v4

Penetration Testing Student

Command Line Scripting

Section 02 | Module 04

Table of Contents

Module 04 | Command Line Scripting

Learning Objectives

Welcome to the Command line scripting module! In this module, we will cover some important concepts about Linux and Windows command line scripting.

By the end of this module, you should have a better understanding of:

- What a bash shell is, as well as its features
- How to create simple linux scripts
- What a Windows command line is and its features
- How to automate simple Windows task

4.1

Bash shell

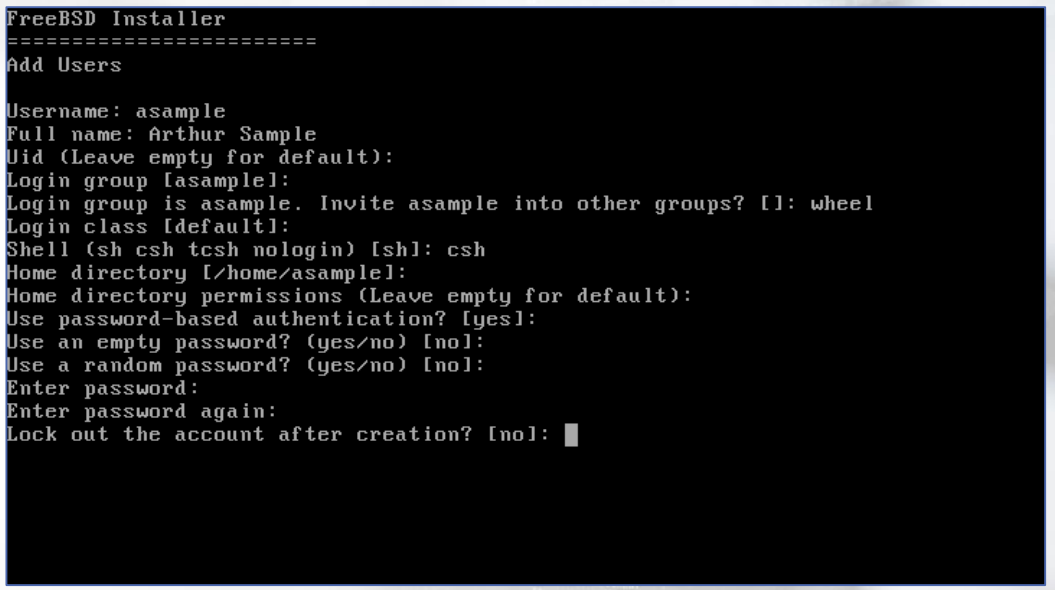
4.1 Bash shell

On Linux systems, the main non-graphical tool to interact with the operating system is **Shell**. You might have seen it being referred to as a **console, terminal or bash**.

If you were to install a legacy unix system like FreeBSD, which does not have any graphical interface like Kali Linux, the only way to interact with the undelying OS is through **shell**.

4.1 Bash shell

A FreeBSD shell looks similar to the following. As you can see, there is no GUI at all.



https://www.freebsd.org/doc/handbook/bsdinstall/bsdinstall-adduser2.png

4.1 Bash shell

Linux shell is a command interpreter. There are various types of shells, and Bash is the most popular of them.

While each shell behavior differs, their main purpose is the same – to provide a command line interface in order to interact with the operating system. Other notable shells are:

- ksh
- zsh
- dash

4.1 Bash shell

In this course, we will focus on **Bash** shell; however, you should keep in mind that this knowledge can be applied to **most type of shells.**

Bash Environment

4.2 Bash Environment

Whenever you do an „Open Terminal” on your Kali Linux machine, a new bash shell is started.

Before you can use it, the operating system initializes your **bash environment**.

4.2 Bash Environment

Upon the start of the shell, the operating system checks for the existence of several files like **~/.bashrc, ~/.bash_login** or **~/.bash_profile**. These files may contain some instruction to help set up the environment properly. The same thing also happens when closing it, with the **~/.bash_logout** file.

4.2 Bash Environment

As a penetration tester, keep in mind the aforementioned files in case you need to backdoor a linux user account some day.

4.2.1 Environment Variables

An important part of the environment are **environment variables**.

You can think of them as **normal programming variables**, that come predefined upon starting your program interpreter – **the bash shell.**

4.2.1 Environment Variables

Environment variables contain information from the operating system that is required to properly use system functionalities.

You can view environment variables by typing **„env”** in the bash window.

