**CISS 360**

**Assignment 2 - Process Script**

**he script must include at least three functions.**

**Some examples of comparisons the script may perform:**

**- Display the two processes most nearly identical in terms of run-time, memory size, etc.**

**- Display the processes which closely match specific input parameters for process attributes.**

**- Display processes which were launched from a particular directory or directory tree, or which were launched from similar locations.**

**- Display processes in adjacent or nearly adjacent memory locations.**

**- Display processes where Usernames have matching characteristics.**

**- Create an on-going list of all process names which have been used on the system.**

The first step is often the hardest, but don’t let that stop you. If you’ve ever wanted to learn how to write a shell script but didn’t know where to start, this is your lucky day.

If this is your first time writing a script, don’t worry — shell scripting is not that complicated. That is, you can do some complicated things with shell scripts, but you can get there over time. If you know how to run commands at the command line, you can learn to write simple scripts in just 10 minutes. All you need is a text editor and an idea of what you want to do. Start small and use scripts to automate small tasks. Over time you can build on what you know and wind up doing more and more with scripts.

Starting Off

Each script starts with a “shebang” and the path to the shell that you want the script to use, like so:

#!/bin/bash

The “#!” combo is called a shebang by most Unix geeks. This is used by the shell to decide which interpreter to run the rest of the script, and ignored by the shell that actually runs the script. Confused? Scripts can be written for all kinds of interpreters — bash, tsch, zsh, or other shells, or for Perl, Python, and so on. You could even omit that line if you wanted to run the script by sourcing it at the shell, but let’s save ourselves some trouble and add it to allow scripts to be run non-interactively.

What’s next? You might want to include a comment or two about what the script is for. Preface comments with the hash (#) character:

#!/bin/bash

# A simple script

Let’s say you want to run an rsync command from the script, rather than typing it each time. Just add the rsync command to the script that you want to use:

#!/bin/bash

# rsync script

rsync -avh --exclude="\*.bak" /home/user/Documents/ /media/diskid/user\_backup/Documents/

Save your file, and then make sure that it’s set executable. You can do this using the chmod utility, which changes a file’s mode. To set it so that a script is executable by you and not the rest of the users on a system, use “chmod 700 scriptname” — this will let you read, write, and execute (run) the script — but only your user. To see the results, run ls -lh scriptname and you’ll see something like this:

-rwx------ 1 jzb jzb 21 2010-02-01 03:08 echo

The first column of rights, rwx, shows that the owner of the file (jzb) has **r**ead, **w**rite, and e**x**ecute permissions. The other columns with a dash show that other users have no rights for that file at all.

Variables

The above script is useful, but it has hard-coded paths. That might not be a problem, but if you want to write longer scripts that reference paths often, you probably want to utilize variables. Here’s a quick sample:

#!/bin/bash

# rsync using variables

SOURCEDIR=/home/user/Documents/

DESTDIR=/media/diskid/user\_backup/Documents/

rsync -avh --exclude="\*.bak" $SOURCEDIR $DESTDIR

There’s not a lot of benefit if you only reference the directories once, but if they’re used multiple times, it’s much easier to change them in one location than changing them throughout a script.

Taking Input

Non-interactive scripts are useful, but what if you need to give the script new information each time it’s run? For instance, what if you want to write a script to modify a file? One thing you can do is take an argument from the command line. So, for instance, when you run “script foo” the script will take the name of the first argument (foo):

#!/bin/bash

echo $1

Here bash will read the command line and echo (print) the first argument — that is, the first string after the command itself.

You can also use read to accept user input. Let’s say you want to prompt a user for input:

#!/bin/bash

echo -e "Please enter your name: "

read name

echo "Nice to meet you $name"

That script will wait for the user to type in their name (or any other input, for that matter) and use it as the variable $name. Pretty simple, yeah? Let’s put all this together into a script that might be useful. Let’s say you want to have a script that will back up a directory you specify on the command line to a remote host:

#!/bin/bash

echo -e "What directory would you like to back up?"

read directory

DESTDIR=

This e-mail address is being protected from spambots. You need JavaScript enabled to view it

:$directory/

rsync --progress -avze ssh --exclude="\*.iso" $directory $DESTDIR

That script will read in the input from the command line and substitute it as the destination directory at the target system, as well as the local directory that will be synced. It might look a bit complex as a final script, but each of the bits that you need to know to put it together are pretty simple. A little trial and error and you’ll be creating useful scripts of your own.

Hope this may help you

1. **Overview**

For this assignment the student will write a BASH script that helps to manage processes and/or threads.

1. **Script Requirements**

Each student will write a Bash script which provides information to the user comparing the statuses of processes and or threads.

The specific information to be provided to the user will be determined by the student, but must include at least three different values.

The script will use the information from the /proc directory to provide the output.

The script will support multiple uses, which will include a help option which will display basic usage information.

Some examples of comparisons the script may perform:

- Display the two processes most nearly identical in terms of run-time, memory size, etc.

- Display the processes which closely match specific input parameters for process attributes.

- Display processes which were launched from a particular directory or directory tree, or which were launched from similar locations.

- Display processes in adjacent or nearly adjacent memory locations.

- Display processes where Usernames have matching characteristics.

- Create an on-going list of all process names which have been used on the system.

The script must allow for and use at least 2 command line arguments (in addition to help).

* The command line arguments may either be used as a part of the determination (e.g. user or dates)
* Or, to change the behavior of the script (e.g. count remote vrs. local access).

The script must include at least one function.

The script will follow the “Scripting Best Practices” as provided.

1. **Process**

The bash script needs to be built and tested using appropriate entries. In addition, it is recommended that the student have another student test their script to ensure it works correctly.

Submit a report following the standard format, along with a copy of the script.

With the script completed and tested, discuss how the script could be used by a security professional to aid in spotting, researching, or analyzing a potential security issue.

1. **References**

The following sources are useful in developing BASH scripts:

Garrels, Machtelt. *Bash Guide for Beginners*. 2008. <http://tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html#sect_01_02>

Chadwick, Ryan. *Ryans Tutorials: Bash Scripting Tutorial*. <https://ryanstutorials.net/bash-scripting-tutorial/>

Banas, Derek. Shell Scripting Tutorial. YouTube. 2016. <https://www.youtube.com/watch?v=hwrnmQumtPw>

Nemeth, Evi et al. *Unix and Linux System Administration Handbook, 4th Ed*. Prentis Hall Upper Saddle River, NJ. 2011.

Finally, type: ***man bash*** at the command prompt on any Linux/Unix system.

The following resources are useful for figuring out what is in /proc:

Krul, Rob. *TechMint: Exploring /proc File System in Linux*. 2015. <https://www.tecmint.com/exploring-proc-file-system-in-linux/>

The Linux Documentation Project. *Linux Filesystem Hierarchy: Chapter 1. Linux Filesystem Hierarchy: 1.14. /proc.*  2018. <https://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html>

**Scripting Best Practices**

When run with inappropriate arguments, scripts should print a usage message and exit. The same message could be printed with a **–help** argument.

Validate inputs and sanity check derived values.

Return an appropriate exit code: zero for success and nonzero for failure.

Use appropriate naming conventions for variables, scripts, and routines.

User variable names that reflect the values they store.

Start every script with a comment block that tells what the script does and what parameters it takes. Include your name and the date. If the script requires non-standard tools, libraries, or modules to be installed, list those as well.

Comment at a useful level. More complexity requires more comments.

Error messages should:

* Go to STDERR, not STDOUT.
* Include the name of the program that’s issuing the error.
* State the function of operation that failed.
* If a system call fails, include the perror string.
* Exit with a code other than 0.