Bash Scripting

[Bash](https://en.wikipedia.org/wiki/Bash_(Unix_shell)) is the scripting language we use to communicate with Unix-based OS and give commands to the system. Since May 2019, Windows provides a [Windows Subsystem for Linux](https://docs.microsoft.com/en-us/windows/wsl/about) that allows us to use Bash in a Windows environment. It is essential to master the language to work efficiently with it. The main difference between scripting and programming languages is that we don't need to compile the code to execute the scripting language, as opposed to programming languages.It is also essential to learn how to combine several commands and work with individual results. This is where scripting comes in, increasing our speed and efficiency. Like a programming language, a scripting language has almost the same structure, which can be divided into:

* Input & Output
* Arguments, Variables & Arrays
* Conditional execution
* Arithmetic
* Loops, Branches
* Comparison operators
* Functions
* Debugging

Let us look at such a script and see how they can be created to get specific results. If we execute this script and specify a domain, we see what information this script provides.

A computer screen shot of a computer

Description automatically generated

**Conditional Execution:**

Conditional execution allows us to control the flow of our script by reaching different conditions. This function is one of the essential components. Otherwise, we could only execute one command after another.

When defining various conditions, we specify which functions or sections of code should be executed for a specific value. If we reach a specific condition, only the code for that condition is executed, and the others are skipped. As soon as the code section is completed, the following commands will be executed outside the conditional execution. Let us look at the first part of the script again and analyze it.

A screen shot of a computer program

Description automatically generated

In summary, this code section works with the following components:

* #!/bin/bash - Shebang.
* if-else-fi - Conditional execution.
* echo - Prints specific output.
* $# / $0 / $1 - Special variables.
* domain - Variables.

The conditions of the conditional executions can be defined using variables ($#, $0, $1, domain), values (0), and strings, as we will see in the next examples. These values are compared with the comparison operators (-eq) that we will look at in the next section.

**Shebang**

The shebang line is always at the top of each script and always starts with "#!". This line contains the path to the specified interpreter (/bin/bash) with which the script is executed. We can also use Shebang to define other interpreters like Python, Perl, and others.

A blue and black background

Description automatically generated with medium confidence

**If-Else-Fi**

One of the most fundamental programming tasks is to check different conditions to deal with these. Checking of conditions usually has two different forms in programming and scripting languages, the if-else condition and case statements. In pseudo-code, the if condition means the following:

if [ the number of given arguments equals 0 ]

then

Print: "You need to specify the target domain."

Print: "<empty line>"

Print: "Usage:"

Print: " <name of the script> <domain>"

Exit the script with an error

else

The "domain" variable serves as the alias for the given argument

finish the if-condition

**If-Only.sh - Execution**

1372000@htb[/htb]**$** bash if-only.sh 5

1372000@htb[/htb]**$** bash if-only.sh 12

Given argument is greater than 10.

When adding Elif or Else, we add alternatives to treat specific values or statuses. If a particular value does not apply to the first case, it will be caught by others.

**If-Elif-Else.sh**

**#!/bin/bash**

value=$1

if [ $value -gt "10" ]

then

echo "Given argument is greater than 10."

elif [ $value -lt "10" ]

then

echo "Given argument is less than 10."

else

echo "Given argument is not a number."

fi

**If-Elif-Else.sh - Execution**

1372000@htb[/htb]**$** bash if-elif-else.sh 5

Given argument is less than 10.

1372000@htb[/htb]**$** bash if-elif-else.sh 12

Given argument is greater than 10.

1372000@htb[/htb]**$** bash if-elif-else.sh HTB

if-elif-else.sh: line 5: [: HTB: integer expression expected

if-elif-else.sh: line 8: [: HTB: integer expression expected

Given argument is not a number.

Arguments, variables and Arrays:

The advantage of bash scripts is that we can always pass up to 9 arguments ($0-$9) to the script without assigning them to variables or setting the corresponding requirements for these. 9 arguments because the first argument $0 is reserved for the script. As we can see here, we need the dollar sign ($) before the name of the variable to use it at the specified position. The assignment would look like this in comparison:

