1.

1) overflow1 运行结果：

```
lx-161220076@ubuntu:~/workspace/lab05/161220076$ ./overflow1
why u r here?!
```

2) (1) 0xbfffef1c　(2) 0xbfffef24

3)

return address
ebp

```
mov     $0x804843b,%eax
mov     %eax,0x4(%ebp)
```

这两条指令将立即数 0x804843b 放入寄存器 ebp 向上偏移四位的地址上，也就是原本存放 foo 函数返回地址的位置，而这个立即数正好是函数 why_here 的开始地址，导致函数 foo 执行完之后继续执行了函数 why_here。

2.

1) 16
2) 都是 a
3) 0xbfffef30+N~0xbfffef30+N+3 是 ebp
0xbfffef30+N+3~0xbfffef30+N+7 存放的是函数发返回地址
4) fgets()：从文件结构体指针 stream 中读取数据，每次读取一行。读取的数据保存在 buf 指向的字符数组中，每次最多读取 bufsize-1 个字符（第 bufsize 个字符赋'\0'），如果文件中的该行，不足 bufsize 个字符，则读完该行就结束。所以用 fgets()函数在此处不会发生覆盖 ebp 和函数返回地址的情况。
而 gets()函数只会在读入到 EOF 或者换行结束。输入的字符数过多污染了 ebp。
5) 栈随机化：使得栈的位置在程序每次运行时都有变化。因此，即使许多机器都运行同样的代码，它们的栈地址都是不同的。
栈破坏检测：加入栈保护着机制，当栈被破坏时，程序会自动异常退出。

3.

1) 以不同参数的运行情况：

```
lx-161220076@ubuntu:~/workspace/lab05/161220076$ ./overflow3 20
malloc 80 bytes
loop time: 20[0x14]
lx-161220076@ubuntu:~/workspace/lab05/161220076$ ./overflow3 1073741824
malloc 0 bytes
loop time: 1073741824[0x40000000]
Segmentation fault (core dumped)
```

2) 范围为 0~（1<<32）-1。第一次分配了 20*sizeof（int）个字节，第二次分配了 0 个字节。

3)

```
buf = malloc(len*sizeof(int));
```

在分配内存空间时，len*sizeof(int)超出了 unsigned int 的表示范围，发生了溢出，溢出之后正好为 0，所以分配了 0 个字节，在下面 for 循环中就会发生缓冲区溢出。

4) 为了避免由于溢出等错误而导致的程序崩溃等问题。允许分配 0 个字节空间就相当于没有得到空间。

时 c 运行库设计的问题。