

Penetration Testing Student

Command Line Scripting

Section 02 | Module 04

© Caendra Inc. 2019
All Rights Reserved

Table of Contents

Module 04 | Command Line Scripting

4.1 Bash shell

4.2 Bash environment

4.3 Bash commands and programs

4.4 Bash output redirectors and special characters

4.5 Bash conditional statements and loops

4.6 Windows Command Line

4.7 Windows environment

4.8 Windows commands and programs

4.9 Windows output redirectors and special characters

4.10 Windows conditional statements and loops



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



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 underlying 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.

```
FreeBSD Installer
=====
Add Users

Username: asample
Full name: Arthur Sample
Uid (Leave empty for default):
Login group [asample]:
Login group is asample. Invite asample into other groups? [1]: wheel
Login class [default]:
Shell (sh csh tcsh nologin) [sh]: csh
Home directory [/home/asample]:
Home directory permissions (Leave empty for default):
Use password-based authentication? [yes]:
Use an empty password? (yes/no) [no]:
Use a random password? (yes/no) [no]:
Enter password:
Enter password again:
Lock out the account after creation? [no]:
```

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**.

```
25 # @param result - the experiment result as a dict
26 # @param observations - an array of Observations, in experiment
27 # @param control - the control observation
28
29 def initialize(experiment, observations = [], control = null)
30   @experiment = experiment
31   @observations = observations
32   @control = control
33   @candidates = observations - [control]
34   evaluate_candidates
35
36   freeze
37
38   context
39   experiment.context
40   nil
41
42   # Initialize the name of the experiment
43   def experiment_name
44     experiment.name
45   end
46
47   # @param result - the result of a match between an experiment
48   def matched?
49     # ...
50   end
51
52   @experiment/result.rb 1.1
```



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**.

```
20 def initialize_experiment_data():
21     # Create a new experiment
22     # Create a new experiment
23     # Create a new experiment
24     # Create a new experiment
25     # Create a new experiment
26     # Create a new experiment
27     # Create a new experiment
28     # Create a new experiment
29     # Create a new experiment
30     # Create a new experiment
31     # Create a new experiment
32     # Create a new experiment
33     # Create a new experiment
34     # Create a new experiment
35     # Create a new experiment
36     # Create a new experiment
37     # Create a new experiment
38     # Create a new experiment
39     # Create a new experiment
40     # Create a new experiment
41     # Create a new experiment
42     # Create a new experiment
43     # Create a new experiment
44     # Create a new experiment
45     # Create a new experiment
46     # Create a new experiment
47     # Create a new experiment
48     # Create a new experiment
49     # Create a new experiment
50     # Create a new experiment
51     # Create a new experiment
52     # Create a new experiment
53     # Create a new experiment
54     # Create a new experiment
55     # Create a new experiment
56     # Create a new experiment
57     # Create a new experiment
58     # Create a new experiment
59     # Create a new experiment
60     # Create a new experiment
61     # Create a new experiment
62     # Create a new experiment
63     # Create a new experiment
64     # Create a new experiment
65     # Create a new experiment
66     # Create a new experiment
67     # Create a new experiment
68     # Create a new experiment
69     # Create a new experiment
70     # Create a new experiment
71     # Create a new experiment
72     # Create a new experiment
73     # Create a new experiment
74     # Create a new experiment
75     # Create a new experiment
76     # Create a new experiment
77     # Create a new experiment
78     # Create a new experiment
79     # Create a new experiment
80     # Create a new experiment
81     # Create a new experiment
82     # Create a new experiment
83     # Create a new experiment
84     # Create a new experiment
85     # Create a new experiment
86     # Create a new experiment
87     # Create a new experiment
88     # Create a new experiment
89     # Create a new experiment
90     # Create a new experiment
91     # Create a new experiment
92     # Create a new experiment
93     # Create a new experiment
94     # Create a new experiment
95     # Create a new experiment
96     # Create a new experiment
97     # Create a new experiment
98     # Create a new experiment
99     # Create a new experiment
100    # Create a new experiment
```