4.2.1 Environment Variables

```
root@0xluk3:~# env
SHELL=/bin/bash
SESSION_MANAGER=local/0xluk3:@/tmp/.ICE-unix/1257,unix/0xluk3:/tmp/.ICE-unix/1257
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
SSH_AUTH_SOCK=/run/user/0/keyring/ssh
DESKTOP_SESSION=gnome
SSH_AGENT_PID=1315
GTK_MODULES=gail:atk-bridge
XDG_SEAT=seat0
PWD=/root
XDG_SESSION_DESKTOP=gnome
LOGNAME=root
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/0/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/0/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
GDM_LANG=en_US.UTF-8
HOME=/root
USERNAME=root
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37
;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=0
1;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31
```

4.2.1 Environment Variables

One of the most important environment variables is PATH. You can see its content by using the „echo” command.

In bash, if you want to refer to a variable, you should put a dollar sign before it. For example, to see the „PATH” variable’s content, you will need to type:

echo $PATH

```
root@0xluk3:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

4.2.2 PATH Variable

PATH has a format of

[location]:[location]:[location]:...:[location]

and it contains information on where bash should look for executable files when you enter a command into the bash window.

4.2.2 PATH Variable

When you type garbage into the bash window it will output:

```
root@0xluk3:~# qweqewqwe
bash: qweqewqwe: command not found
```

That means, bash checked all the locations contained in the **PATH** variable for the existence of the executable „qweqewqwe”. As it was not there, the **command was not found**.

4.2.2 PATH Variable

The PATH variable is one of the execution helpers.

Specifically, if you put a program within locations held in PATH, it will be executed when its name is typed in the bash window.

4.2.2 PATH Variable

Let’s analyze this case thoroughly. Once again, let’s check the content of PATH:

```
root@0xluk3:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

We see that the „/bin” location is held there. Therefore, anything put in „bin” should be executed when its name is typed into the bash shell. Let’s check this!

4.2.2 PATH Variable

```
root@0xluk3:~# qweqwe
bash: qweqwe: command not found
root@0xluk3:~# cp /bin/ping /tmp/qweqwe
root@0xluk3:~# qweqwe
bash: qweqwe: command not found
root@0xluk3:~# cp /tmp/qweqwe /bin/qweqwe
root@0xluk3:~# qweqwe
Usage: ping [-aAbBdDfhLnOqrRUvV64] [-c count] [-i interval] [-I interface]
            [-m mark] [-M pmtudisc_option] [-l preload] [-p pattern] [-Q tos]
            [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp_option]
            [-w deadline] [-W timeout] [hop1 ...] destination
Usage: ping -6 [-aAbBdDfhLnOqrRUvV] [-c count] [-i interval] [-I interface]
             [-l preload] [-m mark] [-M pmtudisc_option]
             [-N nodeinfo_option] [-p pattern] [-Q tclass] [-s packetsize]
             [-S sndbuf] [-t ttl] [-T timestamp_option] [-w deadline]
             [-W timeout] destination
```

4.2.2 PATH Variable

What happened there?

First, we again try to execute „qweqwe”. Obviously, such a file does not exist in the filesystem, so the command is not found.

```
root@0xluk3:~# qweqwe
bash: qweqwe: command not found
```

4.2.2 PATH Variable

Next, we copy the existing program **/bin/ping** (which should be familiar to you) to the location **/tmp/qweqwe**.

```
root@0xluk3:~# cp /bin/ping /tmp/qweqwe
root@0xluk3:~# qweqwe
bash: qweqwe: command not found
```

The program with such a name now exists, but it is not in any location held in PATH; therefore, the command is not found again.

4.2.2 PATH Variable

Now, **/tmp/qweqwe** is copied to **/bin/qweqwe**. **/bin/** is a location within **PATH**, so now the command is found and executed.

```
root@0xluk3:~# cp /tmp/qweqwe /bin/qweqwe
root@0xluk3:~# qweqwe
Usage: ping [-aAbBdDfhLnOqrRUvV64] [-c count] [-i interval] [-I interface]
            [-m mark] [-M pmtudisc_option] [-l preload] [-p pattern] [-Q tos]
            [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp_option]
            [-w deadline] [-W timeout] [hop1 ...] destination
Usage: ping -6 [-aAbBdDfhLnOqrRUvV] [-c count] [-i interval] [-I interface]
             [-l preload] [-m mark] [-M pmtudisc_option]
             [-N nodeinfo_option] [-p pattern] [-Q tclass] [-s packetsize]
             [-S sndbuf] [-t ttl] [-T timestamp_option] [-w deadline]
             [-W timeout] destination
```

4.3

Bash Commands and Programs

4.3 Bash Commands and Programs

Bash itself has some built-in commands that provide basic functionality.

However, it strongly relies on extension programs that are, by default, kept in PATH locations like **/bin** or **/usr/bin**.

4.3 Bash Commands and Programs

Examples of bash built-in commands are **fg, echo, set, while**.

You can find more here.

4.3 Bash Commands and Programs

Most commands that are used in everyday tasks are external mini-programs kept in PATH locations like /bin. Examples are **ls, ping, passwd, chmod, more**

If you want to know the command's real location, you can check it using **which**.

```
root@0xluk3:~# which chmod
/usr/bin/chmod
```

4.3.1 Man Pages

There is a plethora of commands in the Linux OS.

