Digital Forensics Workshop (CSE3157)

scrip in Penetration Testing

Dr. Rourab Paul

Computer Science Department, SOA University

Why Scripting?

1. Automating Repetitive Tasks

- Penetration testing often involves repetitive tasks like scanning, enumeration, and brute-forcing.
- Scripts can automate these tasks, saving time and ensuring consistency across tests.
- Example: A script to scan multiple IPs for open ports using nmap

2. Customization of Tools

- Off-the-shelf tools may not cover all edge cases or unique scenarios.
- Scripting allows testers to extend or modify existing tools to meet specific needs.
- Example: Writing a Python script to manipulate HTTP headers for custom web application tests.

3. Faster Exploitation

- Writing scripts for exploit payloads or vulnerability verification can speed up exploitation.
- Example: Creating a script to exploit a buffer overflow vulnerability specific to an application version.

Why Scripting?

4. Efficient Data Parsing

- Penetration testers often deal with large amounts of output data (e.g., scan results, logs).
- Scripting can parse and analyze this data to identify critical information quickly.
- Example: A script to filter and highlight vulnerable services from a Nessus scan report.

5. Reproducibility

- Scripts ensure that testing processes are reproducible and can be executed consistently across multiple engagements.
- Example: Writing a script to automate a specific sequence of tests for every engagement.

6. Target-Specific Attacks

- Many penetration tests require target-specific payloads or attacks.
- Scripting allows testers to craft custom payloads tailored to the vulnerabilities of the target.
- Example: Generating a unique SQL injection payload for a misconfigured database.

Popular Options

We will look at writing programs to automate various useful tasks in multiple programming languages. Even though we use prebuilt
software for the majority of this book, it is useful to be able to create your own programs.

- **bash:** Ideal for automating tasks on Linux systems, which are common in penetration testing. Works on most Unix-like systems without additional dependencies. Limited libraries for advanced tasks like web scraping or interacting with APIs. Difficult to maintain and debug for complex logic.
- **python:** Suitable for almost every penetration testing task, from reconnaissance to exploitation. Requires Python runtime on the target machine (if running scripts there). Overkill for simple, single-line tasks.
- **powershel:** Ideal for penetration tests on Windows environments. Limited to Windows unless using cross-platform tools like PowerShell Core. Detection by antivirus solutions can be high for common payloads

Other Options:

Ruby:

- Used in frameworks like Metasploit for writing and extending modules.

JavaScript:

- Essential for web application testing (e.g., XSS payloads or testing APIs).

Perl:

- Occasionally used for legacy systems and scripts found during reconnaissance.

Recommendation

- **Beginner:** Start with **Bash** for basic Linux automation and reconnaissance tasks.
- **Intermediate:** Learn **Python** to craft custom exploits, payloads, and automate advanced tasks.
- **Windows Testing:** Add **PowerShell** to your toolkit for Windows-specific environments.
- **Web Application Testing:** Include **JavaScript** for XSS, API testing, and browser-based attacks.

Stage: 1 of the Penetration Test

- pre-engagement phase, which involves talking tothe client about their goals for the pentest, mapping out the scope (the extent and parameters of the test), and so on. When the pentester and the client agree about scope, reporting format, and other topics, the actual testing begins.

Stage: 2 of the Penetration Test

- In the information gathering phase, the pentester searches for publicly available information about the client and identifies potential ways to connect to its systems. In the threat-modeling phase, the tester uses this information to determine the value of each finding and the impact to the client if the finding permitted an attacker to break into a system. This evaluation allows the pentester to develop an action plan and methods of attack.

Stage: 3 of the Penetration Test

- vulnerability analysis. In this phase, the pentester attempts to discover vulnerabilities in the systems that can be taken advantage of in the exploitation phase. A successful exploit might lead to a post-exploitation phase, where the result of the exploitation is leveraged to find additional information, sensitive data, access to other systems, and so on.

Final Stage of the Penetration Test

Finally, in the reporting phase, the pentester summarizes the findings for both executives and technical practitioners.

For more information on pentesting

http://www.pentest-standard.org/

Type of Pen test

Black Box: No prior knowledge of the system.

White Box: Full knowledge of the system.

Gray Box: Partial knowledge of the system.

Benefits of Pen test

- Protects against **data breaches** and **financial loss.**
- Helps meet **compliance** and **regulatory requirements.**
- Enhances trust with stakeholders.

Ping

We'll call our first script *pingscript.sh*. When it runs, this script will perform a *ping sweep* on our local network that sends Internet Control Message Protocol (ICMP) messages to remote systems to see if they respond.

