Bash Scripting Knowledge Check

Bash is the default shell for most UNIX systems. Therefore, as a cyber w arfare operator it w ill be necessary for you to understand Bash scripting.

This course w ill follow the outline below. Each section w ill have a basic lesson and then one or more know ledge checks per area. Answ ers can be found in the accompanying document.

Course Outline

Variables

A variable is defined w ith '=', and there cannot be spaces betw een the variable name, equal sign, and value of the variable.
Strings can be defined w ith single or double quotes:
Single quotes: '<text>' *fully* captures the text, meaning the text is taken as literal characters.
Double quotes: "<text>" *partially* captures the text, meaning the text w ill expand variables.
A pound sign '#' begins a comment, w hich is not interpreted by Bash. These are simply human-readable notes.

Example:

```
#!/bin/bash
# A simple script to print strings
MY_STR='Hello World!'
echo $MY_STR
echo "My string is: $MY_STR"
echo 'My string is: $MY_STR'
```

Output:

```
Hello World!
My string is: Hello World!
My string is: $MY_STR
```

There are also a number of special variables that Bash can use:

- $0 - The name of the Bash script
- $1 - $9 - The first 9 arguments provided to the Bash script
- $# - How many arguments w ere passed to the Bash script
- $@ - All the arguments supplied to the Bash script
- $? - The exit status of the most recently run process

- $$ - The process ID of the current script
- $USER - The username of the user running the script
- $HOSTNAME - The hostname of the machine the script is running on
- $SECONDS - The number of seconds since the script w as started
- $RANDOM - Returns a different random number each time is it referred to
- $LINENO - Returns the current line number in the Bash script

We can save the output of a command to a variable like so: `variable=$( <command> )`. This is know n as command substitution.

> ## *Knowledge Check*
>
> ```
> #!/bin/bash
> MY_STR="Hi there!"
> echo 'My string is: $MY_STR'
> echo "My string is: $MY_STR"
> ```
>
> 1. What w ill be the first line output by this script?
> 2. What w ill be the second line output by this script?
>
> `./example_script 0 1 2`
>
> ```
> #!/bin/bash
> echo $0
> echo "Printing $3, $2, $1"
> echo $#
> ```
>
> 3. What w ill be the first line output by this script?
> 4. What w ill be the second line output by this script?
> 5. What w ill be the third line output by this script?

Arithmetic

The operators +, -, *, /, var++, var--, and % can be used to add, subtract, multiply, divide, increment, decrement, and modulus, respectively.

The command `let` evaluates an arithmetic expression and can be used to save the result to a variable.

Example:

```
#!/bin/bash
# Demonstrate usage of let with arithmetic expressions
let a=5+4
echo $a # prints 9
let "a = 5 + 4" # quotes with let allows us to use spaces
echo $a # prints 9
let "a = $1 + 30" # recall $1 is the first argument provided to the script
echo $a # prints the result of $1 plus 30
```

Output:

Calling `./example_script 9` w ill produce the follow ing:

```
9
```

```
9
39
```

The command `expr` also evaluates an arithmetic expression but instead of saving to a variable it prints the answer.

Example:

```
#!/bin/bash
# Demonstrate usage of expr with arithmetic expressions
expr 5 + 4 # you don't use quotes with expr, and must have spaces
expr "5 + 4" # if quotes are used, expr simply prints the expression
expr 5+4 # if spaces are not used, expr simply prints the expression
expr 5 \* 4 # here, Bash requires an escape of the asterisk for multiplication
a=$( expr 5 + 4 ) # expr can be combined with command substitution
echo $a
```

Output:

```
9
5 + 4
5+4
20
9
```

Double parentheses can also be used to evaluate an arithmetic expression, like `expr`. This syntax can be combined with command substitution like so: `variable=$(( <expression> ))`

Note, you cannot simply write a line like `b=a+3` in Bash like you can in other programming languages. Bash must be given a command to evaluate the expression with mechanisms like `let`, `expr`, or `(( <expression> ))`. In fact, the error you will receive is `command not found`.

Example:

```
#!/bin/bash
# Demonstrate usage of (( )) with arithmetic expressions
a=$(( 4 + 5 )) # the double parentheses evaluates the arithmetic expression
echo $a # print 9
a=$((3+5)) # with this syntax, we can omit the spaces and it still works
echo $a # prints 8
b=$(( a + $a )) # you can choose whether to use $ with a variable, either works
echo $b # prints 16
(( b++ )) # the double parentheses will evaluate this arithmetic expression
echo $b # prints 17
c=$(( 4 * 5 )) # here, Bash does not need to escape the asterisk for multiplication
echo $c # prints 20
```

Output:

```
9
8
16
17
20
```

This is not truly arithmetic, but `${#var}` is extremely useful. It produces the length of the variable (its number of characters). For example, if `a='Hello World'`, then `${#a}` is 11. If `b=4953`, then `${#b}` is 4.