Oftentimes we want to remind ourselves what specific commands do. In such cases, the **man** (short for manual) command will prove useful.

4.3.1 Man Pages

For example, below is the result of issuing the „**man ping**” command:

```
PING(8)                                    iputils                                    PING(8)

NAME
       ping - send ICMP ECHO_REQUEST to network hosts

SYNOPSIS
       ping [-aAbBdDfhLnOqrRUvV46] [-c count] [-F flowlabel] [-i interval] [-I interface] [-l preload]
            [-m mark] [-M pmtudisc_option] [-N nodeinfo_option] [-w deadline] [-W timeout] [-p pattern]
            [-Q tos] [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp option] [hop...] destination

DESCRIPTION
       ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a
       host or gateway. ECHO_REQUEST datagrams (pings) have an IP and ICMP header, followed by a struct
       timeval and then an arbitrary number ofpadbytes used to fill out the packet.

       ping works with both IPv4 and IPv6. Using only one of them explicitly can be enforced by specifying -4
       or -6.

       ping can also send IPv6 Node Information Queries (RFC4620). Intermediate hops may not be allowed,
       because IPv6 source routing was deprecated (RFC5095).

OPTIONS
       -4
           Use IPv4 only.

       -6
           Use IPv6 only.
```

4.3.1 Man Pages

You can also browse online for Linux commands. Here is an example of a free resource that includes Linux command manuals:

https://linux.die.net/man/

4.3.2 Relative Paths

When you want to run a program that is not in PATH, you can specify its **absolute path**; for example, type **/bin/ping** to run it in case it is not in PATH.

4.3.2 Relative Paths

You may also want to run a program using a **relative path.** Relative path means what its location from the current directory is. You may need to run something from the current directory itself by putting a dot in front of it, for instance:

```
root@0xluk3:~# cd /bin
root@0xluk3:/bin# ./ping
Usage: ping [-aAbBdDfhLnOqrRUvV64] [-c count] [-i interval] [-I interface]
            [-m mark] [-M pmtudisc_option] [-l preload] [-p pattern] [-Q tos]
            [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp_option]
            [-w deadline] [-W timeout] [hop1 ...] destination
Usage: ping -6 [-aAbBdDfhLnOqrRUvV] [-c count] [-i interval] [-I interface]
            [-l preload] [-m mark] [-M pmtudisc_option]
            [-N nodeinfo_option] [-p pattern] [-Q tclass] [-s packetsize]
            [-S sndbuf] [-t ttl] [-T timestamp_option] [-w deadline]
            [-W timeout] destination
```

4.3.2 Relative Paths

Or, you can make the system traverse multiple directories to find the desired executable:

```
root@0xluk3:~/Desktop# ../../bin/ping
Usage: ping [-aAbBdDfhLnOqrRUvV64] [-c count] [-i interval] [-I interface]
            [-m mark] [-M pmtudisc_option] [-l preload] [-p pattern] [-Q tos]
            [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp_option]
            [-w deadline] [-W timeout] [hop1 ...] destination
Usage: ping -6 [-aAbBdDfhLnOqrRUvV] [-c count] [-i interval] [-I interface]
             [-l preload] [-m mark] [-M pmtudisc_option]
             [-N nodeinfo_option] [-p pattern] [-Q tclass] [-s packetsize]
             [-S sndbuf] [-t ttl] [-T timestamp_option] [-w deadline]
             [-W timeout] destination
```

4.4

Bash Output Redirection and Special Characters

4.4.1 Bash Special Characters

When typing into the bash shell, you should pay attention to characters that have special meaning. Let’s discuss a few of them.

4.4.1 Bash Special Characters

~ is the current user's home directory

```
root@0xluk3:~/Desktop# echo ~
/root
```

* is a wildcard, that can be used for choosing only certain types of files

```
root@0xluk3:~/Desktop# ls /etc/*.conf
/etc/adduser.conf                /etc/fuse.conf          /etc/nftables.conf      /etc/rsyslog.conf
```

4.4.1 Bash Special Characters

Data between `` or **$()** will be evaluated before the whole statement and will become part of this statement.

```
root@0xluk3:~/Desktop# file `ls /etc/*.conf`
/etc/adduser.conf:                ASCII text
/etc/apg.conf:                    ASCII text
/etc/appstream.conf:              ASCII text
/etc/ca-certificates.conf:        UTF-8 Unicode text
/etc/chkrootkit.conf:             ASCII text
/etc/debconf.conf:                ASCII text
/etc/deluser.conf:                ASCII text
/etc/dleyna-server-service.conf: ASCII text
/etc/dns2tcpd.conf:               ASCII text
/etc/foremost.conf:               ASCII text
```

4.4.2 Bash Output Redirectors

When executing a command in a bash shell, its result is called **output**. The default behavior of every command is to print the output in the bash shell. However, the user can alter this behavior and redirect the output to somewhere else.

