缓冲区溢出攻击实验

本文的实验来源于 CSAPP 《Computer Systems A Programmer's Perspective》（深入理解计算机系统）一书中第三章的一个实验，即习题 3.38。

作者给出了一个含有缓冲区溢出的程序 bufbomb.c，你需要做的，就是注入给缓冲区些特殊的数据，到底利用缓冲区的目的。

在做这个题目之前，你当然要知道什么是帧栈结构（请参阅《深入理解计算机系统》第三章）或者之前的博文栈针&溢出，了解%ebp 和%esp 的含义

```
    //bufbomb.c

    /* Bomb program that is solved using a buffer overflow a
ttack */

    #include <stdio.h>

    #include <stdlib.h>

    #include <ctype.h>

    /* Like gets, except that characters are typed as pairs
of hex digits.

     Nondigit characters are ignored.  Stops when encounter
s newline */

    char*getxs(char*dest)

    {

       int c;

       int even =1; /* Have read even number of digits */

       int otherd =0; /* Other hex digit of pair */

       char*sp = dest;

       while ((c = getchar()) != EOF && c !='\n') {

           if (isxdigit(c)) {
```

```
            int val;
            if ('0'<= c && c <='9')
          val = c -'0';
            else if ('A'<= c && c <='F')
          val = c -'A'+10;
            else
          val = c -'a'+10;
            if (even) {
          otherd = val;
          even =0;
        }
        else {
            *sp++= otherd *16+ val;
          even =1;
        }
      }
    }
     *sp++='\0';
     return dest;
}

/* $begin getbuf-c */
int getbuf()
{
     char buf[12];
  getxs(buf);
     return1;
}
```

```
    void test()
    {
       int val;
      printf("Type Hex string:");
      val = getbuf();
      printf("getbuf returned 0x%x\n", val);
    }
    /* $end getbuf-c */


    int main()
    {


       int buf[16];
     /* This little hack is an attempt to get the stack to b
e in a
       stable position
     */
       int offset = (((int) buf) &0xFFF);
       int*space = (int*) alloca(offset);
       *space =0; /* So that don't get complaint of unused v
ariable */
      test();
       return0;
    }
```

函数 getxs 的功能类似于库函数 gets 的功能，除了它是以十六进制数字对的编码方式读入的字符。例如，要读入字符串“0123”，你得给出输入字符串“30 31 32 33”，这个函数会忽略空格。

分析这个程序,可以得知,正常情况下,这个函数会在 getbuf 中,调入 getxs 函数读入数字对，然后不管任何情况下，都会对 test 函数返回 0x1,然后由 test 中的 printf 函数打印处 getbuf 的返回值。

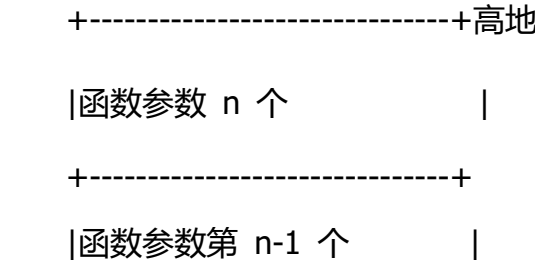
现在你的任务，就是，利用缓冲区溢出的漏洞，输入些特殊的数字，使得屏幕中打印的是 0xdeadbeef。

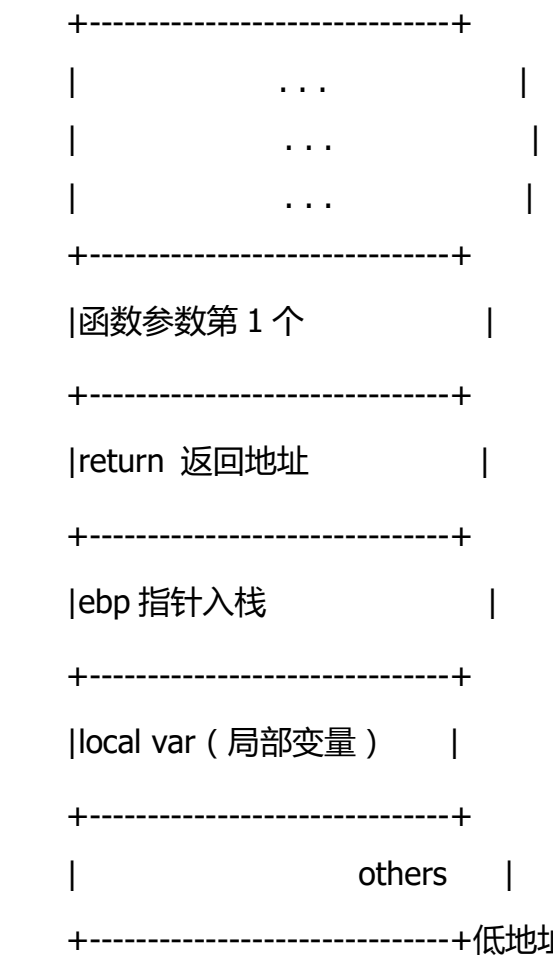
由于这个程序中含有 alloca 函数，因而要添加#include <new>#include <excpt.h>#include <malloc.h>（这三个是我在网上随便搜的，反正引进之后编译之后不报错）。