We'll use the ping tool to determine which hosts are reachable on a network. (Although some hosts may not respond to ping requests and may be up despite not being "pingable," a ping sweep is still a good place to start.) By default, we supply the IP address or hostname to ping. For example, to ping our Windows XP target, enter the bold code in Listing 3-1.

```
root@kali:~/# ping 192.168.20.10
PING 192.168.20.10 (192.168.20.10) 56(84) bytes of data.
64 bytes from 192.168.20.10: icmp_req=1 ttl=64 time=0.090 ms
64 bytes from 192.168.20.10: icmp_req=2 ttl=64 time=0.029 ms
64 bytes from 192.168.20.10: icmp_req=3 ttl=64 time=0.038 ms
64 bytes from 192.168.20.10: icmp_req=4 ttl=64 time=0.050 ms
^C
--- 192.168.20.10 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999 ms
rtt min/avg/max/mdev = 0.029/0.051/0.090/0.024 ms
```

Listing 3-1: Pinging a remote host

We can tell from the ping output that the Windows XP target is up and responding to ping probes because we received replies to our ICMP requests. (The trouble with ping is that it will keep running forever unless you stop it with CTRL-C.)

A Simple Bash Script

Let's begin writing a simple Bash script to ping hosts on the network. A good place to start is by adding some help information that tells your users how to run your script correctly.

```
#!/bin/bash
echo "Usage: ./pingscript.sh [network]"
echo "example: ./pingscript.sh 192.168.20"
```

The first line of this script tells the terminal to use the Bash interpreter. The next two lines that begin with *echo* simply tell the user that our ping script will take a command line argument (network), telling the script which network to ping sweep (for example, 192.168.20). The `echo` command will simply print the text in quotes.

NOTE *This script implies we are working with a class C network, where the first three octets of the IP address make up the network.*

After creating the script, use `chmod` to make it executable so we can run it.

```
root@kali:~/# chmod 744 pingscript.sh
```

Running Our Script

Previously, when entering Linux commands, we typed the command name at the prompt. The filesystem location of built-in Linux commands as well as pentest tools added to Kali Linux are part of our `PATH` environmental variable. The `PATH` variable tells Linux which directories to search for executable files. To see which directories are included in our `PATH`, enter **`echo $PATH`**.

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

Notice in the output that the */root* directory is not listed. That means that we won't be able to simply enter `pingscript.sh` to run our Bash script. Instead we'll enter **`./pingscript.sh`** to tell the terminal to run the script from our current directory. As shown next, the script prints the usage information.

```
root@kali:~/# ./pingscript.sh
Usage: ./pingscript.sh [network]
example: ./pingscript.sh 192.168.20
```

Adding Functionality with if Statements

Now let's add in a bit more functionality with an `if` statement, as shown in Listing 3-2.

```
#!/bin/bash
if [ "$1" == "" ] ❶
then ❷
echo "Usage: ./pingscript.sh [network]"
echo "example: ./pingscript.sh 192.168.20"
fi ❸
```

Listing 3-2: Adding an `if` statement

Typically a script needs to print usage information only if the user uses it incorrectly. In this case, the user needs to supply the network to scan as a command line argument. If the user fails to do so, we want to inform the user how to run our script correctly by printing the usage information.

To accomplish this, we can use an `if` statement to see if a condition is met. By using an `if` statement, we can have our script echo the usage information only under certain conditions—for example, if the user does not supply a command line argument.

The `if` statement is available in many programming languages, though the syntax varies from language to language. In Bash scripting, an `if` statement is used like this: `if [`*`condition`*`]`, where *`condition`* is the condition that must be met.

In the case of our script, we first see whether the first command line argument is null ❶. The symbol `$1` represents the first command line argument in a Bash script, and double equal signs (`==`) check for equality. After the `if` statement, we have a `then` statement ❷. Any commands between the `then` statement and the `fi` (`if` backward) ❸ are executed only if the conditional statement is true—in this case, when the first command line argument to the script is null.

When we run our new script with no command line argument, the `if` statement evaluates as true, because the first command line argument is indeed null, as shown here.

```
root@kali:~/# ./pingscript.sh
Usage: ./pingscript.sh [network]
example: ./pingscript.sh 192.168.20
```

As expected we see usage information echoed to the screen.