4.4.2 Bash Output Redirectors

Basic output redirections can have two directions: to **file** and to **another command**.

4.4.2 Bash Output Redirectors

In order to redirect the output to a file:

- Use the **command > file.txt** format to create create a file containing the command's output or overwrite an existing file with the command's output
- Use the **command >> file.txt** format to create a file containing the command's output or append the command's output to an existing file

4.4.2 Bash Output Redirectors

Example:

```
root@0xluk3:~# echo "test" > file.txt
root@0xluk3:~# cat file.txt
test
root@0xluk3:~# echo "test2" >> file.txt
root@0xluk3:~# cat file.txt
test
test2
root@0xluk3:~#
```

4.4.2 Bash Output Redirectors

Another type of redirection, is redirection to another command. For this purpose, a pipe „|” is used.

4.4.2 Bash Output Redirectors

Example:

```
root@0xluk3:~# ls /etc/*.conf | sort
/etc/adduser.conf
/etc/apg.conf
/etc/appstream.conf
/etc/ca-certificates.conf
/etc/chkrootkit.conf
/etc/debconf.conf
```

The output of the **ls** command is sent to the **sort** command. The last command in the chain presents the output – in this case, sorted files that match the *.conf pattern and reside in the /etc/ directory.

4.4.3 Bash Commands Chaining

Chaining commands is a quite powerful Bash feature. Using the previously mentioned special characters, it is possible to chain multiple commands, creating an one-line script that can automate many tedious tasks. Such scripts are often called **„oneliners”.**

4.4.3 Bash Commands Chaining

What does this oneliner do?

```
file `ls /etc/*.conf | sort` > test.txt && cat test.txt | wc -l
```

```
root@0xluk3:~# file `ls /etc/*.conf | sort` > test.txt && cat test.txt | wc -l
44
```

4.4.3 Bash Commands Chaining

First, let's split it into three blocks divided by redirection operators.

```
file `ls /etc/*.conf | sort` > test.txt && cat test.txt | wc -l
```

4.4.3 Bash Commands Chaining

As we already know, the `` operator will be evaluated first. Bash will list all files with the name *.conf in the /etc/ directory and sort them; however, this will not be visible to the user.

```
file `ls /etc/*.conf | sort` > test.txt && cat test.txt | wc -l
```

For each file listed by the **ls** command, a **file** command (file information) will be executed and this output will be redirected to file test.txt

4.4.3 Bash Commands Chaining

Next, there is the „&&” operator, which means that „if the previous command succeeds, then proceed to the next command”.

```
file `ls /etc/*.conf | sort` > test.txt && cat test.txt | wc -l
```

Since, in this case, we successfully redirected the output to a file, the „cat test.txt” command will be executed. However, its output is once again redirected.

4.4.3 Bash Commands Chaining

The „wc –l” command prints the number of lines of whatever text is “fed” to it. In this case, test.txt is the input.

```
file `ls /etc/*.conf | sort` > test.txt && cat test.txt | wc -l
```

Then, only the number of lines (44 in this case) is printed.

4.5

Bash Conditional Statements and Loops

4.5.1 Bash Script Files

Apart from writing bash statements inside shell windows, we can also save them in script files.

Linux should be able to execute any file containing bash instructions; however, following the common standards, a bash script file should have a **.sh** extension.

4.5.1 Bash Script Files

Bash script files should also contain information at the very beginning that indicates **which shell they should be interpreted**.

We want to use **Bash**, so the first line of any Bash script should be the following:

#!/bin/bash

4.5.1 Bash Script Files

When creating a Bash file, upon saving, we should also make sure it is an **executable**.

Since Linux permissions are a vast and somewhat advanced topic, for the sake of this scripting task you should simply make sure you use the **chmod +x <scriptname>** command on any newly created script before running it. Then, it can be run by issuing the **./<scriptname>** command.

4.5.1 Bash Script Files

```
root@0xluk3:~# echo '#!/bin/bash' > script.sh
root@0xluk3:~# echo 'ls /tmp | wc -l' >> script.sh
root@0xluk3:~# chmod +x script.sh
root@0xluk3:~# ./script.sh
13
```

4.5.2 Bash Conditional Statements

Like other scripting languages, Bash offers conditional statements functionality.

Basic syntax is:

if <condition>; then

<commands>

fi

4.5.2 Bash Conditional Statements

Additionally, some similar constructs like **else** of **else if (elif)** are also applicable:

```
if [ x ]; then
        docommand
elif [ y ]; then
        doothercommand
else
        dosomethingelse
fi
```

4.5.2 Bash Conditional Statements

Writing a conditional statement in bash is a bit tricky since it requires us to use different patterns for different types of comparisons.

