Pwn专题 - 1

2024.07.11

Esifiel / 颜尔汛

Outline

- 前置知识
  - 二进制基础与工具使用
  - 常见保护技术与checksec
  - libc，ld获取与patchelf使用
- ROP
  - ROPgadget与one_gadget
  - ret2plt与ret2libc
  - stack pivot
- 格式化字符串漏洞（fsb）
  - printf与格式化字符串原理
  - 利用fsb泄露数据与修改内存
  - 栈上的fsb利用
  - 非栈上的fsb利用

Part 0 - 前置知识

x86汇编（64位下）

- 寄存器：rax，rbx，rcx，rdx，rsi，rdi，r8，r9，r10，r11，r12，r13，r14，r15，rsp，rbp，rip
- 指令：
  - 赋值：mov
  - 运算：add，sub，mul，div，and，or，xor，shl，shr，sar
  - 比较：cmp
  - 栈操作：push，pop
  - 跳转：jmp，j[e/ne/z/nz/l/le/g/ge/b/be/a/ae]
  - 函数调用与返回：call，ret
  - 系统调用：syscall
  - 空指令：nop
- 函数传参顺序：rdi，rsi，rdx，rcx，r8，r9，栈
- 函数返回值：rax

gdb命令

- 运行：r，c
- 单步调试：s，n，si，ni
- 断点：b <func name>，b *<addr>，bp <addr>，pie b <offset>，bp $rebase(<offset>)，brva <offset>
  - b main：在main函数起始地址下断点
  - 在指定地址下断点：b *0x555555555000，bp 0x555555555000 (pwndbg)
  - 在内存偏移0x1000处下断点：pie b 0x1000 (gef)，bp $rebase(0x1000) (pwndbg)，brva 0x1000 (pwndbg)
- 查看值：p(rint)/[d/x]
  - p/x $rax：以十六进制格式打印寄存器rax的值
  - p/d 1+2+3+4：以十进制格式打印表达式的结果
- 查看内存：x/<count>[b/w/g/s] <addr>，tele(scope) <addr>
  - x/20gx：以8字节为单位打印20个带有0填充的十六进制值
  - x/s：查看字符串
- 程序状态：i(nfo)，vmmap，ctx

IDA快捷键

- F5：反编译
- n：为变量、函数重命名
- y：设置变量、函数的类型
- v：一键将函数设置为无参数无返回值：void f(void)
- x / Ctrl + x：查找符号的交叉引用
- Ctrl + s：跳转到指定段
- Shift + F12：字符串窗口
- Tab：切换汇编与反编译窗口
- Space：切换完整二进制信息和控制流图窗口
- g：快速跳转到指定地址
- Alt + t：文本搜索
- Alt + b：二进制搜索

GOT与PLT

- Global Offset Table：全局偏移表
- Procedure Linkage Table：过程链接表

```
  0x555555555070 <malloc@plt+0>    endbr64
→ 0x555555555074 <malloc@plt+4>    bnd    jmp QWORD PTR [rip+0x2f55]        # 0x555555557fd0 <malloc@got.plt>
```

```
.got:0000000000003FB0                         ; ===========================================================================
.got:0000000000003FB0
.got:0000000000003FB0                         ; Segment type: Pure data
.got:0000000000003FB0                         ; Segment permissions: Read/Write
.got:0000000000003FB0                         _got            segment qword public 'DATA' use64
.got:0000000000003FB0                                         assume cs:_got
.got:0000000000003FB0                                         ;org 3FB0h
.got:0000000000003FB0 C0 3D 00 00 00 00 00 00 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got:0000000000003FB8 00 00 00 00 00 00 00 00 qword_3FB8      dq 0                    ; DATA XREF: sub_1020↑r
.got:0000000000003FC0 00 00 00 00 00 00 00 00 qword_3FC0      dq 0                    ; DATA XREF: sub_1020+6↑r
.got:0000000000003FC8 28 40 00 00 00 00 00 00 free_ptr        dq offset free          ; DATA XREF: _free+4↑r
.got:0000000000003FD0 38 40 00 00 00 00 00 00 malloc_ptr      dq offset malloc        ; DATA XREF: _malloc+4↑r
.got:0000000000003FD8 30 40 00 00 00 00 00 00 __libc_start_main_ptr dq offset __libc_start_main
.got:0000000000003FD8                                                                 ; DATA XREF: _start+1F↑r
.got:0000000000003FE0 48 40 00 00 00 00 00 00 _ITM_deregisterTMCloneTable_ptr dq offset _ITM_deregisterTMCloneTable
.got:0000000000003FE0                                                                 ; DATA XREF: deregister_tm_clones+13↑r
.got:0000000000003FE8 50 40 00 00 00 00 00 00 __gmon_start___ptr dq offset __gmon_start__
.got:0000000000003FE8                                                                 ; DATA XREF: _init_proc+8↑r
.got:0000000000003FF0 58 40 00 00 00 00 00 00 _ITM_registerTMCloneTable_ptr dq offset _ITM_registerTMCloneTable
.got:0000000000003FF0                                                                 ; DATA XREF: register_tm_clones+24↑r
.got:0000000000003FF8 40 40 00 00 00 00 00 00 __cxa_finalize_ptr dq offset __imp___cxa_finalize
.got:0000000000003FF8                                                                 ; DATA XREF: __cxa_finalize+4↑r
.got:0000000000003FF8                                                                 ; __do_global_dtors_aux+E↑r
.got:0000000000003FF8                         _got            ends
```

pwntools使用 - 配置

context:

- log_level: info, debug
- arch: i386, amd64, arm, aarch64...
- bits: 32, 64
- os: linux, windows
- encoding

```
context(arch = “amd64”, log_level = “debug”, encoding = “latin1”)
```

pwntools使用 – ELF文件加载与使用

- program = ELF(“./hello”), libc = ELF(“./libc.so.6”)
- libc.address = 0x7ffffff0000
- program.sym[‘main’], libc.sym[‘__free_hook’]
- program.got[‘puts’]
- program.plt[‘read’]

pwntools使用 - 连接

- p = process(“./hello”)
- p = remote(“127.0.0.1”, 8888)
- p = gdb.debug(“./hello”, gdbscript = “c\n”) (需要安装gdbserver)
- p = process(“./hello”) ... gdb.attach(p)：将调试器挂到正在运行的p上
- p.interactive()

pwntools使用 - 交互

- p.send(“hello”)
- p.sendline(“hello”)
- p.sendafter(“$ ”, “hello”)
- p.sendlineafter(“$ ”, “hello”)
- p.recv(1024)
- p.recvn(4)
- p.recvline()
- p.recvuntil(“Input your name: ”)

pwntools使用 - 工具函数

- [p/u][8/16/32/64]
  - p64(0xdeadbeefcafebabe)
  - u64(b“\x11\x22\x33\x44\x55\x66\x77\x88”)
- flat([1, 2, 0xdeadbeef, b“aaaaaaaa”, libc.sym[‘puts’]])

常见保护技术 – NX / DEP

- No-eXecute / Data Execution Prevention：数据段（heap，stack等）不可执行
- 在开启NX保护的情况下，ret2sc失效
- 如果不加任何参数，gcc编译出的ELF默认保护全开，需要添加-zexecstack参数才能使得栈具有可执行权限

常见保护技术 – Canary

- 在函数开始时插入栈底的随机值，用于防止栈溢出覆盖rbp和返回地址
- 如果想关闭canary保护，编译时需要添加-fno-stack-protector参数

```
.text:0000000000001189 F3 0F 1E FA                      endbr64
.text:000000000000118D 55                               push    rbp
.text:000000000000118E 48 89 E5                         mov     rbp, rsp
.text:0000000000001191 48 81 EC 10 01 00 00             sub     rsp, 110h
.text:0000000000001198 64 48 8B 04 25 28 00 00+         mov     rax, fs:28h
.text:0000000000001198 00
.text:00000000000011A1 48 89 45 F8                      mov     [rbp+var_8], rax
.text:00000000000011A5 31 C0                            xor     eax, eax
.text:00000000000011A7 48 8D 85 F0 FE FF FF             lea     rax, [rbp+var_110]
.text:00000000000011AE 48 89 C6                         mov     rsi, rax
.text:00000000000011B1 48 8D 05 4C 0E 00 00             lea     rax, format     ; "buffer address: %p\n"
.text:00000000000011B8 48 89 C7                         mov     rdi, rax        ; format
.text:00000000000011BB B8 00 00 00 00                   mov     eax, 0
.text:00000000000011C0 E8 BB FE FF FF                   call    _printf
.text:00000000000011C5 48 8D 85 F0 FE FF FF             lea     rax, [rbp+var_110]
.text:00000000000011CC 48 89 C6                         mov     rsi, rax
.text:00000000000011CF 48 8D 05 42 0E 00 00             lea     rax, aS         ; "%s"
.text:00000000000011D6 48 89 C7                         mov     rdi, rax
.text:00000000000011D9 B8 00 00 00 00                   mov     eax, 0
.text:00000000000011DE E8 AD FE FF FF                   call    ___isoc99_scanf
.text:00000000000011E3 B8 00 00 00 00                   mov     eax, 0
.text:00000000000011E8 48 8B 55 F8                      mov     rdx, [rbp+var_8]
.text:00000000000011EC 64 48 2B 14 25 28 00 00+         sub     rdx, fs:28h
.text:00000000000011EC 00
.text:00000000000011F5 74 05                            jz      short locret_11FC
.text:00000000000011F7 E8 74 FE FF FF                   call    ___stack_chk_fail
.text:00000000000011FC                  ; ---------------------------------------------------------------------------
```

常见保护技术 - Canary

- 特征：最低字节为0
- 一般需要通过其他方法泄露canary的值，才能继续进行栈溢出的利用

```
→  Desktop ./example4
buffer address: 0x7ffcd5623a80
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
[1]    5140 IOT instruction (core dumped)  ./example4
```

