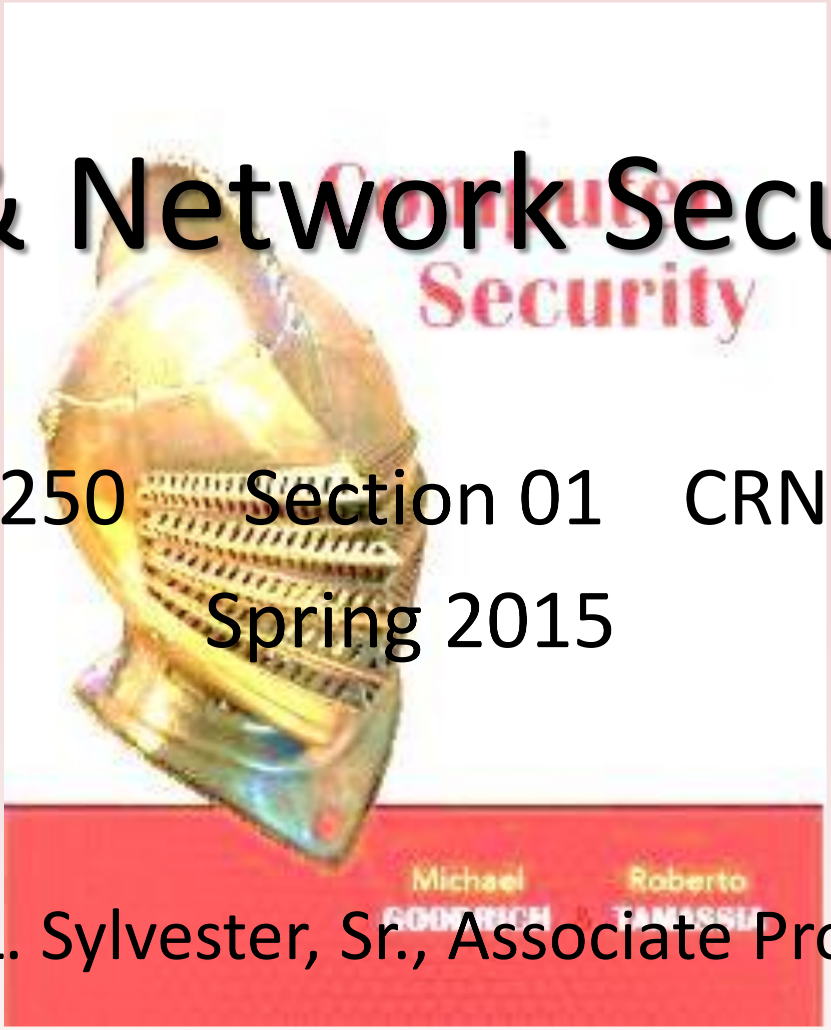
PC & Network Security



CNET – 250 Section 01 CRN: 20653

Spring 2015

David L. Sylvester, Sr., Associate Professor

Chapter 3

Operating System Security

Operating Systems Concepts

Operating system, abbreviated (OS) provides the interface between the user’s of a computer and the computer’s hardware. It manages:

- The way applications access the resources
  - Disk drives
  - CPU
  - Main memory
  - Input devices
  - Output devices and
  - Network interfaces

Operating systems allow users and the applications to interact with the hardware of the computer.