A for Loop

If we run the script again with a command line argument, nothing happens. Now let's add some functionality that is triggered when the user runs the script with the proper arguments, as shown in Listing 3-3.

```
#!/bin/bash
if [ "$1" == "" ]
then
echo "Usage: ./pingscript.sh [network]"
echo "example: ./pingscript.sh 192.168.20"
else ❶
for x in `seq 1 254`; do ❷
ping -c 1 $1.$x
done ❸
fi
```

Listing 3-3: Adding a for loop

After our `then` statement, we use an `else` statement ❶ to instruct the script to run code when the `if` statement evaluates as false—in this case, if the user supplies a command line argument. Because we want this script to ping all possible hosts on the local network, we need to loop through the numbers 1 through 254 (the possibilities for the final octet of an IP version 4 address) and run the `ping` command against each of these possibilities.

An ideal way to run through sequential possibilities is with a `for` loop ❷. Our `for` loop, `for x in `seq 1 254`; do`, tells the script to run the code that follows for each number from 1 to 254. This will allow us to run one set of instructions 254 times rather than writing out code for each instance. We denote the end of a `for` loop with the instruction `done` ❸.

Inside the `for` loop, we want to ping each of the IP addresses in the network. Using ping's man page, we find that the `-c` option will allow us to limit the number of times we ping a host. We set `-c` to `1` so that each host will be pinged just once.

To specify which host to ping, we want to concatenate the first command line argument (which denotes the first three octets) with the current iteration of the `for` loop. The full command to use is `ping -c 1 $1.$x`. Recall that the `$1` denotes the first command line argument, and `$x` is the current iteration of the `for` loop. The first time our `for` loop runs, it will ping 192.168.20.1, then 192.168.20.2, all the way to 192.168.20.254. After iteration 254, our `for` loop finishes.

When we run our script with the first three octets of our IP address as the command line argument, the script pings each IP address in the network as shown in Listing 3-4.

```
root@kali:~/# ./pingscript.sh 192.168.20
PING 192.168.20.1 (192.168.20.1) 56(84) bytes of data.
64 bytes from 192.168.20.1: icmp_req=1 ttl=255 time=8.31 ms ❶

--- 192.168.20.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.317/8.317/8.317/0.000 ms
PING 192.168.20.2(192.168.20.2) 56(84) bytes of data.
64 bytes from 192.168.20.2: icmp_req=1 ttl=128 time=166 ms

--- 192.168.20.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 166.869/166.869/166.869/0.000 ms
PING 192.168.20.3 (192.168.20.3) 56(84) bytes of data.
From 192.168.20.13 icmp_seq=1 Destination Host Unreachable ❷

--- 192.168.20.3 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
--snip--
```

Listing 3-4: Running the ping sweep script

Your results will vary based on the systems in your local network. Based on this output, I can tell that in my network, the host 192.168.20.1 is up, and I received an ICMP reply ❶. On the other hand, the host 192.168.20.3 is not up, so I received a host unreachable notification ❷.

Streamlining the Results

All this information printed to screen is not very nice to look at, and anyone who uses our script will need to sift through a lot of information to determine which hosts in the network are up. Let's add some additional functionality to streamline our results.

In the previous chapter we covered `grep`, which searches for and matches specific patterns. Let's use `grep` to filter the script's output, as shown in Listing 3-5.

```
#!/bin/bash
if [ "$1" == "" ]
then
echo "Usage: ./pingscript.sh [network]"
echo "example: ./pingscript.sh 192.168.20"
else
for x in `seq 1 254`; do
ping -c 1 $1.$x | grep "64 bytes" ❶
done
fi
```

Listing 3-5: Using grep to filter results

Here we look for all instances of the string `64 bytes` ❶, which occurs when an ICMP reply is received when pinging a host. If we run the script with this change, we see that only lines that include the text `64 bytes` are printed to the screen, as shown here.

```
root@kali:~/# ./pingscript.sh 192.168.20
64 bytes from 192.168.20.1: icmp_req=1 ttl=255 time=4.86 ms
64 bytes from 192.168.20.2: icmp_req=1 ttl=128 time=68.4 ms
64 bytes from 192.168.20.8: icmp_req=1 ttl=64 time=43.1 ms
--snip--
```

We get indicators only for live hosts; hosts that do not answer are not printed to the screen.

But we can make this script even nicer to work with. The point of our ping sweep is to get a list of live hosts. By using the `cut` command discussed in Chapter 2, we can print the IP addresses of only the live hosts, as shown in Listing 3-6.

```
#!/bin/bash
if [ "$1" == "" ]
then
echo "Usage: ./pingscript.sh [network]"
echo "example: ./pingscript.sh 192.168.20"
else
for x in `seq 1 254`; do
ping -c 1 $1.$x | grep "64 bytes" | cut -d" " -f4 ❶
done
fi
```

Listing 3-6: Using cut to further filter results

We can use a space as the delimiter and grab the fourth field, our IP address, as shown at ❶.