常见保护技术 – PIE，ASLR

- Position Independent Executable 地址无关可执行文件：程序每次加载时的基址将发生变化
- Address Space Layout Randomization 地址空间随机化：系统级的地址随机化，程序每次加载时所有内存段基址将发生变化，包括程序段、共享库、heap、stack等
- gcc关闭pie保护：-no-pie
- 查看系统是否启用aslr：/proc/sys/kernel/randomize_va_space
- gdb开关aslr：aslr on/off

常见保护技术 – FULL RELRO

- RELocation Read-Only 重定位表只读：防止针对GOT表覆写的攻击
- Partial RELRO：GOT表可写
  - 通过参数-z lazy开启，即延迟绑定，直到程序首次调用某个符号时才做解析
- FULL RELRO：GOT表不可写
  - 通过指定-z now开启（默认），即立即绑定，在程序被载入内存后立即解析完所有符号

checksec

- pwntools附带的命令行工具，用于检查程序开启的保护

```
➜  Desktop checksec ./example4
[*] '/home/ctfer/Desktop/example4'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

```
➜  Desktop checksec ./example4
[*] '/home/ctfer/Desktop/example4'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

patchelf

patchelf：用于修改程序的动态链接库和链接器路径，适配题目给的libc和ld

- 设置ld：--set-interpreter <ld_path>
- 设置其他libc搜索路径： --set-rpath <dir_name>

示例：ld位于当前目录下，名字为ld-linux-x86-64.so.2；libc也位于当前目录下且文件名为libc.so.6

- patchelf --set-interpreter ./ld-linux-x86-64.so.2 --set-rpath . ./hello
- 修改前：

```
➜ Desktop ldd ./hello
        linux-vdso.so.1 (0x00007fff14f8a000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe0e3000000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe0e341b000)
```

- 修改后：

```
➜ Desktop patchelf --set-interpreter ./ld-linux-x86-64.so.2 --set-rpath . ./hello
➜ Desktop ldd ./hello
        linux-vdso.so.1 (0x00007ffe6c57f000)
        libc.so.6 => ./libc.so.6 (0x00007f904fe00000)
        ./ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007f905020a000)
```

glibc-all-in-one

- 项目地址：https://github.com/matrix1001/glibc-all-in-one
- 快速下载指定版本的glibc库文件和调试符号，包括libc、ld等等
- update_list：用于更新list和old_list
- download <id_in_list>：下载LTS版本系统上对应的libc
  - 2.23, 2.27, 2.31, 2.35...
- download_old <id_in_old_list>：下载非LTS版本系统上对应的libc
  - 2.24, 2.26, 2.28, 2.29, 2.30, 2.32, 2.34, 2.36...

libc版本搜索

- https://libc.rip/

Search

Symbol name: __libc_start_main_ret
Address: 083
REMOVE

Symbol name
Address
REMOVE

FIND

Results

libc6_2.31-0ubuntu9.10_amd64

Download	Click to download
All Symbols	Click to download
BuildID	965ff93b372ec6e456142d04b7d3795aefdcf0c5
MD5	bf729448dee0966904d3bff97467fdbe
__libc_start_main_ret	0x24083
dup2	0x10e8c0
printf	0x61c90
puts	0x84420
read	0x10dfc0
str_bin_sh	0x1b45bd
system	0x52290
write	0x10e060

libc6_2.31-0ubuntu9.9_amd64
libc6_2.31-0ubuntu9.4_amd64
libc6_2.31-0ubuntu9.8_amd64
libc6_2.31-0ubuntu9.5_amd64
libc6_2.31-0ubuntu9.11_amd64