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.

```
38 def initialize_experiment(experiment, observations = [], control = null)
39   # Initialize the experiment's context
40   # Control: the control observation
41   # Candidates: the candidates for the control observation
42   # Observations: the observations
43   # Candidates: the candidates for the control observation
44   # Candidates: the candidates for the control observation
45   # Candidates: the candidates for the control observation
46   # Candidates: the candidates for the control observation
47   # Candidates: the candidates for the control observation
48   # Candidates: the candidates for the control observation
49   # Candidates: the candidates for the control observation
50   # Candidates: the candidates for the control observation
51   # Candidates: the candidates for the control observation
52   # Candidates: the candidates for the control observation
53   # Candidates: the candidates for the control observation
54   # Candidates: the candidates for the control observation
55   # Candidates: the candidates for the control observation
56   # Candidates: the candidates for the control observation
57   # Candidates: the candidates for the control observation
58   # Candidates: the candidates for the control observation
59   # Candidates: the candidates for the control observation
60   # Candidates: the candidates for the control observation
61   # Candidates: the candidates for the control observation
62   # Candidates: the candidates for the control observation
63   # Candidates: the candidates for the control observation
64   # Candidates: the candidates for the control observation
65   # Candidates: the candidates for the control observation
66   # Candidates: the candidates for the control observation
67   # Candidates: the candidates for the control observation
68   # Candidates: the candidates for the control observation
69   # Candidates: the candidates for the control observation
70   # Candidates: the candidates for the control observation
71   # Candidates: the candidates for the control observation
72   # Candidates: the candidates for the control observation
73   # Candidates: the candidates for the control observation
74   # Candidates: the candidates for the control observation
75   # Candidates: the candidates for the control observation
76   # Candidates: the candidates for the control observation
77   # Candidates: the candidates for the control observation
78   # Candidates: the candidates for the control observation
79   # Candidates: the candidates for the control observation
80   # Candidates: the candidates for the control observation
81   # Candidates: the candidates for the control observation
82   # Candidates: the candidates for the control observation
83   # Candidates: the candidates for the control observation
84   # Candidates: the candidates for the control observation
85   # Candidates: the candidates for the control observation
86   # Candidates: the candidates for the control observation
87   # Candidates: the candidates for the control observation
88   # Candidates: the candidates for the control observation
89   # Candidates: the candidates for the control observation
90   # Candidates: the candidates for the control observation
91   # Candidates: the candidates for the control observation
92   # Candidates: the candidates for the control observation
93   # Candidates: the candidates for the control observation
94   # Candidates: the candidates for the control observation
95   # Candidates: the candidates for the control observation
96   # Candidates: the candidates for the control observation
97   # Candidates: the candidates for the control observation
98   # Candidates: the candidates for the control observation
99   # Candidates: the candidates for the control observation
100  # Candidates: the candidates for the control observation
```

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=01;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@oxluk3:~# 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@0x1uk3:~# 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@0x1uk3:~# 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
```

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**.

```
24 # @experiment - the experiment data result as a list
25 # @observations - an array of Observations, in the format of
26 # @control - the control observation
27
28 def skillRanking(experiment, observations = [], control = null)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations - {control}
33   evaluate_candidates
34
35   freeze
36
37   experiment.context
38
39   @experiment.name
40   experiment.name
41
42   nil
43
44   # @return - the result a match between all candidates
45   def match?
46     ...
47   end
48
49   @experiment.result[0] 1:1
```



4.3 Bash Commands and Programs

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

You can find more [here](http://manpages.ubuntu.com/manpages/bionic/man7/bash-builtins.7.html).

