external libraries into their address space. These external libraries contain instructions that are commonly used by many processes, system calls, and other low-level operating system code. Because they are loaded into the process's address space in a reserved section of memory, they are in predictable memory locations. Attackers can use knowledge of these external libraries to perform a trampolining attack. For example, an attacker might be aware of a particular assembly code instruction in a Windows core system DLL and suppose this instruction tells the processor to jump to the address stored in one of the processor's ***registers***, such as ESP. If the attacker can manage to place his malicious code at the address pointed to by ESP and then overwrite the return address of the current function with the address of this known instruction, then on returning, the application will jump and execute the `jmp esp` instruction, resulting in execution of the attacker's malicious code. Once again, specific examples will vary depending on the application and the chosen library instruction, but in general this technique provides a reliable way to exploit vulnerable applications that is not likely to change on subsequent attempts on different machines, provided all of the machines involved are running the same version of the operating system.

The Return-to-libc Attack

A final attack technique, known as a ***return-to-libc attack***, also uses the external libraries loaded at runtime—in this case, the functions of the C library, ***libc***. If the attacker can determine the address of a C library function within a vulnerable process's address space, such as system() or execv, this information can be used to force the program to call this function. The attacker can overflow the buffer as before, overwriting the return address with the address of the desired library function. Following this address, the attacker must provide a new address that the libc function will return to when it is finished execution (this may be a dummy address if it is not necessary for the chosen function to return), followed by addresses pointing to any arguments to that function. When the vulnerable stack frame returns, it will call the chosen function with the arguments provided, potentially giving full control to the attacker. This technique has the added advantage of not executing any code on the stack itself. The stack only contains arguments to existing functions, not actual shellcode. Therefore, this attack can be used even when the stack is marked as nonexecutable.

Shellcode

Once an attacker has crafted a stack-based buffer overflow exploit, they have the ability to execute arbitrary code on the machine. Attackers often choose to execute code that spawns a terminal or shell, allowing them to issue further commands. For this reason, the malicious code included in an exploit is often known as ***shellcode***. Since this code is executed directly on the stack by the CPU, it must be written in assembly language, low-level processor instructions, known as ***opcodes***, that vary by CPU architecture. Writing usable shellcode can be difficult. For example, ordinary assembly code may frequently contain the null character, 0x00. However, this code cannot be used in most buffer overflow exploits, because this character typically denotes the end of a string, which would prevent an attacker from successfully copying his payload into a vulnerable buffer; hence, shellcode attackers employ tricks to avoid null characters.

Buffer overflow attacks are commonly used as a means of privilege escalation by exploiting SetUID programs. Recall that a SetUID program can be executed by low-level users, but is allowed to perform actions on behalf of its owner, who may have higher permissions. If a SetUID program is vulnerable to a buffer overflow, then an attack might include shellcode that first executes the setuid() system call, and then spawns a shell. This would result in the attacker gaining a shell with the permissions of the exploited process's owner, and possibly allow for full system compromise.