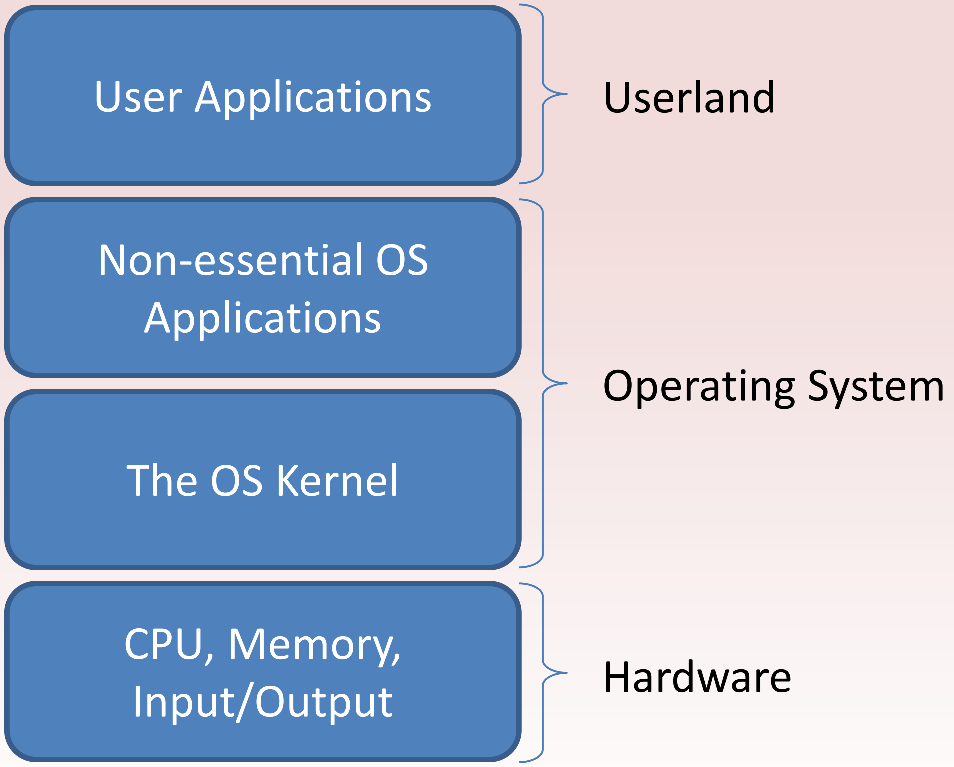
Operating systems:

- Allow application developers to write programs in high level languages
- Allow applications to run by users in a simple and consistent way
- Handle complex tasks related to security problems
  - Multi-users with different levels of access on the same computer
  - Protect against malicious activity
- Allows multiple programs to run at the same time (multitasking)
  - OS must manage the programs so that one program will not interfere with another
    - Shared resources among programs
    - Share the same file system

Thus, an OS must have measures in place so that applications cannot maliciously or mistakenly damage resources needed by other applications.

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. (keyboard, mouse, video display) Operating systems define tasks associated with the kernel in terms of layers, where the CPU, memory and I/O devices being on the bottom.



Layers of a computer system.

Input/Output Devices

The input/output devices of a computer include devices such as:

- Keyboard
- Mouse
- Video display
- Network cards
- Scanner
- WiFi interface
- Video camera
- USB port

Each of these devices is configured by a device driver, which encapsulates the details of how interaction with that device should be done.

The application programmer interface (API), allows programs to interact with I/O devices at a fairly high level, while the operating system performs the low level interactions that make the device work.

System Calls

Since user applications do not communicate directly with low-level hardware components, but instead delegate these tasks to the kernel, there must be a mechanism by which user applications can request the kernel to perform action on their behalf. In fact, there are several such mechanisms, but one of the most common techniques is known as the system calls or syscalls for short.

System calls are usually contained in a collection of programs (a library), that allows applications to use a predefined series of application programmer interfaces that define the functions for communicating with the kernel.

System Calls	
File I/O	Running application programs
Open, close, read, write	Exec (execute)

Many systems implement system calls as software interrupts, which are requests made 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.

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. The use of RAM allows multiple copies of the same program to run by having multiple processes initialized with the same program code.

Ex: *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 process is 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 happen so fast, all the active processes appear to be running at the same time to us humans.

Users and the Process Tree

Most modern computers 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, to run some program, the kernel sees this as an existing process asking to create a new process. Thus, processes are created by a mechanism called forking, where a new process is created (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 permission of its parent, unless the parent deliberately forks a new child process with lower permission than itself. Due to the forking mechanism for process creation, processes are organized in a rooted tree, known as the process tree.

Each process running on a given computer is identified by a unique non-negative integer, called the process ID (PID).

Process Privileges

To grant appropriate privileges to processes, an operating system associates information about the one whose behalf the process is being executed with each process. 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, in hexadecimal notation, that uniquely identifies each user. Typically, UID 0 is reserved for the root or administrator account. The GID is a number with 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 process are run with their EUID set to the ID of the application's owner.

Inner-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 – filesystem – so communicating this way is simple. File handling typically involves reading from or writing to an external hard drive, which is often more 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 RAM memory. As long as the kernel manages the shared and private memory space appropriately.

Two additional solutions for process communication are known as pipes and sockets. Both of these mechanisms 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. Piles act as a conduit (a channel through which something is conveyed), allowing two processes to communicate.

Signals

Unix-based systems incorporates 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 or trigger when a particular signal is received).

If a signal handler exists, then the 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.