```
1 def initialize_experiment(experiment, observations = [], control = null)
2   # Initialize the experiment
3   # Create the experiment object
4   # Create the experiment object
5   # Create the experiment object
6   # Create the experiment object
7   # Create the experiment object
8   # Create the experiment object
9   # Create the experiment object
10  # Create the experiment object
11  # Create the experiment object
12  # Create the experiment object
13  # Create the experiment object
14  # Create the experiment object
15  # Create the experiment object
16  # Create the experiment object
17  # Create the experiment object
18  # Create the experiment object
19  # Create the experiment object
20  # Create the experiment object
21  # Create the experiment object
22  # Create the experiment object
23  # Create the experiment object
24  # Create the experiment object
25  # Create the experiment object
26  # Create the experiment object
27  # Create the experiment object
28  # Create the experiment object
29  # Create the experiment object
30  # Create the experiment object
31  # Create the experiment object
32  # Create the experiment object
33  # Create the experiment object
34  # Create the experiment object
35  # Create the experiment object
36  # Create the experiment object
37  # Create the experiment object
38  # Create the experiment object
39  # Create the experiment object
40  # Create the experiment object
41  # Create the experiment object
42  # Create the experiment object
43  # Create the experiment object
44  # Create the experiment object
45  # Create the experiment object
46  # Create the experiment object
47  # Create the experiment object
48  # Create the experiment object
49  # Create the experiment object
50  # Create the experiment object
51  # Create the experiment object
52  # Create the experiment object
53  # Create the experiment object
54  # Create the experiment object
55  # Create the experiment object
56  # Create the experiment object
57  # Create the experiment object
58  # Create the experiment object
59  # Create the experiment object
60  # Create the experiment object
61  # Create the experiment object
62  # Create the experiment object
63  # Create the experiment object
64  # Create the experiment object
65  # Create the experiment object
66  # Create the experiment object
67  # Create the experiment object
68  # Create the experiment object
69  # Create the experiment object
70  # Create the experiment object
71  # Create the experiment object
72  # Create the experiment object
73  # Create the experiment object
74  # Create the experiment object
75  # Create the experiment object
76  # Create the experiment object
77  # Create the experiment object
78  # Create the experiment object
79  # Create the experiment object
80  # Create the experiment object
81  # Create the experiment object
82  # Create the experiment object
83  # Create the experiment object
84  # Create the experiment object
85  # Create the experiment object
86  # Create the experiment object
87  # Create the experiment object
88  # Create the experiment object
89  # Create the experiment object
90  # Create the experiment object
91  # Create the experiment object
92  # Create the experiment object
93  # Create the experiment object
94  # Create the experiment object
95  # Create the experiment object
96  # Create the experiment object
97  # Create the experiment object
98  # Create the experiment object
99  # Create the experiment object
100 # Create the experiment object
```

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 of padbytes 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@xluk3:~# cd /bin
root@xluk3:/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
```

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@0x1uk3:~/Desktop# echo ~  
/root
```

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

```
root@0x1uk3:~/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@0x1uk3:~/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/dleya-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**.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
This script is a simple example of a Bash script that
demonstrates the use of output redirection.
"""

# Import the necessary modules
import sys
import os

# Define the main function
def main():
    # Create a file named 'output.txt'
    with open('output.txt', 'w') as f:
        # Write some text to the file
        f.write('This is a test file.\n')
        f.write('It contains some text.\n')
        f.write('And some more text.\n')

    # Print the contents of the file
    with open('output.txt', 'r') as f:
        print(f.read())

# Call the main function
if __name__ == '__main__':
    main()
```

4.4.2 Bash Output Redirectors

In order to redirect the output to a file:

- Use the **command > file.txt** format to 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**“.

```
77 def context
78   experiment.context
79 end
80
81 # Fetch the name of the experiment
82 def experiment_name
83   experiment.name
84 end
85
86 # Fetch the result of the experiment
87 def match
88   match
89 end
90
91 # Fetch the result of the experiment
92 def result
93   result
94 end
```



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.

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.

```
20 # ...
21 # ...
22 # ...
23 # ...
24 # ...
25 # ...
26 # ...
27 # ...
28 # ...
29 # ...
30 # ...
31 # ...
32 # ...
33 # ...
34 # ...
35 # ...
36 # ...
37 # ...
38 # ...
39 # ...
40 # ...
41 # ...
42 # ...
43 # ...
44 # ...
45 # ...
46 # ...
47 # ...
48 # ...
49 # ...
50 # ...
51 # ...
52 # ...
53 # ...
54 # ...
55 # ...
56 # ...
57 # ...
58 # ...
59 # ...
60 # ...
61 # ...
62 # ...
63 # ...
64 # ...
65 # ...
66 # ...
67 # ...
68 # ...
69 # ...
70 # ...
71 # ...
72 # ...
73 # ...
74 # ...
75 # ...
76 # ...
77 # ...
78 # ...
79 # ...
80 # ...
81 # ...
82 # ...
83 # ...
84 # ...
85 # ...
86 # ...
87 # ...
88 # ...
89 # ...
90 # ...
91 # ...
92 # ...
93 # ...
94 # ...
95 # ...
96 # ...
97 # ...
98 # ...
99 # ...
100 # ...
```



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`