Preventing Stack-Based Buffer Overflow Attacks

Many measures have been developed to combat buffer overflow attacks. First, the root cause of buffer overflows is not the operating system itself, but rather insecure programming practices. Programmers must be educated about the risks of insecurely copying user-supplied data into fixed-size buffers, and ensure that their programs never attempt to copy more information than can fit into a buffer. Many popular programming languages, including C and C++, are susceptible to this attack, but other languages do not allow the behavior that makes buffer overflow attacks possible. To fix the previous example, the safer strncpy function should be used, as in Code Fragment 3.6.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  // Create a buffer on the stack
  char buf[256];
  // Only copies as much of the argument as can fit in the buffer
  strncpy(buf, argv[1], sizeof(buf));
  // Print the contents of the buffer
  printf("%s\n",buf);
  return 1;
}
```

Code Fragment 3.6: A C program protected against a stack buffer overflow.

Because of the dangers of buffer overflows, many operating systems have incorporated protection mechanisms that can detect if a stack-based buffer overflow has occurred (at which point the OS can decide how to deal with this discovery). One such technique directly provides stack-smashing protection by detecting when a buffer overflow occurs and at that point prevent redirection of control to malicious code.

There are several implementations of this technique, all of which involve paying closer attention to how data is organized in the method stack. One such implementation, for instance, reorganizes the stack data allotted to programs and incorporates a ***canary***, a value that is placed between a buffer and control data (which plays a similar role to a canary in a coal mine). The system regularly checks the integrity of this canary value, and if it has been changed, it knows that the buffer has been overflowed and it should prevent malicious code execution. (See Figure 3.16.)

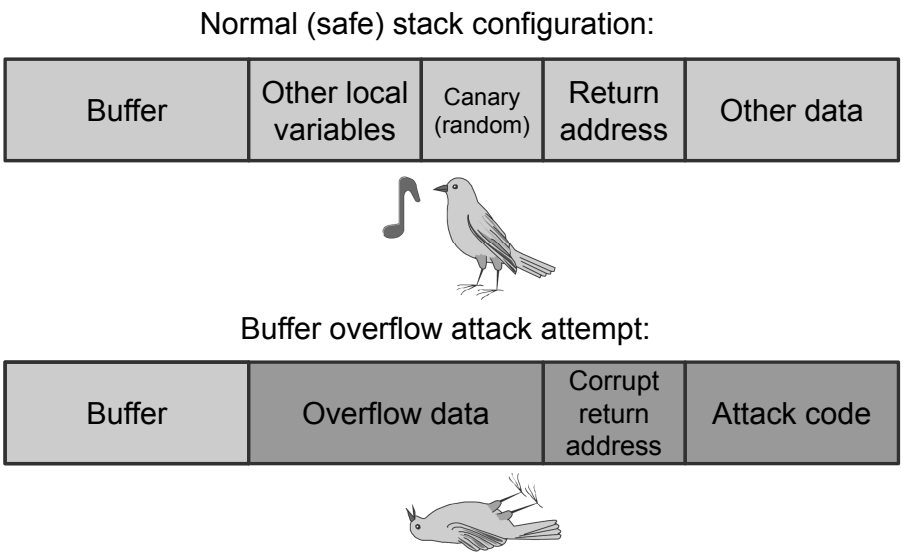


Figure 3.16: Stack-based buffer overflow detection using a random canary. The canary is placed in the stack prior to the return address, so that any attempt to overwrite the return address also overwrites the canary.

Other systems are designed to prevent the attacker from overwriting the return address. Microsoft developed a compiler extension called PointGuard that adds code which XOR-encodes any pointers, including the return address, before and after they are used. As a result, an attacker would not be able to reliably overwrite the return address with a location that would lead to a valid jump. Yet another approach is to prevent running code on the stack by enforcing a no-execution permission on the stack segment of memory. If the attacker's shellcode were not able to run, then exploiting an application would be difficult. Finally, many operating systems now feature ***address space layout randomization*** (***ASLR***), which rearranges the data of a process's address space at random, making it extremely difficult to predict where to jump in order to execute code.

Despite these protection mechanisms, researchers and hackers alike have developed newer, more complicated ways of exploiting buffer overflows. For example, popular ASLR implementations on 32-bit Windows and Linux systems have been shown to use an insufficient amount of randomness to fully prevent brute-force attacks, which has required additional techniques to provide stack-smashing protection. The message is clear, operating systems may have features to reduce the risks of buffer overflows, but ultimately, the best way to guarantee safety is to remove these vulnerabilities from application code. The primary responsibility rests on the programmer to use safe coding practices.

3.4.4 Heap-Based Buffer Overflow Attacks

Memory on the stack is either allocated statically, which is determined when the program is compiled, or it is allocated and removed automatically when functions are called and returned. However, it is often desirable to give programmers the power to allocate memory dynamically and have it persist across multiple function calls. This memory is allocated in a large portion of unused memory known as the ***heap***.

Dynamic memory allocation presents a number of potential problems for programmers. For one, if programmers allocate memory on the heap and do not explicitly deallocate (free) that block, it remains used and can cause ***memory leak*** problems, which are caused by memory locations that are allocated but are not actually being used.

From a security standpoint, the heap is subject to similar problems as the stack. A program that copies user-supplied data into a block of memory allocated on the heap in an unsafe way can result in overflow conditions, allowing an attacker to execute arbitrary code on the machine. An example of a vulnerable program can be found in Code Fragment 3.7.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  // Allocate two adjacent blocks on the heap
  char *buf = malloc(256);
  char *buf2 = malloc(16);
  // Does not check length of buffer before copying argument
  strcpy(buf, argv[1]);
  // Print the argument
  printf("Argument: %s\n", buf);
  // Free the blocks on the heap
  free(buf);
  free(buf2);
  return 1;
}
```

Code Fragment 3.7: A simple C program vulnerable to a heap overflow.

