[翻译] glibc里的one gadget

原文链接

https://david942j.blogspot.com/2017/02/project-one-gadget-in-glibc.html

glibc里的one-gadget

介绍

one-gadget 是glibc里调用`execve('/bin/sh', NULL, NULL)`的一段非常有用的gadget。在我们能够控制ip（也就是pc）的时候，用one-gadget来做RCE（远程代码执行）非常方便，比如有时候我们能够做一个任意函数执行，gadget就可以搞定了。我之前每次都是用IDA去手动找的，哪怕我原来还找过，所以我就决定写个好用的工具来避免再手动去找。

最后做出来的工具是one_gadget，工具不仅可以找到one gadget还可以把需要满足的条件也给出来。

```
david942j in ~/one_gadget on master via 💎 v2.4.2
➜ one_gadget spec/data/libc-2.19-cf699a15caae64f50311fc4655b86dc39a479789.so
0x46428 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4647c execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xe5765 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xe66bd execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL

david942j in ~/one_gadget on master via 💎 v2.4.2
➜
```


这篇文章主要讲讲one_gadget都干了点什么。

Repository

one_gadget的代码可以在这里找到。

代码包装成了一个ruby gem，在命令行里用`gem install one_gadget`就可以安装。

One Gadget

首先，一个潜在的gadget需要满足以下几个条件：

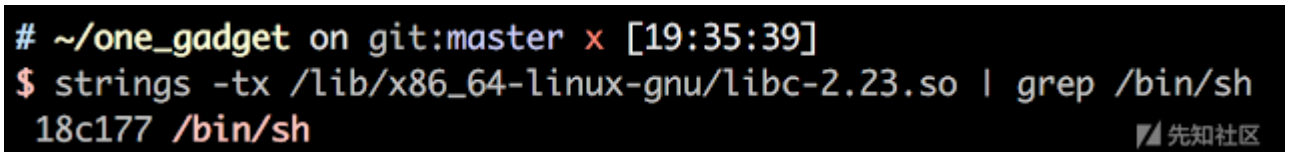
1. 能够访问到'/bin/sh'字符串
2. 调用了exec*系列的函数

为了说的更明白点，看看下面这一段汇编，这时libc-2.23用`objdump`出来的内容：

```
; glibc-2.23 (64bit, 16.04 ubuntu, BuildID: 60131540dadc6796cab33388349e6e4e68692053)
  4526a: mov    rax,QWORD PTR [rip+0x37dc47] # 3c2eb8 <_IO_file_jumps@@GLIBC_2.2.5+0x7d8>
  45271: lea    rdi,[rip+0x146eff]           # 18c177 <_libc_intl_domainname@@GLIBC_2.2.5+0x197>
  45278: lea    rsi,[rsp+0x30]
  45278: mov    DWORD PTR [rip+0x380219],0x0 # 3c54a0 <__abort_msg@@GLIBC_PRIVATE+0x8c0>
  45287: mov    DWORD PTR [rip+0x380213],0x0 # 3c54a4 <__abort_msg@@GLIBC_PRIVATE+0x8c4>
  45291: mov    rdx,QWORD PTR [rax]
  45294: call   cbbc0 <execve@@GLIBC_2.2.5>
```

第45271行相当于`rdi = libc_base + 0x18c177`，而`libc_base + 0x18c177`正好就是'/bin/sh'的字符串。

用strings很容易把字符串的偏移量拿出来：



至于这个gadget的约束，注意一下45278行的`rsi = rsp + 0x30`，从这就可以看出其实最后结果是调用的`execve('/bin/sh', rsp + 0x30, environ)`，这就需要`[rsp + 0x30] == NULL`。

Gadget 0x4526a:

```
execve('/bin/sh', rsp + 0x30, environ)
```

所以找gadget的策略并不麻烦：

1. 把所有访问到'/bin/sh'的汇编代码作为one gadget的备选
2. 把附近没有调用execve的备选都去了
3. 类似`lea rsi, [rsp+0x??]`的汇编就是约束条件