Part 1 - ROP

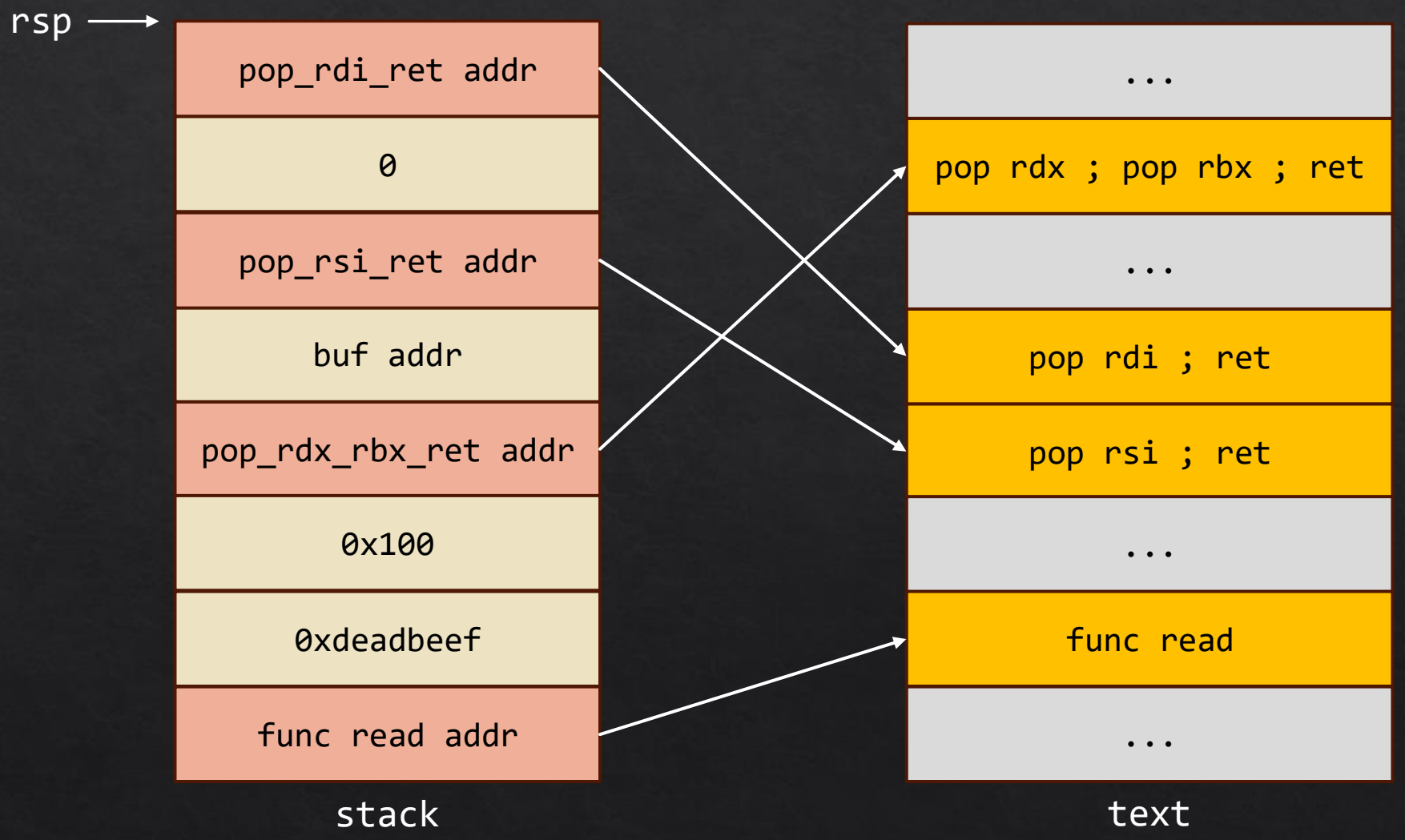
ROP (Return-Oriented Programming)

在栈缓冲区溢出的基础上，利用程序中已有的小片段（gadgets）来改变某些寄存器或者变量的值，从而控制程序的执行流程。所谓 gadgets 就是以 ret 结尾的指令序列，通过这些指令序列，我们可以修改某些地址的内容，方便控制程序的执行流程。

-- 引自CTF Wiki

ROP (Return-Oriented Programming)

通过ROP执行read(0, buf, 0x100)



ROPgadget

- 安装：pip install ROPgadget
- 用于寻找二进制程序中的gadget
- ROPgadget --binary <filename>
- 高版本gcc编译出的ELF没有诸如 pop rdi ; ret 这样好用的gadget，基本只存在于libc中（因为__lib_csu_init函数被去除了）
- 同类工具：ropper、rp++等

```
→ Desktop ROPgadget --binary ./hello
Gadgets information
============================================================
0x000000000000113b : add byte ptr [rax], 0 ; add byte ptr [rax], al ; endbr64 ; jmp 0x10c0
0x00000000000010b3 : add byte ptr [rax], 0 ; add byte ptr [rax], al ; ret
0x000000000000113c : add byte ptr [rax], al ; add byte ptr [rax], al ; endbr64 ; jmp 0x10c0
0x0000000000001161 : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rbp ; ret
0x00000000000010b4 : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x0000000000001036 : add byte ptr [rax], al ; add dl, dh ; jmp 0x1020
0x0000000000001130 : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [
```

one_gadget

- 安装：sudo apt install ruby; sudo gem install one_gadget
- 像后门一样，跳过去就能获得shell的一个地址
- 一般存在于libc中，且需要满足各种寄存器的约束条件才能成功
- 随着libc版本的升级，one gadget的条件越来越苛刻

```
➜ Desktop one_gadget ./libc.so.6
0x4e1c9 posix_spawn(rsp+0xc, "/bin/sh", 0, rbx, rsp+0x50, environ)
constraints:
  address rsp+0x60 is writable
  rsp & 0xf == 0
  rax == NULL || {"sh", rax, r12, NULL} is a valid argv
  rbx == NULL || (u16)[rbx] == NULL

0x4e1d0 posix_spawn(rsp+0xc, "/bin/sh", 0, rbx, rsp+0x50, environ)
constraints:
  address rsp+0x60 is writable
  rsp & 0xf == 0
  rcx == NULL || {rcx, rax, r12, NULL} is a valid argv
  rbx == NULL || (u16)[rbx] == NULL

0xe4159 execve("/bin/sh", rbp-0x50, r13)
constraints:
  address rbp-0x48 is writable
  r12 == NULL || {"/bin/sh", r12, NULL} is a valid argv
  [r13] == NULL || r13 == NULL || r13 is a valid envp

0xe41b3 execve("/bin/sh", rbp-0x50, r13)
constraints:
  address rbp-0x50 is writable
  rax == NULL || {"/bin/sh", rax, NULL} is a valid argv
  [r13] == NULL || r13 == NULL || r13 is a valid envp
```

