题目

出题人：阮震元，2015 年 11 月

栈溢出是黑客常用的攻击手段。代码 2.c 是一段模拟栈溢出的示例程序。2.s 是通过 gcc -m32 -fno-stack-protector -S 2.c 产生的 32 位汇编代码的节选片段。在 2.c 中，若使用适当的 a 值，则可在调用 f(a,b)的过程中修改其活动记录中的返回地址，使得程序跳到黑客指定的地址 b，执行恶意代码。

GCC 提供了编译选项-fstack-protector-all(此选项默认情况下自动开启)，它会启用栈保护，为所有函数插入保护代码，在发生栈溢出时会发出警告信息并迅速结束程序。3.s 是通过 gcc -m32 -S -fstack-protector-all 2.c 产生的 32 位汇编代码的节选片段。

阅读代码，在假设机器环境为 X86/Linux 的情况下回答如下问题。

(1). 结合函数调用时的栈布局，回答 a 为何值时可以修改函数 f 的返回地址。

(2). 在实际中，黑客往往并不知道 a 的合适值，他们一般通过循环调用 f(x,b)(stack 末地址<x≤R)来试探性的修改函数的返回地址。对比 3.s 和 2.s,回答-fstack-protector-all 具体采用了什么措施来保护堆栈安全。(提示 : movl %gs:20, %eax 会把一个随机值放入 EAX 中)。

(3). 分别画出 2.s 与 3.s 中活动记录栈图。两者在局部变量的布局顺序上有何差异?为何 GCC 要这么做?(提示 ： 思考当 i 为一个函数指针时的情形)。

2.c

```
void f(int a, int b) {
    int i = 1;
    int stack[10];
    stack[a] = b;
}
```

2.s

```
.LFB0:
    pushl   %ebp
    movl    %esp, %ebp
    subl $48, %esp
    movl    $1, -4(%ebp)
    movl    8(%ebp), %eax
    movl    12(%ebp), %edx
    movl    %edx,
-44(%ebp,%eax,4)
    leave
    ret
```

3.s

```
.LFB0:
    pushl   %ebp
    movl    %esp, %ebp
    subl $72, %esp
    movl    8(%ebp), %eax
    movl    %eax, -60(%ebp)
    movl    12(%ebp), %eax
    movl    %eax, -64(%ebp)
    movl    %gs:20, %eax
    movl    %eax, -12(%ebp)
    xorl    %eax, %eax
    movl    $1, -56(%ebp)
    movl    -60(%ebp), %eax
    movl    -64(%ebp), %edx
    movl    %edx, -52(%ebp,%eax,4)
    movl    -12(%ebp), %eax
    xorl    %gs:20, %eax
    je      .L2
    call    __stack_chk_fai
.L2:
    leave
```

解答

(1).**学生的解答：** a=12。X86 下为小端存储，stack[10]为 i 的地址，stack[11]为老控制链的地址，stack[12]为函数返回地址的地址。

张昱 20151222：X86 采用小端（little endian，即低地址存储一个整数的低位或数组的低下标元素）存储，stack[10]为 i 的存储单元，stack[11]为老控制链的存储单元，stack[12]为函数返回地址的存储单元。

(2). **学生的解答：** -fstack-protector-all 将一个随机值（防止黑客猜出）安插在 stack 末地址后的-12(%ebp)中。在循环调用 f(a,b)的过程中，会修改掉-12(%ebp)存放的值。在函数返回前执行 xorl %gs:20, %eax 所得结果非 0，使得 call __stack_chk_fail 发生。

张昱 20151222：使用-fstack-protector-all 选项编译生成的代码会将一个随机值（%gs:20 表示当前进程的段寄存器的值+20，防止黑客猜出）安插在 stack 数组之后的-12(%ebp)（即 stack[10]的存储位置）中。倘若黑客在循环调用 f(a,b)的过程中，会修改掉-12(%ebp)存放的值，那么在 f 函数返回前执行 xorl %gs:20, %eax 所得的结果会非 0，从而会执行 call __stack_chk_fail。

(3). **学生的解答**：活动记录栈图略。在局部变量布局顺序中，2.s 将 i 放在高地址，stack 数组放在低地址；3.s 将 i 放在低地址，stack 数组放在高地址。倘若按照 2.s 的布局，即使对控制链和返回地址加以保护，攻击者也可以通过溢出 stack 数组修改 i 的值。当 i 是函数指针时，这种攻击尤其致命，在函数中对 i 的任何脱引用都将使程序跳至攻击者想要执行的代码区域。

张昱 20151222：在 Linux 下，系统栈增长的方向是从高地址向低地址。不论是 2.s 还是 3.s，0(%ebp)为起址的 4 个字节（从 0(%ebp)到 3(%ebp)）存放的是控制链，即 f 的调用者的栈帧指针（或称活动记录的基址指针）；4(%ebp)是形参 a 的存储单元；8(%ebp)是形参 b 的存储单元；12(%ebp)是返回地址的存储单元。2.s 和 3.s 中的第 3 条汇编指令(subl)用于分配局部变量和临时单元，分别占 48 字节和 72 字节，分配的字节数是 8 字节的倍数，表示栈帧按 8 字节对齐。

针对 2.s，局部变量 i 存储在-4(%ebp)到-1(%ebp)中，stack 数组存储在-44(%ebp)到-5(%ebp)这 40 个字节中，其中-44(%ebp)为 stack[0]的存储单元的起址。即 i 放在高地址，stack 数组放在低地址。

针对 3.s，局部变量 i 存储在-56(%ebp)到-53(%ebp)中，stack 数组存储在-52(%ebp)到-13(%ebp)这 40 个字节中，其中-52(%ebp)为 stack[0]的存储单元的起址。即 i 放在低地址，stack 数组放在高地址。-12(%ebp)为起址的 4 字节用于存储随机值(%gs:20)。-60(%ebp)和-64(%ebp)是临时变量所占的空间，分别用于存放 b 和返回地址的值。

倘若按照 2.s 的存储布局，即使对控制链和返回地址加以保护，攻击者也可以通过溢出 stack 数组修改 i 的值。当 i 是函数指针时，这种攻击尤其致命，在函数中对 i 的任何脱引用都将使程序跳至攻击者想要执行的代码区域。