M1522.000800 System Programming
Fall 2018

System Programming Buflab Report

John Doe
2016-00000

1. <lab> Buffer Lab

The goal of the buffer lab is to make proper input which make 'buffer-overflow' attack to target program. Full understand of IA-32 calling convention and stack organization is required to solve this lab.

2. Implementation

The buffer lab consists of five phases: Smoke, Sparkler, Firecracker, Dynamite and Nitroglycerin.

2.1 Smoke

It's base task to solve further tasks. To solve this, I need to modify the return address which is saved in stack frame.

First of all, 'getbuf' function calls 'Gets' to read. Before calling Gets, getbuf put the value of %eax into (%esp) which is going to be a parameter of Gets function. That means, my input will be saved from this address.

I can get more information about stack and registers. %eax = %ebp-0x30 means that I need 48 bytes to fill from %eax to right below to %ebp. I need 4 more bytes to our destination which is return address. Thus, I put 13 * 4 bytes of [00] followed address of 'smoke' function.

'smoke', 'fizz', 'bang' functions are ended by calling exit function which terminates the process. It means that when I entered one of these functions, I don't have to care about messing up other stack frame.

2.2 Sparkler

I can find 2 mov instruction at fizz+0x6 and fizz+0x9.

```
mov 0x8(%ebp), %eax
mov 0xc(%ebp), %edx
```

By IA-32 calling convention, %eax will have 1st parameter and %edx will have 2nd parameter after executing these 2 instructions. After executing few arithmetic instruction, I can find cmp instruction at fizz+0x19. Here, cmp instruction compares %edx and %ecx but there is 'and' instruction right before the cmp. This means that %ecx could be zero regardless of what is stored in [0x804d128] if %eax (1st argument) is 0xffffffff. This also means that If I can put 0xffffffff and 0x0 into 0x8(%ebp) and 0xc(%ebp) respectively, the result of cmp instruction will always be 'equal'.

First of all, I modify the return address of smoke.txt (answer of 2.1 smoke). Before putting 2 arguments consecutively, I have to consider that fizz function push %ebp before setup stack frame with the instruction 'mov %esp, %ebp'. Because of that, I need 1 empty line (4 byte) between return address and 2 arguments.

2.3 Firecracker

‘bang’ simply compares following expression. “compare [0x804d120], [0x804d128] & 0xf0f”
With objdump -D option, I can disassemble whole file. The result includes .bss section, which contains the name of variables. 0X804d128 is the location of ‘cookie’ variable. For double check, I used gdb to look up the register and my cookie was stored correctly. But 0x804d120 which name was ‘global_value’ contains zero but there’s no instructions which modifies this value in the program. Thus, I decided to modify it myself.

When I solve smoke and sparkler, I put the address of destination at return address position manually. This also means that I can overwrite anything on it even the destination is not in .text section and I decided to my own put machine code on stack and execute it.
Right before the ‘cmp’ instruction in bang (which located on bang+0x17), %edx is set to value of (0xf0f & [My Cookie = 0x6968c9d7] = 0x907). After I change the value of Mem[0x804d120], I need to enter the function ‘bang’ which located at 0x8048c81. Thus I generate machine code with following assembly instructions.

```
mov $0x907, 0x804d120
push $0x8048c81
ret
```

These are 16 bytes of machine code. (c7 05 20 d1 04 08 07 09 00 00 68 81 8c 04 08 c3)
I put this from the first line to fourth line (each line contains 4 bytes).

Finally, I used gdb to find out where the stack is located. I set breakpoint at getbuf+0xd and then look up the value of %eax with ‘info registers’. The result was %eax = 0x55683870. I completed my input file with adding 70 38 68 55 (reverse due to little endian) at 14th line.

2.4 Dynamite

The goal of this phase is return to our original function after I do my own thing. From 2.1 to 2.3, I don’t have to worry about stack frame that much because all the functions that I moved on finish with exit function. To solve this phase, I need to preserve ‘old %ebp’ value which is stored on top of return address. I use gdb to look up the registers right before the instruction movl %esp, %ebp at test+0x1. Then, I decided to overwrite the value of old %ebp on itself to overwrite return address of getbuf stack frame.
At 0x8048e7f, there is an instruction mov -0xc(%ebp), %eax which loads a value which are going to be compared with my cookie. Now, all I have to do is write my cookie value into proper position -0xc(%ebp). Thus I decided to move my cookie value from 0x804d128 to -0xc(%ebp) but I need 1 register to mediate this Mem to Mem movement.
And the final result is as follows.

```
push %eax
mov 0x804d128, %eax
mov %eax, -0xc(%ebp)
pop %eax
push $0x8048e65
ret
```

I use push %eax and pop %eax instruction not to mess up the register %eax.