Now we run the script again as shown here.

```
root@kali:~/mydirectory# ./pingscript.sh 192.168.20
192.168.20.1:
192.168.20.2:
192.168.20.8:
--snip--
```

Unfortunately, we see a trailing colon at the end of each line. The results would be clear enough to a user, but if we want to use these results as input for any other programs, we need to delete the trailing colon. In this case, `sed` is the answer.

The `sed` command that will delete the final character from each line is `sed 's/.$//'`, as shown in Listing 3-7.

```
#!/bin/bash
if [ "$1" == "" ]
then
echo "Usage: ./pingscript.sh [network]"
echo "example: ./pingscript.sh 192.168.20"
else
for x in `seq 1 254`; do
ping -c 1 $1.$x | grep "64 bytes" | cut -d" " -f4 | sed 's/.$//'
done
fi
```

Listing 3-7: Using `sed` to drop the trailing colon

Now when we run the script, everything looks perfect, as shown here.

```
root@kali:~/# ./pingscript.sh 192.168.20
192.168.20.1
192.168.20.2
192.168.20.8
--snip--
```

NOTE *Of course, if we want to output the results to a file instead of to the screen, we can use the >> operator, covered in Chapter 2, to append each live IP address to a file. Try automating other tasks in Linux to practice your Bash scripting skills.*

Python Scripting

Linux systems typically come with interpreters for other scripting languages such as Python and Perl. Interpreters for both languages are included in Kali Linux. In Chapters 16 through 19, we'll use Python to write our own exploit code. For now, let's write a simple Python script and run it in Kali Linux just to demonstrate the basics of Python scripting.

For this example we'll do something similar to our first Netcat example in Chapter 2: We'll attach to a port on a system and see if the port is listening. A starting point for our script is shown here.

```
#!/usr/bin/python ❶
ip = raw_input("Enter the ip: ") ❷
port = input("Enter the port: ") ❸
```

In the previous section, the first line of our script told the terminal to use Bash to interpret the script. We do the same thing here, pointing to the Python interpreter installed on Kali Linux at */usr/bin/python* ❶.

We'll begin by prompting the user for data and recording input into variables. The variables will store the input for use later in the script. To take input from the user, we can use the Python function `raw_input` ❷. We want to save our port as an integer, so we use a similar built-in Python function, `input`, at ❸. Now we ask the user to input an IP address and a port to test.

After saving the file, use `chmod` to make the script executable before running the script, as shown here.

```
root@kali:~/mydirectory# chmod 744 pythonscript.py
root@kali:~/mydirectory# ./pythonscript.py
Enter the ip: 192.168.20.10
Enter the port: 80
```

When you run the script, you're prompted for an IP address and a port, as expected.

Now we will add in some functionality to allow us to use the user's input to connect to the chosen system on the selected port to see if it is open (Listing 3-8).

```
import socket #1
ip = input("Enter the IP: ")
port = int(input("Enter the port: "))
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #2
s.settimeout(2)  # Set timeout to avoid hanging indefinitely
try:
    result = s.connect_ex((ip, port))
    if result == 0:
        print(f"Port {port} is open")
    else:
        print(f"Port {port} is closed")
except socket.error as e:
    print(f"Error occurred: {e}")
finally:
    s.close()
```

To perform networking tasks in Python, we can include a library called *socket* using the command `import socket` ❶. The socket library does the heavy lifting for setting up a network socket.

The syntax for creating a TCP network socket is `socket.socket(socket.AF_INET, socket.SOCK_STREAM)`. We set a variable equal to this network socket at ❷.

Connecting to a Port

When creating a socket to connect to a remote port, the first candidate available from Python is the socket function `connect`. However, there is a better candidate for our purposes in the similar function, `connect_ex`. According to the Python documentation, `connect_ex` is like `connect` except that it returns an error code instead of raising an exception if the connection fails. If the connection succeeds, `connect_ex` will return the value `0`. Because we want to know whether the function can connect to the port, this return value seems ideal to feed into an `if` statement.

if Statements in Python

When building `if` statements in Python, we enter `if` *`condition`*`:`. In Python the statements that are part of a conditional or loop are denoted with indentations rather than ending markers, as we saw in Bash scripting. We can instruct our `if` statement to evaluate the returned value of the connection of our TCP socket to the user-defined IP address and port with the command `if s.connect_ex((ip, port)):` ❸. If the connection succeeds, `connect_ex` will return `0`, which will be evaluated by the `if` statement as false. If the connection fails, `connect_ex` will return a positive integer, or true. Thus, if our `if` statement evaluates as true, it stands to reason that the port is closed, and we can present this to the user using the Python `print` command at ❹. And, as in the Bash scripting example, if `connect_ex` returns `0` at ❺, we can use an `else` statement (the syntax is `else:` in Python) to instead inform the user that the tested port is open.

