Attack Lab Report

Chengyuan Zhang, 2017011429

1 Target

Students will learn different ways that attackers can exploit security vulnerablities, get a better understanding of how to write a program that is more secure, and get a deeper understanding of the stack and the x86-64 instructions.

2 Experimental Procedure & Principle

Phase 1

Task

Get CTARGET to execute the code for touch1 when getbuf executes its return statement, rather than returning to test.

Idea

As shown in Figure 1, when <getbuf> returns, *rsp* will point at return address. So if a byte representation of the starting address of <touch1> positions here, the ret instruction at the end of code for <getbuf> will transfer control to <touch1>.

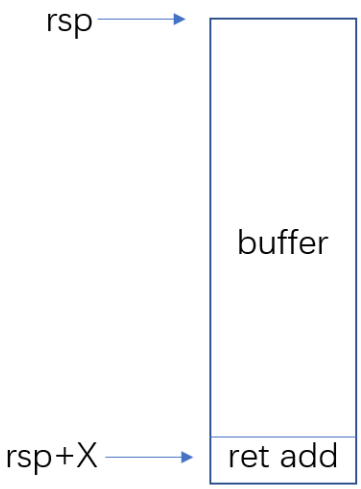

Figure 1-1

Method

First I checked the size of buffer so that I can know the length of the sequence I need to fill in. The instruction in Figure 1-2 shows that the buffersize is 0x18=24 bytes.

```
00000000004017db <getbuf>:
  4017db:       48 83 ec 18             sub    $0x18,%rsp
  4017df:       48 89 e7                mov    %rsp,%rdi
  4017e2:       e8 7e 02 00 00          callq  401a65 <Gets>
  4017e7:       b8 01 00 00 00          mov    $0x1,%eax
  4017ec:       48 83 c4 18             add    $0x18,%rsp
  4017f0:       c3                      retq

00000000004017f1 <touch1>:
  4017f1:       48 83 ec 08             sub    $0x8,%rsp
  4017f5:       c7 05 1d 2d 20 00 01    movl   $0x1,0x202d1d(%rip)        # 60451c <vlevel
  4017fc:       00 00 00
  4017ff:       bf 25 31 40 00          mov    $0x403125,%edi
  401804:       e8 c7 f4 ff ff          callq  400cd0 <puts@plt>
  401809:       bf 01 00 00 00          mov    $0x1,%edi
  40180e:       e8 97 04 00 00          callq  401caa <validate>
  401813:       bf 00 00 00 00          mov    $0x0,%edi
  401818:       e8 33 f6 ff ff          callq  400e50 <exit@plt>
```

Figure 1-2

Then I got the starting address of <touch1>, which is 0x4017f1. So I used 24 "00" to fill in the buffer, and replaced the returning address with the starting address of <touch1>. And I got passed!

Phase 2

Task

Get CTARGET to execute the code for <touch2> rather than returning to test, and pass the cookie as <touch2>'s argument.

Idea

Function <touch2> needs a specific argument, so in order to pass this level some code must be injected. Because the first argument to a function is always passed in register *%rdi*, the instruction **movq cookie,%rdi** is necessary. And *rsp* needs to point at <touch2>'s starting address, so **pushq touch2_add** is also essential. These instructions can be injected in the buffer, so as long as we know the starting address of the buffer, we can let *%rsp* jump to the suitable position and execute the code we want.

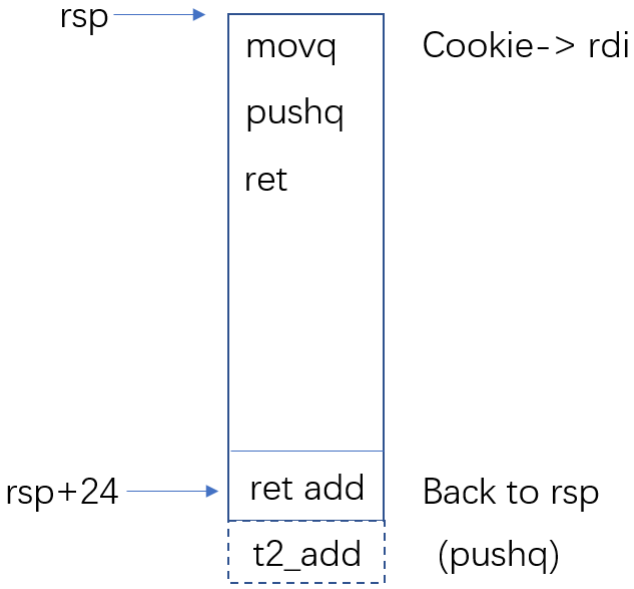

Figure 2-1

Method

First I translated the assembly instructions mentioned above into machine code. I wrote these instructions in 2.s and compile it to generate 2.o, then I used objdump to check 2.o and get the code.

```
2017011429@dell07:~$ objdump -d 2.o

2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 2a e9 61 7d    mov    $0x7d61e92a,%rdi
   7:   68 1d 18 40 00          pushq  $0x40181d
   c:   c3                      retq
```