These instructions are 16 bytes of machine code. (50 a1 28 d1 04 08 89 45 f4 58 68 65 8e 04 08 c3) followed by 00s and finished with old %ebp and modified return address. (d0 38 68 55 70 38 68 55)

After I passed this this phase, I find out shorter machine code as follows with the idea that write my cookie value into destination memory directly.

```
push $0x6968c9d7, -0xc(%ebp)
push $0x8048e65
ret
```

These take only 3 instruction with 13 bytes and also work.
(c7 45 f4 d7 c9 68 69 68 65 8e 04 08 c3)

2.5 Nitroglycerin

While I was solving previous phases, I found out that bufbomb contains 2 functions named 'getbufn' and 'testn' and only notable difference was the size of buffer. I made c program named inputGenerator which generate bigger version of 'dynamite.txt'. But the result contains 1 pass but 4 fails. I draw whole stack frame with MS Excel and track the assembly instructions to find where the loop located.
'main' function of bufbomb generate 5 random offset through the loop from main+0x194 to main+0x1ad. In my case, generated offset and relative offset(related to first loop) was as follows.

1st loop : 0x710 → 0x0
2nd loop : 0x6d0 → 0x30
3rd loop : 0x740 → -0x30
4th loop : 0x780 → -0x70
5th loop : 0x740 → -0x30

Relative offset has opposite order with generated random number because 'sub' instruction which located at 0x80491b8.
Thus, there are 5 locations that input buffer starts by loop.

0x55683698 / 0x556836c8 / 0x55683668 / 0x55683628 / 0x55683668

Problem of my first try was overwriting old %ebp which located right below return address. The value of old %ebp changes from the 2nd loop. Thus I decided not to overwrite value into memory but make machine code to modify %ebp. At testn+0x1, function save %esp into %ebp and then sub 0x28 to %esp. This means, when getbufn returns %old ebp equals to %esp + 0x28. Thus I made following machine code to insert into stack.

```
lea 0x28(%esp), %ebp
mov 0x804d128, %eax
push $0x8048dd1
ret
```
(8d 6c 24 28 a1 28 d1 04 08 68 d1 8d 04 08 c3) → 15 bytes

I tried with new input file but it causes segmentation fault. I found out that overwritten return address indicates outside of allocated memory. I fixed it to placing my machine code middle of the buffer. However, it still didn't work. From 2nd loop, overwritten address indicates empty space where I filled with 00s. I need some machine code on these space to leads PC to my machine code.

But my machine code could be overlapped if I put 4 jmp instruction after some calculation for each loop. But there was a different solution. I decided to fill all 00s with meaningless instruction such as andl %eax, %eax
But size of this and instruction was 4, but my machine code has odd bytes. This means that this solution may work on this lab, but not for universal use. After few hours, I finally recall that there was nop instruction which takes only 1 byte.

Finally my input file looks like below.

```
90 90 90 90
……………
90 90 90 90
8d 6c 24 28
a1 28 d1 04
08 68 d1 8d
04 08 c3 00
00 00 00 00
……………
00 00 00 00
f8 36 68 55        → somewhere in middle of the stack.
```

3. Conclusion

Through these 5 phases, I learned what the buffer overflow attack is and got deeper understand on stack frame and IA-32 function calls.
Honestly, the hardest part was watching monitor for hours which was full of white characters on black terninal. Phase 5 Nitro took most of my time. At the very first time I try to solve this phase, I did not know that this phase will execute testn function 5 times. Also I did not know that there was -n option to hex2raw program. Because I started this lab on the first day and there was no 'BufferLab Hand out.pdf' on eTL. It was uploaded like 1 day after. While solving nitro, I used MS Excel to draw whole stack frame from the main function to getbuf to track all registers and memories.

After I finished this lab, I find more information about buffer overflow attack, I realized why 'MS Visual Studio' forces user to use safe library function such as scanf_s.

CSAP GitLab Repository
https://git.csap.snu.ac.kr/2014-15395/buflab.git
https://git.csap.snu.ac.kr/2014-15395/SysProg.git