M1522.000800 System Programming
Fall 2019

System Programming Buflab Report

Jinje Han
2018-18786

1. <lab> Buffer Lab

The main goal of the buffer lab is to understand how the run-time stack operates when a function is called. The lab is composed of five stages. In each stages I have to make the program to achieve certain goals by making specific inputs. All inputs will overrun the part of stack assigned to them and make the program work differently from the original.

2. Implementation

2.1 Smoke

The objective of this part is calling the function <smoke> by changing the return address of function <test> using input string. Verifying the address of <smoke> could be done in one command. That is, all I had to do was finding where the return address is and how to change it. I put on the break at function <getbuf> and followed each assembly commands. After some lines, a value was put at the top of the stack and function named <strtol> was called. Since argument registers didn't get change any, I could infer the value is the argument of the function which is an address. I checked what was in that address, which turned out to be my student id number in string format. (Specifically, the last digits, since number 5 was added beforehand.) By name of the function <strtol>, it seemed to convert the figure string into integer, and it did.

Using this integer value, [%eax] points to specific part of the stack. Then, the value of [%eax] is loaded to the top of the stack, and function <gets> is called. Obviously, the value of [%eax] is used to assign the starting address to save the input value from <gets>. The return address of <test> was calculated 49 bytes away from that address. Therefore, the input I needed was string with 49 leading characters which didn't matter at all (except 0x0A), followed by the address of <smoke> with little endian style.

2.2 Sparkler

Changing the return address of function <test> to the address of <fizz> was similar to the level above. The difference was that I also had to change arguments of <fizz>. I didn't know if the arguments are passed by register or by stack, so first of all I made input that only returns to <fizz> and see how values were treated. I found that value in [%ebp + 8] is treated as val1 (first argument), and value in [%ebp + 12] as val2 (second argument). That is, arguments were passed through stack.

Right after returning to <fizz>, the val1 was located at 4 bytes below the top of the stack, and the val2 was right below that. Therefore, I had to add 12 bytes to the input, last 4 of which being val1, and middle 4 being val2. Value of front 4 bytes didn't matter. All that remained was setting the value. Val1 was bitwise complemented, shifted 8 bits left, and bitwise ANDed with my cookie value. That value should be same with val2. There could be many solutions to make them same, but I simply set val1 to 0 and val2 to cookie value that had 0 in last 8 bits (= 1 byte).

2.3 Firecracker

I looked at the assembly code of <bang> as I done before. Value in address 0x804d120, which was 0, was loaded to [%eax], and value in address 0x804d128 was loaded to [%edx]. As shown in the program guide, [%edx] contained my cookie value. If the value that cookie and 0xF0F are bitwise ANDed is same with the value of [%eax], the instruction I want will be executed. Therefore, I had to change the value in 0x804d120. However, there weren't any instructions that approach there. I had to write my own instruction that change value and execute it. Instructions I wrote were as below.

```
movl $0x804d120, %eax
movl $0x00000d0c, (%eax)
```

```
ret
```

If I write these instructions at the front of the input, they will be written on a certain point the stack, which I found at 2.1 (specifically, address 0x55683c73). Of course, the instructions should be written in byte values, and I used objdump to convert them. I order to make PC to point at these instructions, I had to write this value at the return address of function <test>. In addition, the instruction I wrote also contained RET instruction that is expected to change the value of PC to the address of <bang>, so I had to write that address at the end of the input. As above, these two addresses should be written in little endian, which the instruction byte codes shouldn't. In short, the input consist of (assembly instructions) - (empty space) - address of written instructions (little endian) - address of <bang> (little endian)

2.4 Dynamite

The objective of this process was executing an instruction in function <test> without making irregular operations. Reaching that instruction could be done by changing the value of variable [val] into my cookie, and keeping the value of variable [local] same. In short, I should only manage the value of these two, without changing anything else.

Simply changing the value of [val] by input of <gets> wasn't possible, since the position of variable [local] was below [val]. When the input changes [val], it also changes [local], whose value must be maintained to the value made by function <uniqueval>. The return value of <uniqueval> changed every time I execute the program, so I couldn't predict and write it on the input. I had to find another way to achieve the purpose.

Executing instructions could be the way. As I did in 2.3, I wrote instructions that change the value of [val], and the address of where these instructions will be stored in the input. In addition, parts of the stack except them should have been preserved. I found that the value of old [%ebp] and [%ebx] are stored in the stack where the input would overwrite. I checked their value and modified the corresponding part of the input as same as them.

Instructions I wrote are as below.

```
push %ebx
movl $0x55683cc4, %ebx          //0x55683cc4 : address of [val]
movl $0x3cce2d0c, (%ebx)        //0x3cce2d0c : my cookie
pop %ebx
subl $0x4, %esp
movl $0x8048ec1, (%esp)         //0x8048ec1 : return address from <getbuf> to <test>
ret
```

During execution of the instructions I wrote, I had to use the register [%ebx]. In order to preserve the value of it, I pushed it in the stack and popped it out after using it. Finally, the input was consist of (instructions) – (empty space) – old [%ebx], [%ebp] value – address of written instructions.

2.5 Nitroglycerin

In this level, function <testn> is called instead of <test>, and <getbufn> is called instead of <getbuf>. <getbuf> has bigger stack size and called five times. The position of that function changes each time it is called, but I have to input identical strings and make it return my cookie value each time. Therefore, I have to carefully treat addresses. Fortunately, return value of <getbufn> is passed throught [%eax], no pointer.

First of all, it is obvious that I had to write some instructions that push cookie value in %eax. Plus, I had to make the instruction pointer jump to those instructions. Thus, overwriting return address was inevitable. The problem was that in this process, old [%ebp] value is also overwritten. However, there weren't any way to move this value and save it at another position. I had to infer the original value after overwriting, and the clue was the value of [%esp]. Since <getbufn> is always called by same instruction in <testn> all the time, the stack size of <testn> is same when <getbufn> is called. That is, the gap between [%esp] and old [%ebp] is identical. I wrote instructions that calculate and store value of old [%ebp]. The final instructions I wrote are as below.

```
sub $0x4, %esp
movl $0x8048e2d, (%esp)          //restore original return address
movl %esp, %eax
addl $0x2c, %eax                     //calculate original old [%ebp] value
sub $0x4, %esp
movl %eax, (%esp)                //restore old [%ebp] value
movl $0x3cce2d0c, %eax           //move cookie value to [%eax]
movl %esp, %ebp                      //restore original [%ebp] value
leave
ret
```

The next problem was where to jump(=what return address to overwrite with) and where in the input should I write instructions. As mentioned above, the position of stack varies on every call of <getbufn> and I have to make the program to reach the instructions. The solution I found was writing the instructions as far back as possible (=right before the position of old [%ebp]) and filling the rest as nop instructions. In that way, if the instruction pointer comes to reach one of the nop instructions, it will go down a bunch of nop instructions and reach the code I wrote. The return address that I overwrite with was the address where nop instruction will be located in all 5 calls.

3. Conclusion

Doing this lab I could understand how an assembly program works deeply. What I should know in this lab included how functions pass arguments, what do functions do when they are called and return from their call (especially, how [%ebp] works), and states to preserve before calling a function. I could observe how functions work step by step, and apply what I knew to achieve the objectives. In addition, I could be familiar with writing values with little endian style.

The assembly program surprisingly could not distinguish between values and instructions in the memory. I realized we can make a program operate very differently than intended by just changing it little, and grasped the security weakness if we don’t prepare for an attack gracefully.