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 rely on remote procedure calls (RPC), which allow a process to call a subroutine from another process's program. To terminate a process, Windows make use of a kernel-level API (application programmer interface), 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

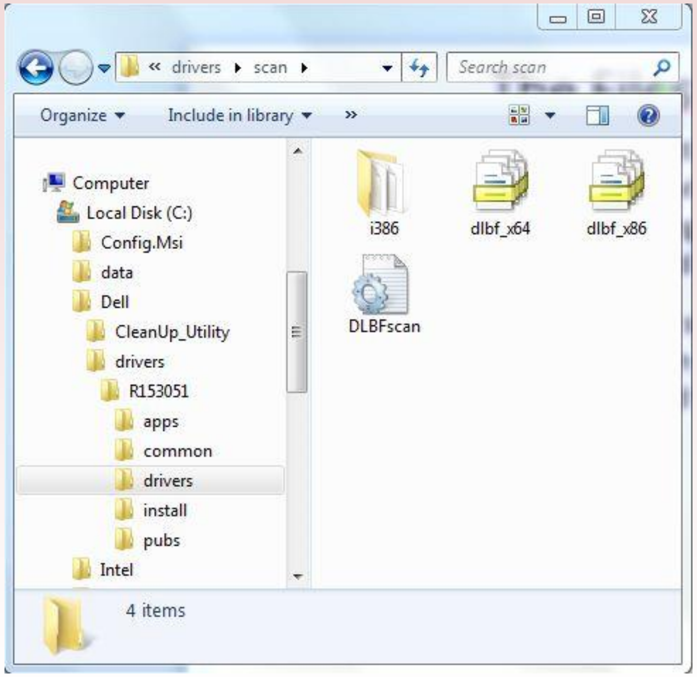
Computers today run dozens of processes that run without any user intervention. In Linux, these background processes known as daemons, and are essentially indistinguishable from any other process. The are typically started by the **init** process and operate with varying levels of permissions. Because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

In Windows, these processes are referred to as services. Unlike daemons, services are easily distinguishable from other processes, and are differentiated in monitoring software such as the Task Manager.

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. A 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.



File Access Control

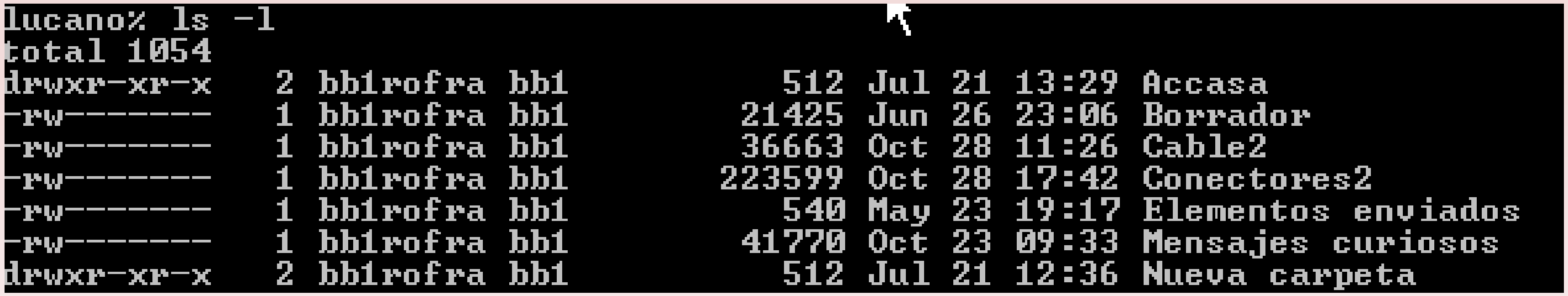
One of the main concerns of operating system security is how to determine which users can access which resources: who can read files, write date, and execute programs. In most cases, this concept is encapsulated in the notation of file permissions, whose specific implementation depends on the operating 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, writeable, 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 permission of the file.

Unix-like operating systems have a mechanism for file permission known as file permission matrix that 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.