Now, run the updated script to test whether TCP port 80 is running on the Windows XP target host as shown here.

```
root@kali:~/# ./pythonscript.py
Enter the ip: 192.168.20.10
Enter the port: 80
Port 80 is open
```

According to our script, port 80 is open. Now run the script again against port 81.

```
root@kali:~/# ./pythonscript.py
Enter the ip: 192.168.20.10
Enter the port: 81
Port 81 is closed
```

This time, the script reports that port 81 is closed.

NOTE *We will look at checking open ports in Chapter 5, and we will return to Python scripting when we study exploit development. Kali Linux also has interpreters for the Perl and Ruby languages. We will learn a little bit of Ruby in Chapter 19. It never hurts to know a little bit of multiple languages. If you are up for a challenge, see if you can re-create this script in Perl and Ruby.*

Writing and Compiling C Programs

Time for one more simple programming example, this time in the C programming language. Unlike scripting languages such as Bash and Python, C code must be compiled and translated into machine language that the CPU can understand before it is run.

Kali Linux includes the GNU Compiler Collection (GCC), which will allow us to compile C code to run on the system. Let's create a simple C program that says hello to a command line argument, as shown in Listing 3-9.

```
#include <stdio.h> ❶
int main(int argc, char *argv[]) ❷
{
    if(argc < 2) ❸
    {
        printf("%s\n", "Pass your name as an argument"); ❹
        return 0; ❺
    }
    else
    {
                printf("Hello %s\n", argv[1]); ❻
                return 0;
    }
}
```

Listing 3-9: "Hello World" C program

The syntax for C is a bit different from that of Python and Bash. Because our code will be compiled, we don't need to tell the terminal which interpreter to use at the beginning of our code. First, as with our Python example, we import a C library. In this case we'll import the *stdio* (short for standard input and output) library, which will allow us to accept input and print output to the terminal. In C, we import *stdio* with the command `#include <stdio.h>` ❶.

Every C program has a function called `main` ❷ that is run when the program starts. Our program will take a command line argument, so we pass an integer `argc` and a character array `argv` to `main`. `argc` is the argument count, and `argv` is the argument vector, which includes any command line arguments passed to the program. This is just standard syntax for C programs that accept command line arguments. (In C, the beginning and end of functions, loops, and so on are denoted by braces {}.)

First, our program checks to see if a command line argument is present. The `argc` integer is the length of the argument array; if it is less than two (the program name itself and the command line argument), then a command line argument has not been given. We can use an `if` statement to check ❸.

The syntax for `if` is also a little different in C. As with our Bash script, if a command line argument is not given, we can prompt the user with usage information ❹. The `printf` function allows us to write output to the terminal. Also note that statements in C are finished with a semicolon (`;`). Once

we're through with our program, we use a `return` statement ❺ to finish the function `main`. If a command line argument is supplied, our `else` statement instructs the program to say hello ❻. (Be sure to use braces to close all of your loops and the `main` function.)

Before we can run our program, we need to compile it with GCC as shown here. Save the program as *cprogram.c*.

```
root@kali:~# gcc cprogram.c -o cprogram
```

Use the `-o` option to specify the name for the compiled program and feed your C code to GCC. Now run the program from your current directory. If the program is run with no arguments, you should see usage information as shown here.

```
root@kali:~# ./cprogram
Pass your name as an argument
```

If instead we pass it an argument, in this case our name, the program tells us hello.

```
root@kali:~# ./cprogram georgia
Hello georgia
```

NOTE *We will look at another C programming example in Chapter 16, where a little bit of sloppy C coding leads to a buffer overflow condition, which we will exploit.*

Summary

In this chapter we've looked at simple programs in three different languages. We looked at basic constructs, such as saving information in variables for later use. Additionally, we learned how to use conditionals, such as `if` statements, and iterations, such as `for` loops, to have the program make decisions based on the provided information. Though the syntax used varies from programming language to programming language, the ideas are the same.

Thank You