KIT304 Server Administration and Security Assurance

Tutorial 4

Goal

In this tutorial you will focus on **processes**. You'll look at how you can determine which processes are running, which are consuming significant resources, and you'll look at a few ways to control them. Finally, you'll look at how you can automate the running of processes to perform administrative tasks.

Introduction

Unix is a multi-tasking operating system. When a program is run from the command-line, it becomes a **process**. Processes use system resources like memory, CPU, and the file system. Every command you run starts at least one new process, and there are a number of system processes that are run automatically by the system as part of the boot sequence. Some of these continue to run until the system is shut down.

Each process is uniquely identified by a **process ID** (PID). Like files, a process has an **owner** and a **group**, which determine what access rights that process has. Processes can only access files and devices to the extent allowed by the ownership and permissions set on those files and devices.

All processes have a **parent process** that spawned them. For example, the shell is a process, and any command started by the shell will have that shell as its parent process. The exception is the initial process that is started when the system boots and which always has a PID of 1. This is called **systemd**, **init**, or **launchd** (depending on the Unix family it comes from). This process does not have a parent process, but every other process is ultimately a descendant of it.

Some processes are designed to be run without continuous user input, and disconnect from the terminal during launch. For example, a web server responds to web requests rather than user input. Mail servers listen for incoming mail connections. These types of programs are often known as **daemons**. The term comes from Greek mythology, and represents an entity that is neither good nor evil, and which invisibly performs useful tasks.

Conventionally on Unix systems the names of programs that run as daemons end with a trailing **d**. For example, the **ssh** server is a daemon that listens for incoming **ssh** connections, and it is named **sshd**. The web server daemon is often named **httpd**.

In this tutorial, you will start by examining the processes that are running on the system, and specifically user-owned processes.

Activity – Processes

1. Start up both **CentOS** and **Kali**, and configure both virtual machines so that they have IP addresses. Confirm that they can see each other with **`ping`**.

2. On **CentOS**, open a terminal window and become the root user, create a new user account and give it a password, and then from a terminal window on **Kali** connect to CentOS as that new user via **`ssh`**. Finally, return to the **CentOS** virtual machine to work in the terminal window where you are the root user.

3. The first tool you are going to use today is **`ps`**, which displays **process status**. Take a few minutes to read the **`ps`** man page – **`ps`** is a very useful tool for seeing which processes are running on a system. Unfortunately, different versions of Unix use different **`ps`** options for producing similar results. The version installed on CentOS and Kali tries to incorporate support for several of these variants, and as a result the man page can be confusing. You'll see that there are a large number of options, but many Unix administrators use only a few in day-to-day operation.

   Try the following **`ps`** command variants (on **CentOS** , as root) and then write down what they are doing. Study the man page to try understand what each option does (this won't be easy, given the previously mentioned support for multiple Unix variants).

   a) **`ps -e`** _____

   b) **`ps axu`** _____

   c) **`ps`** _____

   d) **`ps au`** _____

   e) **`ps aux > /home/export`**

   _____

4. Another command you can use to view running processes is **`top`**. Switch to the **Kali** VM, and in the **`ssh`** connection you established in step 2 above (to CentOS), run the command **`top`** (so you're actually running this command on the CentOS system in a Kali window). What is the fundamental difference between **`top`** and **`ps`**?

   _____

5. Leave **top** running in the ssh session you created, open a second terminal window (still on Kali) and connect to CentOS again as the user you created in step (2). In this new session, change to the **/home/** directory and enter the command **tail –f export**.

   This will display the contents of the file you redirected **ps** command output to in step 3(e), and will then appear to stop without showing a command prompt. The addition of the **–f** flag on the **tail** command adds an interesting twist (which you can look up on your own with **man tail**).

   Back in your **CentOS** VM terminal window, which should still be the root user, run the following commands, and try to explain (both by observation, and by reading the man page for **ps**) what each are doing. For (a), explain how this differs from 3 (d) above.

   a) **ps au** _____

   b) **ps -U root u** _____

   c) **ps --ppid 1** _____

   d) **ps u -C tail** _____

   e) **ps aux | grep sshd**

   _____

6. Here is a third set of examples to explore that demonstrate the extent of what you can discover with **ps** – write down what each of them do**:**

   a) **ps axjf** _____

   b) **ps aux --sort=-pcpu,-pmem | head**

   _____

   c) **ps -e -o pid,comm,etime**

   _____

   d) **pstree**

   _____

7. The **ps** and **top** commands allow you to determine what is happening on the system. If system performance degrades because a process is using too many resources, or a user is running too

many processes, these are valuable tools to determine what is happening so that you can take appropriate action.

8. Processes on Unix systems have a **priority** which determines how much CPU time they are allocated in comparison to other processes on the system. You can see the priority in the **top** output that is still running in the first Kali window you opened – it should be the third column, labelled **PR**. Note that the *lower* a priority value, the *higher* the priority is for that process. Process priorities are determined by the **scheduler** and depend on a number of factors.