1. Owner – responds to the UID for some user
2. Group – determines permission for users in the same group
3. Other – determine permission for users who are neither the owner of the file nor in the same group



```
lucano% ls -l
total 1054
drwxr-xr-x   2 bb1rofra bb1        512 Jul 21 13:29 Accasa
-rw-------   1 bb1rofra bb1      21425 Jun 26 23:06 Borrador
-rw-------   1 bb1rofra bb1      36663 Oct 28 11:26 Cable2
-rw-------   1 bb1rofra bb1     223599 Oct 28 17:42 Conectores2
-rw-------   1 bb1rofra bb1        540 May 23 19:17 Elementos enviados
-rw-------   1 bb1rofra bb1      41770 Oct 23 09:33 Mensajes curiosos
drwxr-xr-x   2 bb1rofra bb1        512 Jul 21 12:36 Nueva carpeta
```

Ex: $\overset{\nearrow}{\text{d}}\ \underbrace{\text{r w x}}_{\text{owner (user)}}\ \underbrace{\text{r w x}}_{\text{group}}\ \underbrace{\text{r w x}}_{\text{other}}$

folder

Unix File Permissions

The read, write and execute bits are implemented in binary, but it is common to express them in decimal notation.

- The execute bit has weight 1
- The write bit has weight 2
- The read bit has weight 4

Thus, each combination of the 3 bits yields a unique number between 0 and 7.

Ex: 3 denotes that both the execute and write bits are set, while 7 denotes that read, right and execute are all set.

Binary representation of permission

$$\text{Equals 3}\left\{\begin{array}{ccc}0 & 1 & 1\\ - & w & x\end{array}\right.$$

Binary representation of permission

$$\text{Equals 7}\left\{\begin{array}{ccc}1 & 1 & 1\\ r & w & x\end{array}\right.$$

Ex: chmod 644 yields: - r w - r - - r - -

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-base 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.

Memory Management

Another service that an operating system provides is memory management. This is the organization and allocation of the memory in a computer. When a process executes, it is allocated a region in memory known as its address space. The address space stores the program code, data, and storage that a process needs during its execution. In Unix memory models the address space is organized into five segments

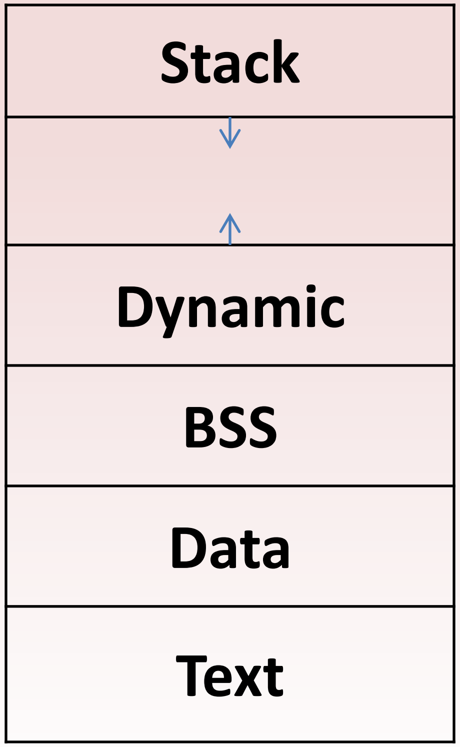
Text - actual machine code for program

Data - static program variables of source code

BSS (block started by symbol) – contains static variables that are uninitialized (or initialize to zero)

Heap (dynamic segment) – stores data generated during the execution of a process (i.e. objects created dynamically in an object-oriented program)

Stack – houses a stack data structure that grows downward and is used for keeping track of the call structure of subroutines (method, function and their arguments)



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, 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.

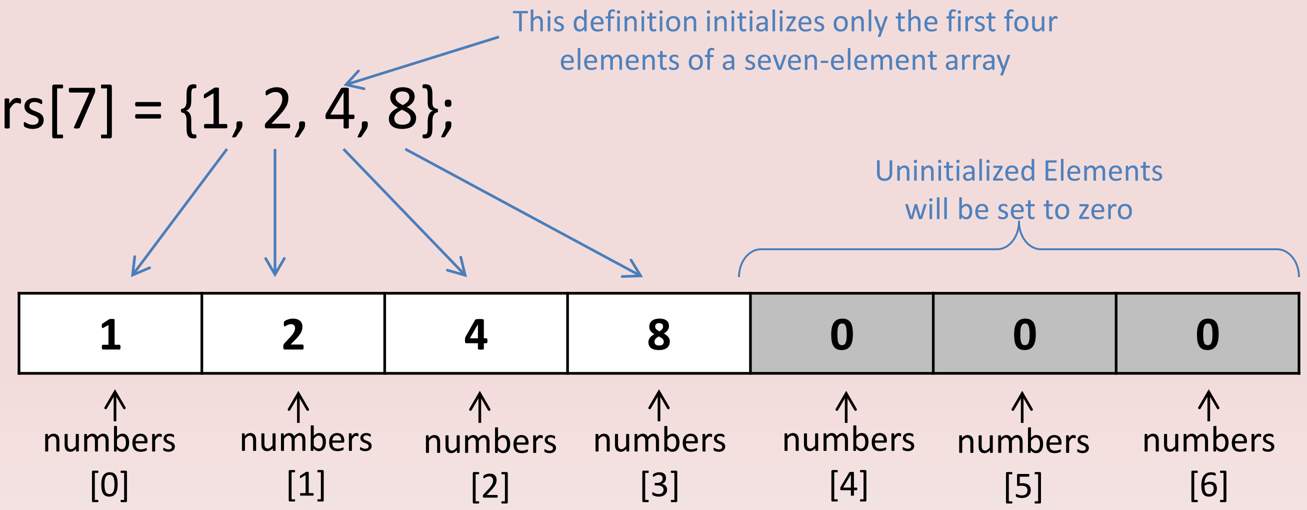
An essential rule of operating systems security is that processes are not allowed to access the address space of other processes, unless they have explicitly requested to share some of that address space with each other. If this rule were not enforced, then processes could alter the execution and data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

