

Chapter 3

Operating Systems Security

Contents

3.1	Operating Systems Concepts	114
3.1.1	The Kernel and Input/Output	115
3.1.2	Processes	116
3.1.3	The Filesystem	121
3.1.4	Memory Management	124
3.1.5	Virtual Machines	128
3.2	Process Security	130
3.2.1	Inductive Trust from Start to Finish	130
3.2.2	Monitoring, Management, and Logging	132
3.3	Memory and Filesystem Security	136
3.3.1	Virtual Memory Security	136
3.3.2	Password-Based Authentication	137
3.3.3	Access Control and Advanced File Permissions	140
3.3.4	File Descriptors	146
3.3.5	Symbolic Links and Shortcuts	148
3.4	Application Program Security	149
3.4.1	Compiling and Linking	149
3.4.2	Simple Buffer Overflow Attacks	150
3.4.3	Stack-Based Buffer Overflow	152
3.4.4	Heap-Based Buffer Overflow Attacks	159
3.4.5	Format String Attacks	162
3.4.6	Race Conditions	163
3.5	Exercises	166

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.)

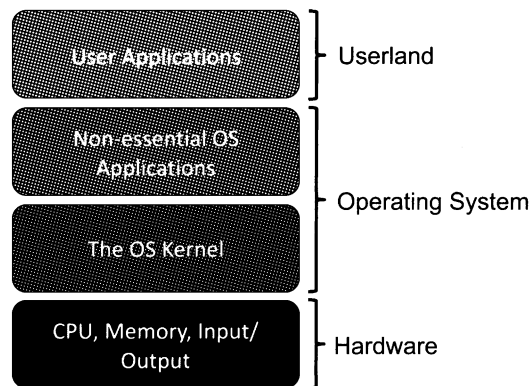


Figure 3.1: The layers of a computer system.

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 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.

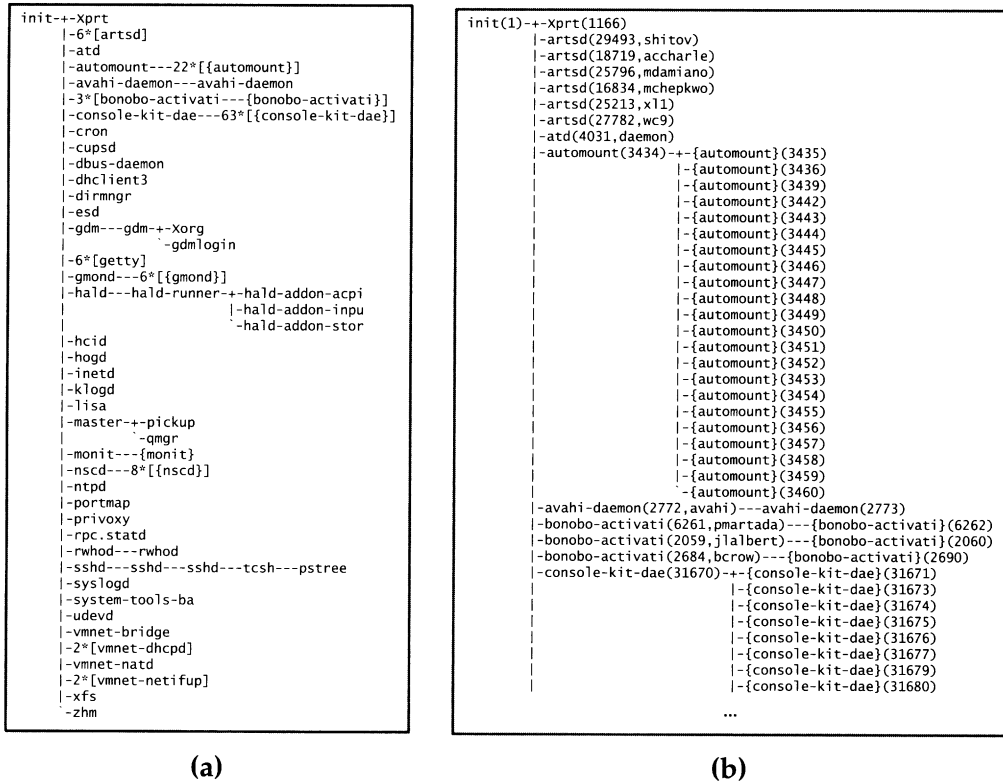


Figure 3.2: The tree of processes in a Linux system produced by the `ps` command. The process tree is visualized by showing the root on the upper left-hand corner, with children and their descendants to the right of it. (a) Compact visualization where children associated with the same command are merged 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 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.

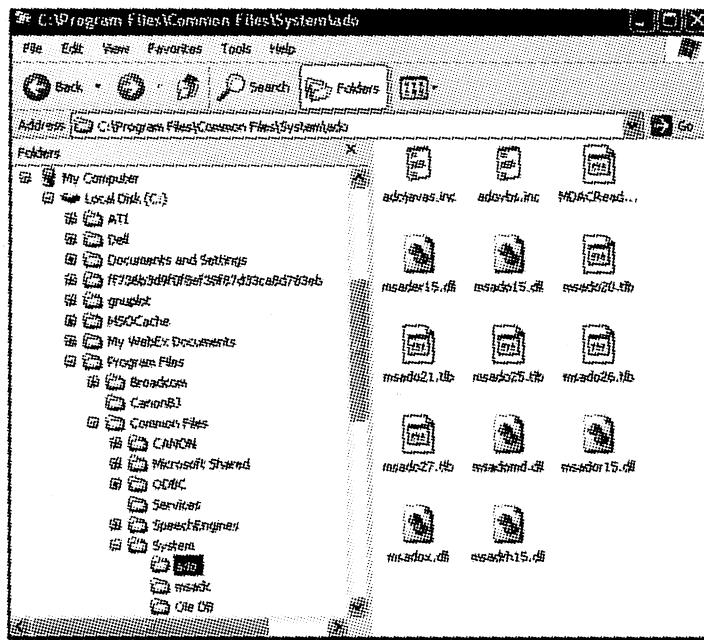


Figure 3.3: A filesystem as a tree, displayed by Windows Explorer.

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.

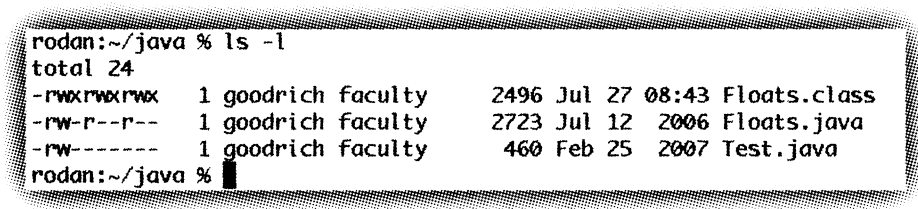
File Permissions

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, writable 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.