Processes and Signals: Section 1 Transcript

Introduction

Welcome to the Processes and Signals module. In this module, we will discuss UNIX processes; specifically, we will examine some of the most important fields of the process descriptor. We will review the purpose of signals and discuss several examples of signals that are sent to processes. Finally, we will discuss how to analyze a UNIX system process list to detect abnormal behavior.

Throughout this module, you'll be presented with opportunities to assess and apply what you've learned. At the end of this module, you will be able to:

- Explain how UNIX operating systems represent processes,
- Explain the purpose of signals in UNIX operating systems, and
- Use a process list to analyze a UNIX system.

Prerequisites

Specifically, as you are traversing through this module, you are expected to use Internet search engines and UNIX `man` pages to discover and learn about a variety of UNIX commands. Some of the commands that you need to research and become proficient with are: `chmod`, `chown`, `fuser`, `kill`, `pgrep`, `pkill`, `pmap`, `ps`, `top`, `pargs`, `pcred`, `pfiles`, `pldd`, `preap`, `prstat`, `prun`, `psig`, `pstop`, `ptime`, `ptree`, `pwdx`, `atop`, `htop`, `lsof`, and `pstree`. Note that some commands are common across UNIX and UNIX-like variants, yet other commands are specific to Solaris, Linux, etc. We suggest that you use Internet search engines and UNIX `man` pages to conduct your initial research of these and other commands.

Bypass Exam Introduction

If you are already familiar with the subject matter presented in this module, you can choose to take a Bypass Exam to skip this module.

The Bypass Exam option provides a single opportunity to successfully demonstrate your competence with the material presented within the module. If you pass, you'll receive credit for completing the module, unlocking the content within, and you will be free to proceed to the next module. If you do not pass, you will need to successfully complete the module, including all exercises and the Module Exam, to receive credit.

Click the Next Section button to continue.

Processes and Signals: Section 2 Transcript

What is a process? 1/12

As we already know, a process is a program in execution along with all the resources allocated to that execution. A process manifestation in UNIX, like any other operating system, is characterized by the program code that is being executed by the processor, the program counter and other processor registers currently involved in the execution of the code, and various data structures used to manage the process and the memory address space allocated to the process. Next we will learn about important data structures that are involved in the management of processes.

The Process Structure 2/12

UNIX operating systems maintain structures that are used to manage processes. One of the most important structures is the process descriptor. The process descriptor is a very complex structure that holds pointers to multiple other structures and is used for most of the actions the kernel performs on a process.

From an operational standpoint, some of the most important fields of the process descriptor include process numbers, credentials, process state, file descriptors table, signal descriptors table, and some others that we will discuss in this module.

Let's get started by looking at a few of these in more detail.

The Process Structure: Process Numbers 3/12

There are three numbers that are assigned to a process:

- First, the Process Identifier (PID), which is a unique number that allows the system to identify and track the process. The PID maximum value is a kernel tunable parameter on most UNIX systems. If the system creates more processes than the maximum value, PID values wrap back to the beginning and the system starts recycling them. On Solaris, the maximum value can be set in the `/etc/system` file and changes will only take effect after the next reboot. However in Linux, the changes can either be made in the `/etc/sysctl.conf` file, requiring a reboot to take effect, or by using the **sysctl** command or by directly accessing the `/proc/sys/kernel/pid_max`, to change the currently running configuration. In both cases, the maximum value set through configuration can never be bigger than the absolute maximum. That absolute maximum is set to 32768 (2^15) on 32-bit Linux, 4194304 (2^22) on 64-bit Linux, 30000 on Solaris, and 99999 on BSD variants including MacOS.
- Next is the Parent Process Identification (PPID), which is usually the PID of the process that created the current process via the *fork()* system call.
- Third is the Processes Group identification (PGID) which identifies a group of processes to which the process is assigned. The significance of a process group is for signaling purposes. A signal that is sent to a process group will be delivered to all the processes in the group. Process groups are a very important concept for situations when multiple processes share the

same resource such as a terminal where they should all be informed if the status of the resource has changed. For each new process, the default value of the process group is the same value as the process group of the parent process.