> ## *Knowledge Check*
>
> `./example_script 2 3`
>
> ```
> let "num_1 = $1 * $2"
> echo $num_1
> expr $1 \* $2
> expr $1 * $2
> num_2=$(( num_1 / $1 ))
> echo $num_2
> ```
>
> 1. What will be the first line output by this script?
>   - ☐ 2
>   - ☐ 3
>   - ☐ 6
>   - ☐ Error
> 2. What will be the second line output by this script?
>   - ☐ 2
>   - ☐ 3
>   - ☐ 6
>   - ☐ Error
> 3. What will be the third line output by this script?
>   - ☐ 2
>   - ☐ 3
>   - ☐ 6
>   - ☐ Error
> 4. What will be the fourth line output by this script?
>   - ☐ 2
>   - ☐ 3
>   - ☐ 6
>   - ☐ Error
>
> ```
> #!/bin/bash
> expr $RANDOM % 100
> ```
>
> 5. What will this script output?
> 6. Write a script to convert a fahrenheit value to celsius, to the nearest integer. (Hint: pass the fahrenheit value to the script as a command line argument.) For a challenge, output the answer to 3 decimal places.

Comparisons and Operators

Bash makes the Boolean comparison OR using double pipes '||' and AND using double ampersands '&&'.

The following conditionals can be used in bash. Note the different usage depending on whether you're comparing numbers or comparing strings.

Description	Numeric Comparison	String Comparison
less than	-lt	<
greater than	-gt	>

equal	-eq	= or ==
not equal	-ne	!=
less or equal	-le	N/A
greater or equal	-ge	N/A

Square brackets are a reference to the command `test`. That command evaluates whatever is in the square brackets and exits with a status. A status of 0 means it exited with success; a status of 1 means it exited with failure. In other words:
0 -> True
1 -> False
(If you've programmed in other languages, take note. This convention to return 0 for success is the same as defined in C. Don't confuse it with the typical Boolean association of 0 with False and 1 with True.)

Example:

```
#!/bin/bash
string_a="UNIX"
string_b="GNU"
echo "Are the strings $string_a and $string_b equal?"
[ $string_a = $string_b ]
echo $? # Recall from above, this is the exit status of the most recently run process
```

Output:

```
Are the strings UNIX and GNU equal?
1
```

Example:

```
#!/bin/bash
num_a=100
num_b=100
echo "Is $num_a equal to $num_b?"
[ $num_a -eq $num_b ]
echo $?
```

Output:

```
Is 100 equal to 100?
0
```

In addition to the comparators for numbers and strings shown above, you can use these operators for more advanced functionality:

Description	Operator
length of string is greater than zero (not null)	-n <string\>
length of string is zero (empty)	-z <string\>
file exists	-e, -a <file\>

file is regular i.e. not a directory or device	-f <file\>
file size is greater than zero (not empty)	-s <file\>
file is a pipe	-p <file\>
file is a symbolic link	-h, -L <file\>
file is a block device	-b <file\>
file is a character device	-c <file\>
file is associated w ith a terminal device	-t <file\>
check if stdin in a given shell is a terminal	-t 0
check if stdout in a given shell is a terminal	-t 1
check read, w rite, execute permission	-r, -w, -x <file\>

Knowledge Check

```
#!/bin/bash
[ -z $1 ]
echo $?
```

1. What is the output if the script is called like so?: `./example_script Hello`
   - ☐ 0
   - ☐ 1
   - ☐ Hello
   - ☐ No output
2. What is the output if the script is called like so?: `./example_script`
   - ☐ 0
   - ☐ 1
   - ☐ Hello
   - ☐ No output

If Statements

There are tw o common w ays to w rite an if statement:

```
#!/bin/bash
if [<test>]; then
  echo "The test returned True"
fi

if [<test>]
then
  echo "The test returned True"
fi
```

It's programmer's choice w hich w ay to w rite the statement. Also, it is optional to indent the contents of the if statement. How ever, indention is the accepted convention for readability, so it's the best practice. It becomes even more important w ith nested If statements.