1372000@htb[/htb]**$** ./script.sh ARG1 ARG2 ARG3 ... ARG9

ASSIGNMENTS: **$**0 $1 $2 $3 ... $9

**Special Variables**

Special variables use the [Internal Field Separator](https://bash.cyberciti.biz/guide/$IFS) (IFS) to identify when an argument ends and the next begins. Bash provides various special variables that assist while scripting. Some of these variables are:

| **IFS** | **Description** |
| --- | --- |
| $# | This variable holds the number of arguments passed to the script. |
| $@ | This variable can be used to retrieve the list of command-line arguments. |
| $n | Each command-line argument can be selectively retrieved using its position. For example, the first argument is found at $1. |
| $$ | The process ID of the currently executing process. |
| $? | The exit status of the script. This variable is useful to determine a command's success. The value 0 represents successful execution, while 1 is a result of a failure. |

Of the ones shown above, we have 3 such special variables in our if-else condition.

| **IFS** | **Description** |
| --- | --- |
| $# | In this case, we need just one variable that needs to be assigned to the domain variable. This variable is used to specify the target we want to work with. If we provide just an FQDN as the argument, the $# variable will have a value of 1. |
| $0 | This special variable is assigned the name of the executed script, which is then shown in the "Usage:" example. |
| $1 | Separated by a space, the first argument is assigned to that special variable. |

**Variables**

We also see at the end of the if-else loop that we assign the value of the first argument to the variable called "domain". The assignment of variables takes place without the dollar sign ($). The dollar sign is only intended to allow this variable's corresponding value to be used in other code sections. When assigning variables, there must be no spaces between the names and values.

<SNIP>

else

domain=$1

fi

<SNIP>

In contrast to other programming languages, there is no direct differentiation and recognition between the types of variables in Bash like "strings," "integers," and "boolean." All contents of the variables are treated as string characters. Bash enables arithmetic functions depending on whether only numbers are assigned or not. It is important to note when declaring variables that they do not contain a space. Otherwise, the actual variable name will be interpreted as an internal function or a command.

**Declaring a Variable - Error**

1372000@htb[/htb]**$** variable = "this will result with an error."

command not found: variable

**Declaring a Variable - Without an Error**

1372000@htb[/htb]**$** variable="Declared without an error."

1372000@htb[/htb]**$** echo $variable

Declared without an error.

**Arrays**

There is also the possibility of assigning several values to a single variable in Bash. This can be beneficial if we want to scan multiple domains or IP addresses. These variables are called arrays that we can use to store and process an ordered sequence of specific type values. Arrays identify each stored entry with an index starting with 0. When we want to assign a value to an array component, we do so in the same way as with standard shell variables. All we do is specify the field index enclosed in square brackets. The declaration for arrays looks like this in Bash:

Comparison Operators

To compare specific values with each other, we need elements that are called [comparison operators](https://www.tldp.org/LDP/abs/html/comparison-ops.html). The comparison operators are used to determine how the defined values will be compared. For these operators, we differentiate between:

* string operators
* integer operators
* file operators
* boolean operators

String Operators

If we compare strings, then we know what we would like to have in the corresponding value.

| **Operator** | **Description** |
| --- | --- |
| == | is equal to |
| != | is not equal to |
| < | is less than in ASCII alphabetical order |
| > | is greater than in ASCII alphabetical order |
| -z | if the string is empty (null) |
| -n | if the string is not null |

String comparison operators "< / >" works only within the double square brackets [[ <condition> ]]. We can find the ASCII table on the Internet or by using the following command in the terminal. We take a look at an example later.

1372000@htb[/htb]**$** man ascii

A screenshot of a computer

Description automatically generated

ASCII stands for American Standard Code for Information Interchange and represents a 7-bit character encoding. Since each bit can take two values, there are 128 different bit patterns, which can also be interpreted as the decimal integers 0 - 127 or in hexadecimal values 00 - 7F. The first 32 ASCII character codes are reserved as so-called [control characters](https://en.wikipedia.org/wiki/Control_character).

| **Operator** | **Description** |
| --- | --- |
| -eq | is equal to |
| -ne | is not equal to |
| -lt | is less than |
| -le | is less than or equal to |
| -gt | is greater than |
| -ge | is greater than or equal to |

**File Operators**

The file operators are useful if we want to find out specific permissions or if they exist.

| **Operator** | **Description** |
| --- | --- |
| -e | if the file exist |
| -f | tests if it is a file |
| -d | tests if it is a directory |
| -L | tests if it is if a symbolic link |
| -N | checks if the file was modified after it was last read |
| -O | if the current user owns the file |
| -G | if the file’s group id matches the current user’s |
| -s | tests if the file has a size greater than 0 |
| -r | tests if the file has read permission |
| -w | tests if the file has write permission |
| -x | tests if the file has execute permission |

**Boolean and Logical Operators**

We get a boolean value "false" or "true" as a result with logical operators. Bash gives us the possibility to compare strings by using double square brackets [[ <condition> ]]. To get these boolean values, we can use the string operators. Whether the comparison matches or not, we get the boolean value "false" or "true".

