**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。

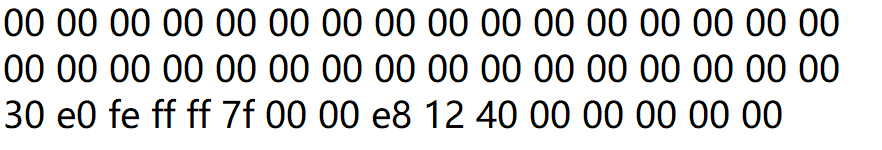
源程序包括 bufbomb.c、buf.c、support.c 。

但是编译链接时，要加多种编译开关，下面有更详细的说明。

程序运行的用法如下： ./bufbomb 学号 攻击串文件 关的编号

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

攻击串中的文件内容示例如下。



（2） 编译开关

要带自己学号的 末位数字,即： -D U1

要去掉一些栈保护，一些检查代码的生成，否则，修改这些地方的数据，会导致程序运行异常。

gcc -g -fno-stack-protector bufbomb.c buf.c support.c -o bufbomb

gcc -g -fno-stack-protector -no-pie bufbomb.c buf.c support.c -o bufbomb -DU1

gcc -g bufbomb.c buf.c support.c -o bufbomb -D U1

gcc -g -D U1 -fno-stack-protector -no-pie -fcf-protection=none bufbomb.c buf.c support.c -o bufbomb

gcc -g -D U1 -fno-stack-protector -no-pie -fcf-protection=none -z execstack bufbomb.c buf.c support.c -o bufbomb

gcc -g -D U1 -fno-stack-protector -no-pie -fcf-protection=none bufbomb\_new.c buf.c support.c -o bufbomb\_new