As with stack overflows, these problems can be mitigated by using safe programming practices, including replacing unsafe functions such as strcpy() with safer equivalents like strncpy(). (See Code Fragment 3.8.)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  // Allocate two adjacent blocks on the heap
  char *buf = malloc(256);
  char *buf2 = malloc(16);
  // Only copies as much of the argument as can fit in the buffer
  strncpy(buf, argv[1], 255);
  // Print the argument
  printf("Argument: %s\n", buf);
  // Free the blocks on the heap
  free(buf);
  free(buf2);
  return 1;
}
```

Code Fragment 3.8: A simple C program protected against a heap overflow.

Heap-based overflows are generally more complex than the more prevalent stack-based buffer overflows and require a more in-depth understanding of how garbage collection and the heap are implemented. Unlike the stack, which contains control data that if altered changes the execution of a program, the heap is essentially a large empty space for data. Rather than directly altering control, heap overflows aim to either alter data on the heap or abuse the functions and macros that manage the memory on the heap in order to execute arbitrary code. The specific attack used varies depending on the particular architecture.

An Example Heap-Based Overflow Attack

As an example, let us consider an older version of the GNU compiler (GCC) implementation of malloc(), the function that allocates a block of memory on the heap. In this implementation, blocks of memory on the heap are maintained as a linked list—each block has a pointer to the previous and next blocks in the list. When a block is marked as free, the unlink() macro is used to set the pointers of the adjacent blocks to point to each other, effectively removing the block from the list and allowing the space to be

reused. One heap overflow technique takes advantage of this system. If an attacker provides user input to a program that unsafely copies the input to a block on the heap, the attacker can overflow the bounds of that block and overwrite portions of the next block. If this input is carefully crafted, it may be possible to overwrite the linked list pointers of the next block and mark it as free, in such a way that the unlink routine is tricked into writing data into an arbitrary address in memory. In particular, the attacker may trick the unlink routine into writing the address of his shellcode into a location that will eventually result in a jump to the malicious code, resulting in the execution of the attacker's code.

One such location that may be written to in order to compromise a program is known as .dtors. Programs compiled with GCC may feature functions marked as constructor or destructor functions. Constructors are executed before main(), and destructors are called after main() has returned. Therefore, if an attacker adds the address of his shellcode to the .dtors section, which contains a list of destructor functions, his code will be executed before the program terminates. Another potential location that is vulnerable to attacks is known as the ***global offset table*** (***GOT***). This table maps certain functions to their absolute addresses. If an attacker overwrites the address of a function in the GOT with the address of his shellcode and this function is called, the program will jump to and execute the shellcode, once again giving full control to the attacker.

Preventing Heap-Based Buffer Overflow Attacks

Prevention techniques for heap-based overflow attacks resemble those for stack-based overflows. Address space randomization prevents the attacker from reliably guessing memory locations, making the attack more difficult. In addition, some systems make the heap nonexecutable, making it more difficult to place shellcode. Newer implementations of dynamic memory allocation routines often choose to store heap metadata (such as the pointers to the previous and next blocks of heap memory) in a location separate from the actual data stored on the heap, which makes attacks such as the unlink technique impossible. Once again, the single most important preventive measure is safe programming. Whenever a program copies user-supplied input into a buffer allocated on the heap, care must be taken to ensure that the program does not copy more data than that buffer can hold.

3.4.5 Format String Attacks

The printf family of C library functions are used for I/O, including printing messages to the user. These functions are typically designed to be passed an argument containing the message to be printed, along with a ***format string*** that denotes how this message should be displayed. For example, calling printf("%s",message) prints the message variable as a string, denoted by the format string %s. Format strings can also write to memory. The %n format string specifies that the print function should write the number of bytes output so far to the memory address of the first argument to the function.

When a programmer does not supply a format string, the input argument to the print function controls the format of the output. If this argument is user-supplied, then an attacker could carefully craft an input that uses format strings, including %n, to write to arbitrary locations in memory. This could allow an attacker to seize control and execute arbitrary code in the context of the program by overwriting a return address, function pointer, etc. An example of a program with a format string vulnerability can be found in Code Fragment 3.9, where the printf() function is called without providing a format string.

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  printf("Your argument is:\n");
  // Does not specify a format string, allowing the user to supply one
  printf(argv[1]);
}
```

Code Fragment 3.9: A C program vulnerable to a format string bug.

