人肉反编译练习题

ywy_c_asm

免责声明：皆为原创，仅作参考，如有雷同，绝非泄题！

温馨提示：①如果你认为题目可能有多个答案，其实，言之有理即可，真正的期末考试题会尽量避免或在判卷时容忍这种情况的。②部分题目难度可能会稍难于考试难度。

1.小 Y 写了一个简单的 C 语言程序，以下是 main 函数的汇编代码：

```
1  main:
2      push   %rbx
3      sub    $0x10,%rsp
4      movabs $0x726f57306c6c6568,%rax
5      mov    %rax,0x2(%rsp)
6      movl   $0x646c,0xa(%rsp)
7      mov    $0x0,%ebx
8  label1:
9      xor    %edi,%edi
10     movslq %ebx,%rax
11     mov    0x2(%rsp,%rax,1),%dil
12     test   %dil,%dil
13     je     label2 <main+0x4e>
14     callq  1050 <putchar@plt>
15     add    $0x1,%ebx
16     jmp    label1 <main+0x2c>
17 label2:
18     mov    $0x0,%eax
19     add    $0x10,%rsp
20     pop    %rbx
21     retq
```

(1)第 14 行标为绿色的函数调用，传入的参数存在_____里。

(2)该程序的 main 函数内仅有一个局部变量或数组，它的类型是_____。

(3)解释第 10 行和 11 行标为蓝色的两条指令的作用。

(4)解释第 4~6 行标为黄色的 3 条指令的作用，并结合你在本课程所学的程序优化相关知识，说明这样处理的好处。

(5)这个程序最终会输出什么？

2.小 Y 写了一个简单的 C 语言程序，以下是其中的一个非 void 函数 func 的反汇编代码：

```
func:
    sub    $0x1,%edi
    cmp    $0x8,%edi
    ja     label
    mov    $0x1,%eax
    retq
label:
    mov    $0x0,%eax
    retq
```

(1)这个函数的参数 arg 的类型是什么？

(2)分别写出 C 语言表达式 func(-1)、func(0)、func(1)、func(12)的值。

(3)这个函数体内仅有一条 C 语言语句，它可以是_____。

(4)如果把其中的指令 ja 换成 jg，则(2)中的表达式的值会变成什么？结合相关知识解释这种差异产生的原因。

3.小 Y 写了一个简单的 C 语言程序，以下是其中的一个非 void 函数 func 的汇编代码：

```
1   func:
2       sub     %edi,%esi
3       mov     %esi,%eax
4       shr     $0x1f,%eax
5       add     %esi,%eax
6       sar     $1,%eax
7       retq
```

(1)解释第 4 行标蓝指令的作用。

(2)请写出该函数的一个可能的 C 语言代码实现。

(3)你认为该函数可能有什么实际用途？

(4)小 Y 认为他的代码实现的很好，该函数全过程都没有进行内存访问，请你利用所学知识反驳他。

4.小 Y 写了一个简单的函数 int func(char* str)，功能为将输入的字符串 str 转换为十进制数字（例如：func(“12345”)=12345）。以下为 func 的反汇编代码：

```
1   func:
2       push   %rbx
3       mov    %rdi,%rbx
4       callq  1149 <strlen>
5       mov    %eax,%edi
6       mov    $0x0,%edx
7       mov    $0x0,%eax
8   label1:
9       cmp    %edi,%edx
10      jge    label2
11      lea    (%rax,%rax,4),%esi
12      xor    %rcx,%rcx
13      mov    (%rbx,%rdx,1),%cl
14      lea    -0x30(%rcx,%rsi,2),%eax
15      add    $0x1,%edx
16      jmp    label1
17  label2:
18      pop    %rbx
19      retq
```

(1)解释第 2 行和第 18 行标绿指令的作用，说明编译器为什么要生成这两条指令。

(2)解释第 11~14 行标蓝指令的作用，并针对第 11 行和第 14 行的两条 lea 指令，说明编译器这样做的好处。

(3)输入的参数 str 必须满足什么条件才能保证 func 的计算结果正确？

5. 小 Y 写了一个简单的 C 语言程序，以下是其中的一个非 void 函数 func 的汇编代码：

```
func:
    movslq %edi,%rdi
    add    (%rdx),%rdi
    sub    %rsi,%rdi
    mov    %rdi,(%rdx)
    add    (%rcx),%rdi
    movslq %r8d,%rax
    add    %rax,%rdi
    movslq %r9d,%rax
    add    %rdi,%rax
    sub    0x8(%rsp),%rax
    add    0x10(%rsp),%rax
    mov    %rax,(%rdx)
    retq
```

