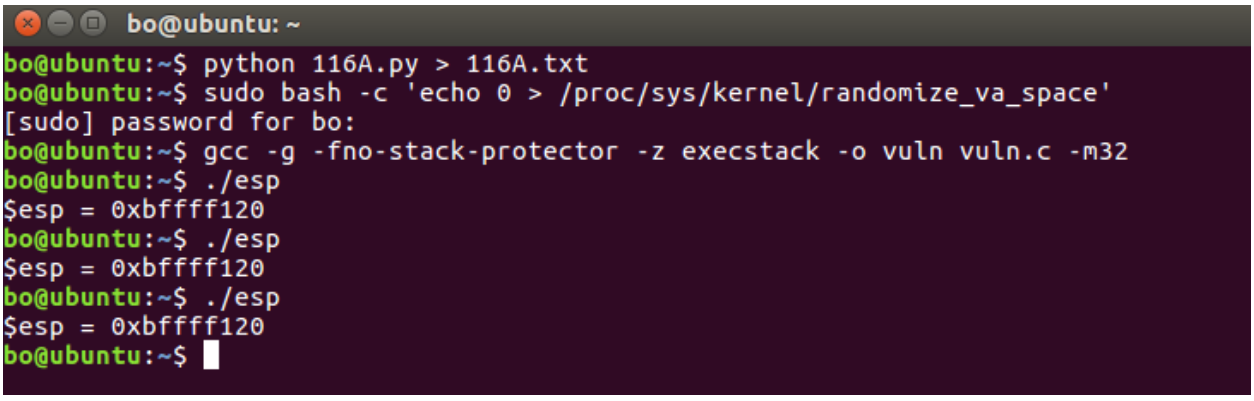
Exploration about buffer Overflow

Bo Wen

The buffer overflow has long been a feature of the computer security landscape (Peter Bright). Because computer OS deal with the data all the time, buffer overflow becomes common thing that attackers can use. Basically operating system deal with data from user, file or network, and store those data into the buffer, when the size of data larger than the size of buffer, buffer overflow happens. Buffer overflow happens on the program uses the processor's instruction, which is called native code. In C, strcpy, get and cin functions can easily cause buffer overflow.

In this lab, I explored and finished the lab by using C++. Because of the input from command line directly copy into the fixed size of buffer, which is 100 bytes long. From the stack architecture, the main function layout starts from register ESP and ends up EBP. At the middle are the buffer fill of 100 chars. The next memory address is the return address.

In this lab, first I disabled the ASLR and turn off stack protector. Make sure the stack pointer is always on the same position. There is the C library can direct see what is value in ESP.



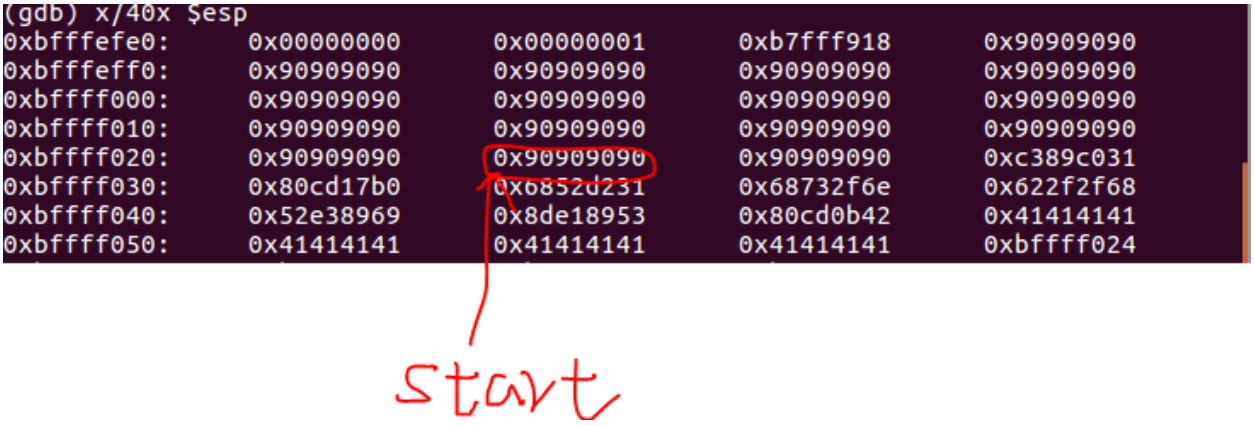
For the 100 A characters, go inside the gdb debug mode, can see esp and ebp registers.

```
(gdb) info registers
eax            0xbfffeffc       -1073745924
ecx            0xbffff370       -1073745040
edx            0xbffff05e       -1073745826
ebx            0x0      0
esp            0xbfffeff0       0xbfffeff0
ebp            0xbffff068       0xbffff068
esi            0xb7fbb000       -1208242176
edi            0xb7fbb000       -1208242176
eip            0x8048459        0x8048459 <func+30>
```

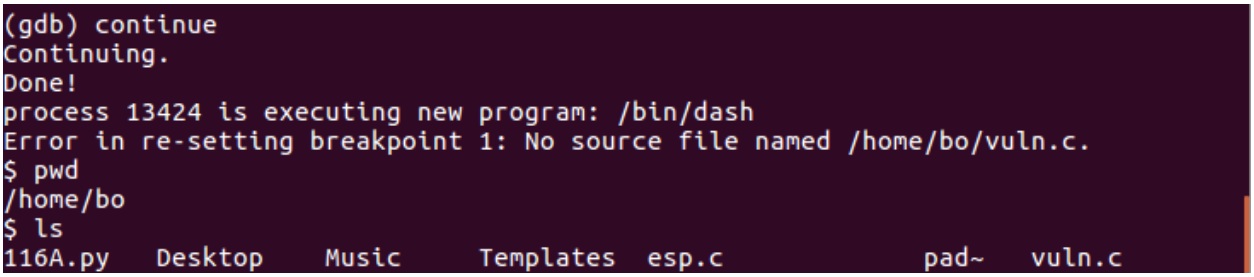
```
(gdb) x/40x $esp
0xbfffeff0:     0x00000000      0x00000001      0xb7fff918      0x41414141
0xbffff000:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff010:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff020:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff030:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff040:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff050:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff060:     0x00000000      0x00000000      0xbffff088      0x0804848a
0xbffff070:     0xbffff134      0xbffff134      0xbffff140      0x080484c1
```

There are 100 “\41” inside the stack. Because ebp is 0xbfffff068. I need to over write the return address back to the range between 00 to 30, then insert the NOP and shellcode, the shellcode opens dash which is 32 bytes long. For this lab, I chose “0xbfffff024”. However, any range between that and has enough space for shellcode is enough. The arithmetic is here. “116” bytes for total, “32” bytes for shellcode, “4” bytes for return address, “16” bytes A for padding between shellcode and return address. The NOP is “116-32-4-16=64”. The assemble the

payload and write into program.



And everything looks good. The program returns to those NOP then runs into the shellcode.



There is the new dash program is running. And I can run “pwd” and “ls” shell script.

The problem, I had because I didn’t split the strcpy into another function, if I write strcpy inside the main function, the buffer overflow can’t overwrite the return value. It is a fun lab, I learn how the stack memory is and how it works.

Work Citation

Peter, Bright How security flaws work: The buffer overflow. August 25, 2015

https://arstechnica.com/information-technology/2015/08/how-security-flaws-work-the-buffer-overflow/