MODULE 5. COMPUTER SECURITY AND CASE STUDY OF LINUX OS

5.1 THE SECURITY PROBLEM

We have discussed earlier that each file/directory can be secured by controlling access to them. But, total security cannot be achieved by this. Security violations of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse. Malicious access has following forms:

- Unauthorized reading of data
- Unauthorized modification of data
- Unauthorized destruction of data
- Preventing legitimate use of the system

To protect the system, we must take security measures at four levels:

1. **Physical:** The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders.
2. **Human:** Users must be screened carefully to reduce the chance of authorizing a user who then gives access to an intruder.
3. **Network:** Data being transferred via network is prone to attack.
4. **Operating System:** The system must protect itself security breaches.

5.2 USER AUTHENTICATION

The major security problem for OS is authentication. The protection system depends on an ability identify the programs and processes currently executing. Generally authentication is based on one or more of three items:

- User possession (a key or a card)
- User knowledge (identifier or password)
- User attribute (fingerprint, retina, signature etc)

5.2.1 Passwords

Passwords are used to protect the data in the computer, when there are no complete protection schemes. User is asked to provide the username and password during the access. If the password matches with the one stored in the system, access is allowed.

5.2.2 Password Vulnerabilities

Though passwords are popularly used because of its easy usage and understanding, it is vulnerable to threats. They can be guessed, accidentally exposed, illegally transferred from one to other etc. To avoid these various possibilities can be tried:

- keeping non-guessable passwords
- keep changing the passwords frequently
- keeping encrypted passwords
- using One – Time Passwords (OTP)
- using biometrics authentication

5.3 PROGRAM THREATS

When a program written by one user is used by another user, misuse and unexpected behavior may occur. Following are the common methods by which such behavior may occur:

- **Trojan Horse:** A code segment that misuses its environment is called as Trojan horse. For example, a hacker would have created a fake login screen on your computer. When you enter username and password, it will store the data, but displays the message as 'Login failed'. You may feel that you have typed wrongly. That code will now terminate and actual login pops-up. You will once again give login credentials and the system logs-in. Such a code which created fake screen is Trojan Horse.
- **Trap Door:** The designer of a program or system may leave a loop-hole in the software purposefully. Only he/she knows it and tries to collect information from a particular user without their knowledge. This type of security breach is known as trap door. A trap door can also be included in a compiler.
- **Stack and Buffer Overflow:** A bug is introduced in a program such a way that, it tries to access stack or buffer beyond its limits. Hence, memory safety is violated.

5.4 SYSTEM THREATS

Most OS allows processes to spawn (creating child process) other processes. In such a situation, it is possible that OS resources and user files are misused. The methods for misusing are discussed here:

- **Worms:** A worm is a process that uses the spawn mechanism to attack system performance. It creates multiple copies of itself using system resources and locking all other processes from using the system.
- **Viruses:** It is a code embedded within a legitimate program. They spread into other programs and mess-up the system – modifying/destroying files, causing system crash etc.
- **Denial of Service:** Here, the hacker does not gain any information, but disables legitimate user from accessing system.

CASE STUDY OF LINUX OS

5.5 LINUX HISTORY

Linux is a modem, free operating system based on UNIX standards. It is developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility. Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet. It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms. The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code.

5.6 DESIGN PRINCIPLES

Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools. Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model. Main design goals are speed, efficiency, and standardization. Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification. The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior.

5.6.1 Components of Linux System

There are three major components in Linux:

- **Kernel:** It is responsible for maintaining all the important abstractions of the OS, including virtual memory and processes. Kernel code executes in *kernel mode* with full access to all the physical resources of the computer. All kernel code and data structures are kept in the same single address space.
- **System Libraries:** They define a standard set of functions through which applications can interact with the kernel. They also implement much of the OS functionality that does not need the full privileges of kernel code.
- **System Utilities:** These are the programs which perform individual specialized management tasks.

Figure 5.1 illustrates various components of Linux system.

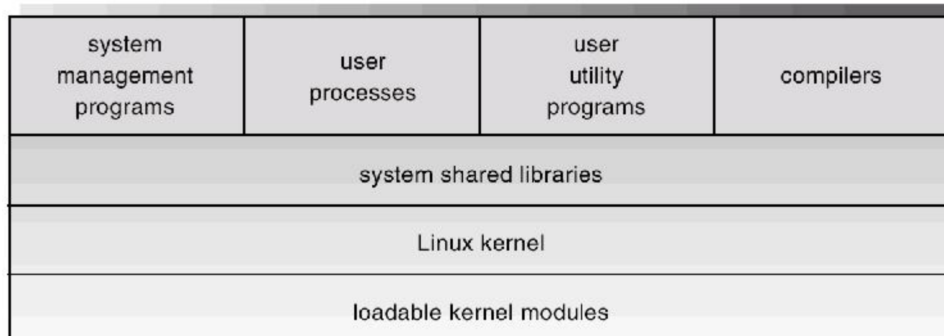

Figure 5.1 Components of Linux System

5.7 KERNEL MODULES

Kernel modules are the sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel. A kernel module may typically implement a device driver, a file system, or a networking protocol. The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL (General Public Library). Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in. Three components to Linux module support are discussed here.

