

漏洞利用实验报告

1. 实验目标

通过分析网络学堂给定压缩包内的二进制程序 vul32，找出漏洞，并写出利用脚本，获取远程服务器上的 flag 文件。

2. 实验原理

- Canary 保护

canary 在 stack overflow 发生的高危区域的尾部插入一个值，当函数返回之时检测 canary 的值是否经过了改变, 以此来判断 stack/buffer overflow 是否发生.

开启 canary 后的 stack 结构:



- 栈溢出原理

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。可以通过缓冲区溢出的手段修改 **return address**，从而达到控制程序的目的。

- ROP

随着 NX 保护的开启，以往直接向栈或者堆上直接注入代码的方式难以继续发挥效果。攻击者们也提出相应的方法来绕过保护，目前主要的是 ROP (Return Oriented Programming)，其主要思想是在栈缓冲区溢出的基础上，利用程序中已有的小片段 (gadgets) 来改变某些寄存器或者变量的值，从而控制程序的执行流程。所谓 gadgets 就是以 ret 结尾的指令序列，通过这些指令序列，我们可以修改某些地址的内容，方便控制程序的执行流程。

3. 实验工具

- IDA 程序反汇编工具
- python2.7

- ubuntu16.04 虚拟机
- checksec

4. 实验过程

1. 查看程序保护机制

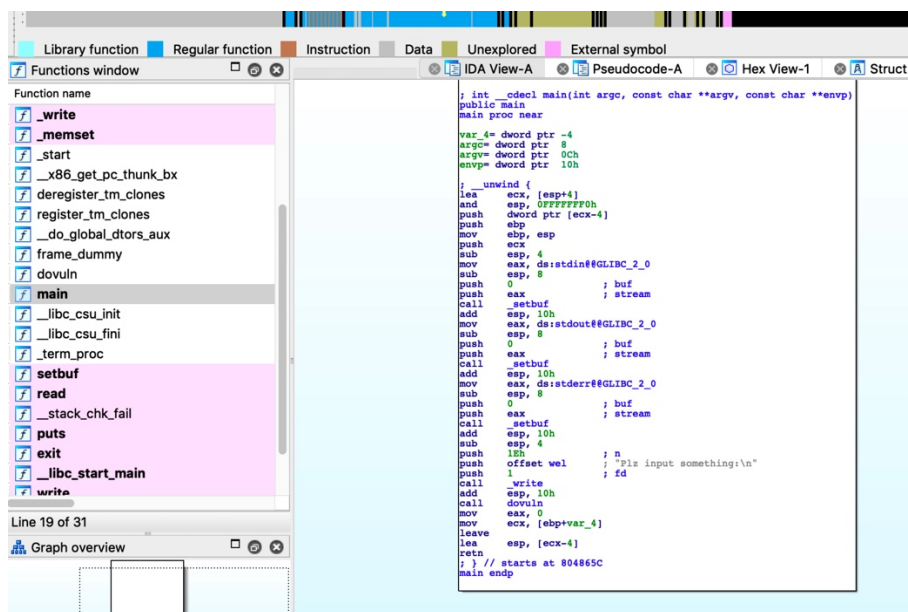
checksec ./vul32

```
chaihj15@ubuntu:/mnt/hgfs/Network_Lab/Shared$ checksec vul32
[*] '/mnt/hgfs/Network_Lab/Shared/vul32'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

由图可知，vul32 程序开启了 Canary 保护和 NX 保护。

2. 查看程序，寻找可利用漏洞

- a) 利用 IDA 工具进行程序逆向



使用 F5，进行反汇编，获取程序 main 函数的 c 代码，如下图所示：

```
IDA View-A Pseudocode-A
1 int dovuln()
2 {
3     int v0; // eax
4     char buf; // [esp+4h] [ebp-44h]
5     char v3[51]; // [esp+5h] [ebp-43h]
6     int v4; // [esp+38h] [ebp-10h]
7     unsigned int v5; // [esp+3Ch] [ebp-Ch]
8
9     v5 = __readgsdword(0x14u);
10    memset(v3, 0, 0x30u);
11    v4 = 0;
12    while ( 1 )
13    {
14        if ( read(0, &buf, 1u) != 1 )
15            exit(0);
16        if ( buf == 10 )
17            break;
18        v0 = v4++;
19        v3[v0] = buf;
20    }
21    return puts(v3);
22 }
```

b) 分析程序，寻找漏洞

通过观察，我们知道程序存在缓冲区溢出问题。对于 char v3[51]并未进行数据大小检查，可以利用这个问题进行栈溢出攻击。再次观察可知，v3 的索引 v4 的距离为 51。所以，当我们利用 v3 的字符溢出攻击时，会在第 52 位修改掉 v4 的值，从而使得之后的溢出字符会写入新的 v4 所指向的地址。

v4 与 ebp 间的偏移距离为 0x10，所以当我们把 v4 的值覆盖为 ret addr 时可以完美避开 Canary 的保护机制。

我们将返回地址改为系统调用 system 的位置，同时将 system 的返回地址覆盖为一个无效地址，并将/bin/sh 字符串作为参数填充可以获取到系统的 shell。

c) 编写利用脚本

实验脚本主要参考 <https://ctf-wiki.github.io/ctf-wiki/pwn/linux/stackoverflow/basic-rop/> 的内容。

难点在于如何获取 system 和/bin/sh 字符的位置。

经过同学提醒，想要获取系统调用，需要使用 libc 中的函数。而 libc.so.6 中的各函数间的相对偏移量是固定的。可以使用 libc-database 工具中 dump 得到如下图所示：