(1)已知 func 所有的参数都在函数体中被使用，func 的参数有_____个，它们的类型分别是_____。

(2)画出调用 func 时的栈结构（只考虑与 func 相关的部分）。

(3)请写出 func 函数的一个可能的 C 语言实现。

6.小 Y 写了一个简单的 C 语言程序，以下是 main 函数的 objdump 反汇编的结果：

```
0000000000003193 <main>:
    3193:   53                      push   %rbx
    3194:   48 83 ec 10             sub    $0x10,%rsp
    3198:   48 8d 5c 24 01          lea    0x1(%rsp),%rbx
    319d:   48 89 df                mov    %rbx,%rdi
    31a0:   b8 00 00 00 00          mov    $0x0,%eax
    31a5:   e8 c6 fe ff ff          callq  3070 <gets@plt>
    31aa:   48 89 df                mov    %rbx,%rdi
    31ad:   e8 ae fe ff ff          callq  3060 <puts@plt>
    31b2:   b8 00 00 00 00          mov    $0x0,%eax
    31b7:   48 83 c4 10             add    $0x10,%rsp
    31bb:   5b                      pop    %rbx
    31bc:   c3                      retq
```

(1)main 中唯一的局部变量的类型为_____，它在运行时栈中的地址是_____。

(2)地址 0x31aa 处的指令能否删去？结合所学知识阐述你的观点。

(3)这个程序的作用是什么？

(4)这个程序在安全性上有何缺陷？结合所学知识，阐述至少两条改进策略。

(5)小 Y 的程序里还有另外一个简单的函数 void func()，这个函数的作用仅仅是显示一个字符串，以下为 func 的 objdump 反汇编结果：

```
000000000000316d <func>:
    316d:   48 83 ec 18             sub    $0x18,%rsp
    3171:   48 b8 48 61 63 6b 65    movabs $0x64656b636148,%rax
    3178:   64 00 00
    317b:   48 89 44 24 08          mov    %rax,0x8(%rsp)
    3180:   48 8d 7c 24 08          lea    0x8(%rsp),%rdi
    3185:   e8 d6 fe ff ff          callq  3060 <puts@plt>
    318a:   48 83 c4 18             add    $0x18,%rsp
    318e:   c3                      retq
```

很显然，正常情况下 main 根本不会调用到 func，func 根本不会被执行。假如你是用户（显然你不可能修改程序代码），你正在运行小 Y 写的这个程序，请你构造该程序的一个可行的用户输入，使得 func 能够被执行，并结合所学知识解释你的构造策略。(在本题中，我们姑且假设 main 函数是被地址 0x5678 处的__libc_start_main 调用的)

7.小 Y 写了一个简单的 C 语言程序，作用是计算一个数组的和并输出。它的 3 个函数 **main**、**sum**（非 **void**）、**print** 的反汇编结果如下所示：

```
main:
    mov    $0x4,%esi
    mov    $0x4020,%rdi
    callq  116e <sum>
    mov    $0x0,%eax
    retq
```

```
sum:
    push   %rbx
    mov    $0x0,%eax
    mov    $0x0,%ebx
label1:
    cmp    %esi,%eax
    jge    label2
    movslq %eax,%rdx
    add    (%rdi,%rdx,4),%ebx
    add    $0x1,%eax
    jmp    label1
label2:
    mov    %ebx,%edi
    callq  1149 <print>
    mov    %ebx,%eax
    pop    %rbx
    retq
```

```
print:
    mov    %edi,%esi
    mov    $0x4010,%rdi
    callq  1050 <__printf@plt>
    retq
```