4.5.2 Bash Conditional Statements

In this course's context, we will focus on numbers comparison, but you can see other possible comparisons here.

4.5.2 Bash Conditional Statements

Numbers can be compared using the following operators:

-eq # equal

-ne # not equal

-lt # less than

-le # less than or equal

-gt # greater than

-ge # greater than or equal

4.5.2 Bash Conditional Statements

EXAMPLE - Pay attention to the spaces within the square brackets, as they are important!

```
#!/bin/bash

x=231
y=321

if [ "$a" -eq "$b" ];then
  echo "They're equal";
fi
```

4.5.3 Bash Loops

In this course's context, we will use two basic loops in Bash, **for** and **while**.

4.5.3.1 Bash For Loop

A For loop is repsonsible for iterating over items in the **loop condition**. For example this is how you print every item in current directory:

```
#!/bin/bash
for i in $( ls ); do
   echo item: $i
done
```

4.5.3.1 Bash For Loop

If you need to iterate over numbers, you can use the bash „seq” command:

```
#!/bin/bash
for i in `seq 1 10`;
do
  echo $i
done
```

4.5.3.2 Bash While Loop

The „While” loop can be very useful when iterating over items in a file. Below we see how the basic syntax looks like:

```
while [condition]; do <command1>;<command2>; done
```

4.5.3.2 Bash While Loop

To read and print items from a file, the following oneliner is a viable option:

```
while read line; do echo $line; done < file.txt
```

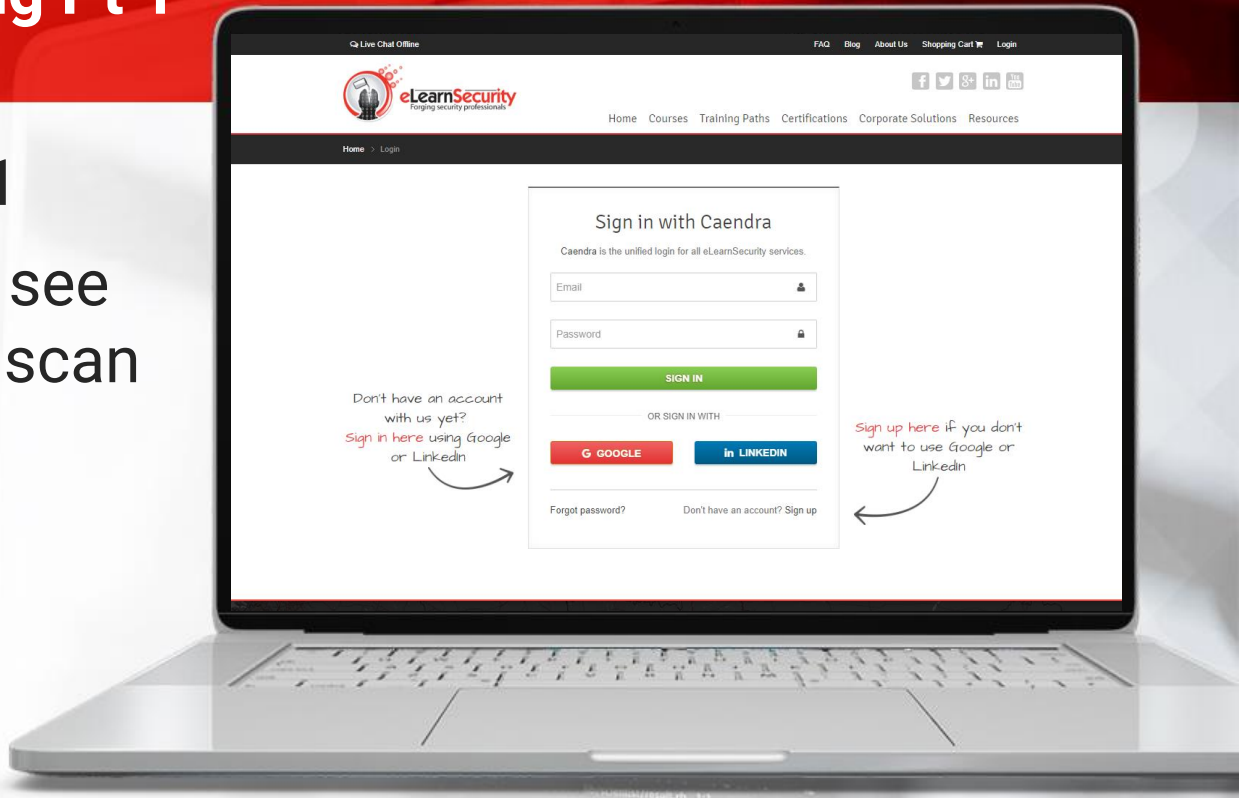
4.5.3.3 Bash Scripting Summary

In following videos, you will see exemplary use of Bash scripting to automate some everyday tasks performed by penetration testers.