我是在 WindowsXP，visual c++6.0 环境解决这个问题的。。

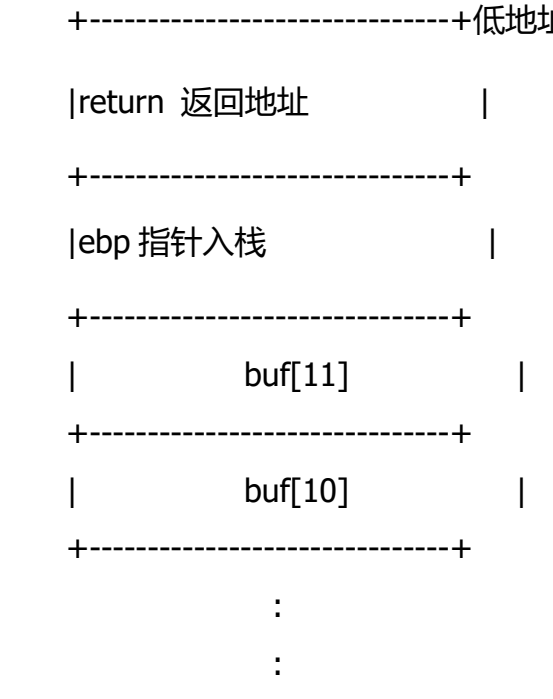
题目中已经说了，“分析这个程序，可以得知，正常情况下，这个函数会在 getbuf 中，调入 getxs 函数读入数字对，然后不管任何情况下，都会对 test 函数返回 0x1,”那我们该怎么办了？我们马上可以想到在 getbuf 这个函数里定义的 char buf[12]上做手脚，可以看到在 getxs 函数里的 while 循环，结束条件只是以回车或者是 eof 结束符为判断标准,所以,根本没对 char 输入的数量做判断！这样的话，我们可以输入多于 12 个的数，从而缓冲区溢出！

在这里还是提一下帧栈结构，如下：





按照上面说的函数栈的存放情况，在 getbuf 这个函数里，函数参数没有，我们不管 然后就是 return 返回地址 然后就是 ebp 指针 然后就是 char buf[12]。



```
                    :
    +-----------------------------+
    |                buf[0]                |
    +-----------------------------+
    |               others                 |
    +-----------------------------+高地址
```

如果我们对 buf 溢出，能改写 ebp 和 return 地址！下面看看，ebp 是多少，return 地址是多少。

要知道这里的%ebp 存的是 test 函数的%ebp，因而我们在调试的时候就可以在 test 函数得到%ebp 的值，它应该是我们写入的 buf[12]-buf[15]的值，而且它要保持原来的值，不然返回之后就乱套了，在我机器上是 0x0012efa0。这个很容易，解决了第一步。

下面我们再来看返回地址，先看一段汇编码（不同的机器有所不同）：

```
58:       val = getbuf();
004011C5  call      @ILT+10(getbuf) (0040100f)
004011CA  mov       dword ptr [ebp-4],eax
59:       printf("getbuf returned 0x%x\n", val);
004011CD  mov       eax,dword ptr [ebp-4]
004011D0  push      eax
004011D1  push      offset string "getbuf returned 0x%x\n"
(0042001c)
004011D6  call      _printf (00401510)
004011DB  add       esp,8
```

在 getbuf()返回后，肯定会接着执行 004011CA ，我们能让它从这执行吗？当然不行！不然就要 push eax，那是我们不想看到的，因为 eax 的值就是 1。因而我们会想到能不能跳过这？当然能，改返回地址啊！顺水推舟，我们通过 buf

数组来覆盖返回地址。此时，我们想要它直接跳到 004011D1 处，因而可以通过设置 buf[16]-buf[19]的值来覆盖返回地址。

到了考虑如何加进 deadbeef 了，在返回后，将直接执行 push offset string "getbuf returned 0x%x\n" (0042001c)，没 eax 怎么行了？不然 printf 函数就少了参数。再回到帧栈结构一下，printf 在调用之前，要压入参数，先压入 val，再压入 offset string "getbuf returned 0x%x\n"，也就是说参数 val（等同那个 eax）在 offset string "getbuf returned 0x%x\n "之上，而且紧挨着。此时我们可以想到，既然返回之后（返回地址及其以下的元素都已弹出，返回地址的上一个字节成了栈顶）就执行 push offset string "getbuf returned 0x%x\n" (0042001c)进行压栈 ,那此时栈顶一定是参数 val ,而它原来就在返回地址上面，因而我们可以通过设置 buf[20]-buf[23]的值来覆盖这个地方。

综上所述，%ebp 的值为 0x0012efa0,修改后的返回地址为 0x004011D1，因而我们可以输入

00000000 00000000 00000000 a0ef1200 d1114000 efbeadde

这 24 个 0 可以输入别的，不影响，关键是后面 24 个数。