这个简单的策略在glibc-2.19和glibc-2.23能找到3个one gadget，如下：

```
; glibc-2.19(64bit, 14.04 ubuntu, BuildID: cf699a15caae64f50311fc4655b86dc39a479789)
0x4647c execve('/bin/sh', rsp+0x30, environ)
0xe5765 execve('/bin/sh', rsp+0x50, environ)
0xe66bd execve('/bin/sh', rsp+0x70, environ)

; glibc-2.23(64bit, 16.04 ubuntu, BuildID: 60131540dadc6796cab33388349e6e4e68692053)
0x4526a execve('/bin/sh', rsp+0x30, environ)
0xef6c4 execve('/bin/sh', rsp+0x50, environ)
0xf0567 execve('/bin/sh', rsp+0x70, environ)
```

由于这些gadget的约束只要求stack上的一些特定位置值为0，所以非常有用。

但是，在32位的libc上，这办法完全用不了。

下面我们来看下一个32位libc的潜在one gadget长啥样：

```
; glibc-2.23 (32bit, 16.04 ubuntu, BuildID: 926eb99d49cab2e5622af38ab07395f5b32035e9)
  3ac69: mov    eax,DWORD PTR [esi-0xb8]
  3ac6f: add    esp,0xc
  3ac72: mov    DWORD PTR [esi+0x1620],0x0
  3ac7c: mov    DWORD PTR [esi+0x1624],0x0
  3ac86: push   DWORD PTR [eax]
  3ac88: lea    eax,[esp+0x2c]
  3ac8c: push   eax
  3ac8d: lea    eax,[esi-0x567d5]
  3ac93: push   eax
  3ac94: call   b0670 <execve@@GLIBC_2.0>
```

32和64主要有这两点区别：

1. 数据访问： 32位是用`[<reg> - 0x??]`来访问只读数据的
2. 调用约定：32位里参数只在栈上，64位用的是寄存器

下面我们来看下为什么这两点不一样的地方会导致one gadget在32位的libc上会很难去找，也很难用。

数据访问方法

在64位libc里访问data段是用rip相对偏移去访问的，而在32位libc里，汇编大概长这样：

```
11f995: mov    ebx,DWORD PTR [esp]
 11f998: ret
 11f999: mov    eax,DWORD PTR [esp]
 11f99c: ret
 11f99d: mov    edx,DWORD PTR [esp]
 11f9a0: ret
 11f9a1: mov    esi,DWORD PTR [esp]
 11f9a4: ret
 11f9a5: mov    edi,DWORD PTR [esp]
```

```
 11f9a8: ret
 11f9a9: mov    ebp,DWORD PTR [esp]
 11f9ac: ret
 11f9ad: mov    ecx,DWORD PTR [esp]
 11f9b0: ret
```

在不同的函数里可能会用不同的寄存器为基础去访问数据，比如fexecve的前6行：

```
000b06a0 <fexecve@@GLIBC_2.0>:
  b06a0: push   ebp
  b06a1: push   edi
  b06a2: push   esi
  b06a3: push   ebx
  b06a4: call   11f995 <__frame_state_for@@GLIBC_2.0+0x375>
  b06a9: add    ebx,0x101957
  b06af: sub    esp,0x8c
```

在执行了`add ebx, 0x101957`之后，ebx就是`libc_base + 0xb06a9 + 0x101957 = libc_base + 0x1b2000`，这里0x1b2000是dynamic tag pltgot的值：

```
$ readelf -d libc.so.6 | grep PLTGOT
0x00000003 (PLTGOT)                     0x1b2000
```

在我们找one gadget的时候，本不应该在一个函数前几行出现的是，所有的32位one gadget都有一个约束要求特定寄存器（一般是ebx或者esi)指向libc的GOT区域。

这个约束看起来非常强，因为ebx和esi在x86里是callee safe的，也就是说在一段程序返回之前会被pop回来，但是在实际当中，由于esi或者rdi已经在main里被赋值为了需要的值，也就是在__libc_start_main里设置的，所

调用约定

在32位里，参数被放在了[esp], [esp+4], [esp+8].
这里有两种方法来做到，一种是直接用mov来设置这些值，另外一种是使用push指令。两种指令在找gadget的时候都需要被考虑到，这样就比64位复杂一些，不过还好不是

