作业4

本次作业为《计算机系统概论》课的期中复习，其中包含5道书面题，每题8分。

Q1 溢出（8分）

有如下的C程序：

```c
#include <stdio.h>
int overflow(void);
int one = 1;
int overflow() {
    char buf[4];
    int val, i = 0;
    while (scanf("%x", &val) != EOF) buf[i++] = (char)val;
    return 15213;
}

int main() {
    int val = overflow();
    val += one;
    if (val != 15213)
        printf("Boom!\n");
    else
        printf("????\n");
    exit(0);
}
```

> **`scanf` 回顾**
>
> `scanf("%x", &val);` 函数从标准输入读取由空格分割的代表一个十六进制整数的字符或字符序列，并将该字符或字符序列转换为一个32位 `int`，将转换结果赋给 `val`。
>
> `scanf` 返回 `1` 表示转换成功，返回 `EOF`（即 `-1`）则表示 `stdin` 已经没有更多的输入序列了（通常输入 `Ctrl`+`D` 表示 `EOF`）。
>
> 例如，在输入字符串 `"0 a ff"` 上调用 `scanf` 四次将有以下结果：

1. 1st call: `val=0x0` and `scanf` returns `1`.
2. 2nd call: `val=0xa` and `scanf` returns `1`.
3. 3rd call: `val=0xff` and `scanf` returns `1`.
4. 4th call: `val=??` and `scanf` returns `EOF`.

对应的X86-64架构下的（反）汇编代码：

```
00000000004005d6 <overflow>:
 4005d6: 55                      push %rbp
 4005d7: 48 89 e5                mov %rsp, %rbp
 4005da: 48 83 ec 28             sub $0x28, %rsp
 4005db: 53                      push %rbx
 4005df: bb 00 00 00 00          mov $0x0, %ebx
 4005e4: eb 0d                   jmp 4005f3 <overflow+0x1d>
 4005e6: 48 63 c3                movslq %ebx, %rax
 4005e9: 8b 55 dc                mov -0x8(%rbp), %edx        # edx = val
 4005ec: 88 54 05 e0             mov %dl, -0x4(%rbp, %rax, 1) #buf
 4005f0: 8d 5b 01                lea 0x1(%rbx), %ebx         # i++
 4005f3: 48 8d 75 dc             lea -0x8(%rbp), %rsi
 4005f7: bf d4 06 40 00          mov $0x4006d4, %edi         # "%x"地址是 0x4006d4
 4005fc: b8 00 00 00 00          mov $0x0, %eax
 400601: e8 ba fe ff ff          callq 4004c0 <scanf>
 400606: 83 f8 ff                cmp $0xffffffff, %eax       # EOF 值为-1
 400609: 75 db                   jne 4005e6 <overflow+0x10>
 40060b: b8 6d 3b 00 00          mov $0x3b6d, %eax           # 0x3b6d=15213
 400610: 5b                      pop %rbx
 400614: 48 83 c4 28             add $0x28, %rsp
 400615: 5d                      pop %rbp
 400616: c3                      retq
0000000000400617 <main>:
 400617: 55                      push %rbp
 400618: 48 89 e5                mov %rsp, %rbp
 40061b: e8 b6 ff ff ff          callq 4005d6 <overflow>
 400620: 03 05 22 0a 20 00       add 0x200a22(%rip), %eax    # val += one
 400626: 3d 6d 3b 00 00          cmp $0x3b6d, %eax
 40062b: 74 0c                   je 400639 <main+0x22>
 40062d: bf d7 06 40 00          mov $0x4006d7, %edi
 400632: e8 69 fe ff ff          callq 4004a0 <puts>         # printf("Boom!\n")
 400637: eb 0a                   jmp 400643 <main+0x2c>
 400639: bf dd 06 40 00          mov $0x4006dd, %edi
 40063e: e8 5d fe ff ff          callq 4004a0 <puts>         # printf("????\n")
 400643: bf 00 00 00 00          mov $0x0, %edi
 400648: e8 43 fe ff ff          callq 400490 <exit>         # exit 不使用 rbp
```

1. 请参照 `buf[0]` 、 `buf[1]` 的地址表示方式表示表格内其余对象的地址（4分）。

Stack object	Address of stack object
return address	`&buf[0]` + _____
old `%rbp`	`&buf[0]` + _____
`buf[3]`	`&buf[0]` + _____
`buf[2]`	`&buf[0]` + _____

Stack object	Address of stack object
`buf[1]`	`&buf[0]` + 1
`buf[0]`	`&buf[0]` + 0

2. 为了让这个程序输出`"????"`，你需要在命令行输入什么字符串？以下每个下划线都是一个 1 位或 2 位的十六进制数字，下划线之间是空格（4分）。

__ __ __ __ __ __ __ __ __ __ __ __ __

Q2 结构体（8分）

有如下对应的C代码与对应的X86-64汇编代码：

```
struct matrix_entry {
    char a;
    char b;
    double d;
    short c;
};
struct matrix_entry matrix[5][__空格(1)__];

int return_entry(int i, int j) { return matrix[i][j].c; }
```

```
return_entry:
    movslq      %esi, %rsi
    movslq      %edi, %rdi
    leaq        (%rsi,%rsi,2), %rax
    leaq        0(,%rax,8), %rdx
    leaq        (%rdi,%rdi,4), %rax
    leaq        (%rdi,%rax,4), %rcx
    __空格(2)__  0(,%rcx,8), %rax
    movswl      matrix+ __空格(3)__(%rdx,%rax), %eax
    ret

    .comm       matrix, __空格(4)__
```

```
# .comm后的第二个参数表示matrix占据空间的大小，以字节为单位
```

请对照着填上代码中缺失的部分（数字请用十进制表示）。

空格	值
(1)	