在程序运行时，虚拟地址 **0x4000** 开始处存储的 64 个字节如图所示：

```
4000    00 00 00 00 00 00 00 00 08 40 00 00 00 00 00 00
4010    72 65 73 20 69 73 20 25 64 00 00 00 00 00 00 00
4020    34 02 00 00 56 04 00 00 01 00 00 00 00 10 00 00
4030    00 00 00 00 00 00 00 01 02 10 00 00 00 00 00 01
```

(1)画出 **print** 函数执行时的栈结构。

(2)函数 **sum** 的参数类型为_____，返回值类型为_____。

(3)写出程序的输出结果。

(4)假如使用某种方式，将地址 **0x4020** 处开始的 16 个字节改为：

```
4020    00 00 00 80 00 00 00 00 00 00 00 80 00 00 00 00
```

那么程序的输出结果将会变为什么？为什么会得到这样的结果？假如小 Y 仍然想达到他的编程目的，请你给小 Y 提一些改进建议（假设该程序不涉及无符号整数）。

8.小 Y 写了一个简单的 C 语言程序，以下是它的函数 main、func 的反汇编结果：

```
main:
    sub    $0x10,%rsp
    movl   $0x2,0xc(%rsp)
    lea    0xc(%rsp),%rdi
    callq  1149 <func>
    mov    0xc(%rsp),%edx
    mov    $0x0,%eax
    add    $0x10,%rsp
    retq
```

```
func:
    mov    $0x5,%eax
label1:
    cmp    $0x0,%eax
    jb     label2
    shll   (%rdi)
    sub    $0x1,%eax
    jmp    label1
label2:
    retq
```

(1)函数 func 唯一的参数类型为_____，函数 main 中仅有一个局部变量，它的类型为_____，分析这个局部变量的存储位置，并利用所学知识解释编译器这样处理的原因。

(2)小 Y 发现该程序运行时行为异常，请你结合所学知识分析原因。

(3)请写出该程序的一种可能的 C 语言源代码实现。

9.小 Y 写了一个简单的 C 语言程序，其中的一个函数 long func(long x)的反汇编结果如下所示：

```
func:
    mov    %edi,%eax
    mov    %rdi,-0x8(%rsp)
    mov    $0xff,%al
    sub    %ah,%al
    mov    %al,-0x7(%rsp)
    mov    -0x8(%rsp),%rax
    retq
```

(1)画出 func 执行时的栈结构。

(2)写出 func(0x3f3f3f3f)的结果在内存中从低地址到高地址的字节表示。

(3)请你写出一个效果与 func 等价的函数 long func1(long x)，要求：函数体中只能有一条 C 语言语句，且不能使用指针操作。

答案解析

1.小 Y 写了一个简单的 C 语言程序，以下是 main 函数的汇编代码：

```
1  main:
2      push   %rbx
3      sub    $0x10,%rsp
4      movabs $0x726f57306c6c6568,%rax
5      mov    %rax,0x2(%rsp)
6      movl   $0x646c,0xa(%rsp)
7      mov    $0x0,%ebx
8  label1:
9      xor    %edi,%edi
10     movslq %ebx,%rax
11     mov    0x2(%rsp,%rax,1),%dil
12     test   %dil,%dil
13     je     label2 <main+0x4e>
14     callq  1050 <putchar@plt>
15     add    $0x1,%ebx
16     jmp    label1 <main+0x2c>
17 label2:
18     mov    $0x0,%eax
19     add    $0x10,%rsp
20     pop    %rbx
21     retq
```

(1)第 14 行标为绿色的函数调用，传入的参数存在___edi(或 rdi)_____里。

(putchar 就一个参数)

(2)该程序的 main 函数内仅有一个局部变量或数组，它的类型是___char[]_____。

(见(3)和(4)的分析)

(3)解释第 10 行和 11 行标为蓝色的两条指令的作用。

指令 movslq %ebx,%rax 将 ebx 作 64 位符号扩展存到 rax 中。

指令 mov 0x2(%rsp,%rax,1),%dil 将地址为 rsp+2+rax*1 的内存单元的字节取到 dil 寄存器中。

这两条指令的作用是将基址为 rsp+2 的局部 char 数组的 ebx 下标处的字符取到 dil 中，由于 ebx 是 32 位，参与地址计算前需要进行 64 位符号扩展。

