实验 5.1

1.

```
hyc@161220049:~/workspace/lab05/161220049$ ./overflow1
why u r here?!
```

2.

```
0804845b <foo>:
 804845b:       55                      push   %ebp
 804845c:       89 e5                   mov    %esp,%ebp
 804845e:       83 ec 10                sub    $0x10,%esp
 8048461:       b8 3b 84 04 08          mov    $0x804843b,%eax
 8048466:       89 45 04                mov    %eax,0x4(%ebp)
 8048469:       b8 00 00 00 00          mov    $0x0,%eax
 804846e:       c9                      leave
 804846f:       c3                      ret
```

由反汇编程序可以看出，第 6 行中，将 why_here 赋值给 buff[2]，buff[2]位于%ebp 的上方 4 位，即 0x4(%ebp)

返回地址	0xbfffef30
保存的%edp	0xbfffef2c
保存的%eax	0xbfffef28

由于 buff[2]位于%ebp 上方四位，故 buff[2]的地址为 0xbfffef30，又因为数组的各元素地址连续，所以 buff 数组的开始地址为 0xbfffef28.

3.

2 中 buff[2]的地址与 foo 函数的返回地址重合，当调用 buff[2] = (int)why_here 时，函数把 why_here 函数的地址进行返回，相当于调用了 why_here 函数，所以输出了”why u r here?!”。

实验 5.2

1.

```
hyc@161220049:~/workspace/lab05/161220049$ ./overflow2
So..The End...
ffffffffffffff
ffffffffffffff
or...maybe not?
hyc@161220049:~/workspace/lab05/161220049$ ./overflow2
So..The End...
ffffffffffffffff
ffffffffffffffff
or...maybe not?
Segmentation fault (core dumped)
```

需要连续输入 16 个‘f’才能产生 segmentation fault。

2.

```
8048444:       8d 45 f0                lea    -0x10(%ebp),%eax
8048447:       50                      push   %eax
8048448:       e8 b3 fe ff ff          call   8048300 <gets@plt>
```

由上图可知，数组 buf 的开始地址为-16（%ebp)。又因为每个 char 字符占用 1 位，当连续输入 16 个 f 时，由于字符串末尾还有空字符’\0’（这个空字符由读取的换行符转化而成），会破坏更高地址的%ebp，从而引发段错误。
当连续输入 N+6=22 个字符时，不妨用栈来表示内存的使用情况：

返回地址(低位为 00 buf[21] buf[20])	0xbfffef44-0xbfffef47
buf[19] buf[18] buf[17] buf[16]	0xbfffef40-0xbfffef43
buf[15] buf[14] buf[13] buf[12]	0xbfffef3c-0xbfffef3f
buf[11] buf[10] buf[9] buf[8]	0xbfffef38-0xbfffef3b
buf[7] buf[6] buf[5] buf[4]	0xbfffef34-0xbfffef37
buf[3] buf[2] buf[1] buf[0]	0xbfffef30-0xbfffef33

因此内存地址范围 0xbfffef30-0xbfffef45 存放 buf 数组从前往后的各元素

3.未被污染前，0xbfffef40-0xbfffef43 存放着%ebp，0xbfffef44-0xbfffef47 存放着 doit 函数的返回地址.

4.

```
hyc@161220049:~/workspace/lab05/161220049$ ./overflow2
So..The End...
fffffffffffffff
fffffff
or...maybe not?
hyc@161220049:~/workspace/lab05/161220049$ ./overflow2
So..The End...
ffffffffffffffff
fffffff
or...maybe not?
```

此时没有发生 segmentation fault。
fgets 和 gets 的相同点：都是标准输入，一次读取；不同点：gets 可以无限读取，不会判断上限，而 fgets 的参数中有读取的字符串长度，当读完规定的长度时，不再读入。因此 fgets 在这里只会读前 8 个字符，后面的输入不影响程序，也就不会完成攻击。所以 fgets 相对 gets 更加安全。

5.
栈随机化思想：栈的位置在程序每次运行时都有变化，基于攻击者猜测随机化空间位置的可能性降低，通过增加搜索空间的方式来实现，它提供更多的熵存在于随机偏移中时时更有效的。包括随机排列程序的关键数据区域的位置，包括可执行的部分、堆、栈及共享库的位置。

栈破坏检测思想：在栈中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀值，在程序每次运行时随机产生，攻击者没有简单的方法检测它。在恢复寄存器状态和从函数返回之前，程序检查这个金丝雀值是否被函数的某个操作或者函数调用的某个操作改变，若是，则程序异常终止。

实验 5.3
1. 用./overflow3 20 运行程序：

```
hyc@161220049:~/workspace/lab05/161220049$ ./overflow3 20
malloc 80 bytes
loop time:20[0x14]
```

用./overflow3 1073741824 运行程序：

```
hyc@161220049:~/workspace/lab05/161220049$ ./overflow3 1073741824
malloc 0 bytes
loop time:1073741824[0x40000000]
Segmentation fault (core dumped)
```

2.

32 位平台 size_t 类型的范围为 0-4294967295.

第一次运行成功分配了 80 字节的内存，第二次运行分配了 0 字节 的内存但发生了 segmetation fault.

3.

argv[1]指向在 DOS 命令行中执行程序后的第一个字符串，随后通过 atoi 转化为 int 类型的值，作为 foo 函数的第一个参数传递到函数中，并乘以 sizeof(int)分配给 buf 数组内存。

对于第一次运行，所以 argv[1]的值为 20，在计算 malloc()时乘以 4 得到 80，成功分配内存；

对于第二次运行，输入了 “1073741824”，转化为整数 len=1073741824，为 buf 分配的内存为 1073741824*4=4294967296，恰好超出 size_t 的最大值 1，因此输出是 malloc 0 bytes，但系统认为 buf 非空（否则不会输出 loop time）由于指针指向的数组内存大小为 0，在执行 for 循环的操作时会访问出错，打印出 segmentation fault.

4.

现有 Linux 分配 0 个字节的合理性：允许分配 0 字节内存空间，但是返回值仍是 NULL 或者是可以被 free 的指针。

这可能是 c 运行库设计的问题。