```
chaihj15@ubuntu: /mnt/hgfs/Network_Lab/Shared/libc-database$ ./find __libc_start_
main 540
ubuntu-xenial-i386-libc6 (id libc6_2.23-0ubuntu10_i386)
libc.so.6 (id local-19d65d1678e0fa36a3f37f542e1afd31e439f1bd)
chaihj15@ubuntu: /mnt/hgfs/Network_Lab/Shared/libc-database$ ./dump local-19d65d1
678e0fa36a3f37f542e1afd31e439f1bd
offset__libc_start_main_ret = 0x18637
offset_system = 0x0003a940
offset_dup2 = 0x000d4b50
offset_read = 0x000d4350
offset_write = 0x000d43c0
offset_str_bin_sh = 0x15902b
chaihj15@ubuntu: /mnt/hgfs/Network_Lab/Shared/libc-database$
```

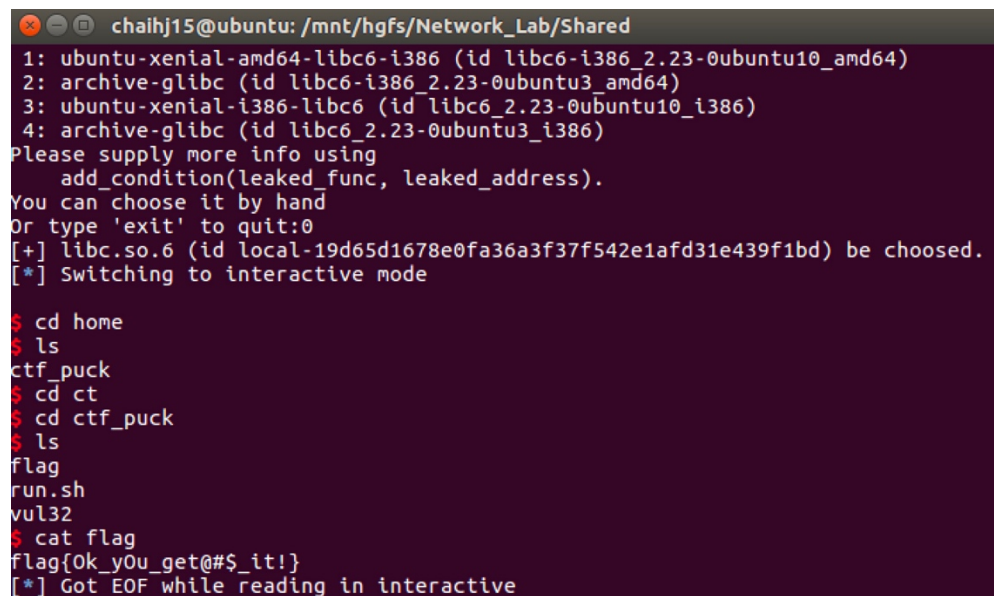
因为 libc.so.6 在 vul32 执行过程中是链接使用的，所以入口的位置与 libc.so.6 中的入口位置不同，使用 ida 观察 vul32 程序可以得出 __libc_start_main 在 vul32 中的绝对地址为 0x00018540。

00018540: __libc_start_main (Synchronized with Hex View-1)

由此可以推出 system 和 ‘/bin/sh’ 在 vul32 中的绝对位置，从而可以使用栈溢出的攻击方式获得目标的 shell。

具体利用脚本见附件。

d) 使用脚本，获取 shell，拿到 flag



```
chaihj15@ubuntu: /mnt/hgfs/Network_Lab/Shared
1: ubuntu-xenial-amd64-libc6-i386 (id libc6-i386_2.23-0ubuntu10_amd64)
2: archive-glibc (id libc6-i386_2.23-0ubuntu3_amd64)
3: ubuntu-xenial-i386-libc6 (id libc6_2.23-0ubuntu10_i386)
4: archive-glibc (id libc6_2.23-0ubuntu3_i386)
Please supply more info using
  add_condition(leaked_func, leaked_address).
You can choose it by hand
Or type 'exit' to quit:0
[+] libc.so.6 (id local-19d65d1678e0fa36a3f37f542e1afd31e439f1bd) be choosed.
[*] Switching to interactive mode

$ cd home
$ ls
ctf_puck
$ cd ct
$ cd ctf_puck
$ ls
flag
run.sh
vul32
$ cat flag
flag{0k_y0u_get@#$_it!}
[*] Got EOF while reading in interactive
```

5. 实验结果

获取的 flag 为 flag{0k_y0u_get@#\$_it!}

6. 实验总结体会

这次实验让我对程序的漏洞利用有了更深、更形象的理解，学会了利用相关工具进行程序的漏洞分析和利用，同时也让我知道了程序漏洞所带来的严重的安全问题。很高兴能够通过这次实验了解到一点点网络攻防的知识，非常开心，希望之后能有机会能继续接触安全领域的知识。