The Process Structure: Credentials

When a process is created, the kernel assigns two types of user and group identification information to the process: the Real User and Group Identifiers (RUID and RGID) and the Effective User and Group Identifiers (EUID and EGID).

The RUID and RGID determine what user on the system owns the process. The RUID and RGID of a process always match the UID and the GID of the user that started the process as they are defined in the `/etc/passwd` file. As a general rule, the RUID and RGID are inherited from parent to child processes.

The EUID and EGID are usually equal to their real counterparts. They also get inherited whenever child processes get created. However, when an executable file has its **setuid** bit set, the process that is created from that executable has its EUID set to the UID of the owner of the file. The same concept holds true for the EGID when an executable has the **setgid** bit set. A process whose EUID is 0 (the user ID of root) is called a privilege process.

Privilege processes can change their credentials to any UID and GID on the system as needed in order to impersonate any user.

Let's discuss an example to illustrate how credentials are passed from one process to another. When a user logs in, the login process, which is a root-owned process, consults the `/etc/passwd` file to get the UID and GID of the authenticated user. The login process then sets the RUID, RGID, EUID, and EGID of the shell to those values. Whenever a process gets created from the shell, which is always done via *fork()* and *exec()* system calls, the child process will usually inherit the credentials from the shell. However, if the process is created from a setuid and setgid file, the process has its EUID set to the UID of the owner of the file, and its EGID set to the GID of the owner of the file.

The Process Structure: Credentials

On most UNIX variants, the kernel mostly uses the Real IDs and Effective IDs to determine what permission to grant to the process when it performs various operations. For instance, if a non-privilege process is trying to access a file, the kernel verifies that its EUID and EGID match the UID and GID of the owner of the file to allow the action to proceed. Similarly, a non-privilege process can only be allowed to send a signal to another process if the RUID or EUID of the sending process matches the RUID of the receiving process. Privilege processes have all the privileges of the superuser and are therefore permitted to do everything.

The real and effective IDs are also used for other purposes. For example, every file created by a process has its owner information set to the effective IDs of the process.

An exception to the signal sending rule is the **init** process. The init process can only be sent signals for which it has a handler installed. This prevents the system administrator from accidentally

terminating init, which is essential to the operation of the system.

We will work with the various user and group identifiers in much more depth later when we discuss Enumeration and Hardening.

The Process Structure: Process State

In UNIX, a process goes through different states during its lifetime:

- **NEW:** The process is being created. The process descriptor is the first element about the process that gets allocated. As the kernel allocates memory resources to the process, the process descriptor gets populated until all the resources are allocated.
- **READY:** The process creation is done, and the process is added to the run queue by the scheduler.
- **RUNNING:** When the process is allocated to the CPU, it is placed into the running queue by the scheduler.
- **WAITING:** The process is waiting for some event to occur such as the result of an I/O request. The scheduler moves the process from the run queue to the sleep queue. When the event occurs, the process is set back to the READY state so it can be allocated CPU time once again.
- **TERMINATED:** When the process finishes execution or receives a SIGKILL signal, it is placed in the terminated state and all memory resources are de-allocated before it is removed from the system. In some circumstances, a process might become a **zombie**. A zombie is a process that has all its resources de-allocated or freed except its entry in the process table. The kernel maintains that entry until the parent requests the exit status of the process using the *wait()* system call. Zombies can be identified in the ps command output by "Z" in the STATE column. In Solaris, in addition of the "Z" state, zombies also have the command section set to the **<defunct>** string. To get rid of those processes, use the Solaris `preap` command.

In all UNIX variants, when a process dies, its children, including zombies, are immediately inherited by the init process. Init will then make sure of reaping all the zombies by requesting their exit status from the kernel using the *wait()* system call.