Figure 2-2

Function <getbuf> calls function <Gets> to create buffer, so I set a breakpoint after callq < Gets > so that I can print *$rsp* and get its address, which is 0x5565a038.

```
(gdb) r -q
Starting program: /home/2017011429/ctarget -q
Cookie: 0x7d61e92a
Type string:000000000

Breakpoint 1, getbuf () at buf.c:16
16      buf.c: Permission denied.
(gdb) disass
Dump of assembler code for function getbuf:
   0x00000000004017db <+0>:     sub    $0x18,%rsp
   0x00000000004017df <+4>:     mov    %rsp,%rdi
   0x00000000004017e2 <+7>:     callq  0x401a65 <Gets>
=> 0x00000000004017e7 <+12>:    mov    $0x1,%eax
   0x00000000004017ec <+17>:    add    $0x18,%rsp
   0x00000000004017f0 <+21>:    retq
End of assembler dump.
(gdb) print $rsp
$1 = (void *) 0x5565a038
(gdb)
```

Figure 2-3

So finally the string includes 3 parts: the machine code I want to inject, “00” fulfilling the buffer, and the starting address of the buffer.

Phase 3

Task

Get CTARGET to execute the code for <touch3> rather than returning to test, and pass a special string as <touch3>’s argument.

Idea

The idea is basically the same with phase 2. The biggest difference is that the cookie is an instance number so that we can operate it directly; while we cannot move a string into *rdi*. As a string is represented in C as a sequence of bytes, the sequence must be placed somewhere in the stack and the injected code must set register *%rdi* to the address of the string. However, <touch3> calls function <hexmatch> and <strncmp>, which will push data onto the stack and overwrite portions of memory that held the buffer. So the string must be placed outside the buffer.

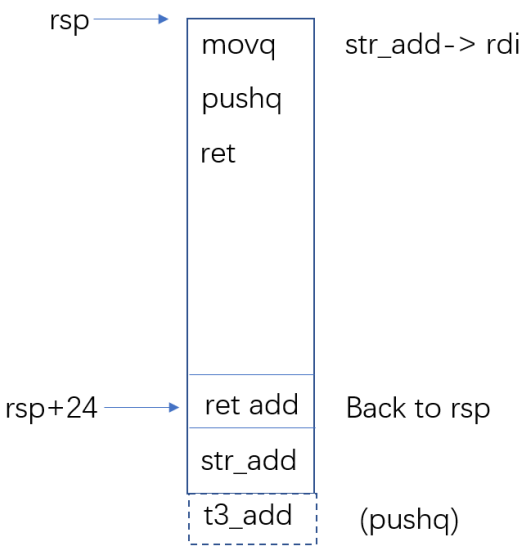

Figure 3-1

Method

First I calculated the address of str_add, which was

$$\text{rsp} + 32 = 0\text{x}5565\text{a}038 + 32 = 0\text{x}5565\text{a}058$$

Then I translated assembly instructions into machine code and got this:

```
3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 58 a0 65 55    mov    $0x5565a058,%rdi
   7:   68 2e 19 40 00          pushq  $0x40192e
   c:   c3                      retq
2017011429@dell07:~$
```

Figure 3-2

So the string contains 4 parts: the machine code I want to inject, “00” fulfilling the buffer, the starting address of the buffer (38 a0 56 55 00 00 00 00), and the byte sequence representation of <touch3>’s argument.

Phase 4

Task

Repeat the attack of phase 2 on program RTARGET using gadgets from the gadget farm.