类似的与堆栈保护有关的编译开关

**-fstack-protector 、-fstack-protector-all、-fstack-protector-strong、**

**-fstack-protector-explicit、-fstack-check、-fno-stack-limit、-fsplit-stack、**

**-fstack-limit-register=reg、-fstack-limit-symbol=sym、**

程序的调试

gdb bufbomb

list main

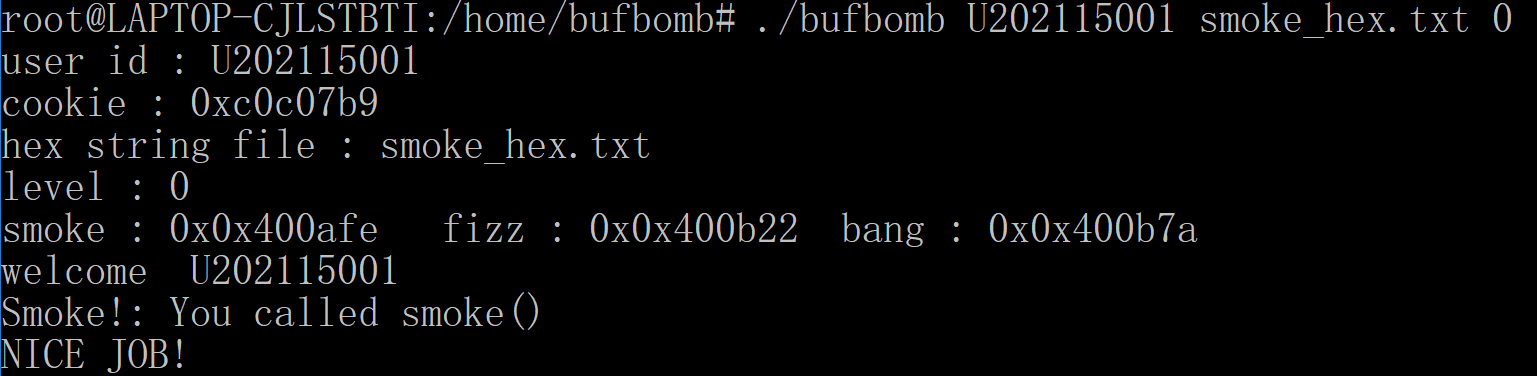
b \*\*\* 在适当的行上设置断点

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

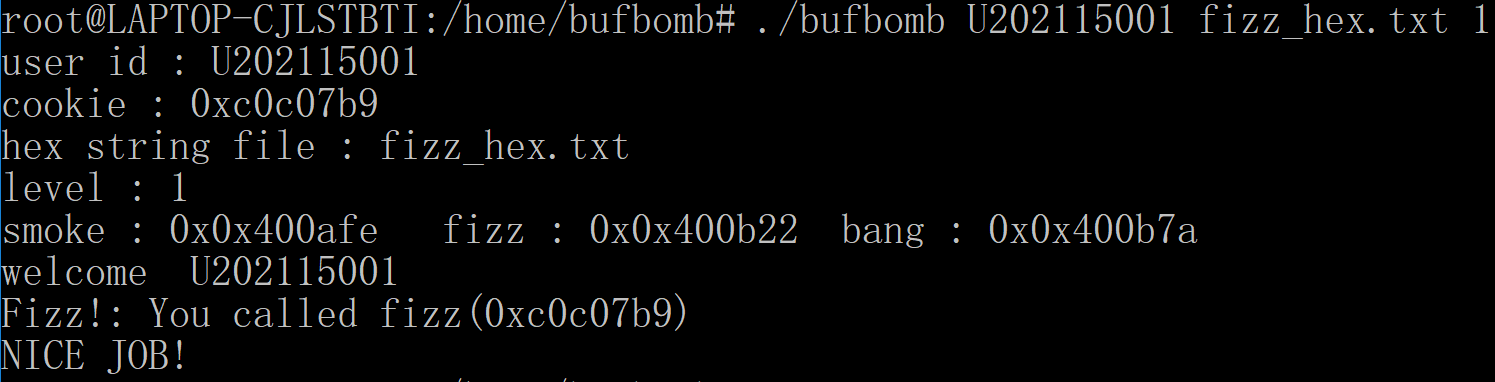
1. smoke ,只要将要返回的断点地址设为 smoke的起始地址即可。即在字符串的相应位置填上smoke的入口地址即可，该位置之前的字符串可随意设置。



1. 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。

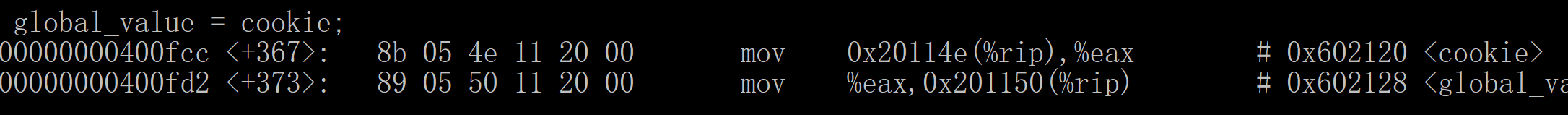


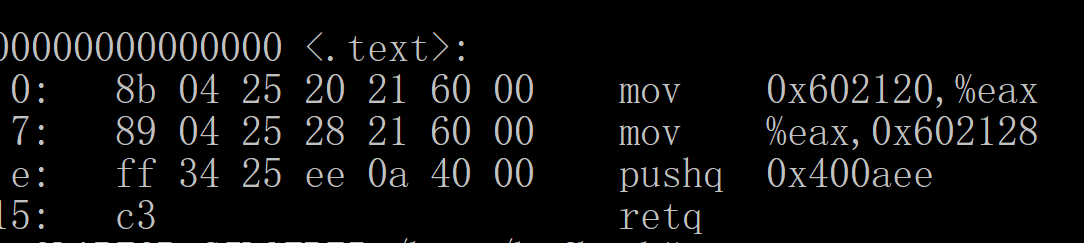
3. bang

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。

此关在 64位程序中，实现有难度。主要是全局变量的地址 使用了 %rip+位移量来寻址，将 rip执向 buf缓冲区的首地址时，位移量变得非常大，以至于机器指令不支持 64位的位移量，而直接写地址也不行。





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；

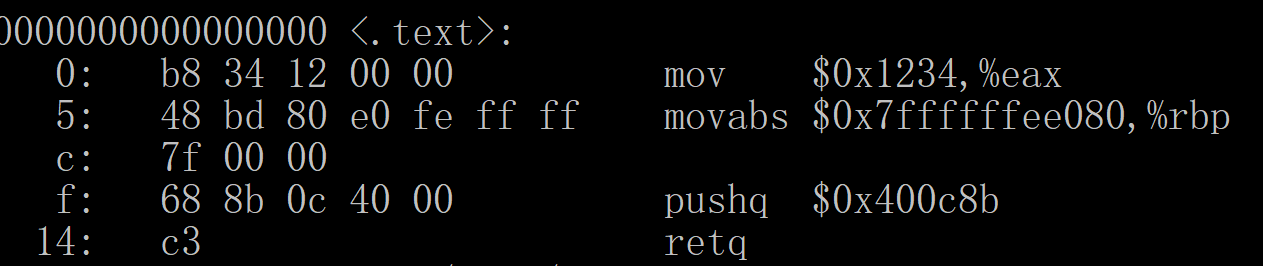
（2） 恢复 %rbp, 这样回到主程序时，才能正常执行；

（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。



* Gcc 参考手册 **https://www.gnu.org/software/gcc/**

**-fstack-protector**

Emit extra code to check for buffer overflows, such as stack smashing attacks.

