IT 231 FALL 18

CYBERSECURITY ANALYST (CYSA+)

Project #4: Bash Scripting

Document Version: 2018-9-12

The development of this document is funded by the Information and Communications Techology/Digital Media Sector grant #16-158-006 from the California Community Colleges Chancellor's Office, Workforce and Economic Development Division.

Contents

Introduction

The Bash scripting language is the glue that connects together the various commands in the Linux operating system. While not mandatory to learn your understanding of how Linux works and what you can do with it will increase massively by having at least a basic working knowledge of Bash. The ability to use the shell to automate repetitive tasks, solve basic problems, and parse/interact with files are in demand skills associated with both security and basic systems administration in the Linux world.

Lab Topology

We will be doing all of our scripting on the CentOS7-Class-VM machine.

Client	Password
user1	abc123
root	password

1 Getting Started with Bash

1.1 Variables

Variables are a way to store a piece of data and associate it with an easy to read name that we can use to retrieve that data later. We chose the name so that it is easy to understand what the data that is stored in it represents. For example it is easy to understand what is stored in this variable

Name=”John Gardner”

Let try it out:

Create a variable (Note: No $ when setting):

```
Name=”John Gardner”
```

Output the variable to the screen (Note: Use the $ to retrieve the data in the variable):

```
printf "$Name\n"
John Gardner
or
echo $Name
```

Output the character count (length) of a string:

```
printf  ${#Name}”\n”
13
Or
echo ${#Name}
```

To store a string including special characters [Note: put the $ before the string for output]:

```
test="Hello\nWorld\n"

printf "$test"
Hello
World
```

1.2 Interacting with the User

1.2.1 Output:

Using printf instead of echo:

http://wiki.bash-hackers.org/commands/builtin/printf

Echo automatically inserts a new line character (old way)

```
echo "Hello World"
```

Printf doesn't insert a new line character unless you tell it to (\n).
Note: ALWAYS use quotes when using printf

```
printf "Hello World\n"
```

Learn more: https://linuxconfig.org/bash-printf-syntax-basics-with-examples

1.2.2 Input:

Request input from the user and store it in varFirstName

```
read -p "Enter your first name: " FirstName
```

Learn more: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_08_02.html

1.2.2.1 Input validation, case, OR (||)

Input data need to be validated before using it. The following script demonstrate how we can validate the input.

```
nano input_validation.sh
#!/bin/bash
read -p "Enter the month# at which you were born  1-12: " month
while [[ $month -lt 1 || $month -gt 12 ]]
do
        read -p "Enter a valid month# 1-12: " month
done
case $month in
    1)  printf "You were born in January\n"
        ;;
    2)  printf "You were born in February\n"
        ;;
    3)  printf "You were born in March\n"
        ;;
    *)  printf "You were born between April and December\n"
esac

bash input_validation.sh
chmod u+x input_validation.sh
```


./input_validation.sh

1.2.2.2 Input validation, AND (&&), NOT (!)

nano input_validation_1.sh

```
#!/bin/bash
read -p “Enter your Age: “ age
while ! [[ $age -gt 0 && $age -lt 110 ]]
 do
        read -p “Enter a valid age: “ age
done
printf “Your age is  $age years\n”
```

```
bash input_validation_1.sh
chmod u+x input_validation_1.sh
./input_validation_1.sh
```

1.2.2.3 Assignment (==, =, ++, --)

nano assignment.sh

```
#!/bun/bash
read -p “Enter your Name: “ Name
if [[ $Name == “John Gardner” ]]
  then
    printf “Your name is John Gardner\n”
else
   printf “Your name is $Name\n”
fi
```

1.2.2.4 Algorithm and Data Structure

Data structures are constructs for storing and organizing data in a computer so that this data can be stored and then fetched again efficiently. Array is the simplest type of data structure. Algorithm is a step-by-step procedure to solve a problem. As an example, write a script to find the smallest number in an array of integers. Here is the algorithm: Save the first element in the array in a variable call it smallest. Then compare each element in the array with smallest. If the element value is smaller than smallest then assign the value of the element to smallest. At the end smallest contain the value of the smallest element in the array. This is not the most efficient algorithm.