VIDEO: Bash Scripting Pt 1

Bash Scripting Part 1

In this video, you will see how to format nmap scan results using Bash.

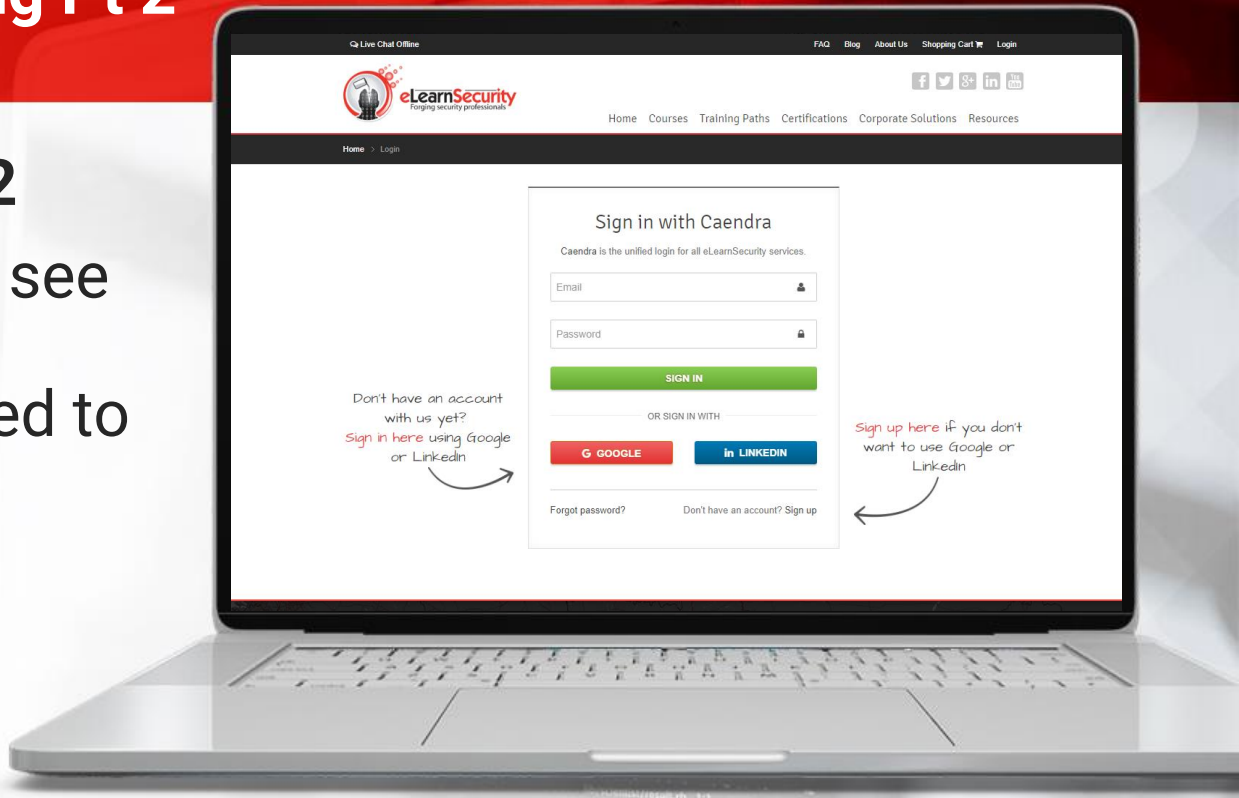


Videos are only available in Full or Elite Editions of the course. To upgrade, click HERE. To access, go to the course in your members area and click the resources drop-down in the appropriate module line.

VIDEO: Bash Scripting Pt 2

Bash Scripting Part 2

In this video, you will see how to create a bash script that can be used to probe multiple web applications for their HTTP status.



**Videos are only available in Full or Elite Editions of the course. To upgrade, click HERE. To access, go to the course in your members area and click the resources drop-down in the appropriate module line.*

4.6

Windows Command Line

4.6 Windows Command Line

Windows command line (cmd.exe) is the Microsoft equivalent of the Linux Bash shell.

Its usual location is **C:\Windows\system32\cmd.exe**.

4.6 Windows Command Line

Contrary to Bash, the windows command line relies mainly on built-in functionalities – commands. Some of them are **dir, cls, move or del**.

4.6 Windows Command Line

Due to Microsoft moving most command line utilities to PowerShell, we will briefly cover cmd.exe capabilities. You can find more about Windows command line commands here.

PowerShell for penetration testers is widely covered in the PTP course.

