**缓冲区溢出攻击的 要点提示**

1. 实验包里提供了所有的源程序， 但要自己编译生成执行程序。

源程序包括 bufbomb.c 和 buf.c 。

1. 自己编译生成执行程序

编译链接时要加多种编译开关。

gcc -g -fno-stack-protector -no-pie -DU\* -fcf-protection=none -z execstack bufbomb.c buf.c -o bufbomb

-DU\* 中 \*为自己学号的最后一位。例，学号为 U202215001，命令为:

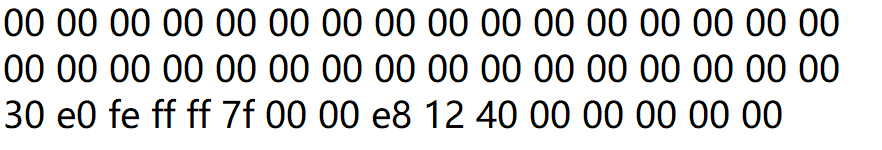
gcc -g -fno-stack-protector -no-pie -fcf-protection=none  -z execstack -DU1 bufbomb.c buf.c -o bufbomb

(3) 程序使用方法

./bufbomb 学号 攻击串文件 关的编号

Example : ./bufbomb U202215001 smoke\_hex.txt 0

攻击串中的文件（smoke\_hex.txt）内容示例如下。



（4）函数参数传递提示

linux 64位程序中的函数调用，函数的第一个（最左）、第二、第三、第四参数依次放在 %rdi、%rsi、 %rdx、%rcx中；或者 %edi,%esi,%edx,%ecx中。用64/32/16/8位的寄存器，取决于参数的类型。

(5) 程序的调试

gdb bufbomb

list main

b \*\*\* 在适当的行上设置断点

run U123456781 smoke\_hex.txt 0 调试带命令行参数的程序，在run 后 给出命令行参数

cat smoke\_hex.txt |./hex2raw |./bufbomb 使用管道操作符连接不同的程序

攻击字符串生成提示

1. smoke ,只要将要返回的断点地址设为 smoke的起始地址即可。即在字符串的相应位置填上smoke的入口地址即可，该位置之前的字符串可随意设置。
2. fizz，对于 64位程序，使用的是寄存器 edi 来传递 int型的参数 val。直接修改 edi 是很困难的。一种巧妙的办法是，不要跳到 fizz函数的起始地址，而直接跳到 if (val==cookie) 处。此时，val 的值已存放在栈中地址为 -0x4（%rbp）处。只要 %rbp-0x4 与 cookie 对应同一个单元，则if的条件就会成立。如果只是简单绕开if语句的检查，之后的打印语句打印 val的值会发现它与cookie并不一致。

构造的字符串要点：正确设置要执行的 fizz语句的地址（非fizz的第一条机器指令的地址）；正确的设置rbp。在保留返回地址（“断点地址”）的前面，是保存原 rbp的值，要将其修改为期望的值。

在32位程序中，直接使用栈传递参数时，可以将cookie的值存放在攻击串的相应位置。这也是另外一种解决问题的方法。

3. bang

全局变量 global\_value 并没有存储在栈中，要真正的修改它的值为cookie，无法直接使用攻击字符串来更改，只能编写一段代码来修改 global\_value的值，并且要让这段代码得到执行。

首先，通过 gdb 确定 global\_value的地址，以及cookie的值。直接写汇编源程序（如bang.s）,含有对global\_value的修改，以及跳转到bang相应位置的指令（用ret指令）。编译后得到指令的机器码放到 buf的开头。修改getbuf的返回地址，使其跳转到 buf缓冲区的开头。

bang.s 示例程序：

mov 0x404148, %eax 假设cookie的地址是 0x0000000000404148

源操作数是直接寻址方式，执行后，eax 中的内容就是 cookie的值

………

jmpq \*%rdx 注意，要使用 寄存器寻址方式，跳到 bang 的相应位置

对bang.s 编译生成目标文件，如 bang.o

再使用 objdump -s -d bang.o 得到 16进制的指令编码；用于构造buf的攻击串。

4. boom

**要实现 boom 的无感攻击，要做到以及几点：**

（1） 将 eax 设置为 cookie；

（2） 恢复 %rbp, 这样回到主程序时，才能正常执行；

（3） 将主程序的断点地址送给 %rip

这就需要事先知道 cookie的值，知道保存的 %rbp的值，以及原始的断点地址。可以通过调试bufbomb ，获取这些信息。

另外编写一段程序，实现cookie 值-> eax；原来保存的rbp -> rbp; 原断点地址压栈； ret。

将这段程序的机器码，写到buf 缓冲区的开头。原来保存断点处的地址，改为 buf 缓冲区的起始地址。

其他注意事项：  
Linux上地址空间随机化(Address Space Layout Randomization)（ASLR）分为0/1/2三级，用户可以通过内核参数randomize\_va\_space进行等级控制，对应效果如下：  
0：没有随机化，即关闭ASLR  
1：保留的随机化，即共享库、栈、mmap()以及VSDO将被随机化  
2：完全的随机化，在1的基础上，通过brk()分配的内存空间也将随机化  
ASLR不负责代码段和数据段的随机化工作，负责栈的地址随机化。  
  
ubuntu下地址随机化的启闭可以执行以下命令（其中X代表上述0-2三级）：  
sudo sh -c &apos;echo X >/proc/sys/kernel/randomize\_va\_space&apos;  
如关闭地址随机化命令为：  
sudo sh -c &apos;echo 0 >/proc/sys/kernel/randomize\_va\_space&apos;  
  
通关不仅要在gdb调试模式下实现，也需要在直接运行模式下实现。

以关卡3为例，因为要固定访问栈内的地址（buf），可以在gets中增加打印buf的地址，看看每次直接执行时buf地址是否变化。如果不一样，说明没有关闭地址随机化，需要先行关闭才能通关。