The Process Structure: File Descriptors Table

The file descriptors table maintains information about each file opened by the process. Every entry in the table is indexed by the file descriptor and is a reference to the following information: reference count, access mode, offset of next I/O operation, the filesystem inode information, etc.

The Solaris `pfiles` command can be used to retrieve information about the descriptor table. Linux systems hold the `lsof` utility which is roughly equivalent to `pfiles`. Alternatively, this information can be interrogated directly from the `/proc` directory structure.

Another very useful tool that allows the user to check all processes that are accessing a specific file is `fuser`. The `fuser` utility is available on most major versions of UNIX and on all versions of Linux.

The Process Structure: Signal Descriptors Table

UNIX operating systems use signals as a way for inter-process communication and for the kernel to send notifications to processes. For instance, a signal can be used to notify a process when network data becomes available on a socket, when a connection has been closed on the other end, or even when data is available via Standard In, which is usually after the user presses the enter key. The signal descriptors table holds a reference to the following information about each signal:

- The signal handler location in memory
- Flags that will modify the default behavior of the signal
- The signal mask that determines if the signal should be ignored or not
- The state of the signal such as blocked or pending

The `kill` command is a very trivial way to send a simple signal to a process. The `kill` command has various options depending on the version of UNIX that is used. We will learn more about signals and the kill command later on in this module.

The `trap` command can be used to handle a signal synchronously. This means that the process waits for the signal before continuing its execution. When the process installs a signal handler that will be called by the kernel later on when the signal is sent, it is said that that the signal is handled asynchronously.

The Process Structure: Other

The process descriptor also holds other information that can be useful. For example:

- **Accounting information** indicates the total CPU time used per **thread** in user mode and in kernel mode, total time used per thread, total time used by the whole process, time usage limits, and total number of file descriptors. As far as limits go, the `ulimit` command is a good starting point to restrict resource usages to a process.
- **Memory mapping information** is the virtual and physical memory address space that constitutes the process memory segments which are text or code segment, stack segment, the data segment, the heap segment, page tables, and so on. Most UNIX variants provide the `pmap` utility which is used to display information about the address space of a process.
- **Process pointers** link the process to a list of all its threads, as well as, to its parent, children, and siblings processes.
- **Priority information** is used by the scheduler to decide in what priority queue the process will be put in. The scheduler maintains queues depending on the priority value of each process. After each time slice, the scheduler uses an algorithm to decide what process gets CPU time next. Usually processes with higher priorities get selected first. A process inherits its priority from its parent's priority. However, you can run a process with a given priority, or change the priority of a running process, using the nice and `renice` commands. The priority information is listed in a process list output under the "PRI" column.
- **Environment variables** are strings that are stored in the userspace portion of the process descriptor. They are inherited by children processes by default and can be viewed using the `pargs` command.

Multithreaded Model

Historically, UNIX operating systems used a single-threaded model. In this model every process is characterized by a single execution path and processes are alternatively executed by the microprocessor in order to present the illusion of parallel execution.

As the need for better performance grew over time, multi-processor architectures were developed along with techniques to allow concurrent processes to run side by side. However, this approach results in a cost imposed by inter-process communication, if we need processes to work together. In order to alleviate the cost of inter-process communication, another technique was developed that allowed processes to share their memory address space. Concurrently, the Institute of Electrical and Electronics Engineers (IEEE) was perfecting their normalization of the thread concept that would later become a Portable Operating System Interface (POSIX) standard. Therefore, the concept of using threads involved having multiple execution paths within the same process where each execution path uses a limited set of resources within that process.