nano smallest.sh

```
#!/bin/bash
num=(100 50 90 110 5 60 55 70 75 120)
smallest=${num[0]}
for i in ${num[*]}
do
        if [[ $i -lt $smallest ]]
        then
                smallest=$i
        fi
done
printf “The smallest number in the array is $smallest\n”
```


bash smallest.sh
chmod u+x smallest.sh
./smallest.sh

1.2.2.5 sequential and parallel execution

Parallel programming involves the concurrent computation or simultaneous execution of processes or threads at the same time. While Sequential programming involves a consecutive and ordered execution of processes one after another. The following example demonstrate sequential execution to find the largest number.
https://mivuletech.wordpress.com/2011/01/12/difference-between-sequential-and-parallel-programming/

```
nano sequential.sh
#!/bin/bash
read -p “Enter the first number: “ num1
read -p “Enter the second number: “ num2
read -p “Enter the third number: “ num3
largest=$num1
if [[ $num2 -gt $largest ]]
  then
      largest=$num2
 fi
if [[ $num3 -gt $largest ]]
then
     largest=$num3
fi
printf “The largest number is $largest\n”
```

bash largest.sh
chmod u+x largest.sh
./largest.sh

1.2.2.6 XOR

Encryption routines can use a variety of cryptographic functions and logical operations. One such technique is the XOR. Using the XOR function, which of the following is correct?

10101100

00110101

a. 01100110

b. 10011001

c. 10101100

d. 10010011

1.2.2.7 Bounds Checking, Type Checking, and Parameter Validation

In computer programming, bounds checking is any method of detecting whether a variable is within some bounds before it is used. Type checking is done by the type checker which verifies that the type of a construct (constant, variable, array, list, object) matches what is expected in its usage context. This ensures certain types of programming errors will be detected and reported. For example consider the following expression involving modulo operator 8 % 3.5 this expression will result in error as modulo operator expects two integers. Parameter validation is the process to validate the accuracy of parameters passed to a module. Parameter validation can be used to defend against cross-site scripting attacks. The following script demonstrate bounds checking.

nano example0.sh

```
#!/bin/bash
read –p “Enter your Age 0-120 : “ Age
if [[ $Age –ge 0 && $Age –le 120 ]]
        then
            printf  “You are $Age years old\n”
        else
                while [[ $Age –lt 0 || $Age –gt 120]]
                    do
                      read –p “Enter a valid Age 0-120: “ Age
                    done
             printf  “You entered a valid age\n”
fi
```

1.2.3 Combining Input/Output:

```
read -p "Enter your first name: " FirstName
read -p "Enter your last name: " LastName

printf "Your first name is $FirstName\nYour last name is
$LastName\n"
```

1.3 Arrays [Lists]

Arrays are a way to store a set of related data under a single variable name. Think about a list of names that we might want to store.

What the “myArray” array looks like:

[0]				[1]				[2]		
0	**1**	**2**	**3**	**5**	**6**	**7**	**8**	**10**	**11**	**12**
M	i	k	e	B	i	l	l	B	o	b

Put data in the array elements:

```
WeekArray=(Monday Tuesday Wednesday Thursday Friday)
```

Note:Curly brackets {} are how bash can tell the difference between a variable (a single element) and an array.

Print out the full array:

```
printf "${WeekArray[*]}"
Monday Tuesday Wednesday Thursday Friday
```

To get the number of elements in the array:

```
printf "${#WeekArray[*]}"
5
```

Print out individual elements in an array:

```
printf "${WeekArray[0]}"
Monday

printf "${WeekArray[1]}"
Tuesday

printf "${WeekArray[2]}"
Wednesday
```

1.4 foor loop [Repetition]

1.4.1 Basic For Loop (count to 10)

{1..10} Generates an array of numbers 1 - 10. We print 1 array element out per line.

Script:

```
for num in {1..10}
do
        printf "$num\n"
done
```

Single Line:

```
for num in {1..10}; do printf "$num\n"; done
```

1.4.2 Loop over an array

We build the array our self and then loop over it printing one array element per line.