   A feature related to priority is **niceness**. Read the man page for **nice**. What does the **nice** command do?

   _____

   What are the maximum and minimum nice values?

   _____

   Launch a third terminal in Kali, and in it open another **ssh** connection to CentOS as the user you created earlier, and then use **nice** to launch a new instance of the shell with a high niceness level, like this:

   ```
   nice -n 19 bash
   ```

   This shell (and, through inheritance, every process it creates) is running with the highest niceness value, which means it (and its children) will be some of the lowest priority processes on the system.

   Next, in the bash session you just launched above with **nice**, create a process that consumes a lot of CPU so that you can see the niceness and priority values in the **top** output:

   ```
   yes > /dev/null
   ```

   This consumes as much CPU as possible, and you will see it appear in the top of the **top** process list, but since it is running at the lowest priority, almost any other process that needs the CPU will get it instead.

9. A related tool that can change an already-running processes niceness value is **renice**. To use this, you need the process id of the process you want to change the priority of. You can look this up with **ps** or **top**, although **pidof** is also an easy way to look up the process id of a program. Open a second terminal window on CentOS, become the root user, and using one of the tools just described, find the process ID of your **yes** process and **renice** it several times to see the

effect in the **top** output. Note that you can only renice processes you own (unless you're the **root** user, or have appropriate **sudo** privileges).

10. Altering a running process's priority may not actually be enough to achieve what is needed. To end a process that is currently running you can use the **kill** command.

    kill sends a **signal** to a process to request its termination. The three most common signal values are **1**, **9** and **15**. What does each of these do? (This will require that you explore some **man** and **info** pages in more detail than you have needed so far).

    1 _____

    9 _____

    15 _____

    In your second CentOS root terminal, find the PID of the **yes** process that is still consuming a lot of CPU, and send it a **SIGTERM** with **kill**. Write down the command you used:

    _____

    Open a third terminal window on CentOS, and use the **su** *username* command to become the user you created in step 2, and try to **kill** the **tail** process that is still running in the Kali terminal session with a **SIGHUP** signal. What was the last thing printed in the Kali window that was running the **tail** command?

    _____

    Enter the command **exit** in the same CentOS window you just used, so that you exit out of the **su** session and return to the **student** user session, then try to **kill** the **top** process that should still be running using a **SIGKILL** signal. What was displayed?

    _____

    Why did this happen?

    _____

11. Another way to kill processes is with the command **pkill**. What option could you use with **pkill** to kill *all* processes owned by a specific user (you would need to be the root user for this to work)?

    _____

Activity - Automating Tasks

When administering a system there are a wide range of tasks that need to be undertaken – for example, printing reports, rotating log files, or backing up user data. Some of these tasks can be automated through the use of scripts or commands. Often you need to execute these commands or scripts at times when the system is not under heavy use (such as early in the morning), or you might need to repeat a task on a set schedule. In this next section you will explore two of the tools that let you do this.

12. The first tool is called **`at`**, which (among other things) can be used to execute a command at a specific time. Have a look at its **`man`** page and write below which version of the command lists pending jobs:

    _____

13. When you use **`at`**, you specify a time in the future at which a command should be run, followed by (on separate lines) the command (or commands) that you want to run at that time. Write down the command that will create a job at 9am on your birthday, that will write a birthday message into a text file in your home directory:

    > ➢ Note: Type control-D to leave the **`at`** console when you have finished typing the command(s) you want to run at the scheduled time.

    _____

    _____

    > ➢ Note: You can't use **`at`** to schedule commands that interact with the terminal, since there's no guarantee that a user would be logged in at the time that the event is scheduled to run, and there's no way to indicate which terminal the interaction should happen on. That's why the suggestion above is to just write the message to a file.

14. In addition to being able to schedule events at specific dates and times with **`at`**, you can specify relative time values. For example, the command **`at now + 1 minute`** schedules an event to be executed one minute in the future. Try a few variations of this to gain some experience with **`at`**.

    How can you remove a scheduled event?

    _____

15. What variant of **at** allows you to schedule an activity to occur once there is a drop in system activity, and therefore enough resources to complete it?

    _____

16. Clearly **at** is a useful tool, but it only runs the command one time, and then removes it from the queue. Often, a system administrator will want to run a command periodically. Unix has a powerful subsystem called **cron** for this purpose.

    Using the **ps** tool you should be able to locate a daemon called **crond**. This daemon starts up at boot time, and spends most of its time sleeping, but it wakes up every minute to kick off any cron jobs that need to be executed at specific times.