```
24 # The experiment will result in 200 observations
25 # observations - an array of Observations, an Experiment
26 # control - the control observation
27
28 def initialize(experiment, observations = [], control = null)
29   @experiment = experiment
30   @observations = observations
31   @control = control
32   @candidates = observations - [control]
33   evaluate_candidates
34
35   freeze
36
37   context
38   experiment.context
39 end
40
41 # Define the name of the experiment
42 def experiment_name
43   experiment.name
44 end
45
46 # Define what the result is match between an experiment
47 def match
48   # ...
49   @experiment.result
50 end
```

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

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

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

```
17 # ...
18 # ...
19 # ...
20 # ...
21 # ...
22 # ...
23 # ...
24 # ...
25 # ...
26 # ...
27 # ...
28 # ...
29 # ...
30 # ...
31 # ...
32 # ...
33 # ...
34 # ...
35 # ...
36 # ...
37 # ...
38 # ...
39 # ...
40 # ...
41 # ...
42 # ...
43 # ...
44 # ...
45 # ...
46 # ...
47 # ...
48 # ...
49 # ...
50 # ...
51 # ...
52 # ...
53 # ...
54 # ...
55 # ...
56 # ...
57 # ...
58 # ...
59 # ...
60 # ...
61 # ...
62 # ...
63 # ...
64 # ...
65 # ...
66 # ...
67 # ...
68 # ...
69 # ...
70 # ...
71 # ...
72 # ...
73 # ...
74 # ...
75 # ...
76 # ...
77 # ...
78 # ...
79 # ...
80 # ...
81 # ...
82 # ...
83 # ...
84 # ...
85 # ...
86 # ...
87 # ...
88 # ...
89 # ...
90 # ...
91 # ...
92 # ...
93 # ...
94 # ...
95 # ...
96 # ...
97 # ...
98 # ...
99 # ...
100 # ...
```



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**

```
24 # ...
25 # ...
26 # ...
27 # ...
28 # ...
29 # ...
30 # ...
31 # ...
32 # ...
33 # ...
34 # ...
35 # ...
36 # ...
37 # ...
38 # ...
39 # ...
40 # ...
41 # ...
42 # ...
43 # ...
44 # ...
45 # ...
46 # ...
47 # ...
48 # ...
49 # ...
50 # ...
51 # ...
52 # ...
53 # ...
54 # ...
55 # ...
56 # ...
57 # ...
58 # ...
59 # ...
60 # ...
61 # ...
62 # ...
63 # ...
64 # ...
65 # ...
66 # ...
67 # ...
68 # ...
69 # ...
70 # ...
71 # ...
72 # ...
73 # ...
74 # ...
75 # ...
76 # ...
77 # ...
78 # ...
79 # ...
80 # ...
81 # ...
82 # ...
83 # ...
84 # ...
85 # ...
86 # ...
87 # ...
88 # ...
89 # ...
90 # ...
91 # ...
92 # ...
93 # ...
94 # ...
95 # ...
96 # ...
97 # ...
98 # ...
99 # ...
100 # ...
```

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
```

```
#!/bin/bash  
  
# Example script for a simple experiment  
# This script demonstrates basic bash syntax and conditional statements  
# It sets variables for experiment parameters and uses an if statement to  
# check if a condition is met.  
  
# Set variables for experiment parameters  
experiment_name="Experiment 1"  # Name of the experiment  
experiment_contact="John Doe"  # Contact information  
experiment_date="2023-10-27"  # Date of the experiment  
experiment_location="Lab A"  # Location of the experiment  
experiment_status="Running"  # Status of the experiment  
experiment_duration="1h"  # Duration of the experiment  
experiment_budget="$1000"  # Budget for the experiment  
experiment_team="Team Alpha"  # Team responsible for the experiment  
experiment_notes="Initial setup complete. Data collection begins."  # Notes on the experiment  
  
# Check if the experiment is running  
if [ "$experiment_status" = "Running" ]; then  
    echo "Experiment is running. Data collection is in progress."  # Message if running  
else  
    echo "Experiment is not running. Please check the status."  # Message if not running  
fi  
  
# End of script
```