ret2plt

ret2plt.c

- 想要通过ROP执行某些函数如system、puts等，且程序本身恰好用到了该函数
- 无需泄露libc地址，通过跳转plt间接执行库函数

ret2libc

ret2libc.c

- 想get shell但是程序本身没有用到system函数?
- system函数和“/bin/sh”字符串总会存在于libc中
- 通用的ROP思路

CTF pwn的ROP基本思路

ropbasic.c

- 泄露canary
- 泄露libc，计算libc基址，进一步计算出各个所需的函数或gadget的地址
- ret2libc

stack pivot – 覆盖old rbp实现栈迁移

- 如果溢出长度只够刚好覆盖掉rbp和返回地址?
- leave ; ret == mov rsp, rbp ; pop rbp ; ret

```
.text:00000000000011D1 C9                                   leave
.text:00000000000011D2 C3                                   retn
.text:00000000000011D2                              ; } // starts at 1169
.text:00000000000011D2                              main    endp
```

- 覆盖old rbp, 连续执行两次 leave ; ret 就能修改rsp的值
- 如果fake rbp是一段攻击者可控的内存空间, 就可以执行更长的ROP链

Part 2 - 格式化字符串漏洞

printf是如何工作的

printf的函数声明：`int printf(const char *format, ...);`

printf的参数个数是可变的（通过va_list实现）：

- `printf("Hello!\n");`
- `printf("Hello %s!\n", name);`
- `printf("There are %d lights!", 5);`
- `printf("The average of %d, %d, and %d is %f", 1, 2, 3, (float)(1+2+3)/3.0);`

正常的参数由两部分组成：

- 格式化字符串（format string）
- 理论上应该与格式化字符串对应的参数

其他格式化字符串函数

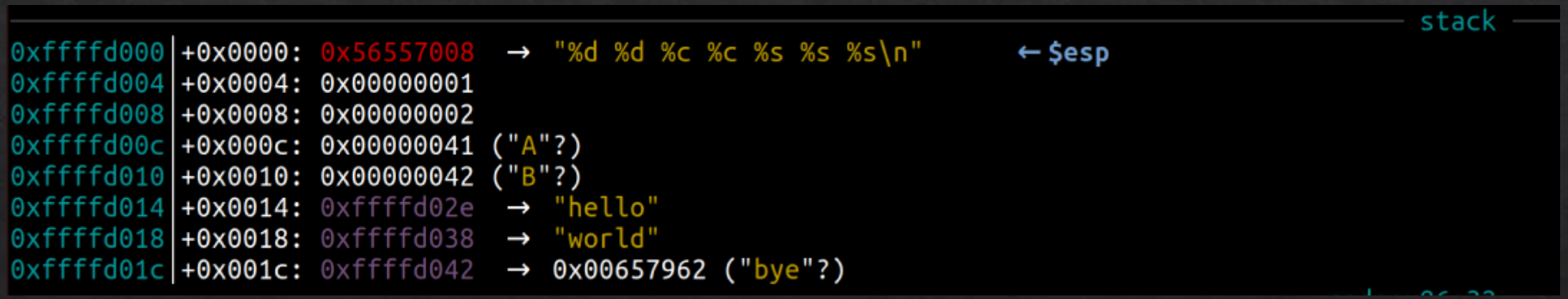
- sprintf, snprintf, fprintf...
- scanf, sscanf...

man 3 printf:

```
SYNOPSIS
       #include <stdio.h>

       int printf(const char *restrict format, ...);
       int fprintf(FILE *restrict stream,
                   const char *restrict format, ...);
       int dprintf(int fd,
                   const char *restrict format, ...);
       int sprintf(char *restrict str,
                   const char *restrict format, ...);
       int snprintf(char *restrict str, size_t size,
                   const char *restrict format, ...);

       #include <stdarg.h>

       int vprintf(const char *restrict format, va_list ap);
       int vfprintf(FILE *restrict stream,
                   const char *restrict format, va_list ap);
       int vdprintf(int fd,
                   const char *restrict format, va_list ap);
       int vsprintf(char *restrict str,
                   const char *restrict format, va_list ap);
       int vsnprintf(char *restrict str, size_t size,
                   const char *restrict format, va_list ap);
```