This is done by adding a guard variable to functions with vulnerable objects.

This includes functions that call alloca, and functions with buffers larger than

8 bytes. The guards are initialized when a function is entered and then checked

when the function exits. If a guard check fails, an error message is printed and

the program exits.

**-fstack-protector-all**

Like ‘-fstack-protector’ except that all functions are protected.

**-fstack-protector-strong**

Like ‘-fstack-protector’ but includes additional functions to be protected

— those that have local array definitions, or have references to local frame

addresses.

**-fstack-protector-explicit**

Like ‘-fstack-protector’ but only protects those functions which have the

stack\_protect attribute.

**-fstack-check**

Generate code to verify that you do not go beyond the boundary of the stack.

You should specify this flag if you are running in an environment with multiple

threads, but you only rarely need to specify it in a single-threaded environment

since stack overflow is automatically detected on nearly all systems if there is

only one stack.

Note that this switch does not actually cause checking to be done; the operating

system or the language runtime must do that. The switch causes generation of

code to ensure that they see the stack being extended.

You can additionally specify a string parameter: ‘no’ means no checking,

‘generic’ means force the use of old-style checking, ‘specific’ means use the

best checking method and is equivalent to bare ‘-fstack-check’.

Old-style checking is a generic mechanism that requires no specific target sup-

port in the compiler but comes with the following drawbacks:

1. Modified allocation strategy for large objects: they are always allocated

dynamically if their size exceeds a fixed threshold.

2. Fixed limit on the size of the static frame of functions: when it is topped

by a particular function, stack checking is not reliable and a warning is

issued by the compiler.

3. Inefficiency: because of both the modified allocation strategy and the

generic implementation, code performance is hampered.

Note that old-style stack checking is also the fallback method for ‘specific’ if

no target support has been added in the compiler.

**-fstack-limit-register=reg**

**-fstack-limit-symbol=sym**

**-fno-stack-limit**

Generate code to ensure that the stack does not grow beyond a certain value,

either the value of a register or the address of a symbol. If a larger stack is

required, a signal is raised at run time. For most targets, the signal is raised

before the stack overruns the boundary, so it is possible to catch the signal

without taking special precautions.

For instance, if the stack starts at absolute address ‘0x80000000’ and grows

downwards, you can use the flags ‘-fstack-limit-symbol=\_\_stack\_limit’

and ‘-Wl,--defsym,\_\_stack\_limit=0x7ffe0000’ to enforce a stack limit of

128KB. Note that this may only work with the GNU linker.

You can locally override stack limit checking by using the no\_stack\_limit

function attribute (see Section 6.31 [Function Attributes], page 429).

**-fsplit-stack**

Generate code to automatically split the stack before it overflows. The resulting

program has a discontiguous stack which can only overflow if the program is unable to allocate any more memory. This is most useful when running threaded

programs, as it is no longer necessary to calculate a good stack size to use for

each thread. This is currently only implemented for the x86 targets running

GNU/Linux.

When code compiled with ‘-fsplit-stack’ calls code compiled without

‘-fsplit-stack’, there may not be much stack space available for the

latter code to run. If compiling all code, including library code, with

‘-fsplit-stack’ is not an option, then the linker can fix up these calls so that

the code compiled without ‘-fsplit-stack’ always has a large stack. Support

for this is implemented in the gold linker in GNU binutils release 2.21 and

later.

**linux64 溢出,64位Linux下的栈溢出**

<https://blog.csdn.net/weixin_29710403/article/details/116872724>

不过即便内存地址有64位长，用户空间也只能使用前47位。要牢记这点，因为当你指定一个大于0x00007fffffffffff的地址时，会抛出一个异常。

**linux 函数栈溢出,64位Linux下的栈溢出**

<https://blog.csdn.net/weixin_30175731/article/details/116716798?spm=1001.2101.3001.6661.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-1-116716798-blog-116872724.pc_relevant_multi_platform_whitelistv4&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-1-116716798-blog-116872724.pc_relevant_multi_platform_whitelistv4&utm_relevant_index=1>

32位环境在的 操作 （不一定成功）

sudo dpkg --add-architecture i386

sudo apt-get install libc6:i386

sudo apt-get install lib32ncurses5

sudo apt-get install lib32z1

查看支持的 结构 dpkg --print-architecture

dpkg --print-foreign-architectures

sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 zlib1g:i386