Script: nano example1.sh


```
#!/bin/bash
myArray=(Monday Tuesday Wednesday Thursday)

for day in ${myArray[*]}
do
    printf "$day\n"
done
```

bash example1.sh
chmod u+x example1.sh
./example1.sh

Single Line: nano example2.sh

```
#!/bin/bash
myArray=(Monday Tuesday Wednesday Thursday Friday)
for day in ${myArray[*]}; do  printf "$day\n";  done
```

bash example2.sh
chmod a+x example2.sh
./example2.sh

1.4.3 Read a file one line at a time with a loop

Note: You need to run this one as root (sudo su)

su - root

Script: nano example3.sh

```
#!/bin/bash
for line in $(cat /etc/shadow)
do
        printf "$line\n"
done
```

bash example3.sh
chmod a+x example3.sh

Single Line:nano example3a.sh

```
#!/bin/bash
for line in $(cat /etc/shadow); do printf "$line\n"; done
```

bash example3a.sh

1.4.4 Generate a list of IP Addresses with a loop

This will generate a class C subnet address range.

Script: nano example4.sh

```
#!/bin/bash
for host in {1..255}
do
    printf "192.168.1.$host\n";
done
```

bash example4.sh
chmod u+x example4.sh

./example4.sh

Single Line: nano example4a.sh

```
#!/bin/bash
for host in {1..255}; do printf "192.168.1.$host"\n; done
```

bash example4a.sh

1.5 IF [Conditional Decisions]

In order to understand how the if statement works in Bash we have to take a look at how boolean (true/false) values work and what the exit status of a function is.

1.5.1 Boolean Values

In bash the boolean values (true/false) are actually commands. (IMPORTANT!!!)

- true - running this sets an exit status of 0 [SUCCESS]
- false - running this sets an exit status of 1 [FAIL]

The if statement that we will be studying later uses these values to run code based on the result true or false.

Ex: nano example5.sh

Note: we will be studying the "if" statement later in the course, so no need to understand this thoroughly now.

```
#!/bin/bash
if true
then
	printf "Boolean true was entered\n"
else
	printf "Boolean false was entered\n"
fi
```

bash example5.sh
chmod u+x example5.sh
./example5.sh
echo $?

1.5.2 Exit Status Variable

$? = This stores the return value (code) of the last command that you ran. This is nothing more than a special variable used to store the return value.

Note: The command "true" has an exit status of 0 and "false" has an exit status of 1. This is how bash handles boolean values in Linux.

Ex: using true and false commands: nano example6.sh


```
#!/bin/bash
true
echo $?

false
echo $?
```

bash example6.sh
chmod u+x example6.sh
./example6.sh

Ex: Using ls command: nano example7.sh

```
#!/bin/bash
ls example1.sh
echo $?
!0 (result = 0 Success)

ls example1a.sh
echo $?
!2 (result = 2, Error)
```

bash example7.sh

1.5.3 if [Conditional "true/false" decision making]

"if" statements are how we make decisions using a script.

Note: the "if" statement is looking for the exit code of the command the is run right next to it. An exit status of 0 (or boolean "true") will run the "then" code block, an exit status of 1 (or boolean "false") will run the else statement.

nano example8.sh

```
#!/bin/bash
if true
then
        printf "It is True\n"
else
        printf "IT is False\n"
fi
```

bash example8.sh

1.5.4 What can we do with this?

Test if a file exists from a list of files

Study This: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

Create a file containing a list of file paths:

```
echo /etc/passwd >> my_test_files.txt
echo /etc/group >> my_test_files.txt
echo /etc/shadow >> my_test_files.txt
echo /etc/somecrap >> my_test_files.txt
```

Script to check if files exist: nano example8.sh

```
#!/bin/bash

# read in a list of filenames from a file you create
for file in $(cat my_test_files.txt)
do
        if [ -a $file ]
                then
                        printf "file is here:  $file\n"
                else
                        printf "file does not exist:  $file\n"
        fi
done
```

bash example8.sh
chmod example8.sh
./example8.sh

Test if a given user exists

Note: What were are checking for here is the value of $?. It is automatically returned when we run a command. If the user exists, id will return 0, if the user does not exist id will return an error code > 0.

One Liner:

```
if id -u [User Name] &> /dev/null; then printf "Im here\n";
else printf "No User\n"; fi
```

Script: nano example9.sh

```
#!/bin/bash

if $(id -u $1 &> /dev/null)
then
        printf "User exists\n"
else
        printf "User does not exist\n"
fi
```

bash example9.sh
chmod example9.sh
./example9.sh

Test number of lines in a file

Note: This example tests if there are more than 50 lines in a file.