**Logical Operators**

With logical operators, we can define several conditions within one. This means that all the conditions we define must match before the corresponding code can be executed.

| **Operator** | **Description** |
| --- | --- |
| ! | logical negotation NOT |
| && | logical AND |
| || | logical OR |

Arithmetic Operators

In Bash, we have seven different arithmetic operators we can work with. These are used to perform different mathematical operations or to modify certain integers.

| **Operator** | **Description** |
| --- | --- |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulus |
| variable++ | Increase the value of the variable by 1 |
| variable-- | Decrease the value of the variable by 1 |

The output of this script looks like this:

**Arithmetic.sh - Execution**

  1372000@htb[/htb]**$** ./Arithmetic.sh

Addition: 10 + 10 = 20

Subtraction: 10 - 10 = 0

Multiplication: 10 \* 10 = 100

Division: 10 / 10 = 1

Modulus: 10 % 4 = 2

Increase Variable: 2

Decrease Variable: 0

**VarLength.sh**

Code: bash

**#!/bin/bash**

htb="akadmin"

echo ${#htb}

If we look at our CIDR.sh script, we will see that we have used the increase and decrease operators several times. This ensures that the while loop, which we will discuss later, runs and pings the hosts while the variable "stat" has a value of 1. If the ping command ends with code 0 (successful), we get a message that the host is up and the "stat" variable, as well as the variables "hosts\_up" and "hosts\_total" get changed.

<SNIP>

echo -e "\nPinging host(s):"

for host in $cidr\_ips

do

stat=1

while [ $stat -eq 1 ]

do

ping -c 2 $host > /dev/null **2**>**&1**

if [ $? -eq 0 ]

then

echo "$host is up."

((stat--))

((hosts\_up++))

((hosts\_total++))

else

echo "$host is down."

((stat--))

((hosts\_total++))

fi

done

done

<SNIP

Input Control

We may get results from our sent requests and executed commands, which we have to decide manually on how to proceed. Another example would be that we have defined several functions in our script designed for different scenarios. We have to decide which of them should be executed after a manual check and based on the results. It is also quite possible that specific scans or activities may not be allowed to be performed. Therefore, we need to be familiar with how to get a running script to wait for our instructions. If we look at our CIDR.sh script again, we see that we have added such a call to decide further steps.

Output control

We have already learned about the output redirections of output in the Linux Fundamentals module. Nevertheless, the problem with the redirections is that we do not get any output from the respective command. It will be redirected to the appropriate file. If our scripts become more complicated later, they can take much more time than just a few seconds. To avoid sitting inactively and waiting for our script's results, we can use the [tee](https://man7.org/linux/man-pages/man1/tee.1.html) utility. It ensures that we see the results we get immediately and that they are stored in the corresponding files. In our CIDR.sh script, we have used this utility twice in different ways.

When using tee, we transfer the received output and use the pipe (|) to forward it to tee. The "-a / --append" parameter ensures that the specified file is not overwritten but supplemented with the new results. At the same time, it shows us the results and how they will be found in the file.

Input and Output

1372000@htb[/htb]**$** cat discovered\_hosts.txt CIDR.txt

165.22.119.202

NetRange: 165.22.0.0 - 165.22.255.255

CIDR: 165.22.0.0/16

Flow Control – loops

The control of the flow of our scripts is essential. We have already learned about the if-else conditions, which are also part of flow control. After all, we want our script to work quickly and efficiently, and for this, we can use other components to increase efficiency and allow error-free processing. Each control structure is either a branch or a loop. Logical expressions of boolean values usually control the execution of a control structure. These control structures include:

* Branches:
  + If-Else Conditions
  + Case Statements
* Loops:
  + For Loops
  + While Loops
  + Until Loops

For Loops

Let us start with the For loops. The For loop is executed on each pass for precisely one parameter, which the shell takes from a list, calculates from an increment, or takes from another data source. The for loop runs as long as it finds corresponding data. This type of loop can be structured and defined in different ways. For example, the for loops are often used when we need to work with many different values from an array. This can be used to scan different hosts or ports. We can also use it to execute specific commands for known ports and their services to speed up our enumeration process. The syntax for this can be as follows:

Syntax examples:-

for variable in 1 2 3 4

do

echo $variable

done

example 2

for variable in file1 file2 file3

do

echo $variable

done

example 3

for ip in "10.10.10.170 10.10.10.174 10.10.10.175"

do

ping -c 1 $ip

done

we can also write these commands in a single line. Such a command would look like this:

1372000@htb[/htb]**$** for ip in 10.10.10.170 10.10.10.174;do ping -c 1 $ip;done

PING 10.10.10.170 (10.10.10.170): 56 data bytes

64 bytes from 10.10.10.170: icmp\_seq=0 ttl=63 time=42.106 ms

--- 10.10.10.170 ping statistics ---

1 packets transmitted, 1 packets received, 0.0% packet loss

round-trip min/avg/max/stddev = 42.106/42.106/42.106/0.000 ms

PING 10.10.10.174 (10.10.10.174): 56 data bytes

64 bytes from 10.10.10.174: icmp\_seq=0 ttl=63 time=45.700 ms

--- 10.10.10.174 ping statistics ---

1 packets transmitted, 1 packets received, 0.0% packet loss

round-trip min/avg/max/stddev = 45.700/45.700/45.700/0.000 ms

While loops

The while loop is conceptually simple and follows the following principle:

* A statement is executed as long as a condition is fulfilled (true).

We can also combine loops and merge their execution with different values. It is important to note that the excessive combination of several loops in each other can make the code very unclear and lead to errors that can be hard to find and follow.

<SNIP>

stat=1

while [ $stat -eq 1 ]

do

ping -c 2 $host > /dev/null **2**>**&1**

if [ $? -eq 0 ]

then

echo "$host is up."

((stat--))

((hosts\_up++))

((hosts\_total++))

else

echo "$host is down."

((stat--))

((hosts\_total++))

fi

done

<SNIP>

The while loops also work with conditions like if-else. A while loop needs some sort of a counter to orientate itself when it has to stop executing the commands it contains. Otherwise, this leads to an endless loop. Such a counter can be a variable that we have declared with a specific value or a boolean value. While loops run while the boolean value is "True". Besides the counter, we can also use the command "break," which interrupts the loop when reaching this command like in the following example:

**WhileBreaker.sh**

**#!/bin/bash**

counter=0

while [ $counter -lt 10 ]

do

# Increase $counter by 1

((counter++))

echo "Counter: $counter"

if [ $counter == 2 ]

then

continue

elif [ $counter == 4 ]

then

break

fi

done

execution

1372000@htb[/htb]**$** ./WhileBreaker.sh

Counter: 1

Counter: 2

Counter: 3

Counter: 4

Until loops

There is also the until loop, which is relatively rare. Nevertheless, the until loop works precisely like the while loop, but with the difference:

* The code inside a until loop is executed as long as the particular condition is false.

The other way is to let the loop run until the desired value is reached. The "until" loops are very well suited for this. This type of loop works similarly to the "while" loop but, as already mentioned, with the difference that it runs until the boolean value is "False."

**Until.sh**

**#!/bin/bash**

counter=0

until [ $counter -eq 10 ]

do

# Increase $counter by 1

((counter++))

echo "Counter: $counter"

done

execution and results

1372000@htb[/htb]**$** ./Until.sh

Counter: 1

Counter: 2

Counter: 3

Counter: 4

Counter: 5

Counter: 6

Counter: 7

Counter: 8

Counter: 9

Counter: 10

Flow Control Branches

As we have already seen, the branches in flow control include if-else and the case statements. We have already discussed the if-else statements in detail and know how this works. Now we will take a closer look at the case statements.

**Case Statements**

Case statements are also known as switch-case statements in other languages, such as C/C++ and C#. The main difference between if-else and switch-case is that if-else constructs allow us to check any boolean expression, while switch-case always compares only the variable with the exact value. Therefore, the same conditions as for if-else, such as "greater-than," are not allowed for switch-case. The syntax for the switch-case statements looks like this:

**Syntax - Switch-Case**

case <expression> in

pattern\_1 ) statements ;;

pattern\_2 ) statements ;;

pattern\_3 ) statements ;;

esac

Functions

The bigger our scripts get, the more chaotic they become. If we use the same routines several times in the script, the script's size will increase accordingly. In such cases, functions are the solution that improves both the size and the clarity of the script many times. We combine several commands in a block between curly brackets ( { ... } ) and call them with a function name defined by us with functions. Once a function has been defined, it can be called and used again during the script.