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 responsible 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
```

```
21 # def main: Create a seq object
22
23 # def main: Create a seq object
24
25 # def main: Create a seq object
26
27 # def main: Create a seq object
28
29 # def main: Create a seq object
30
31 # def main: Create a seq object
32
33 # def main: Create a seq object
34
35 # def main: Create a seq object
36
37 # def main: Create a seq object
38
39 # def main: Create a seq object
40
41 # def main: Create a seq object
42
43 # def main: Create a seq object
44
45 # def main: Create a seq object
46
47 # def main: Create a seq object
48
49 # def main: Create a seq object
50
51 # def main: Create a seq object
52
53 # def main: Create a seq object
54
55 # def main: Create a seq object
56
57 # def main: Create a seq object
58
59 # def main: Create a seq object
60
61 # def main: Create a seq object
62
63 # def main: Create a seq object
64
65 # def main: Create a seq object
66
67 # def main: Create a seq object
68
69 # def main: Create a seq object
70
71 # def main: Create a seq object
72
73 # def main: Create a seq object
74
75 # def main: Create a seq object
76
77 # def main: Create a seq object
78
79 # def main: Create a seq object
80
81 # def main: Create a seq object
82
83 # def main: Create a seq object
84
85 # def main: Create a seq object
86
87 # def main: Create a seq object
88
89 # def main: Create a seq object
90
91 # def main: Create a seq object
92
93 # def main: Create a seq object
94
95 # def main: Create a seq object
96
97 # def main: Create a seq object
98
99 # def main: Create a seq object
100
```


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
```

```
18 def initialize(experiment, observations = [], candidates = [])
19   @experiment = experiment
20   @observations = observations
21   @control = control
22   @candidates = candidates
23   evaluate_candidates
24 end
25
26 freeze
27 end
28
29 # PARS: the experiment's context
30 def context
31   experiment.context
32 end
33
34 # PARS: the experiment's name
35 def experiment_name
36   experiment.name
37 end
38
39 # PARS: the result's match between an experiment and a result
40 def matched?
41   @experiment.result.up == 1
42 end
```