Modern implementation of the POSIX thread standard in UNIX requires each thread to have the following properties:

- Stack is allocated within the address space of the process
- Thread descriptor is allocated, which contains fields similar to those in the process descriptor and additional fields such as the thread ID for the current thread
- Threads within the same process share the same code section within the process, but can be executing a different section of that code at any given time
- Threads also share the same data section within the process, which makes communication easier and very efficient

Sending Signals to Processes

Signals are a very powerful communication mechanism provided by the kernel in all UNIX operating systems. These are used by the kernel as a process notification medium. For instance, the kernel uses the SIGCHLD signal to inform processes that a child has terminated. This is how the init process knows it is time to **respawn** another **getty** process after a user logs out.

They can also be used for interprocess communication. For instance, a process can just send a signal to another process to inform it to take some action, just like the Apache HTTP server on the diagram reloads its configuration file after receiving a SIGHUP signal.

Signals also come with a default action that a process uses to handle the signal. For example, the default action for SIGINT is for the process to exit immediately, while the default action for the SIGCHLD is to do nothing and continue processing. However, processes may set up **signal handlers**, or functions that the process will run whenever it receives the signal for which the handler was set, which overrides the default action. When that happens, it is said that the process caught the signal. Depending on the signal, failing to properly set handlers for signals can cause adverse consequences; for example, failing to handle SIGCHLD is exactly how we get zombie processes.

Note that the SIGKILL and SIGSTOP signals cannot be handled by processes; these signals are interpreted by the kernel, which takes the appropriate action regarding the process state directly,

without referring to the process itself.

Signal handling and sending is more extensive when used programmatically. However, there are two commands that are very useful to operate with signals on the command line: `kill` and `trap`.

At the most basic level, the `kill` command has the following syntax:

```
kill -s signal pid
```

Where "signal" is the number or the string representation of the signal and PID is the process ID of the process to which we are sending the signal.

Signal numbers are highly dependent on the UNIX variant. However, the SIGKILL signal number is always nine (9) on all UNIX variants and the typical syntax to send the SIGKILL signal to terminate a process is `kill -9 pid`.

To list all the available signal numbers on any UNIX variant, you can use the `kill` command with the `-l` option. The attached table provides a list of the most commonly used signals in UNIX systems.

Section Completed

Processes and Signals: Section 3 Transcript

Process List and the ps Command

The `ps` command generates a snapshot of the active processes on your system. All the information displayed by the `ps` command is retrieved from the system process table and the process descriptor for each process, via the /**proc** directory. By default, the `ps` command just lists the active processes for the current user in a short listing format.

To get all the processes for all users in the system, use the `-e` option. Additionally, you can use either the `-f` or `-l` options for a more detailed listing or a long listing. Many other powerful options are available for different purposes and for different variants of UNIX. Take a few moments to compare the options of the `ps` command for Linux and Solaris by using the `man` pages, the Internet, or both.

The following is a list of attributes that are displayed when you run a `ps` command:

- **F**: Flags associated with the process (different meaning depending on the UNIX variant,
- **S**: State of the process (varies depending on the UNIX variant),
- **UID**: The effective user ID of the process or the associated username if the `-f` option is used,
- **PID**: The process ID of the process,
- **PPID**: The parent process ID,
- **PRI**: The priority of the process,
- **NI**: The nice value (used for priority computation),
- **ADDR**: The memory address of the process,
- **SZ**: The total size of the process in virtual memory, including all mapped files and devices,
- **WCHAN**: The address of an event for which the process is sleeping (this will be blank for running processes),
- **STIME**: The starting time of the process, given in hours, minutes, and seconds,
- **TTY**: The terminal assigned to the process,
- **TIME**: The cumulative execution time for the process, given in minutes and seconds,
- **CMD**: The command name. If the `-f` option is used, the full command name and its arguments are printed up to an 80-characters limit.

Let's now discuss what type of information can be gained by dissecting the output of the `ps` command.

Using a Process List to Detect Abnormal Behavior

A typical install of a UNIX system will commonly have a core set of processes that show up in the process list. Understanding and recognizing the common normal processes will aid in identifying abnormal processes. Once the unusual processes are identified, a further analysis can be performed to determine if they are malicious or can negatively impact operations. To perform a thorough analysis, the following questions need to be answered about each process:

- What is it?
- Does it, or should it, start on boot?
- Where should it be started from?
- Does it, or should it, spawn children?
- What ports does it listen on?
- What files should it have open?