(4)解释第 4~6 行标为黄色的 3 条指令的作用，并结合你在本课程所学的程序优化相关知识，说明这样处理的好处。

这 3 条指令的作用是给基址为 rsp+2 的局部 char[]数组赋初值，通过对 64 位和 32 位数据的传送，使得 12 个字节（字符）68 65 6c 6c 30 57 6f 72 6c 64 00 00 在运行时被动态地加载到局部数组中。使用 64 位或 32 位的传送指令能够一次操作传送多个字符数据，比一个字符一个字符地传送的时间效率高。

(5)这个程序最终会输出什么？

我们可以分析出 main 函数内存在一个典型的 for 循环结构，循环变量是 ebx，一开始将 ebx 赋初值 0，每次将 ebx+=1，并且取局部字符数组的第 ebx 个字符传给 edi，调用 putchar 输出，循环结束条件见 12 行和 13 行的条件转移指令 je，当字符（dil）为 0 时循环结束。因此这个程序的行为就是输出局部 char[]存的字符串。分析其 ASCII 码可以得到“hell0world”。

2.小 Y 写了一个简单的 C 语言程序，以下是其中的一个非 void 函数 func 的反汇编代码：

```
func:
    sub     $0x1,%edi
    cmp     $0x8,%edi
    ja      label
    mov     $0x1,%eax
    retq
label:
    mov     $0x0,%eax
    retq
```

(1)这个函数的参数 arg 的类型是什么？它对 edi 进行操作，因此参数类型为 32 位整数 int/unsigned int。

(2)分别写出 C 语言表达式 func(-1)、func(0)、func(1)、func(12)的值。

注意，ja 是无符号意义下的大于跳转，考虑"参数减 1"在无符号意义下是不是大于 8 的，-1-1=-2=(unsigned)2^32-2，大于 8 返回 0；0-1=-1=(unsigned)2^32-1，大于 8 返回 0；1-1=0，小于等于 8 返回 1；12-1=11，大于 8 返回 0。

因此 func(-1)=0，func(0)=0，func(1)=1，func(12)=0。

(3)这个函数体内仅有一条 C 语言语句，它可以是 return (unsigned)arg-1<=8 。

或者 return arg-1u<=8，或者 return (arg-1<=8u)?1:0，总而言之必须是无符号意义下的比较。

(4)如果把其中的指令 ja 换成 jg，则(2)中的表达式的值会变成什么？结合相关知识解释这种差异产生的原因。

这里考察对无符号比较跳转 ja 和有符号比较跳转 jg 的区别的理解，换成 jg，那么就是有符号意义下的比较，函数就变成了 return (int)arg-1<=8，那么 func(-1)=1，func(0)=1，func(1)=1，func(12)=0。ja 与 jg 差异的本质原因是它们所判断的标志位不同，ja 的转移条件是 CF==0，而 jg 的转移条件是(SF!=0||OF!=0)&&ZF==0，前者是为无符号整数设计的规则，而后者是为有符号整数设计的规则。

3.小 Y 写了一个简单的 C 语言程序，以下是其中的一个非 void 函数 func 的汇编代码：

```
1   func:
2       sub     %edi,%esi
3       mov     %esi,%eax
4       shr     $0x1f,%eax
5       add     %esi,%eax
6       sar     $1,%eax
7       retq
```

(1)解释第 4 行标蓝指令的作用。将 eax 逻辑右移 31 位，相当于把 eax 变成它符号位的 01 数值，等价于 C 语言表达式 eax=(eax<0)。

(2)请写出该函数的一个可能的 C 语言代码实现。

```
int func(int a,int b){
    return (b-a+(b-a<0))/2;
}
```

两个参数 a 和 b，分别通过 edi 和 esi 传入，首先计算 eax=b-a，再通过 shr 指令计算表达式 b-a<0 的值，把它们的和通过算术右移除以 2（向下取整），通过 eax 返回。

在这里务必要注意逻辑右移与算术右移的区别!