Once again, the solution to this attack lies in the hands of the programmer. To prevent format string attacks, programmers should always provide format strings to the printf function family, as in Code Fragment 3.10.

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  printf("Your argument is:\n");
  // Supplies a format string
  printf("%s",argv[1]);
}
```

Code Fragment 3.10: A C program protected against a format string bug.

3.4.6 Race Conditions

Another programming error that can lead to compromise by malicious users is the introduction of what is known as a ***race condition***. A race condition is any situation where the behavior of the program is unintentionally dependent on the timing of certain events.

A classic example makes use of the C functions access() and open(). The open() function, used to open a file for reading or writing, opens the specified file using the effective user ID (rather than the real user ID) of the calling process to check permissions. In other words, if a SetUID program owned by the root user is run by an ordinary user, that program can successfully call open() on files that only the root user has permission to access. The access() function checks whether the real user (in this case, the user running the program) has permission to access the specified file.

Suppose there were a simple program that takes a filename as an argument, checks whether the user running the program has permission to open that file, and if so, reads the first few characters of the file and prints them. This program might be implemented as in Code Fragment 3.11.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
int main(int argc, char * argv[])
{
  int file;
  char buf[1024];
  memset(buf, 0, 1024);
  if(argc < 2) {
    printf("Usage: printer [filename]\n");
    exit(-1);
  }
  if(access(argv[1], R_OK) != 0) {
    printf("Cannot access file.\n");
    exit(-1);
  }
  file = open(argv[1], O_RDONLY);
  read(file, buf, 1023);
        close(file);
  printf("%s\n", buf);
  return 0;
}
```

Code Fragment 3.11: A C program vulnerable to a race condition.

The Time of Check/Time of Use Problem

There is a race condition in the above implementation. In particular, there is a tiny, almost unnoticeable time delay between the calls to access() and open(). An attacker could exploit this small delay by changing the file in question between the two calls. For example, suppose the attacker provided /home/joe/dummy as an argument, an innocent text file that the attacker can access. After the call to access() returns 0, indicating the user has permission to access the file, the attacker can quickly replace /home/joe/dummy with a symbolic link to a file that he does not have permission to read, such as /etc/passwd.

Next, the program will call open() on the symbolic link, which will be successful because the program is SetUID root and has permission to open any files accessible to the root user. Finally, the program will dutifully read and print the contents of the file.

Note that this type of attack could not be done manually; the time difference between two function calls is small enough that no human would be able to change the files fast enough. However, it would be possible to have a program running in the background that repeatedly switches between the two files—one legitimate and one just a symbolic link—and runs the vulnerable program repeatedly until the switch occurred in exactly the right place.

In general, this type of vulnerability is known as a Time of Check/Time of Use (TOCTOU) problem. Any time a program checks the validity and authorizations for an object, whether it be a file or some other property, before performing an action on that object, care should be taken that these two operations are performed ***atomically***, that is, they should be performed as a single uninterruptible operation. Otherwise, the object may be changed in between the time it is checked and the time it is used. In most cases, such a modification simply results in erratic behavior, but in some, such as this example, the time window can be exploited to cause a security breach.