空格	值
(2)	
(3)	
(4)	

Q3 跳转表（8分）

课堂上我们结合`switch`语句示例讲解了跳转表（jump table）的应用。课上的例子采用的是绝对地址定位的方式，即跳转表中的每一项存放的是各个代码块的绝对地址。后面我们学习了“共享库中的全局变量寻址”，知道共享库被不同进程装载的时候，其绝对地址是不一样的，这就给绝对地址定位的方式带来了难度。一种解决方式是“相对定位”方式，其依据的事实是：不管对象文件被装载到进程的哪个地址，代码段中的任一给定指令与数据段（包括只读数据段）中的任一给定位置之间的“距离”是一个常量。据此编译器生成了`switch`示例的与绝对地址无关（Position Independent Code）代码（X86-64架构）如下（右侧是对应的C函数），其中`.L4`标识了该跳转表。

```
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch (x) {
        case 1:
            w = y * z;
            break;
        case 2:
            w = y / z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

```
switch_eg:
        cmpq            $6, %rdi
        movq            %rdx, %rcx
        ja              .L8                             # __指令1__
        leaq            __空格1__(%rip), __空格2__
        movslq          (__空格3__, __空格4__,  4), %rdi
        __空格5__       %r8, %rdi
        jmp             *%rdi                           # 以%rdi为目标地址直接跳过去

        .section        .rodata                         # 只读数据段
.L4:
        .long           .L8-.L4
        .long           .L3-.L4
        .long           .L5-.L4
        .long           .L9-.L4
        .long           .L8-.L4
        .long           .L7-.L4
        .long           .L7-.L4
        .text                                           # 正文段
.L9:
        movl            __空格6__, %eax
        addq            %rcx, %rax
        ret
.L5:
        movq            %rsi, %rax
        cqto
        idivq           %rcx
        addq            %rcx, %rax
        ret
.L3:
        movq            %rdx, %rax
        imulq           %rsi, %rax
        ret
.L7:
        movl            $1, %eax
        subq            %rdx, %rax
        ret
.L8:
        movl            $2, %eax
        ret
```

请根据上述事实以及C函数语义，回答以下问题：

1. 请问为何 `__指令1__` 处用的是 `ja` 指令来进行带符号数的条件判断？（2分）

2. 填写空格处缺失的值以补齐指令（6分）：

空格	值
(1)	
(2)	
(3)	
(4)	
(5)	
(6)	

Q4 `echo` （8分）

下面是一段 C 代码以及对应的 X86-64 汇编：

```c
void echo() {
    char buf[8];
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushq   %rbx
    xorl    %eax, %eax
    subq    $16, %rsp
    leaq    8(%rsp), %rbx
    movq    %rbx, %rdi
    call    gets
    movq    %rbx, %rdi
    call    puts
    addq    $16, %rsp
    popq    %rbx
    ret
```

填充下面空格缺失的值：

1. gets 的参数通过寄存器 `__空格(1)__` 传递；
2. 寄存器 %rbx 属于 `__空格(2)__` (caller saved/callee saved) 寄存器；
3. 以 `subq` 指令后的 `%rsp` 计算，`buf` 数组的基地址是 `%rsp + __空格(3)__`，`%rbx` 寄存器保存在 `$rsp + __空格(4)__`；
4. 上面的代码有缓冲区溢出的漏洞，已知 `gets` 会从标准输入读入一行字符串，把字符串保存在其第一个参数指向的缓冲区，最后填入 `NULL`（亦即 `'/0'`）作为结尾。请计算，为了满足以下的要求，输入的字符串长度（不计入 `NULL`）需要满足的要求，以区间表示：

- 该字符串输入到上面的程序，`buf` 数组被更新但是缓冲区没有溢出：长度范围是 `[0, __空格(5)__]`
- 该字符串输入到上面的程序，保存在栈上的 `%rbx` 寄存器被更新，但是栈上的返回地址没有被更新：长度范围是 `[__空格6__, __空格7__]`
- 该字符串输入到上面的程序，保存在栈上的 `%rbx` 寄存器和返回地址被更新：长度范围是 `[__空格(8)__ , +inf)`

空格	值
(1)	
(2)	
(3)	
(4)	
(5)	
(6)	
(7)	
(8)	

Q5 Shellcoding（8分）

代码注入攻击（也称为Shellcoding），是在缓冲区溢出的基础上，向缓冲区注入攻击指令（专业术语为shellcode），通过修改函数的返回地址，实现代码的任意执行，步骤如下：

1. 攻击者找到可以缓冲区溢出攻击的函数：

```
int Q() {
    char buf[64];
    gets(buf); // 漏洞!
    ...
    return ...;
}
```

2. 攻击者构造一段输入，包括了shellcode，以及保存了shellcode的栈上地址；
3. 程序执行到 `int Q()` 处，读取攻击者构造的输入，此时返回地址被覆盖为栈上的地址，栈则保存了攻击指令；
4. `int Q()` 执行 `ret` 指令（对应于 `return` 语句），跳转到攻击者构造的攻击指令，进而执行攻击。

回答以下问题：

1. 请说明如何修改源代码，以修复缓冲区溢出的漏洞。（2分）

2. 即使不能修改代码，也有多种措施可以防御代码注入攻击。请结合上面的攻击流程，解释下面的措施为什么可以防御代码注入攻击：（4分）
    i. No-eXecute (NX)：把栈标记为不可执行
    ii. Stack Canaries：在栈上保存"金丝雀值"

3. ROP攻击利用程序已有的指令来攻击，它利用了 x86-64 指令的特性，即指令的后缀可能也是一个合法的指令。对于上一题提到的两个措施，它可以绕过哪一个？但是必不可以绕过哪一个？（2分）