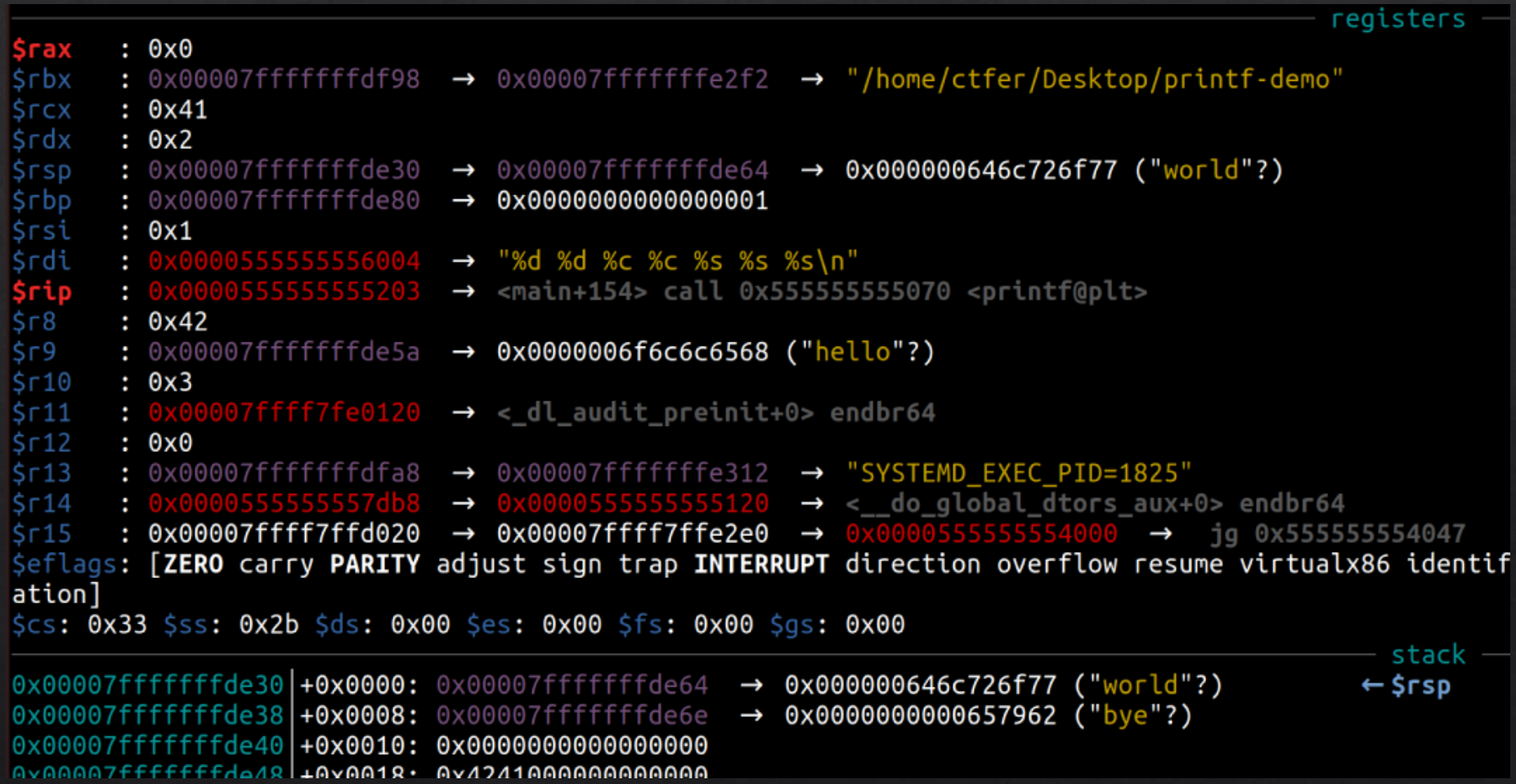
printf的参数准备（32位）

- 32位下的传参顺序：从右到左依次入栈



printf的参数准备（64位）

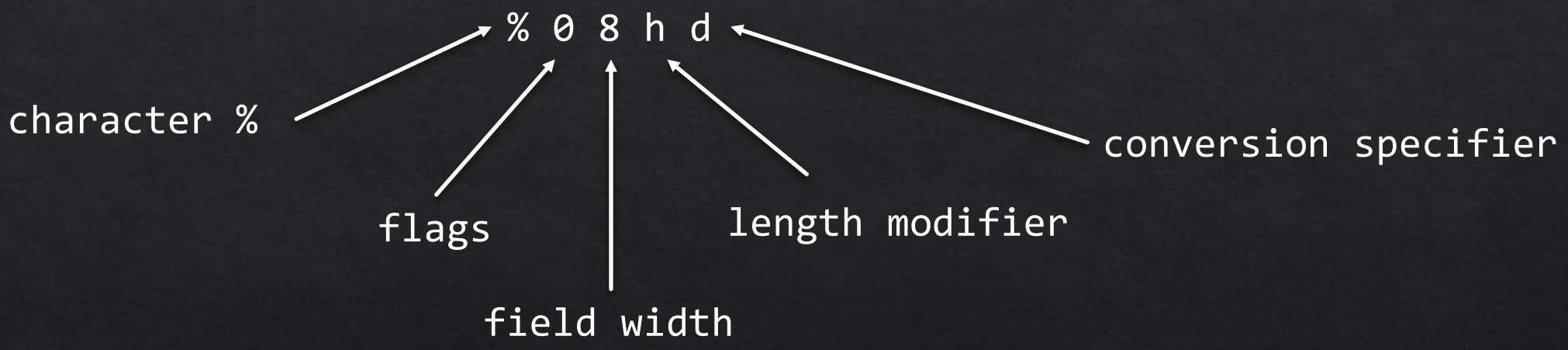
- 64位下的传参顺序：优先寄存器传参（rdi，rsi，rdx，rcx，r8，r9），剩下的通过栈传参