To safely code the example above, the call to access() should be completely avoided. Instead, the program should drop its privileges using seteuid() before calling open(). This way, if the user running the program does not have permission to open the specified file, the call to open() will fail. A safe version of the program can be found in Code Fragment 3.12.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
int main(int argc, char * argv[])
{
  int file;
  char buf[1024];
  uid_t uid, euid;
  memset(buf, 0, 1024);
  if(argc < 2) {
    printf("Usage: printer [filename]\n");
    exit(-1);
  }
  euid = geteuid();
  uid = getuid();
  /* Drop privileges */
  seteuid(uid);
  file = open(argv[1], O_RDONLY);
  read(file, buf, 1023);
  close(file);
  /* Restore privileges */
  seteuid(euid);
  printf("%s\n", buf);
  return 0;
}
```

Code Fragment 3.12: A simple C program that is protected against a race condition.

3.5 Exercises

For help with exercises, please visit **securitybook.net**.

Reinforcement

R-3.1 How can multitasking make a single processor look like it is running multiple programs at the same time?

R-3.2 Give an example of three operating systems services that do not belong in the kernel?

R-3.3 If a process forks two processes and these each fork two processes, how many processes are in this part of the process tree?

R-3.4 What is the advantage of booting from the BIOS instead of booting the operating system directly?

R-3.5 Can a process have more than one parent? Explain.

R-3.6 Describe two types of IPC. What are their relative benefits and weaknesses?

R-3.7 Why would it be bad to mix the stack and heap segments of memory in the same segment?

R-3.8 Describe the difference between a daemon and a service.

R-3.9 What are the benefits of virtual memory?

R-3.10 Why should a security-conscious Windows user inspect processes with Process Explorer instead of Task Manager?

R-3.11 What is the purpose of salting passwords?

R-3.12 If a password is salted with a 24-bit random number, how big is the dictionary attack search space for a 200,000 word dictionary?

R-3.13 Eve has just discovered and decrypted the file that associates each userid with its 32-bit random salt value, and she has also discovered and decrypted the password file, which contains the salted-and-hashed passwords for the 100 people in her building. If she has a dictionary of 500,000 words and she is confident all 100 people have passwords from this dictionary, what is the size of her search space for performing a dictionary attack on their passwords?

R-3.14 Suppose farasi is a member of group hippos in a system that uses basic Unix permissions. He creates a file pool.txt, sets its group as hippos and sets its permissions as u=rw,g=. Can farasi read pool.txt?

R-3.15 Dr. Eco claims that virtual machines are good for the environment. How can he justify that virtualization is a green technology?

R-3.16 Alice, who uses a version of Unix, requires a better program to manage her photos. She wants Bob to code this program for her. However, she does not want Bob to be able to see some confidential files she has in her account (for example, the solutions of some homework). On the other hand, Bob wants to make sure that Alice does not read his code, since this will probably be her CS032 final project. Explain how this can be achieved by using the `setuid` and `chmod` functions provided by UNIX. Also, assume for this question only (regardless of real systems' behavior), that a user cannot revert to the real UID after using the effective UID that was set by the `setuid` feature. Specifically consider the fact that Bob could embed code in his program to transfer data it has access to, to a public folder and/or a web server.

R-3.17 Is it possible to create a symbolic link to a symbolic link? Why or why not?

R-3.18 Why is it pointless to give a symbolic link more restrictive access privileges than the file it points to?

R-3.19 Describe the main differences between advanced file permissions in Linux and Windows NTFS. Give an example to illustrate each difference.

R-3.20 Dr. Blahbah claims that buffer overflow attacks via stack smashing are made possible by the fact that stacks grow downwards (towards smaller addresses) on most popular modern architectures. Therefore, future architectures should ensure that the stack grows upwards; this would provide a good defense against buffer overflow. Do you agree or disagree? Why?

R-3.21 Why is it important to protect the part of the disk that is used for virtual memory?

R-3.22 Why is it unsafe to keep around the `C:\hiberfil.sys` file even after a computer has been restored from hibernation?

Creativity

C-3.1 Bob thinks that generating and storing a random salt value for each userid is a waste. Instead, he is proposing that his system administrators use a SHA-1 hash of the userid as its salt. Describe whether this choice impacts the security of salted passwords and include an analysis of the respective search space sizes.

C-3.2 Alice has a picture-based password system, where she has each user pick a set of their 20 favorite pictures, say, of cats, dogs, cars, etc. To login, a user is shown a series of pictures in pairs—one on the left and one on the right. In each pair, the user has to pick the one that is in his set of favorites. If the user picks the correct 20 out of the 40 he is shown (as 20 pairs), then the system logs him in. Analyze the security of this system, including the size of the search space. Is it more secure than a standard password system?

C-3.3 Charlie likes Alice's picture-password system of the previous exercise, but he has changed the login so that it just shows the user 40 different pictures in random order and they have to indicate which 20 of these are from their set of favorites. Is this an improvement over Alice's system? Why or why not?

C-3.4 Dr. Simplex believes that all the effort spent on access control matrices and access control lists is a waste of time. He believes that all file access permissions for every file should be restricted to the owner of that file, period. Describe at least three scenarios where he is wrong, that is, where users other than a file's owner need some kind of allowed access privileges.

C-3.5 On Unix systems, a convenient way of packaging a collection of files is a ***SHell ARchive***, or ***shar file***. A shar file is a shell script that will unpack itself into the appropriate files and directories. Shar files are created by the shar command. The implementation of the shar command in a legacy version of the HP-UX operating system created a temporary file with an easily predictable filename in directory /tmp. This temporary file is an intermediate file that is created by shar for storing temporary contents during its execution. Also, if a file with this name already exists, then shar opens the file and overwrites it with temporary contents. If directory /tmp allows anyone to write to it, a vulnerability exists. An attacker can exploit such a vulnerability to overwrite a victim's file. (1) What knowledge about shar should the attacker have? (2) Describe the command that the attacker issues in order to have shar overwrite an arbitrary file of a victim. Hint: the command is issued before shar is executed. (3) Suggest a simple fix to the shar utility to prevent the attack. Note that this is *not* a setuid question.

C-3.6 Java is considered to be "safe" from buffer overflows. Does that make it more appropriate to use as a development language when security is a concern? Be sure and weigh all of the risks involved in product development, not just the security aspects.

C-3.7 Dr. Blahbah has implemented a system with an 8-bit random canary that is used to detect and prevent stack-based buffer overflow

attacks. Describe an effective attack against Dr. Blahbah's system and analyze its likelihood of success.

C-3.8 Consider the following piece of C code:

```
int main(int argc, char *argv[])
{
char continue = 0;
char password[8];
strcpy(password, argv[1]);
if (strcmp(password, "CS166")==0)
continue = 1;
if (continue)
{
*login();
}
}
```

In the above code, *login() is a pointer to the function login() (In C, one can declare pointers to functions which means that the call to the function is actually a memory address that indicates where the executable code of the function lies). (1) Is this code vulnerable to a buffer-overflow attack with reference to the variables password[] and continue? If yes, describe how an attacker can achieve this and give an ideal ordering of the memory cells (assume that the memory addresses increase from left to right) that correspond the variables password[] and continue of the code so that this attack can be avoided. (2) To fix the problem, a security expert suggests to remove the variable continue and simply use the comparison for login. Does this fix the vulnerability? What kind of new buffer overflow attack can be achieved in a multiuser system where the login() function is shared by a lot of users (both malicious and and nonmalicious) and many users can try to log in at the same time? Assume for this question only (regardless of real systems' behavior) that the pointer is on the stack rather than in the data segment, or a shared memory segment. (3) What is the existing vulnerability when login() is not a pointer to the function code but terminates with a return() command? Note that the function strcpy does not check an array's length.

C-3.9 In the ***StackGuard*** approach to solving the buffer overflow problem, the compiler inserts a ***canary*** value on the memory location before the return address in the stack. The canary value is ran-

domly generated. When there is a return from the function call, the compiler checks if the canary value has been overwritten or not.

Do you think that this approach would work? If yes, please explain why it works; if not, please give a counterexample.

C-3.10 Another approach to protecting against buffer overflows is to rely on address space layout randomization (ASLR). Most implementations of ASLR offset the start of each memory segment by a number that is randomly generated within a certain range at runtime. Thus, the starting address of data objects and code segments is a random location. What kinds of attacks does this technique make more difficult and why?

Projects

P-3.1 Write a program in pseudocode that acts as a ***guardian*** for a file, allowing anyone to append to the file, but to make no other changes to it. This may be useful, e.g., to add information to a log file. Your program, to be named append, should take two strings file1 and file2 as arguments, denoting the paths to two files. Operation append(String file1, String file2) copies the contents of file1 to the end of file2, provided that the user performing the operation has read permission for file1 and file2. If the operation succeeds, 0 is returned. On error, -1 is returned.

Assume that the operating system supports the setuid mechanism and that append is a setuid program owned by a user called guardian. The file to which other files get appended (file2) is also owned by guardian. Anyone can read its contents. However, it can be written only by guardian. Write your program in pseudocode using the following Java-style system calls:

(1) int open(String path_to_file, String mode) opens a file in a given mode and returns a positive integer that is the descriptor of the opened file. String mode is one of READ_ONLY or WRITE_ONLY. (2) void close(int file_descriptor) closes a file given its descriptor. (3) byte[] read(int file_descriptor) reads the content of the given file into an array of bytes and returns the array. (4) void write(int file_descriptor, byte[] source_buffer) stores a byte array into a file, replacing the previous content of the file. (5) int getUid() gets the real user ID of the current process. (6) int getEuid() gets the effective user ID of the current process. (7) void setEuid(int uid) sets the effective user ID of the current process, where uid is either the real user ID or the saved effective user ID of the process.

Error conditions that occur in the execution of the above system calls (e.g., trying to open a file without having access right to it or using a nonexistent descriptor) trigger exception SystemCallFailed, which should be handled by your program. Note that you do not need to worry about buffer overflow in this question.

