实验 5.1

1. 运行程序 overflow1，观察出现的实验现象。

```
noob@noob-virtual-machine:~/workspace/lab05$ ./overflow1
why u r here?!
```

2. 反汇编源程序，根据汇编代码回答下面的问题：

(1) 假设内存地址 0xbfffef30-0xbfffef33 保存 foo 函数的返回地址，请推断出数组 buf 的开始地址。

0xbfffef28

(2) 根据（1）的结果，推断出变量 buf[2]在内存中的开始地址。

0xbfffef30

3.根据 2 的分析结果，解释出现实验现象的原因。

对 buf 数组的越界访问导致 why_here 函数的起始地址覆盖了 foo 函数的返回地址，于是当 foo 函数返回时，跳转到 why_here 函数打印出该字符串.

实验 5.2

1. 运行 overflow2，尝试输入任意个数的字符，如字符’a’，猜测需要连续输入几个字符才能产生 segmentation fault？

```
noob@noob-virtual-machine:~/workspace/lab05$ ./overflow2
So... The End...
012345678901234
012345678901234
or... maybe not?
noob@noob-virtual-machine:~/workspace/lab05$
noob@noob-virtual-machine:~/workspace/lab05$ ./overflow2
So... The End...
0123456789012345
0123456789012345
or... maybe not?
Segmentation fault (core dumped)
```

16 个字符.

2. 反汇编源程序，验证 1 中的猜测。假设数组 buf 的开始地址为 0xbfffef30，输入 N 个字符’a’后出现 segmentation fault。查阅 gets 函数的资料，如 man。请写出输入 N+6 个字符后，内存地址范围[0xbfffef30, 0xbfffef30+N+5]的存放情况。

由

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

可知, N 为 16.

假设输入的 N+6 个字符全为’0’.

则内存存放情况为: 逐字节全为 0x30(‘0’的 ascii 编码的十六进制).

3.根据 2 的假设，请写出未被污染前，地址范围 0xbfffef30+N 到 0xbfffef30+N+7 原始存放的数据。

地址范围	值	解释
N + 7 ~ N + 4	08 04 84 88	返回地址
N + 3 ~ N	bf ff f0 78	old %ebp

4.将源程序中的 gets 函数替换为 fgets 函数，然后重新实验。比较 fgets 函数和 gets 函数的异同，简要说明采用 fgets 函数为何无法完成上述攻击。

异:

fgets 函数需要指定读取字符个数.

fgets 函数可以用于文件读取.

fgets 函数不会丢弃末尾的换行符

同:

都可以从标准输入中读取一行字符串.

遇到换行符和 EOF 都结束.

原因:

fgets 函数会指定读取字符个数，若读取字符的个数超过了数组大小，该函数会自动根据定义数组的长度截断.

5. 查阅相关资料，简要阐述抵御缓冲区溢出的两种防御方法（栈随机化和栈破坏检测）的主要思想。

栈随机化:

缓冲区溢出攻击的原理是要预先知道返回地址，然后用恶意代码覆盖返回地址，但是随机化以后，缓冲区的位置会随着每一次运行程序而改变，导致无法预先知道返回地址.

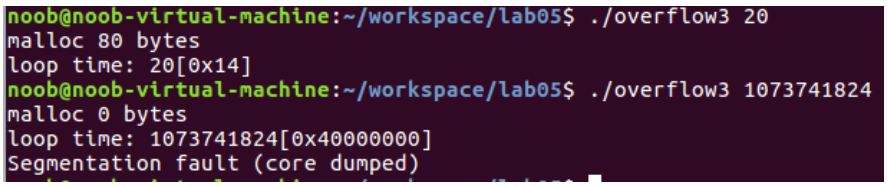
栈破坏检测:

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. 这个金丝雀值是在程序每次运行时随机产生的，因此，攻击者没有简单的办法知道它是什么.

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该函数的某个操作或者函数调用的某个操作改变了. 如果是，那么程序异常终止.

实验 5.3

1. 分别以参数./main 20 和./main 1073741824 运行程序，观察出现的现象。



2. 回答 32 位平台 size_t 类型的范围，计算出两次分配内存大小的不同。

范围: 0 - 2^{32} – 1. 80 bytes.

3. 解释出现的实验现象。

第一次成功分配了 len(0x14) * sizeof(int)(4) = 80 个字节.

第二次分配时，由于 len(0x40000000) * sizeof(int)(4) = 0x1 0000 0000 产生溢出，系统截取低 32 位作为有效值，其值为 0，于是分配了 0 字节.

4. 尝试 malloc 0 个字节，并分析现有 Linux 允许 malloc 0 个字节的合理性（开放题）。

可以 malloc 0 个字节，函数会返回一个不能用的指针，但确实分配了内存.

合理性:

malloc 分配的内存是堆内存，由于堆没有自己的机器指令，所以要有系统自己编写算法来管理这片内存，通常的做法是用链表，在每片被分配的内存前加个表头，里面存储了被分配内存的起始地址和大小，malloc 返回的就是表头里的起始指针，这个地址是由一系列的算法得来的，通常不会为 0. 一旦分配成功，就返回一个有效的指针.

对于分配 0 空间来说，算法已经算出可用内存的起始地址，但是占用 0 空间.