Example:

```
#!/bin/bash
num_a=100
num_b=200
if [ $num_a -lt $num_b ]; then
    echo "$num_a is less than $num_b!"
fi
```

Output:

```
100 is less than 200!
```

> ## *Knowledge Check*
>
> ```
> #!/etc/bash
> num_a=1000
> num_b=1001
> if [ $num_a != $num_b ]
> then
>   echo "$num_a and $num_b are not the same"
> fi
> ```
>
> What is the output of this script?
>
> - [ ] 1000 and 1001 are not the same
> - [ ] No output

If / Else if / Else:

For more complicated scenarios, Bash can evaluate an `if` and if that's false, check an 'else if' expression using `elif`. If that's false, it can check another 'else if', and if false, another 'else if', and so on. Finally, if none of the expressions have been true, Bash can execute the commands under `else`.

Example:

```
#!/bin/bash
# Demonstrate the use of if, elif, and else
if [ $1 -ge 18 ]
then
  echo You may go to the party.
elif [ $2 == 'yes' ]
then
  echo You may go to the party but be back before midnight.
else
  echo You may not go to the party.
fi
```

Output:
`./example_script 23`

```
You may go to the party.
```

`./example_script 17 yes`

```
You may go to the party but be back before midnight.
```

`./example_script 17 no`

```
You may not go to the party.
```

Combined with the Boolean operations already described, you can evaluate more complicated situations.

Example:

```
#!/bin/bash
# Check if a file is both readable and has a size greater than zero.
if [ -r $1 ] && [ -s $1 ]; then
  echo This file is useful.
fi
```

To evaluate expressions with OR or AND, wildcards, or REGEX, you can also use double brackets [[ ]]:

Example:

```
#!/bin/bash
if [[ $1 == n* || $1 == N* ]]
then
  echo "The string starts with 'n' or 'N'"
else
  echo "The string does not start with 'n' or 'N'"
fi
```

> ## *Knowledge Check*
>
> ```
> #!/etc/bash
> if [ $1 -gt $2 ]
> then
>   echo "$1"
> elif [ $2 -gt $1 ]
> then
>   echo "$2"
> else
>   echo "--"
> fi
> ```
>
> What does this script do?

For Loops

The for loop is a loop that iterates over each of the items in a given list. For each item in the list it will perform the given set of commands between the `do` and `done`.

Example:

```
#!/bin/bash
# A simple loop that will print out three names
names='Jim Bob John Greg'

for name in $names
do
  echo $name
done
```

Output:

```
Jim
Bob
John
Greg
```

A for loop can also iterate over a series of numbers. The series of numbers is specified betw een tw o parentheses and the start value and end value are separated by tw o periods. E.g. `for value in {1..5}`. Additionally, the value to increment or decrement can also be specified after the end value like this: `for value in {0..10..2}`. Also note that the start value can be greater than the end value in order to count dow n. E.g. `for value in {5..1}`. Examples show ing the output for these different types of ranges are show n below.

Example:

```
#!/bin/bash
for value in {1..5}
do
  echo $value
done
```

Output:

```
1
2
3
4
5
```

Example:

```
#!/bin/bash
for value in {0..10..2}
do
  echo $value
done
```

Output:

```
0
2
4
6
8
10
```

Example:

```
#!/bin/bash
for value in {5..1}
do
  echo $value
done
```

Output:

```
5
4
3
2
1
```

There are a number of other lists that the for loop can iterate over:

- Numeric range specified w ithout braces: `for val in 1 2 3 4 5`
- List of strings not in a variable: `for val in string1 string2 string3`
- Output of a linux command: `for val in $( <command> )`
- C-like for loop: `for (( c=1; c<=5; c++ ))`

> ## *Knowledge Check*
>
> ```
> #!/bin/bash
> for val in {1..10}
> do
>   echo $val
> done
> ```
>
> 1. What w ill be the first line output by this script?
> 2. What w ill be the fifth line output by this script?
>
> `./example_script /usr`
>
> ```
> #!/bin/bash
> for entry in $(ls $1)
> do
>   echo $entry
> done
> ```
>
> 3. What w ill be the first line output by this script?
> 4. Briefly describe w hat this script is doing

While Loops

The w hile loop evaluates an expression. As long as the expression is true it keeps executing the commands betw een `do` and `done`.

Example:

```
#!/bin/bash
counter=1
while [ $counter -le 5 ]
do
  echo $counter
  let "counter = counter + 1"
done
```

Output:

```
1
2
3
4
5
```

In this case the expression evaluated is `$counter -le 5`. As long as $counter is less than 5 the loop w ill continue to execute.