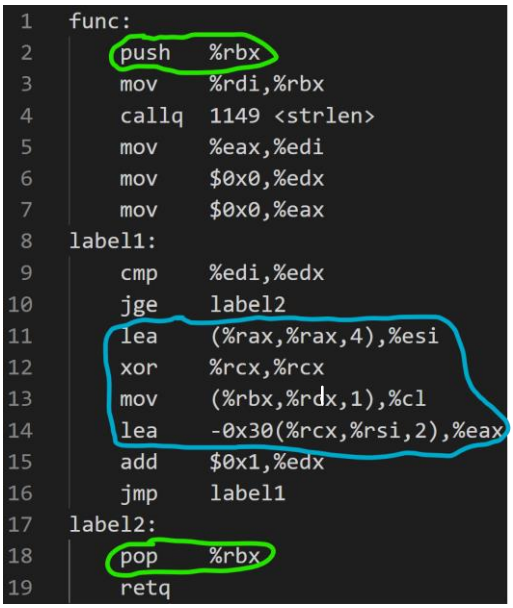
(3)你认为该函数可能有什么实际用途?

计算两数差值的一半，结果向 0 取整。可能会被用来求整数区间的中点。(这个 idea 来源于拆炸弹遇到的取区间中点的操作)

(4)小 Y 认为他的代码实现的很好，该函数全过程都没有进行内存访问，请你利用所学知识反驳他。

①CPU 在执行指令的时候必然要从内存中取出 rip 指向的机器码。②ret 指令进行了弹栈操作，也要做内存访问。

4.小 Y 写了一个简单的函数 int func(char* str)，功能为将输入的字符串 str 转换为十进制数字（例如：func(“12345”)=12345）。以下为 func 的反汇编代码：



```
1  func:
2      push   %rbx
3      mov    %rdi,%rbx
4      callq  1149 <strlen>
5      mov    %eax,%edi
6      mov    $0x0,%edx
7      mov    $0x0,%eax
8  label1:
9      cmp    %edi,%edx
10     jge    label2
11     lea    (%rax,%rax,4),%esi
12     xor    %rcx,%rcx
13     mov    (%rbx,%rdx,1),%cl
14     lea    -0x30(%rcx,%rsi,2),%eax
15     add    $0x1,%edx
16     jmp    label1
17 label2:
18     pop    %rbx
19     retq
```

(1)解释第 2 行和第 18 行标绿指令的作用，说明编译器为什么要生成这两条指令。

指令 push %rbx 将寄存器 rbx 的值入栈，指令 pop %rbx 将栈顶元素弹出到 rbx 中。这样做是因为 func 中使用了寄存器 rbx，而 x86-64 规定了 rbx 是被调用者保存的寄存器，因此 func 需要使用栈暂时保存 rbx 原来的值。

(2)解释第 11~14 行标蓝指令的作用，并针对第 11 行和第 14 行的两条 lea 指令，说明编译器这样做的好处。

指令 lea (%rax,%rax,4),%esi 使得 esi(rsi)=rax+rax*4=rax*5。

指令 xor %rcx,%rcx 将寄存器 rcx 清零。

指令 mov (%rbx,%rdx,1),%cl 将基址为 rbx 的字符数组的下标为 rdx 的字符 ASCII 码传入 cl，此时 rcx 的值也就是这个 ASCII 码。

指令 lea -0x30(%rcx,%rsi,2),%eax 使得 eax=rsi*2+rcx-0x30。

因此这 4 条指令的作用就是实现了这样一个运算：eax=eax*10+str[rdx]-0x30（假设 str 为基址为 rbx 的字符数组）。这两条 lea 指令不仅方便地实现了四则运算，使用了较少的指令，并且它们使用地址加法器内的左移运算实现乘法，更快。

(3)输入的参数 str 必须满足什么条件才能保证 func 的计算结果正确?