Unix operating systems divide the address space into two broad regions: user space, to run applications, and kernel space, for operating system functionality.

Contiguous Address Space

Each process's address space is a contiguous (sequential), block of memory. Arrays are indexed as contiguous memory blocks, so if a program uses a large array, it needs an address space for its data that is contiguous.

```
int  numbers[7] = {1, 2, 4, 8};
```



This definition initializes only the first four elements of a seven-element array
Uninitialized Elements will be set to zero
1
2
4
8
0
0
0
numbers [0]
numbers [1]
numbers [2]
numbers [3]
numbers [4]
numbers [5]
numbers [6]

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. Giving each executing process a continuous slab of real memory would be highly inefficient and in some cases, impossible.

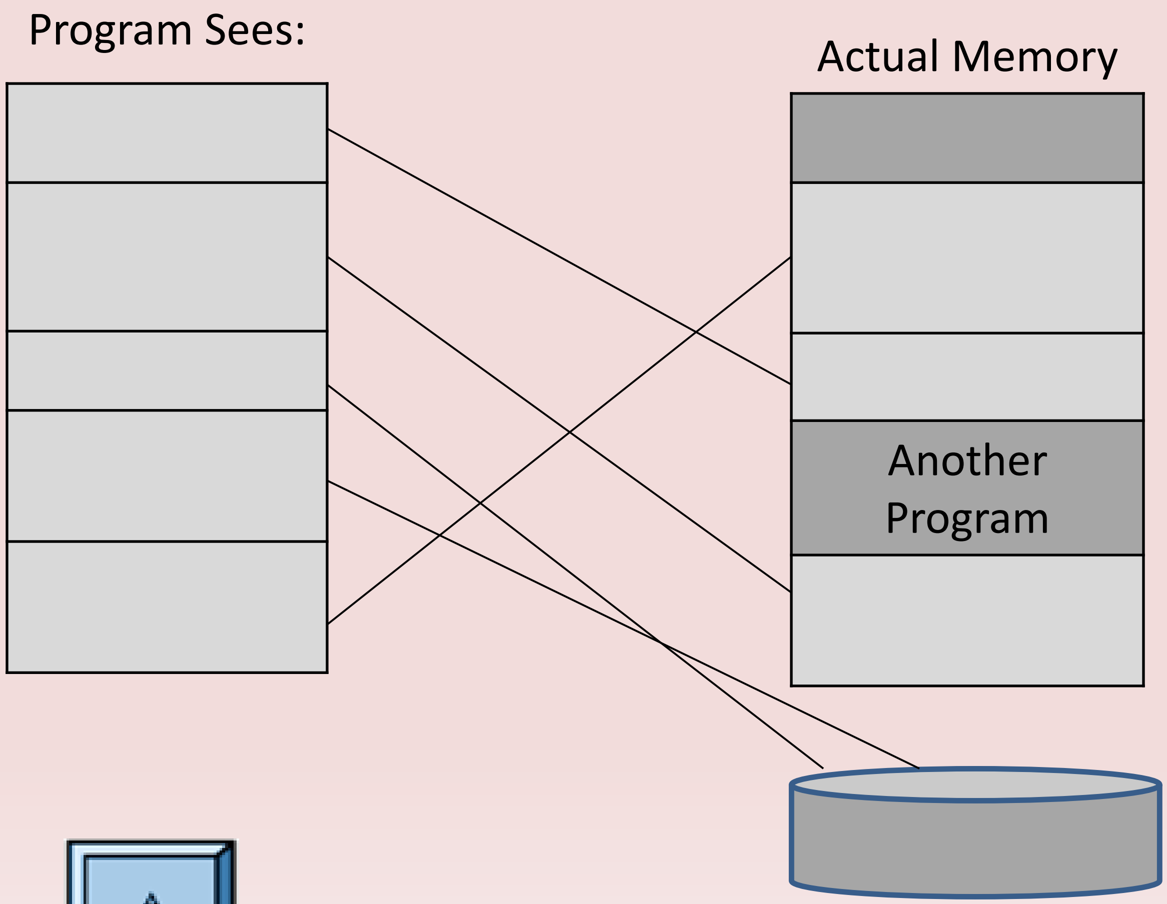
Virtual Memory

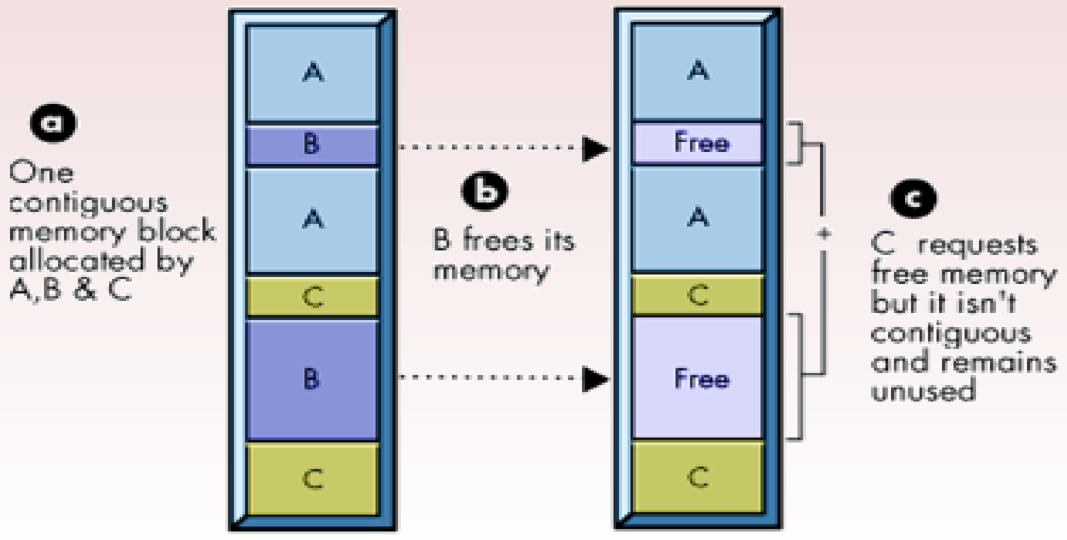
Even if all the processes had address spaces that could fit in memory, there would still be problems. Idle processes in such a scenario would still retain their respective chunks of memory, so if enough processes were running, memory would be needlessly scare.

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 is mapped to and facilitates access. Processes are allowed to act as if there memory is contiguous, when in reality it may be fragmented and spread across RAM.

Mapping Virtual Addresses to Real Addresses




a
One contiguous memory block allocated by A,B & C
A
B
A
C
B
C
b
B frees its memory
A
Free
A
C
Free
C
c
C requests free memory but it isn't contiguous and remains unused

Using RAM to allow the memory to act as if their memory is contiguous allows for several simplifications, such as supporting applications that index into large arrays as contiguous chunks of memory.

Also, virtual memory allows for the total size of the address space of executing processes to be larger than the actual size of 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 whey 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 at all time.

Page Faults

Operating systems use the hard drives 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 on 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.*)

Virtual Machines

Virtual machine technology is a rapidly emerging field that allows an operating system to run without direct contact with its underlying hardware. (Ex: These systems may allow for substantial electrical power savings, by combining the activities of several computer system 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 the guest, and the native operating system is known as the host.

Implementing Virtual Machines

There are two main implementations of VMs.

1. Emulation
2. Virtualization

Emulation. The host operating system simulates virtual interfaces that the guest operating system interacts with. Communication 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.

Advantage: can emulate a virtual environment that can support one process on a machine running an entirely different processor.

Disadvantage: It can typically have decreased performance due to the conversion process associated with the communication between the virtual and real hardware.