> ## *Knowledge Check*
>
> ```
> #!/bin/bash
> counter=5
> while [ counter -gt 0]
> do
>   echo $counter
> done
> ```
>
> 1. What w ill be the first line output by this script?
> 2. What w ill be the fifth line output by this script?
>
> ```
> #!/bin/bash
> counter=1
> while [ $counter -lt 20]
> do
>   echo $counter
>   if [ $(($counter % 2)) -eq 0]; then
>     echo "even"
>   fi
>   let "counter = counter + 1"
> done
> ```
>
> 3. What w ill be the fourth line output by this script?
> 4. What is the sixth line output by this script?
>
> ## Functions
>
> There are tw o formats to w rite a function. Either is valid; the choice is simply programmer's preference.

```
#!/bin/bash
function_name () {
  <commands>
}

function function_name {
  <commands>
}
```

If you choose the implementation with parentheses, keep in mind they are purely decorative in Bash. You won't define arguments here as in other languages.

Another important note: the function definition must occur before any calls to the function within the script.

Calling a function is as simple as typing the name. To pass arguments, simply list them afterwards, like when using a command in the command line. Those arguments are referenced in the function just like command line arguments: `$#` for the number of arguments; `$0` for the function name itself; `$1`, `$2`, etc. for the arguments themselves; and so on.

Bash functions don't return values, per se, like other languages. They can, however, set a return status. That return status can be captured. Typically, an exit status of zero indicates success and a nonzero exit status indicates failures (as you saw earlier), so returning an integer value this way is not the intended purpose but it will work.

Example:

```
#!/bin/bash
# Demonstrate the use of a function returning an integer
print_thing () {
  echo Hello $1
  return ${#1} # recall this is the length of a variable
}

print_thing World
echo "Printed $? characters"
print_thing "Alice and Bob"
echo "Printed $? characters"
print_thing Let\'s\ Escape
echo "Printed $? characters"
```

Output:

```
Hello World
Printed 5 characters
Hello Alice and Bob
Printed 13 characters
Hello Let's Escape
Printed 12 characters
```

An alternative way to capture a result is to have the function print the result (and only the result) paired with command substitution.

Example:

```
#!/bin/bash
lines_in_file () {
  cat $1 | wc -l
}

num_lines=$( lines_in_file $1 )
echo The file $1 has $num_lines lines in it.
```

Output:

```
user@bash#: cat groceries.txt
Tomatoes
Lettuce
```

```
Bread
user@bash#: ./example_script groceries.txt
The file groceries.txt has 3 lines in it.
```

By default, variables in Bash are global. You can define a local variable with `local var_name=<var_value>`.

> ## *Knowledge Check*
>
> ```
> #!/bin/bash
> my_function() {
>   if [ $# -ne 2 ]; then
>     echo "Incorrect"
>   else
>     echo $(($1 + $2))
>   fi
> }
>
> my_val=$(my_function 4 5)
> echo "My first value is $my_val"
> my_val=$(my_function 1 2 3)
> echo "My second value is $my_val"
> ```
>
> 1. What is the first line of output of this script?
>   - [ ] `My first value is 4`
>   - [ ] `My first value is 5`
>   - [ ] `My first value is 9`
>   - [ ] `My first value is Incorrect`
> 2. What is the second line of output of this script?
>   - [ ] `My second value is 2`
>   - [ ] `My second value is 3`
>   - [ ] `My second value is 6`
>   - [ ] `My second value is Incorrect`

Input

To collect input from the user, simply use `read`. With the `-p` flag, you can include a prompt. The `-s` flag keeps the input silent.

Example:

```
echo Hello, what is your name?
read name
echo Hi $name, it\'s nice to meet you.

read -p "How old are you? " age
echo Got it! You are $age years old.
```

Output:

```
Hello, what is your name?
(your input)
Hi (your input), it's nice to meet you.
How old are you? (your input)
Got it! You are (your input) years old.
```

> ## *Knowledge Check*
>
> 1. Write a simple script to collect a username and passw ord from the user. Make sure the passw ord is not visible as the user types it in.
> 2. Now, develop that script to ask for the passw ord tw ice, confirming that the input w as the same both times. Keep asking until the passw ord has been input correctly. (Hint: make use of w hile loops, if/else statements, and user input.)

A script can also send to or receive data via a pipe. The most straightforw ard w ay to do this is by using the system shortcuts `/dev/stdin`, `/dev/stdout`, and `dev/stderr`.