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

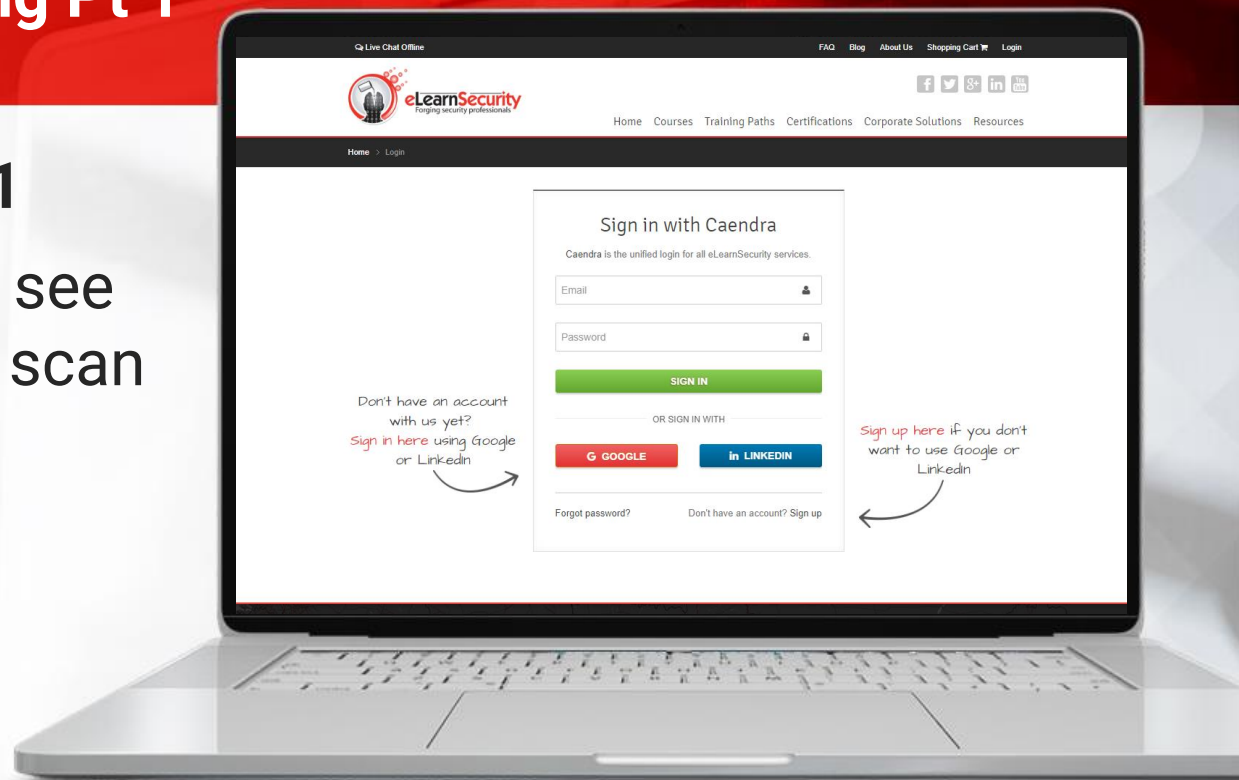
In following videos, you will see exemplary use of Bash scripting to automate some everyday tasks performed by penetration testers.