Advantages of Virtualization

1. Hardware Efficiency – host multiple operating systems on the same machine, ensuring an efficient allocation of hardware resources.
2. Portability – VMs provide portability; the ability to run a program on multiple different machines.
3. Security – functions as a strict sandbox that protects the rest of the machine in the event that the guest operating system is compromised.
4. Management Convenience – the ability to take snapshots of the entire virtual machine state. (*If a host becomes infected, it can be reverted back to the an earlier state from a previous snapshot.*)

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.

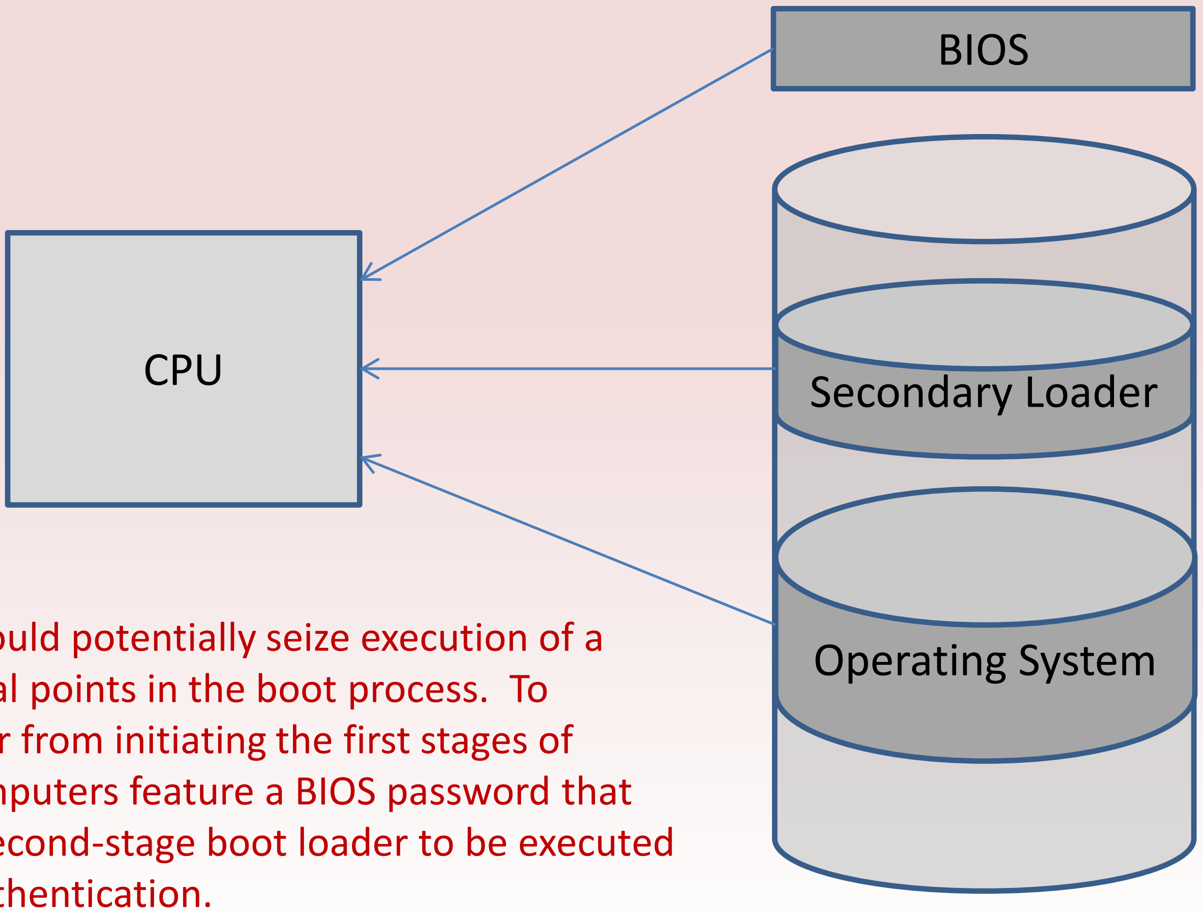
Inductive Trust from Start to Finish

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 process of loading an operating system into memory, from a powered-off state is known as booting, originally called bootstrapping. Typically, all the operating system code is stored in persistent storage (Hard Drive).

When a computer is turned on, it first executes code stored in a firmware component known as the BIOS (basic input/output system). 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.



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.

The Boot Device Hierarchy

Most second-stage boot loaders allow the user to specify which device should be used to load the rest of the operating system. 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. In doing this, one should make sure that the operating system is always booted form 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 this can 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.