    There are two kinds of cron jobs: system cron jobs, and user cron jobs. System cron jobs are stored in **/etc/crontab** and the **cron.hourly**, **cron.daily**, **cron.weekly**, and **cron.monthly** folders inside the **/etc/** directory. You won't be creating any system cron jobs today but open the **/etc/crontab** file and study how it describes the syntax required to schedule a cron job. This same syntax is used to define user cron jobs.

    What is the syntax of a line from the **crontab** file?

    _____

    If a ***** (asterisk) symbol is used in any of the time fields in the cron configuration file, it is treated as a "wildcard", and refers to every instance of that value. How often will a cron job run if the first 5 fields are all set to *****?

    _____

    Each time field can be specified as a numerical value – so a **2** in the day category would specify that the command should be run on the 2nd day of every month.

    The files stored in the **cron.hourly** and related folders are for system cron jobs. If you look inside each of these directories, you will see that some already contain entries. Only the **root** user is able to set up system cron jobs, and they execute at the specific interval indicated by the folder name.

    User cron jobs that can also be managed with by the **crontab** tool. This is a vi-based editor (like **visudo** that was covered in an earlier tutorial). This also includes the root user whose cron jobs are separate to the system cron jobs. The only syntactic change from system cron to user cron is that you don't specify a username – this is because cron runs the job as the user who created the crontab file.

crontab creates and maintains individual cron settings files in **/var/spool/cron/**. This directory is initially empty, but is populated with separate cron settings files for individual users as each user establishes their own cron jobs with the **crontab** command.

> ➢ You should always use the **crontab** tool to edit a cron settings file, and not edit it directly.

17. Here are three crontab entries. Use the **crontab -e** command to add them to your user crontab file (as a non-root user), and write under each one what it does.

```
* * * * * echo "Hello World" >> HelloWorld
```

_____

```
*/2 * * * * date >> minutes.txt
```

_____

```
*/15 10-14 * * * date >> other.text
```

_____

Each of these activities have been simple commands that append text to files in your home directory. But they could be checking whether or not a database is running, testing a network connection, or sending an email if some condition is met. cron jobs can be shell commands, or they can be scripts or programs you've developed.

18. Every time a cron job is run, a record of the event is created in the log file at **/var/log/cron**. If you examine this file (you'll need to be root), you can see all of the cron jobs that have been run to date. If you are waiting for one of your own cron tests to run, use **tail -f /var/log/cron** to monitor the log file, and hence see when the job is started.

19. Both **crontab** and **at** are useful tools, but there may be situations where a system administrator might not want all of their users to be able to schedule jobs to be executed when they are not around. Both tools use a simple whitelist/blacklist approach to control this.

    If a whitelist approach is required, the files **/etc/cron.allow** and **/etc/at.allow** must be created. Then, whenever a user then attempts to use **crontab**, the system checks to see if the user attempting to use it is listed in there. If they are, then they are permitted to create or modify cron jobs. If the user's name is not listed, then they are prevented from using **crontab**. The same behaviour applies for the **at** command with **/etc/at.allow**.

If a blacklist approach is required, where most users are permitted to use these tools, but certain specific users are not, then the files **/etc/cron.deny** and **/etc/at.deny** files should be used instead. If **/etc/cron.deny** exists, and a user wanting to use **crontab** is listed in it, that user will be blocked from running **crontab**. If they are not listed in it, then they are permitted to use **crontab**. The same behaviour applies for **/etc/at.deny**.

If neither of the **.allow** and **.deny** files exist for either **at** or **cron**, then only the root user can use that feature. Conversely, if both files exist, and a user is listed in both of them, then that user can use that feature.

On the CentOS VM, the **/etc/cron.deny** file already exists, so the default mode is a blacklist approach – you add to this file the names of users you do not what to be able to use **cron**. In the CentOS window logged in as the **student** user, verify that you can run the crontab command, and therefore schedule jobs for periodic execution. Next, add the student user to **/etc/cron.deny** using the vi editor (you'll need to be the root user to do this). Now try again as the student user to run **crontab**. What message did you get?

Conclusion

This tutorial has exposed you to more depth in relation to the **ps** command, and process management in general, than you would previously have seen, and then gone on to look at process automation.

Real-world systems are routinely configured to run tasks at different times of the day, the week, and the month. Such tasks might include backing up a database on a regular basis, ingesting data from an external source, rotating system logs so that they do not fill up the system volume, or processing data on a schedule.

Understanding process management and process automation is an important role for system administrators. Hopefully this tutorial has given you a taste of these activities, and the level of flexibility that exists.

Skills to Master

To successfully complete this tutorial, you must have mastered the following skills:

- using the **ps** and **top** commands to identify processes consuming significant resources
- using **kill** and **pkill** to terminate processes in different ways
- controlling process priorities with **nice** and **renice**
- scheduling jobs to run at various times with **at**, **batch**, and cron

- controlling access to cron on a per-user level with **`/etc/cron.allow`** and **`/etc/cron.deny`**

Anything identified in this skills section could be part of an assessment item in this module.