Example:

`salesdata.txt`

```
Susie oranges 5 7 July
Frida apples 20 4 July
Mark watermelons 12 10 July
Terry peaches 7 15 July
```

`summary.sh`

```
#!/bin/bash
echo Here is a summary of the sales data:
echo ------------------------------------

cat /dev/stdin | cut -d' ' -f 2,3 | sort
```

Output:

```
user@bash#: cat salesdata.txt | ./summary.sh
Here is a summary of the sales data:
------------------------------------
apples 20
oranges 5
peaches 7
watermelons 20
```

File Operations

Bash can also be used to interact w ith files on the operating system including reading and w riting to files. This can be especially useful to search for specific information in text files or to clean logs.

Reading a File

A common w ay to read in a file is to do it line-by-line and assign that line to a variable. Here is an example of the syntax. Using `IFS=` before `read` prevents leading and trailing w hitespace from being trimmed. The file being read is /path/to/txt/file and each line of the file is assigned to the variable `line` as it's read. The `-r` flag is commonly included as it prevents backslash escapes from being interpreted. This w ay, the file is read in as-is and its original contents are preserved.

Example:

```
#!/bin/bash
input="/path/to/txt/file"
```

```
while IFS= read -r line
do
  echo "$line"
done < "$input"
```

The filename can also be piped into the w hile loop using cat instead of redirecting it.

```
#!/bin/bash
input="/path/to/txt/file"
cat $input | while IFS= read -r line
do
  echo "$line"
done
```

Output:

```
user@bash#: cat test.txt
Cat
Dog
bird
dinosaur
  these are animals
```

Running either of the previous tw o scripts w ith the path to test.txt as `input` yields:

```
Cat
Dog
bird
dinosaur
  these are animals
```

Writing to a File

Writing to a file w ill make use of the `>` and `>>` operators. Simply redirect your text or variables to the file.

Exam ple:

```
#!/bin/bash
file="/tmp/out.txt"
echo "Text going to output file" > $file
```

Output:

```
user@bash#: ./example_script
user@bash#: cat /tmp/out.txt
Text going to output file
```

> ## *Knowledge Check*
>
> 1. Write a simple script that w rites to a file in "/tmp/" named "bin_files". The first line of the file should have the text "These are the files located in /bin". The next lines should be all the files in your /bin directory. Finally, the last line should have the text "There are files in /bin" w here is the number of files.

> 2. Now, develop a script that will read in the "/tmp/bin_files" file you just created, echo out every filename that starts with a "cu", count the number of files that begin with "d", and output that number.
>   (Hint: if statements using "==" behave differently in double brackets [[==]]. Recall you can use wildcard matching i.e. `[[ test == tes* ]]`)

Capstone

In this final section, you'll need to put together everything you've learned so far. Some of these questions may also require outside research.

> ### *Knowledge Check*
>
> //: # (*### Determine What a Script Does*)
> //: # (Add harder scripts that incorporate all the concepts covered and ask what the output or effects of the script would be.)
>
> ### *Write Your Own Scripts*
>
> **Survey Script**
> Write a script to survey a target Linux box. You may assume you are root. Here are some ideas for what your script can do:
>
> - Check who is logged in / how many users are logged in
> - Check how long the system has been up
> - Check system load averages
> - Check the date of the systems
> - Collect system information e.g. OS and conversion
> - Enumerate the root directory's contents
> - Enumerate the default logging directory's contents
> - Gather the process list
> - Capture the system's network connections
> - Capture its listening ports/services
> - Check file system disk usage
> - Check current remote mounts
> - Collect user-, password-, and group-related files
> - Gather the system's modules
> - Gather the system's services
> - Collect every user's cron jobs
> - Compress all this data so it can be exfil'd
>
> **Log Cleaning Script**
> Write a script to clean logs. Apply it to your survey script so you clean up your activity on the system. Here are some ideas for what your script can do:
>
> - Erase all or a particular number of recent commands from bash history
> - Clean any line with a particular IP address from every log in /var/log/
> - For all logs in /var/log/, remove any lines with a timestamp from the past 2 minutes
>   (Hint: use the `date` command and `egrep`)
>
> //: # (**base64 Conversion Script**)
> //: # (Write a script to read in a file and create a base64-encoded version of the file.)
>
> //: # (**Encrypt / Decrypt Script**)
> //: # (Write a script or scripts to encrypt and decrypt a file with a key.)