Hibernation

Modern machines have the ability to go into a powered-off state known as hibernation. When a system goes into hibernation, the operating system stores the entire contents of the machine’s memory into a hibernation file on the hard drive so that the state of the computer can be quickly be 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 at root of the “C” drive, named hiberfil.sys. Researchers have found ways to reverse the compression algorithm to extract a viewable snapshot of RAM at the time of hibernation.

The Hibernation Attack

1. User closes a laptop computer, putting it into hibernation.

2. Attacker copies the hiberfile.sys file to discover any unencrypted passwords that were stored in memory when the computer was put into hibernation.

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. (NOTE: Windows does not delete the hibernation file after resuming execution. To defend against these attacks, hard disk encryption should be use to protect hibernation files, and swap files.

Monitoring, Management, and Logging

One of the most important aspects of operating system security is situational awareness.

1. Keeping track of what processes are running
2. What other machines have interacted with the system via the internet
3. If the operating system has experienced an unexpected or suspicious behavior

Implementing the steps above can allow for the gathering of important clues not only for troubleshooting ordinary problems, but also for determining the cause of a security breach.

Event Logging

Windows incorporates three sources of logs, "System" , "Application," and "Security."

1. System log - can only be written to by the operating system.
2. Application log – may be written to by ordinary applications.
3. Security log – can only be written to by a special windows service known as the Security Authority Subsystem Service.

Unix and Linux-based files have a different logging mechanism. Typically, log files are stored in **/var/log**, or some similar location. The files are saved as simple text files.

Example:

auth.log – contains records of user authentication

kern.log – keeps track of unexpected kernel behavior

Typically, writing these log files can only be done by a special **syslog** daemon.

Windows log files may allow easier handling when using Microsoft's event logging tools.

Process Monitoring

When using Windows, you can identify and terminate an application using up a lot of CPU cycles or memory, you would use the task manager. In Linux, you could use the **ps**, **top**, **pstree**, and **kill** commands.

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.

Password-Based Authentication

A standard authentication mechanism use 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 system typically keep cryptographic one-way hashes of the passwords in a password file or database. Using the one-way property of cryptographic hash functions, an attacker who gets hold of the password file cannot efficiently derive from it the actual password and has to resort to a guessing attack.

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 bases on data encryption standards known as LAN Manager hash or LM hash. But, this hashing method has weaknesses.

1. The algorithm pads a user's password to 14 characters
2. Converts all lower case letters to uppercase, and
3. Uses each of the 7-byte halves to generate a data encryption standard key.

Because each half of the user's password is treated separately, the task of performing a dictionary attack on an LM hash is made easier, since each half has a maximum of seven characters. In addition, converting all letter to uppercase significantly reduces the search space. Also, the LM hash algorithm does not include a salt, which is a cryptographic technique.

Unix-bases systems feature a similar password mechanism, and stores authentication information at /etc/passwd, possibly in conjunction with /etc/shadow. Most Unix variants use salt and are not as restricted in the choice of hash algorithm, which allow administrators to chose their preference.

Access Control Entries and Lists (File Permission)

An access control entry (ACE) for a given file or folder consists of three parts:

1. Principal – either a user or a group of users.
2. Type – allow or deny.
3. Permission – the action of a file or folder. (read, write, execute)

An access control list (ACL) is an ordered list of ACEs.

There are a number of specific implementation details that must be considered when designing an operating permission scheme. For example:

1. How do permissions interact with the file organization of the system?
2. If a file resides in a folder, does it inherit the permissions of its parent or override them with its own permissions?
3. What happens if a user has permission to write to a file but not to the directory that the files resides in?
4. How read, write and execute permissions affect folder?
5. If not folders aren’t specifically granted or denied, are they implied by default.

Linux Permissions

As stated earlier, Linux features file permission matrices, which determine the privileges various users have. All permissions that are not specifically granted are implicitly denied, so there is no need to explicitly deny permission. Owner of files are given the power to change the permissions on those files – this is known as discretionary access control (DAC).

Windows Permissions

Windows uses an ACL, access control list, 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 full control, which grants all permissions.

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 file. 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, 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 the disk. When finished, the program should issue the close system call to remove the open file description.

File Descriptor Leaks

A common programming error that can lead to serious security problems is known as a file description 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. 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 of the file descriptor was opened with.

Vulnerability problems can arise when a file is open with higher permissions and is not closed, for some reason, and the file descriptor is accessed by a process with lower permissions.