Idea

This time the stack position is random, so it’s impossible to get address of the stack. However some gadgets are provided in this phase and it’s possible to build a chain of gadget execution to achieve the goal. For example, if the cookie is at the top of the stack, then it’s possible to find instructions $\text{popq}\ \%\text{rax}$ and $\text{movq}\ \%\text{rax},\%\text{rdi}$. So the chain of gadget execution is showed below:

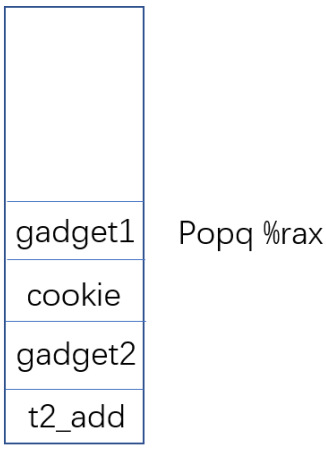

Figure 4-1

It's clear that no specific address is used during the process.

Method

Things to do are very simple: check the assembly code of rtarget and find necessary code.

popq %rax → 58; ret→ c3; movq %rax, %rdi → 48 89 c7.

```
00000000004019e0 <getval_345>:
  4019e0:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  4019e5:       c3                      retq

00000000004019e6 <addval_324>:
  4019e6:       8d 87 fa 95 58 c3       lea    -0x3ca76a06(%rdi),%eax
  4019ec:       c3                      retq

00000000004019ed <setval_453>:
```

Figure 4-2

So the address should be 0x4019e1 and 0x4019ea. And the struct of the string is clear now: 24 "00" to fulfill the stack, address of gadget1, cookie, address of gadget2 and starting address of function touch2.

Phase 5

Task

Repeat the attack of phase 3 on program RTARGET using gadgets from the gadget farm.

Idea

First the string must be stored somewhere in the stack. Because I can't get specific information about the string's address, I have to know the relationship between the positions of the buffer and the string so that I could use $\text{lea}(\%\text{rdi}, \%\text{rsi}, 1), \%\text{rax}$ to visit the address, and accordingly the last step should be $\text{mov}\ \%\text{rax}, \%\text{rdi}$. Then I need to find a chain of gadgets to connect *%rsp* and *%rdi*. Here's my solution:

$$\text{mov}\ \%\text{rsp}, \%\text{rax} \rightarrow \text{mov}\ \%\text{rax}, \%\text{rdi}$$

And *%rsi* should store the distance between *%rsp* and the string. But I don't have $\text{popq}\ \%\text{rsi}$, only $\text{popq}\ \%\text{rax}$ can be used. So I need to build another chain of gadgets to connect *%rsi* and *%rax*. Here's my chain:

$$\text{mov}\ \%\text{eax}, \%\text{ecx} \rightarrow \text{mov}\ \%\text{ecx}, \%\text{edx} \rightarrow \text{mov}\ \%\text{edx}, \%\text{esi}$$

Now I have the whole structure of the stack.

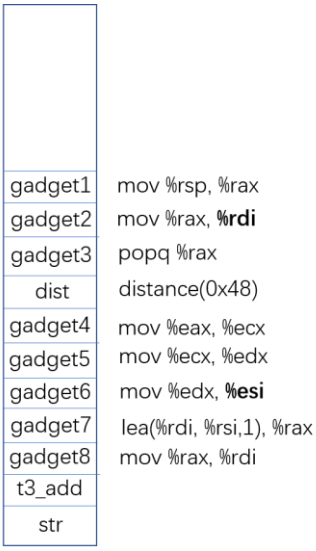

Figure 5-1

There are 9 instructions between str and gadget2, so the distance is 9*8=0x48 bytes.

Method

First I checked the assembly code of rtarget and found necessary code.

mov %rsp, %rax→**48 89 e0**; mov %rax, %rdi→**48 89 c7**;

popq %rax→**58**; mov %eax, %ecx →**89 c1**; mov %ecx, %edx →**89 ca**;

mov %edx, %esi→**89 d6;** lea(%rdi, %rsi, 1), %rax→**<add_xy>**.

Then I wrote every address of these instructions into the exploit string, with 0x48, <touch3>'s starting address and the argument. That was enough to pass the final phase.

3 Difficulties & What I Learned

The major challenge is phase3—if phase 3 get passed then phase 5 is not that difficult. At first I was very confused because I need to put the string outside the stack, and I worried about that the string might cover some important data and cause errors. So I used gdb to check the memory usage[i] before and after calling <hexmatch> and made sure that the place where I put the string was not occupied.

When it's hard to make it clear that where the string should be placed, I just draw a stack to simulate what is happening, and this trick really helped me a lot.

After this experiment I had a better understanding of how x86-64 instructions are encoded, and I gained more experience with GDB and OBJDUMP.

4 Acknowledgement

Thank the teaching assistant for his experimental guidance!

[i] https://blog.csdn.net/small_prince_/article/details/80682060