格式化控制符的构成

man 3 printf – Format of the format string

- %[$][flags][width][.precision][length modifier]conversion



常见的conversion specifier

- d，i：十进制
- o，u，x，X：无符号八进制，无符号十进制，无符号十六进制
- e：科学计数法
- f，F：单精度浮点数
- c：字符
- s：字符串
- p：指针（按照%#x或%#lx输出）

有用的其他字段

- %12345678c：输出时填充空格，将输出共12345678个字符
- %hhd：宽度限制为1字节（length modifier）
  - %hd：2字节
  - %d：4字节
  - %ld：8字节
- %7$p：$可以指定参数位置，将第7个参数作为地址输出
- %n: stores the number of characters written so far into the pointer argument

printf是如何工作的

printf依靠第一个参数格式化字符串（format string）确定参数的个数

如果不给参数会怎么样?

- printf("Hello %s!\n");
- printf("There are %d lights!");
- printf("The average of %d, %d, and %d, is %f");

如果格式化字符串可以由用户指定会怎么样?

- printf(user_input); // format string bug

fsb漏洞利用

fsb能够实现任意地址读写：

- 泄露栈上的敏感信息、栈地址、堆地址、程序段地址、libc地址等等（读）
- 劫持和控制流相关的对象，如返回地址、GOT表项等等（写）

根据格式化字符串位置的不同，利用思路也有所不同

- 栈上的fsb：用户在输入格式化字符串时，还可以在栈上布置任意值作为参数
- 非栈上的fsb：用户在输入格式化字符串时，无法在栈上伪造参数，需要借助栈上原本的数据完成利用

栈上的fsb – Leak

1. 泄露栈上原有的数据
    - %p：将数据打印为带前导0x的十六进制
    - $：指定参数位置。需要计算偏移，确认要打印第几个参数
    - 栈上存在的数据包括栈地址、程序段地址、libc地址，也可能会有堆地址
2. 泄露非栈上的内容
    - %s：解析时会将参数作为地址，访问它指向的内存中的字符串
    - 布置参数，实现任意读

利用fsb修改内存 - %n控制符

- %n：将当前已打印的字节数写入参数指针指向的内存
  - %hhn：写1字节
  - %hn：写2字节
  - %n：写4字节
  - %ln：写8字节
- 设计师认为的正确用法：

```
char *name = "admin";
int namelength = 0;
printf("%s%n\n", name, &namelength);
printf("The name was %d bytes long!\n", namelength);
```

栈上的fsb – Hijack Control Flow

- 我们可以在栈上布置任意地址，通过%s泄露它指向的数据
  - 那么同样也可以通过%n往这个地址指向的内存中写入数据
- %12345678c%7$n：利用宽度对齐来控制写入的值
- 通过fsb覆盖控制流相关对象：
  - 栈上的函数返回地址
  - GOT表
  - libc中的hook函数
  - 和程序逻辑本身有关的变量
  - ...

Tips - printf输出时间太长?

- 通过逐字节写入减少输出的空格数

```
char buf[5] = {0};
// printf("%1145258561c%1$n", buf); // output too long
printf("%65c%1$hhn%c%2$hhn%c%3$hhn%c%4$hhn\n", buf, buf + 1, buf + 2, buf + 3);
puts(buf);
```

- %*10$c%11$n: 将第10个参数作为padding size, 并将输出的字节数写入第11个参数指向的内存

Tips - 一次printf不够怎么办?