VIDEO: Bash Scripting Pt 1

Bash Scripting Part 1

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

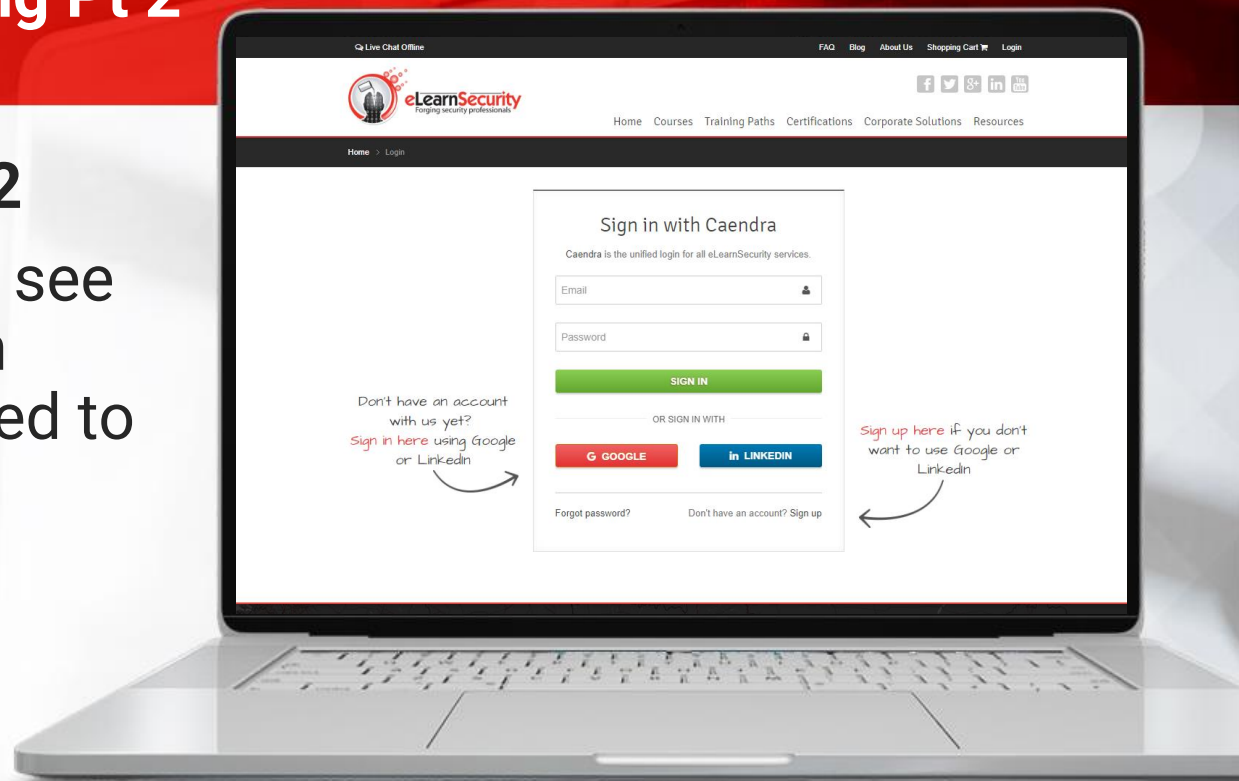


**Videos are only available in Full or Elite Editions of the course. To upgrade, click [HERE](#). To access, go to the course in your members area and click the resources drop-down in the appropriate module line.*

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.*

Windows Command Line



4.6 Windows Command Line

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

```
C:\Users>
```

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.

Windows Environment



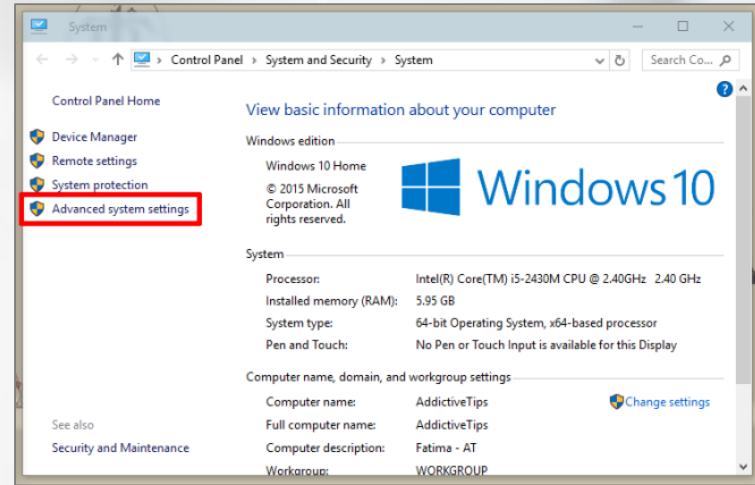
4.7 Windows Environment

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



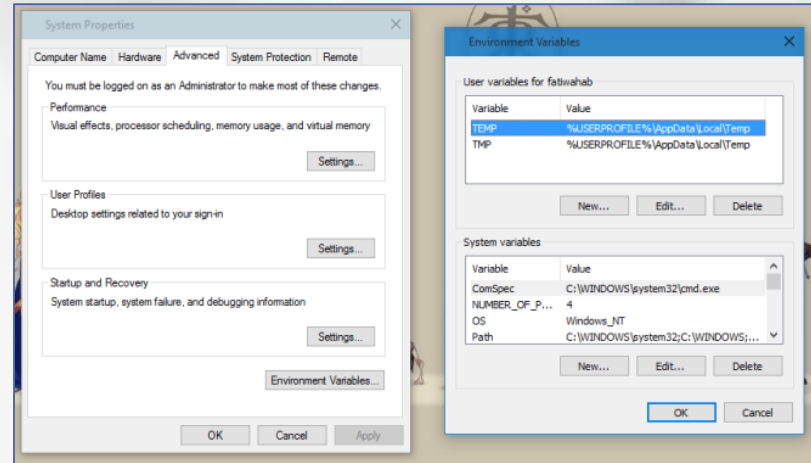
4.7 Windows Environment

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



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**.



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 Components\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
```

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.

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.

EXAMPLE

4.9.2 Windows Output Redirection

```
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

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](#).

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>
```


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/)

<https://ss64.com/nt/>

[Bash programming reference](http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html)

<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)

<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/)

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

[If – Conditionally Perform Command – Windows CMD](https://ss64.com/nt/if.html)

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

[Else – Windows CMD](https://ss64.com/nt/else.html)

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

[For – Looping Commands – Windows CMS](https://ss64.com/nt/for.html)

<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.*