- **Module Management:** It supports to load modules into memory and letting them communicate to the rest of the kernel. Module loading is split into two separate sections:
  - Managing sections of module code in kernel memory
  - Handling symbols that modules are allowed to reference

The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

- **Driver Registration:** Allows modules to tell the rest of the kernel that a new driver has become available. The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time. Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format
- **Conflict Resolution:** A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver. The conflict resolution module aims to:
  - Prevent modules from clashing over access to hardware resources
  - Prevent *autoprobes* from interfering with existing device drivers
  - Resolve conflicts with multiple drivers trying to access the same hardware

5.8 PROCESS MANAGEMENT

A process is the basic context within which all user-requested activity is serviced within the OS. To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX.

5.8.1 The Fork/Exec Process Model

UNIX process management separates the creation of processes and the running of a new program into two distinct operations. The ***fork*** system call creates a new process. A new program is run after a call to ***exec***. Under UNIX, a process encompasses all the information that the operating system must maintain and track the context of a single execution of a single program.

Under Linux, process properties fall into three groups:

- **Process Identity:** The process identity consists of mainly following items:
  - **Process ID (PID)**: The unique identifier for the process; used to specify processes to the OS when an application makes a system call to signal, modify, or wait for another process.
  - **Credentials:** Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files.
  - **Personality:** Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX.

- **Process environment:** The process's environment is inherited from its parent, and is composed of two null-terminated vectors:

- The **argument vector** lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
- The **environment vector** is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.

Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software. The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.

- **Context:** It is the (constantly changing) state of a running program at any point in time. There are various parts:
  - **Scheduling context** is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process.
  - **Accounting:** The kernel maintains accounting information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far.
  - **File table** is an array of pointers to kernel file structures. When making file I/O system calls, processes refer to files by their index into this table.
  - **File System Context:** Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.
  - **Signal-handler table** defines the routine in the process's address space to be called when specific signals arrive.
  - **Virtual-memory context** of a process describes the full contents of its private address space.

5.8.2 Processes and Threads

Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent. A distinction is only made when a new thread is created by the **clone** system call:

- **fork** creates a new process with its own entirely new process context
- **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent

Using **clone** gives an application fine-grained control over exactly what is shared between two threads.

5.9 SCHEDULING

The job of allocating CPU time to different tasks within an OS. While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks. Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

Various aspects of scheduling in Linux are discussed here.

- **Kernel Synchronization:** A request for kernel-mode execution can occur in two ways:
    - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs.
    - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

  Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section.

  Linux uses two techniques to protect critical sections:
    - Normal kernel code is non-preemptable : when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need_resched** flag is set so that the scheduler will run once the system all has completed and control is about to be returned to user mode.
    - The second technique applies to critical sections that occur in an interrupt service routines. By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures.

- **Process Scheduling:** Linux uses two process-scheduling algorithms:
    - A time-sharing algorithm for fair preemptive scheduling between multiple processes
    - A real-time algorithm for tasks where absolute priorities are more important than fairness

  A process's scheduling class defines which algorithm to apply. For time-sharing processes, Linux uses a prioritized, credit based algorithm.

- **Symmetric Multiprocessing:** Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors. To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code.

5.10 MEMORY MANAGEMENT

Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory. It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes.

- **Management of Physical Memory:** The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request. The allocator uses a *buddy-heap* algorithm to keep track of available physical pages:
    - Each allocatable memory region is paired with an adjacent partner.
    - Whenever two allocated partner regions are both freed up they are combined to form a larger region.

  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request.

  Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator).

- **Management of Virtual Memory:** The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required. The VM manager maintains two separate views of a process's address space:
  - A logical view describing instructions concerning the layout of the address space. The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space.
  - A physical view of each address space which is stored in the hardware page tables for the process.

  Virtual memory regions are characterized by:
  - The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (*demand-zero* memory)
  - The region's reaction to writes (page sharing or copy-on-write).

  The kernel creates a new virtual address space
  1. When a process runs a new program with the **exec** system call
  2. Upon creation of a new process by the **fork** system call

- **Executing and Loading User Programs:** Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made. The registration of multiple loader routines allows Linux to support both the ELF and a.out binary formats. Initially, binary-file pages are mapped into virtual memory; only when a program tries to access a given page will a page fault result in that page being loaded into physical memory. An ELF-format binary file consists of a header followed by several page-aligned sections; the ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

5.11 FILE SYSTEMS

To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics. Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS). The Linux VFS is designed around object-oriented principles and is composed of two components:

- A set of definitions that define what a file object is allowed to look like
  - The inode-object and the file-object structures represent individual files
  - the file system object represents an entire file system
- A layer of software to manipulate those objects.

5.12 INPUT AND OUTPUT

The Linux device-oriented file system accesses disk storage through two caches:

- Data is cached in the page cache, which is unified with the virtual memory system
- Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block.

Linux splits all devices into three classes:

- block devices allow random access to completely independent, fixed size blocks of data
- character devices include most other devices; they don't need to support the functionality of regular files.
- network devices are interfaced via the kernel's networking subsystem

5.13 INTER-PROCESS COMMUNICATION

Like UNIX, Linux informs processes that an event has occurred via signals. There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process. The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and wait-queue structures.

Passing of Data among Processes: The pipe mechanism allows a child process to inherit a communication channel to its parent; data written to one end of the pipe can be read by the other. Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. To obtain synchronization, however, shared memory must be used in conjunction with another Inter-process communication mechanism.

****