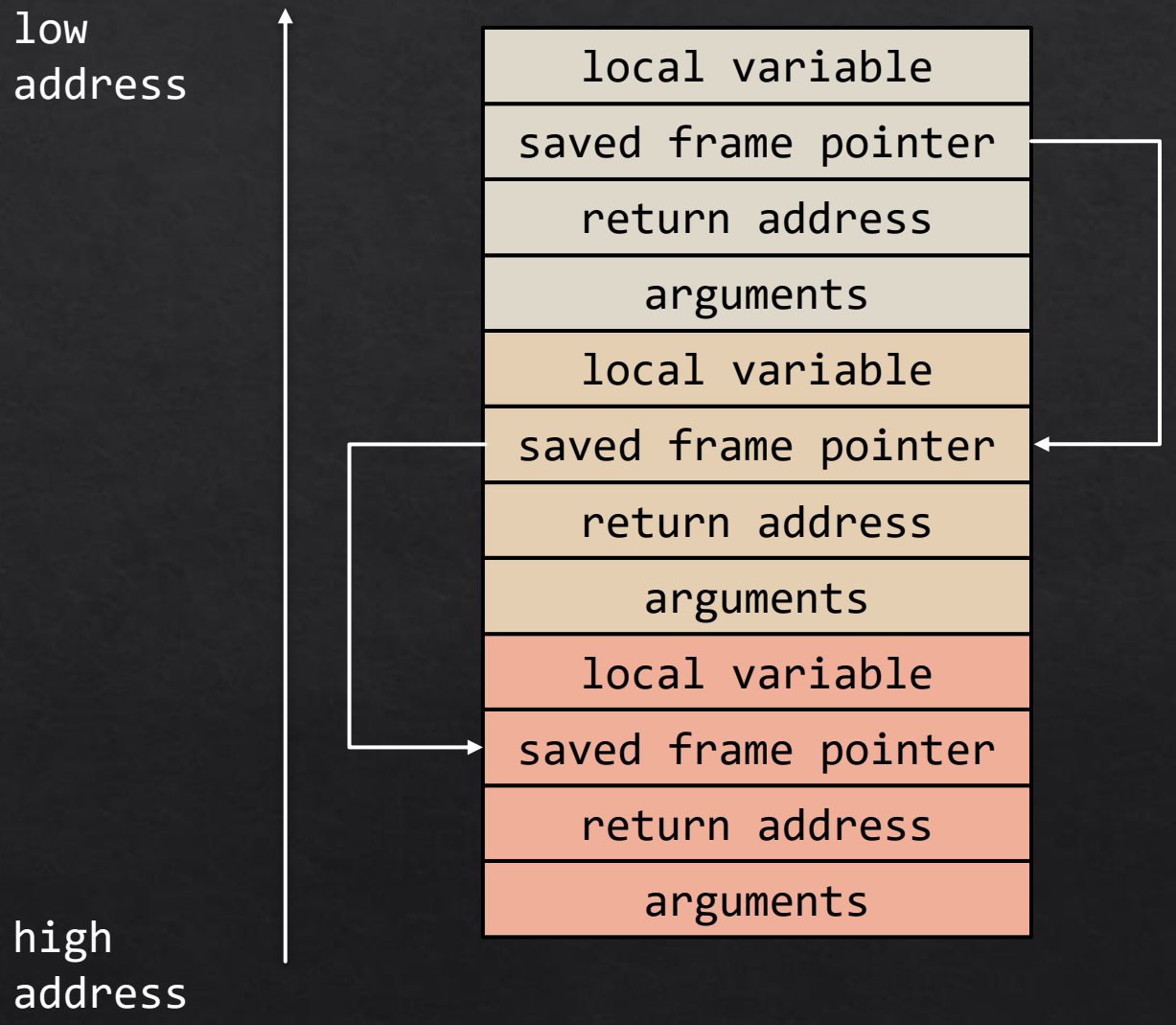
可以构造程序的“无限循环”：劫持控制流重新回到main函数或漏洞函数

- 覆盖exit等函数的GOT表（Partial Relro）
- 覆盖__fini_array中的指针（No Relro）
- 覆盖栈上的返回地址（需要有指向它的指针）
- ...

非栈上的fsb – 复用栈上指针

- 只能将栈上原有的数据作为参数进行利用
  - 如果只是想泄露栈上的数据，没问题
  - 如果栈上的某个地址正好是想要泄露或修改的对象的指针，也没问题
  - 格式化字符串是否在栈上的核心差异：无法直接构造出任意地址来匹配所布置的%s和%n
- 那么，如何构造任意地址读写？
  - 假设栈上有一个栈指针ptrA
  - 借助%n把ptrA指向的内存修改为另一个栈指针ptrB
  - 再借助%s或%n对ptrB进行读写

非栈上的fsb – 利用frame pointer chain




```
push rbp
mov rbp, rsp
...
push rbp
mov rbp, rsp
...
push rbp
mov rbp, rsp
```

非栈上的fsb – 利用frame pointer chain

fsb-global.c

通用思路：

- 找到栈上的rbp链：A -> B -> ?
- 利用fsb，通过A修改B的低字节，使B指向栈上的其他地方，例如main函数的返回地址
- 利用fsb，通过fake B修改其中的内容，布置ROP gadget
- 重复以上两步，布置出完整的ROP链

pwntools - fmtstr函数

- 用于构造fsb的payload
- 主要优点是能够自动对齐参数，不需要自己考虑需要在payload里填充多少个字符
- https://docs.pwntools.com/en/stable/fmtstr.html

Summary

- ROP
  - gadget
  - ret2plt
  - ret2libc
  - stack pivot
- fsb:
  - 格式化控制符写法
  - 利用栈上的fsb任意读
  - 利用栈上的fsb任意写
  - 利用非栈上的fsb