首先，str 内的所有字符都必须是字符’0’~’9’（毕竟 func 不加判断地机械地使用“数字字符 ASCII 码-0x30=数字”的操作）；其次，由于函数的返回值是 int（通过 eax 而不是 rax），需要保证 str 表示的数字不超过 2^31-1，否则会由于溢出导致计算结果不正确。

5. 小 Y 写了一个简单的 C 语言程序，以下是其中的一个非 void 函数 func 的汇编代码：

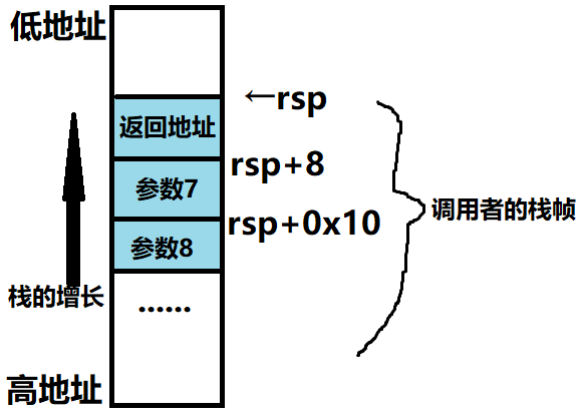
```
func:
    movslq %edi,%rdi
    add    (%rdx),%rdi
    sub    %rsi,%rdi
    mov    %rdi,(%rdx)
    add    (%rcx),%rdi
    movslq %r8d,%rax
    add    %rax,%rdi
    movslq %r9d,%rax
    add    %rdi,%rax
    sub    0x8(%rsp),%rax
    add    0x10(%rsp),%rax
    mov    %rax,(%rdx)
    retq
```

(1)已知 func 所有的参数都在函数体中被使用，func 的参数有__8__个，它们的类型分别是_int,long,long*,long*,int,int,long,long_。

本题考察复杂情况下的参数传递。注意到这个函数使用了所有 6 个传参寄存器 rdi rsi rdx rcx r8 r9，甚至在后面还用了两个栈内的元素作为源操作数，因此该函数有 8 个参数，最后两个参数用栈传递。

根据 movslq %edi,%rdi 的源操作数可以得知对于参数 1，func 只关心它的低 32 位，因此参数 1 是 int，同理可知 r8d 传递的参数 5 和 r9d 传递的参数 6 也是 int。而由 add (%rdx),%rdi 可知 rdx 传递的参数 3 是指针，并且指向 64 位内存单元，所以参数 3 是 long*，同理参数 4 也是 long*。其余参数都完整地参与了 64 位运算，都是 long。

(2)画出调用 func 时的栈结构（只考虑与 func 相关的部分）。



(3)请写出 func 函数的一个可能的 C 语言实现。

```
long func(int arg1,long arg2,long* arg3,long* arg4,int arg5,int arg6,long arg7,long arg8){
    *arg3+=arg1;
    *arg3-=arg2;
    *arg3+=*arg4;
    *arg3+=arg5;
    *arg3+=arg6;
    *arg3-=arg7;
    *arg3+=arg8;
    return *arg3;
}
```

6.小 Y 写了一个简单的 C 语言程序，以下是 main 函数的 objdump 反汇编的结果：

```
0000000000003193 <main>:
    3193:   53                      push   %rbx
    3194:   48 83 ec 10             sub    $0x10,%rsp
    3198:   48 8d 5c 24 01          lea    0x1(%rsp),%rbx
    319d:   48 89 df                mov    %rbx,%rdi
    31a0:   b8 00 00 00 00          mov    $0x0,%eax
    31a5:   e8 c6 fe ff ff          callq  3070 <gets@plt>
    31aa:   48 89 df                mov    %rbx,%rdi
    31ad:   e8 ae fe ff ff          callq  3060 <puts@plt>
    31b2:   b8 00 00 00 00          mov    $0x0,%eax
    31b7:   48 83 c4 10             add    $0x10,%rsp
    31bb:   5b                      pop    %rbx
    31bc:   c3                      retq
```

(1)main 中唯一的局部变量的类型为__char[]__，它在运行时栈中的地址是__rsp+1__。

(通过 gets 和 puts 的参数传递，以及 lea 0x1(%rsp),%rbx)

(2)地址 0x31aa 处的指令能否删去？结合所学知识阐述你的观点。

不能，尽管前面已经有一条 mov %rbx,%rdi，但 rdi 并非被调用者保存的寄存器，在调用 gets 后它的值可能会发生变化，所以需要对 rdi 重新赋值。