For instance, if a process is started at boot, it is an indication that it is a system service. We can then continue our analysis by looking at the child processes of the service. What children should the service spawn? Are any of those processes unusual (for example, a shell)?

Conversely, what if a process started at boot is not a typical service? You can analyze the parent of this process to determine if it is started with a legitimate service. You can also examine some of the configuration files that define the boot process, such as the `/etc/inittab` file, the rc scripts, and the various init scripts that were run.

Since we have discussed what to look for, let's now discuss some typical process list examples for various UNIX systems.

Analyzing the process list for Solaris

Let's examine this Solaris partial process list output:

- In Solaris, it is common to have **sched** as process 0 in the kernel thread. This lets us know that it is a Solaris box right off the bat.
- The init process is running in `/sbin` and not `/etc`, which lets us know that it is probably an OpenSolaris (Solaris 10 box).
- Notice that pageout and fsflush are both kernel processes (PPID == 0) started by init, they confirm further that this is a Solaris box, and pageout is the VM swapper.
- Next we have nscd, which is a Domain Name Service (DNS) cache program that is common to Solaris.
- The next two items, svc.startd and svc.configd are Service Management Facility (SMF) functions which provide further evidence that this is a Solaris 10 box.
- Then we see snmp, a User Datagram Protocol (UDP) network protocol that monitors the network for network attached devices. You will learn in the security concepts section that there is a significant amount of information you can use by querying snmp daemons.
- Next we have nfs4cbd, which is a callback daemon for network file systems.
- Then automountd is a mount point manager mainly used for network file systems and home directories.
- Finally in our example we have lockd, which is a Network File Service (NFS) file locking daemon.

Notice the PIDs in this diagram; this gives you a sense of when everything was started on boot. Using the Parent PID (PPID) field, you can (and should almost always) create "trees" of execution that allow you to trace how a particular program started. Recall that a UNIX process is started by calls to the kernel to *fork()* and *exec()* a new process image. By tracing this you can usually find missing processes and abnormalities, if they exist. Every process should have an existing parent. Note in the example that the parent of `automount` is missing!

Analyzing the process list for Linux

Let's now examine this Linux 2.6 partial process list output:

The `ps` command output fields are the same because they conform to similar standards. The processes, however, are wildly different. Unlike Solaris, Linux chooses to show many of its kernel threads, which are denoted here with square brackets. Also note that, in this distribution, the runlevel is shown after init. The slash zero in kernel threads denotes the CPU the thread is running on, and most will run copies on each processor.

Sometimes less obvious information can also be gleaned from the process list. For example, in Linux, the memory page cache threads have changed names several times. As a result, the user can tell what kernel version this system is running. If the page cache process is named [pdflush], then the system is running a Linux kernel version from 2.6.0 up to 2.6.31. As of kernel 2.6.32, the thread was renamed [flush-#:#] (where the #'s are replaced by numbers). Before kernel 2.6, the cache process was named [bdflush].

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Processes and Signals: Section 4 Transcript

Summary

You have completed the Processes and Signals module. In this module, we explained how UNIX operating systems represent processes, examined some of the most important fields in the process descriptor, explained the purpose of signals (including several examples of signals that are sent to processes), and finally we analyzed a UNIX system process list to detect abnormal behavior. In addition to these topics, as you traversed through this module, you were expected to conduct research to learn about a variety of UNIX commands. You should now be able to:

- Explain how UNIX operating systems represent processes,
- Explain the purpose of signals in UNIX operating systems, and
- Use a process list to analyze a UNIX system.

To receive credit and advance to the next module, you must achieve a passing score on the Module Exam. Click the Next Section button to begin the Module Exam.