在我找到这段gadget之前一切都还很美好。。。

```
3ac69: mov    eax,DWORD PTR [esi-0xb8]
  3ac6f: add    esp,0xc
  3ac72: mov    DWORD PTR [esi+0x1620],0x0
  3ac7c: mov    DWORD PTR [esi+0x1624],0x0
  3ac86: push   DWORD PTR [eax]
  3ac88: lea    eax,[esp+0x2c]
  3ac8c: push   eax
  3ac8d: lea    eax,[esi-0x567d5]
  3ac93: push   eax
  3ac94: call   b0670 <execve@@GLIBC_2.0>
```

第一眼看过去我们可能会觉得这段gadget会调用`execve('/bin/sh', esp+0x2c, environ)`，但是这其实是不对的。在3ac88行把argv设置为了esp+0x2c，esp的值在`3ac6f: add esp, 0xc`和`3ac86: push DWORD PTR [eax]`被改动了，所以这段gadget的真实结果是调用了`execve('/bin/sh', esp+0x34, environ)`

由于这种比较复杂的gadget，我决定不用基于规则的策略来找gadget，而是使用符号执行。

符号执行

我用ruby实现了一个非常简单的符号执行来找one gadget。由于我们根本没有考虑条件跳转，所以非常简单。我们要做的只是去找gadget的正确约束要求，比如说下面这一段汇编：

```
mov edx, [eax]
lea esi, [edx + 4]
push esi
call func
```

如果我们想要func的第一个参数是0，那么真正的的约束就是[[eax]+4]等于0.

为了解决这个问题，我们只需要把每个寄存器和每个栈slot都设置为符号变量，符号化的含义可以在wiki里找到。

通过符号执行，我们就可以成功去解析出one gadget的约束，另外我们还可以从glibc里任意位置开始尝试符号执行，看最后函数是不是可以做到`execve('/bin/sh', argv, environ)'.

结论

one_gadget工具还在开发当中，在1.3.1版本中，在glibc-2.23可以找到很多one gadget。在去掉了一些重复或者很难达到的约束之后，64位里有6个one gadget，32位里有3个one gadget可以找到：

64bit libc-2.23.so

```
david942j in 307-cg at ~/one_gadget on master via 💎 v2.3.4
➜ one_gadget spec/data/libc-2.23-60131540dadc6796cab33388349e6e4e68692053.so
0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xcc543 execve("/bin/sh", rcx, r12)
constraints:
  [rcx] == NULL || rcx == NULL
  [r12] == NULL || r12 == NULL

0xcc618 execve("/bin/sh", rax, r12)
constraints:
  [rax] == NULL || rax == NULL
  [r12] == NULL || r12 == NULL

0xef6c4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf0567 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL

0xf5b10 execve("/bin/sh", rcx, [rbp-0xf8])
constraints:
  [rcx] == NULL || rcx == NULL
  [[rbp-0xf8]] == NULL || [rbp-0xf8] == NULL
```


32bit libc-2.23.so

```
david942j in 307-cg at ~/one_gadget on master via 💎 v2.3.4
➜ one_gadget spec/data/libc-2.23-926eb99d49cab2e5622af38ab07395f5b32035e9.so
0x3ac69 execve("/bin/sh", esp+0x34, environ)
constraints:
  esi is the GOT address of libc
  [esp+0x34] == NULL

0x5fbc5 execl("/bin/sh", eax)
constraints:
  esi is the GOT address of libc
  eax == NULL

0x5fbc6 execl("/bin/sh", [esp])
constraints:
  esi is the GOT address of libc
  [esp] == NULL
```


我也试了不同版本的libc，比如在glibc-2.19上的64位和32位，分别可以找到6段和4段。

如果有任何建议，非常欢迎与我联系，感谢您的阅读。

点击收藏 | 0 关注 | 1

上一篇：Windows下三种mysql提权剖析 下一篇：Silence：针对银行的APT攻击

1. 1 条回复

Pizza 2018-09-16 20:28:36

ancyone_gadget刻不容缓

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录