4.7

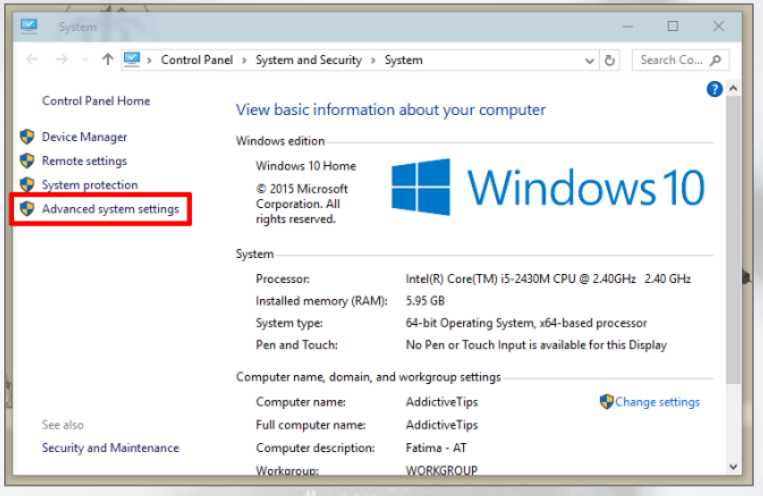
Windows Environment

4.7 Windows Environment

Similarly to Bash, Windows also has an environment; however, it is usually managed via Windows GUI.

4.7 Windows Environment

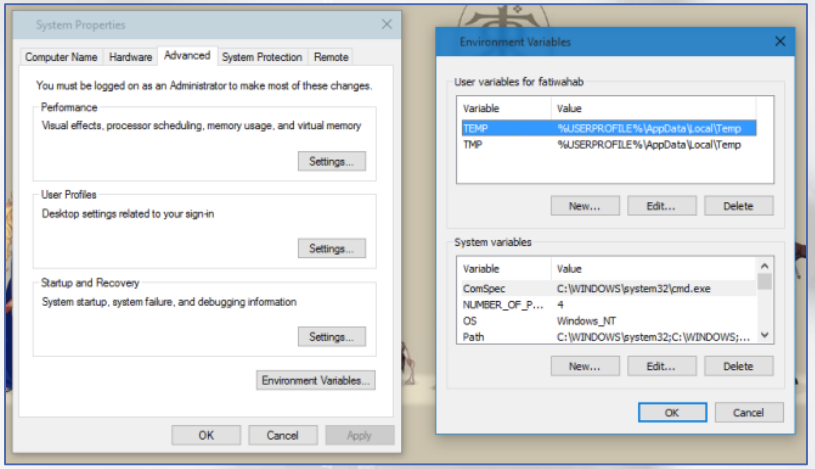
To manage Windows environment variables on Windows 10, you should proceed to Control Panel > System and Security > System > Advanced System Settings.



https://cloud.addictivetips.com/wp-content/uploads/2015/11/win10-CP.png

4.7 Windows Environment

By clicking **Environment variables**, you are able to manage their settings. Whenever you start **cmd.exe**, its environmental settings are pulled from this place. Note that you can manage variables globally (**System variables** - for all users) or for a **current user**.



https://cloud.addictivetips.com/wp-content/uploads/2015/11/win10-sys-env-edit.png

4.7.1 Windows PATH Variable

One of the most interesting variables is the Windows PATH one. It works exactly as the Linux's PATH variable does, providing locations on where to search for **command line programs**.

```
C:\Users>path
PATH=C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\Program Files (x86)\Intel\Intel(R) Management Engine Co
mponents\iCLS\;C:\Program Files\Intel\Intel(R) Management Engine Components\iCLS\;C:\Windows\system32;C:\Windows;C:\Wind
```

In Windows, the executable directories are separated through the „;" symbol.

4.7.1 Windows PATH Variable

When you try to execute something in a **cmd.exe** window, the Windows system checks if it matches any of the **built-in commands**. If what you are trying to execute is not a built-in command, Windows will search in all locations specified in the **PATH** variable to find it.

4.7.2 Absolute and Relative Paths

Like in Bash, you can invoke any other program by providing its location to the command line, relatively or absolutely. You just need to remember that directory slashes are pointing to a different direction than the ones Bash uses. In bash, we use a regular **slash „/”**, while in windows a **backslash „\”**.

```
c:\Users>..\Windows\notepad.exe
```

4.8

Windows Commands and Programs

4.8 Windows Commands and Programs

As mentioned previously, the Windows command line supports more built-in commands than the Linux one. That being said, some common Windows utilities are standalone executables located in **C:\Windows\system32** (which is by default in **PATH**).

One example is the ping.exe utility.

4.8 Windows Commands and Programs

If you would like to make your newly installed software executable from the command line, you should place it within any of your **PATH** locations, or change the **PATH** variable to contain its location. Note that only a few Windows programs support a command line interface, thus executing them from there might result in launching a **GUI** of the software.

You can try, for example, typing „**notepad.exe**” into the cmd.exe window.