(3)这个程序的作用是什么？

通过 gets 将用户输入的字符串保存到基址为 rsp+1 的局部字符数组中，再用 puts 将该字符串输出。

(4)这个程序在安全性上有何缺陷？结合所学知识，阐述至少两条改进策略。

该程序使用了局部数组作为缓冲区，而 gets 属于缓冲区不安全的函数，在将用户输入的字符加载到缓冲区时并不会检查越界，可能会产生缓冲区溢出，破坏 main 的栈帧。

改进策略：①使用更加安全的函数（fgets）替换 gets，②在编译时加入栈保护选项，使编译器加入基于金丝雀的栈破坏检测代码。（实际上这个代码是我故意用-fno-stackprotector 选项生成的 hhh）

(5)小 Y 的程序里还有另外一个简单的函数 void func()，这个函数的作用仅仅是显示一个字符串，以下为 func 的 objdump 反汇编结果：

```
000000000000316d <func>:
    316d:   48 83 ec 18             sub    $0x18,%rsp
    3171:   48 b8 48 61 63 6b 65    movabs $0x64656b636148,%rax
    3178:   64 00 00
    317b:   48 89 44 24 08          mov    %rax,0x8(%rsp)
    3180:   48 8d 7c 24 08          lea    0x8(%rsp),%rdi
    3185:   e8 d6 fe ff ff          callq  3060 <puts@plt>
    318a:   48 83 c4 18             add    $0x18,%rsp
    318e:   c3                      retq
```

很显然，正常情况下 main 根本不会调用到 func，func 根本不会被执行。假如你是用户（显然你不可能修改程序代码），你正在运行小 Y 写的这个程序，请你构造该程序的一个可行的用户输入，使得 func 能够被执行，并结合所学知识解释你的构造策略。(在本题中，我们姑且假设 main 函数是被地址 0x5678 处的__libc_start_main 调用的)

实际上对于这个图我们只需要知道 func 的函数地址是 0x316d，根据缓冲区溢出攻击的原理，需要构造能够溢出缓冲区的字符串输入使得返回地址被数据 0x000000000000316d 覆盖，而原来的返回地址是 0x0000000000005xxx，所以仅修改最低位置的两个字节为 0x6d 和 0x31（它们分别为'm'和'1'的 ASCII 码）即可。分析栈结构可知，输入字符串存放位置从 rsp+1 开始，而返回地址的位置是 rsp+16，那么可以输入一个长度 17 的字符串，最后两个字符是'm'和'1'。

7.小 Y 写了一个简单的 C 语言程序，作用是计算一个数组的和并输出。它的 3 个函数 main、sum（非 void）、print 的反汇编结果如下所示：

```
main:
    mov    $0x4,%esi
    mov    $0x4020,%rdi
    callq  116e <sum>
    mov    $0x0,%eax
    retq
```

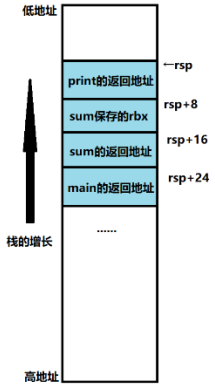
```
sum:
    push   %rbx
    mov    $0x0,%eax
    mov    $0x0,%ebx
label1:
    cmp    %esi,%eax
    jge    label2
    movslq %eax,%rdx
    add    (%rdi,%rdx,4),%ebx
    add    $0x1,%eax
    jmp    label1
label2:
    mov    %ebx,%edi
    callq  1149 <print>
    mov    %ebx,%eax
    pop    %rbx
    retq
```

```
print:
    mov    %edi,%esi
    mov    $0x4010,%rdi
    callq  1050 <__printf@plt>
    retq
```

在程序运行时，虚拟地址 0x4000 开始处存储的 64 个字节如图所示：

```
4000    00 00 00 00 00 00 00 00 08 40 00 00 00 00 00 00
4010    72 65 73 20 69 73 20 25 64 00 00 00 00 00 00 00
4020    34 02 00 00 56 04 00 00 01 00 00 00 00 10 00 00
4030    00 00 00 00 00 00 00 01 02 10 00 00 00 00 00 01
```

(1)画出 print 函数执行时的栈结构。



(2)函数 sum 的参数类型为__int*,int__，返回值类型为_int__。

(3)写出程序的输出结果。

考虑 print 对 printf 的调用，rdi 传入的格式化字符串在 0x4010 处，通过对这里存的字节进行 ASCII 码解析得到字符串“res is %d”，而输出的数通过追溯可以发现是 sum 中