One liner:

```
if [ $(wc -l /etc/passwd | cut -d " " -f 1) -gt "50" ]; then
printf "Is > 50\n"; else printf "Not > 50\n"; fi;
```

Script: nano example10.sh

```
#!/bin/bash

if [ $(wc -l /etc/passwd | cut -d " " -f 1) -gt "50" ]
then
       printf "Is > 50\n"
else
       printf "Not > 50\n"
fi
```

bash example10.sh
chmod u+x exapmle10.sh
./example10.sh

Test if you are root

One Liner:

```
if [ $(whoami) != "root" ]; then printf "I am not root\n"; else
printf "I am root\n"; fi
```

Script: nano p1.sh

```
#!/bin/bash

if [ $(whoami) != root ]
      then
            printf "I am not root\n"
else
            printf "I am root\n"
fi
```

chmod u+x p1.sh
./p1.sh

1.6 Mixing Loops/Conditionals to Make Tools

1.6.1 Putting a test in your loop

Note: For loops work great for counting lists of things. If you can look at the problem you are trying to solve and say "This needs to be done this many times" you want to use a for loop.

Sometimes you only want to loop until you find a specific record.

Note: you need to use (( )) to tell bash to process this as numbers
if you use [[ ]] 10 will be processed as a string and be < 5

Number Hunt: nano p2.sh

```
#!/bin/bash


for num in  {1..10}
do
  if (($num < 5))
  then
        printf "$num is < than 5\n"
  elif (($num == 5))
  then
        printf "$num is = to 5\n"
  else
        printf "$num is > than 5\n"
  fi
done

printf "We are done\n"
```

chmod u+x p2.sh

./p2.sh

1.6.2 While loop [Run until you stop it]

Note: While loops work great for watching an undefined number of things until a given event happens, and then exiting. If you look at a problem you are trying to solve and say "I need to keep running this until a variable changes" use a while loop.

nano p3.sh

```
#!/bin/bash

state=true

while $state
do
        read -p "Type text or type END to close program: " input

        if [ "$input" == "END" ]
        then
                printf "We are done\n"
                break
        else
                printf "You entered $input\n"
        fi
done
```

chmod u+x p3.sh

./p3.sh

Another way to read in a text file one line at a time [Note this could make a great loop for password cracker :)]

nano p4.sh

```
#!/bin/bash

while read line
do
       "printf $line\n"

done < my_test_files.txt
```

chmod u+x p4.sh

./p4.sh

2 Lab Assignment - Building a Custom Password Cracker

Using what we have learned about combining loops and conditional tests lets build a simple password cracker.

2.1 Create a password file (one word per line)

```
echo timmy >> my_password_file.txt
echo password >> my_password_file.txt
echo starwars >> my_password_file.txt
echo mypassword >> my_password_file.txt
echo anotherword >> my_password_file.txt
```

Let's take a look at our file

```
cat my_password_file.txt
```

your output will look like this.

```
timmy
password
starwars
mypassword
anotherword
```

2.2 Write a simple password cracker

Write a script that will read in the password file that you created in the last step test each line against the "$SavedPassword" variable. You can do this using a for loop or (like in this example) a while loop.

Study the two while loop examples right before this lab assignment to get an idea of how to solve this challenge.

Start with this: nano p5.sh

```
#!/bin/bash

# The saved password that you want to match
pass='mypassword'

while read line
do
if [ $line == $pass ]
  then
       printf "The password is $line\n"
       break
fi

done < my_password_file.txt
```

chmod u+x p5.sh

to test if your script work correctly change pass to my_password and run the script again. A better way to write this script is shown below. nano p6

```
#!/bin/bash
# The saved password that you want to match
pass='mypassword'
found='False'
while read line
do
if [ $line == $pass ]
  then
       found='True'
       break
fi

done < my_password_file.txt
if [  $found == 'True' ]
    then
        printf "The password is $line\n"
else
        printf "The password is not found\n"
fi
```

---

chmod u+x p6

./p6

Note: I did not use .sh to show you the file extension is not required.

References:

A Fantastic basics of Linux Shell guide
http://www.linuxcommand.org/index.php

A great FAQ page
http://mywiki.wooledge.org/BashFAQ?action=show&redirect=BashFaq