P-3.2 Implement a system that implements a simple access control list (ACL) functionality, which gives a user the ability to grant file permissions on a user-by-user basis. For example, one can create a file that is readable by joeuser and janeuser, but only writable by janeuser. The operations on the ACL are as follows. (1) setfacl(path, uid, uid_mode, gid, gid_mode) sets a user with uid and/or a group with gid to the ACL for the object (file or directory) specified by path. If the user/group already exists, the access mode is updated. If only (uid, uid_mode) or (gid, gid_mode) is to be set, null is used for the unset arguments. (2) getfacl(path) obtains the entire access control list of the file path. (3) access(uid, access_mode, path) determines whether a user with uid can access the object stored at path in mode access_mode. This method returns a boolean. path contains the full path to a file or a directory, e.g., /u/bob/cs166/homework.doc. You can use groups username to find out the groups that username belongs to. One way to accomplish this ACL would be with a linked list; your solution should be more efficient with respect to the number of users, groups, and files. Describe how to implement the operations with your data structure. You have to consider permissions associated with the parent directories of a file. For this, you are given a method getParent(full_path) that takes a path to a file or directory, and returns the parent directory.

P-3.3 In a virtual machine, install the Linux operating system, which supports the capability-based access control (capabilities are built into the Linux kernel since the kernel version 2.6.24). Use capabilities to reduce the amount of privileges carried by certain SetUID programs, such as `passwd` and `ping`.

P-3.4 In a virtual machine, install a given privileged program (e.g., a SetUID program) that is vulnerable to the buffer overflow attack. Write a program to exploit the vulnerability and gain the adminstrator privilege. Try different attacking schemes, one using shellcode, and the other using the return-to-libc technique. It should be noted that many operating systems have multiple built-in countermeasures to protect them against the buffer overflow attack. First, turn off those protections and try the attack; then turn

them back on and see whether these protections can be defeated (some countermeasures can be easily defeated).

P-3.5 In a virtual machine, install a given privileged program (e.g., a SetUID program) that is vulnerable to the format-string attack. Write a program to exploit the vulnerability and that will crash the privileged program, print out the value of an internal variable secret to the user, and modify the value of this secret variable. Modify the source code of the vulnerable program so it can defeat the format string attack.

P-3.6 In a virtual machine, install a given privileged program (e.g., a SetUID program) that is vulnerable to the race condition attack. Write a program to exploit the vulnerability and gain adminstrator privilege. Modify the source code of the vulnerable program so it can defeat the race condition attack.

P-3.7 Write a term paper describing how buffer overflows are used as vectors for many computer attacks. Discuss how they enable different kinds of attacks and describe how different software engineering practices and languages might encourage or discourage buffer-overflow vulnerabilities.

Chapter Notes

Operating systems are discussed in detail in the textbooks by Doeppner [27] and Silberschatz, Galvin and Gagne [94]. Much of the content in this chapter on Unix-based systems, especially Linux, draws heavily on open source documentation, which can be accessed at http://www.manpagez.com/. Grünbacher describes in detail Linux ACLs and the file access control algorithm based on ACLs [37]. Reference material on the Windows API can be found in the Microsoft Developer Network [60]. A classic introduction to stack-based buffer overflows is given by Aleph One [1]. Lhee and Chapin discuss buffer overflow and format string exploitation [54]. A method for protecting against heap smashing attacks is presented by Fetzer and Xiao [33]. The canary method for defending against stack smashing attacks is incorporated in the StackGuard compiler extension by Cowan et al. [20]. Address space randomization and its effectiveness in preventing common buffer overflow attacks is discussed by Shacham et al. [89]. Project P-3.1 is from Tom Doeppner.