4.9

Windows Output Redirection and Special Characters

4.9.1 Windows Variables

Windows' command line is a less flexible scripting environment than Bash.

If you wish to create advanced scripts in Windows, you should instead use PowerShell, which is not in the scope of this course but is covered in our PTP course.

4.9.1 Windows Variables

In order to access Windows' environment variables, you can refer to them by using **%variablename%.** As you may recall, these variables are those set up in „**Advanced System Settings – Environment variables**". To print them in the command line window, you need to use the **echo** command.

Try the below yourself in your own command line:

- **echo %PATH%**
- **echo %username%**

4.9.1 Windows Variables

If you would like to view another variable, you can do it using the „**SET**” command.

```
C:\Users>set
ALLUSERSPROFILE=C:\ProgramData
```

This command may generate lots of output and also provide you with various system information.

4.9.1 Windows Variables

You can also create your own **variables** or temporarily modify existing ones. Any modifications will not be permanent and will only exist in the **current cmd.exe window**. **If you have another command line window opened, changing variables in one of them will not affect the other one**.

4.9.2 Windows Output Redirection

Output redirection also works in Windows. In order to redirect the output of a command to a file, you can:

- Use the **echo aaa > file.txt** format to create a file with the command's output or overwrite an existing file with the command's output
- Use the **echo bbb >> file.txt** to create a file with the command's output or append the command's output to the an existing file.

You can then view file.txt's content using the „**type**" command.

4.9.2 Windows Output Redirection

EXAMPLE

```
c:\Users>echo aaa > file.txt

c:\Users>type file.txt
aaa

c:\Users>echo bbb >> file.txt

c:\Users>type file.txt
aaa
bbb

c:\Users>
```

4.9.3 Windows Commands Chaining

If you would like to execute subsequent commands in the Windows command line, you can use the following symbols to achieve it:

- **command1 & command2** – execute both regardless of the result
- **command1 && command2** – execute the first command, and if it succeeds, execute the second one

4.9.3 Windows Commands Chaining

If you would like to execute subsequent commands in the Windows command line, you can use the following symbols to achieve it:

- **command1 | command2** – send output from the first command to the second command
- **command1 || command2** – execute the first command, and if it **fails**, execute the second one

4.10

Windows Conditional Statements and Loops

4.10.1 .bat Files

In order to create larger command line scripts, you can save them as .bat files, with one instruction per line.

Windows automatically recognizes this format and allows users to execute such files. You can edit them using a simple text editor like notepad.

4.10.2 Windows Conditional Statements

Like Bash, command line also has multiple possibilities on comparison operators. However, in this course's context, we will cover just simple value comparisons. If you want to learn more about command line conditional statements, please refer to here and here.

https://ss64.com/nt/if.html
https://ss64.com/nt/else.html

4.10.2 Windows Conditional Statements

For comparing values, like if the variable contains a certain word, you can use the following instructions:

```
c:\Users>SET x=qwe

c:\Users>if %x%==qwe (echo true)
true

c:\Users>if %x%==xyz (echo true)

c:\Users>if %x%==xyz (echo true) else (echo "does not contain xyz")
"does not contain xyz"
```

4.10.3 Windows Loops

Windows command line offers basic **FOR loop** with various functionalities. In this course's context, we will cover iterating over files and file contents using **FOR** and **FOR /F**.

There are few more **FOR loop** switches available, if you would like to explore them, please click here.

4.10.3 Windows Loops

For example, if you would like to list files in a directory using the for loop, you can use the following command:

for %i in (*.*) do @echo FILE: %i

„@” before the command means to hide the **command prompt** (like C:\Users>) and just display the **output.**

4.10.3 Windows Loops

Result:

```
c:\Users>for %i in (*.*) do @echo FILE: %i
FILE: 1.txt
FILE: 2.txt

c:\Users>
```

4.11

References

References

This concludes our Command line tutorial. If you want to dig deeper into this topic, here are some references that you can use:

Windows command line reference

https://ss64.com/nt/

Bash programming reference

http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

Unbuntu – Bash Built-in Command Examples

http://manpages.ubuntu.com/manpages/bionic/man7/bash-builtins.7.html

References

Brainasoft Blog: List of all Internal Commands in Command Prompt

https://blog.brainasoft.com/all-internal-commands-of-cmd/

If – Conditionally Perform Command – Windows CMD

https://ss64.com/nt/if.html

Else – Windows CMD

https://ss64.com/nt/else.html

For – Looping Commands – Windows CMS

https://ss64.com/nt/for.html

Videos

Bash Scripting Part 1

In this video, you will see how to format nmap scan results using bash.

Bash Scripting Part 2

In this video, you will see how to create bash script that can be used to probe multiple web application for their HTTP status.

Videos are only available in Full or Elite Editions of the course. To upgrade, click HERE. To access, go to the course in your members area and click the resources drop-down in the appropriate module line.