Functions are an essential part of scripts and programs, as they are used to execute recurring commands for different values and phases of the script or program. Therefore, we do not have to repeat the whole section of code repeatedly but can create a single function that executes the specific commands. The definition of such functions makes the code easier to read and helps to keep the code as short as possible. It is important to note that functions must always be defined logically before the first call since a script is also processed from top to bottom. Therefore the definition of a function is always at the beginning of the script. There are two methods to define the functions:

**Method 1 - Functions**

function name {

<commands>

}

**Method 2 - Functions**

name() {

<commands>

}

**Function Execution**

<SNIP>

case $opt in

"1") network\_range ;;

"2") ping\_host ;;

"3") network\_range && ping\_host ;;

"\*") exit 0 ;;

esac

**Parameter Passing**

Such functions should be designed so that they can be used with a fixed structure of the values or at least only with a fixed format. Like we have already seen in our CIDR.sh script, we used the format of an IP address for the function "network\_range". The parameters are optional, and therefore we can call the function without parameters. In principle, the same applies to the passed parameters as to parameters passed to a shell script. These are $1 - $9 (${n}), or $variable as we have already seen. Each function has its own set of parameters. So they do not collide with those of other functions or the parameters of the shell script.

An important difference between bash scripts and other programming languages is that all defined variables are always processed globally unless otherwise declared by "[local](https://www.tldp.org/LDP/abs/html/localvar.html)." This means that the first time we have defined a variable in a function, we will call it in our main script (outside the function). Passing the parameters to the functions is done the same way as we passed the arguments to our script and looks like this:

**Return Values**

When we start a new process, each child process (for example, a function in the executed script) returns a return code to the parent process (bash shell through which we executed the script) at its termination, informing it of the status of the execution. This information is used to determine whether the process ran successfully or whether specific errors occurred. Based on this information, the parent process can decide on further program **flow.**

| **Return Code** | **Description** |
| --- | --- |
| **1** | **General errors** |
| **2** | **Misuse of shell builtins** |
| **126** | **Command invoked cannot execute** |
| **127** | **Command not found** |
| **128** | **Invalid argument to exit** |
| **128+n** | **Fatal error signal "n"** |
| **130** | **Script terminated by Control-C** |
| **255\\*** | **Exit status out of range** |

**Return.sh - Execution**

1372000@htb[/htb]**$** ./Return.sh

Number of arguments: 0

Function status code: 1

Number of arguments: 1

Function status code: 0

Content of the variable:

Number of arguments: 1

**Debugging**

Bash gives us an excellent opportunity to find, track, and fix errors in our code. The term debugging can have many different meanings. Nevertheless, Bash debugging is the process of removing errors (bugs) from our code. Debugging can be performed in many different ways. For example, we can use our code for debugging to check for typos, or we can use it for code analysis to track them and determine why specific errors occur.

This process is also used to find vulnerabilities in programs. For example, we can try to cause errors using different input types and track their handling in the CPU through the assembler, which may provide a way to manipulate the handling of these errors to insert our own code and force the system to execute it. This topic will be covered and discussed in detail in other modules. Bash allows us to debug our code by using the "-x" (xtrace) and "-v" options. Now let us see an example with our CIDR.sh script.

**CIDR.sh - Debugging**

1372000@htb[/htb]**$** bash -x CIDR.sh

+ '[' 0 -eq 0 ']'

+ echo -e 'You need to specify the target domain.\n'

You need to specify the target domain.

+ echo -e Usage:

Usage:

+ echo -e '\tCIDR.sh <domain>'

CIDR.sh <domain>

+ exit 1

Here Bash shows us precisely which function or command was executed with which values. This is indicated by the plus sign (+) at the beginning of the line. If we want to see all the code for a particular function, we can set the "-v" option that displays the output in more detail

**CIDR.sh - Verbose Debugging**

1372000@htb[/htb]**$** bash -x -v CIDR.sh

**#**!/bin/bash

**#** Check for given argument

if [ **$**# -eq 0 ]

then

echo -e "You need to specify the target domain.\n"

echo -e "Usage:"

echo -e "\t**$**0 <domain>"

exit 1

else

domain=**$**1

fi

+ '[' 0 -eq 0 ']'

+ echo -e 'You need to specify the target domain.\n'

You need to specify the target domain.

+ echo -e Usage:

Usage:

+ echo -e '\tCIDR.sh <domain>'

CIDR.sh <domain>

+ exit 1