的作为累加结果的 ebx，而 main 对 sum 的调用传入地址 0x4020 和 4，通过分析 sum 中典型的 for 循环结构可知是对 0x4020 处存的 4 个 int 进行求和，那么 4 个字节一个 int，注意小端序，可知这 4 个数是 0x234、0x456、0x1、0x1000，加起来是 0x168b，即 5771，所以程序输出结果是 res is 5771。

(4)假如使用某种方式，将地址 0x4020 处开始的 16 个字节改为：

```
4020    00 00 00 80 00 00 00 00 00 00 00 80 00 00 00 00
```

那么程序的输出结果将会变为什么？为什么会得到这样的结果？假如小 Y 仍然想达到他的编程目的，请你给小 Y 提一些改进建议（假设该程序不涉及无符号整数）。

sum 在进行两个 0x80000000（-2147483648）相加的时候会发生溢出，最终得到的 int 结果是 0，程序输出 res is 0。如果仍然想输出正确的结果，那么应当在程序中使用 long 进行计算和输出。

8.小 Y 写了一个简单的 C 语言程序，以下是它的函数 main、func 的反汇编结果：

```
main:
    sub    $0x10,%rsp
    movl   $0x2,0xc(%rsp)
    lea    0xc(%rsp),%rdi
    callq  1149 <func>
    mov    0xc(%rsp),%edx
    mov    $0x0,%eax
    add    $0x10,%rsp
    retq
```

```
func:
    mov    $0x5,%eax
label1:
    cmp    $0x0,%eax
    jb     label2
    shll   (%rdi)
    sub    $0x1,%eax
    jmp    label1
label2:
    retq
```

(1)函数 func 唯一的参数类型为__int*(通过 shll (%rdi)判断)__，函数 main 中仅有一个局部变量，它的类型为__int__，分析这个局部变量的存储位置，并利用所学知识解释编译器这样处理的原因。

这个局部变量存在 main 的栈帧里，而不是通常情况下的寄存器。这是因为 main 需要将该局部变量的地址作为参数传入 func，若存在寄存器中则无法实现取地址，因此编译器不得不将其存入栈中。

(2)小 Y 发现该程序运行时行为异常，请你结合所学知识分析原因。

这个程序会死循环，问题出在函数 func 的指令 cmp $0x0,%eax 和 jb label2 上。jb 是无符号意义下的小于跳转，条件是标志位 CF=1，而 cmp 计算 eax-0 的值，不可能产生进位，CF=0（亦即在无符号意义下所有数都不可能<0），因此 jb 永远都不会跳转，所以 func 的循环永远都不会结束。(这是一个典型的“无符号>=0”的 for 死循环)

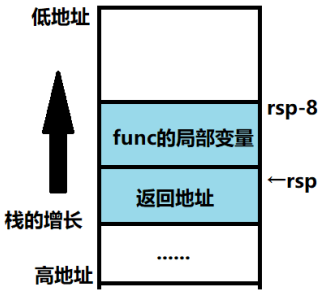
(3)请写出该程序的一种可能的 C 语言源代码实现。

```
void func(int* ptr){
    for(int i=5;i>=0u;i--)
        *ptr+=*ptr;
}
int main(){
    int var;
    func(&var);
    return 0;
}
```

9.小 Y 写了一个简单的 C 语言程序，其中的一个函数 long func(long x)的反汇编结果如下所示：

```
func:
    mov    %edi,%eax
    mov    %rdi,-0x8(%rsp)
    mov    $0xff,%al
    sub    %ah,%al
    mov    %al,-0x7(%rsp)
    mov    -0x8(%rsp),%rax
    retq
```

(1)画出 func 执行时的栈结构。



(2)写出 func(0x3f3f3f3f)的结果在内存中从低地址到高地址的字节表示。

3f c0 3f 3f 00 00 00 00　提示：该函数将输入的 long 的第 2 个字节用 255 减，相当于取反。记得高位补 0!

(3)请你写出一个效果与 func 等价的函数 long func1(long x)，要求：函数体中只能有一条 C 语言语句，且不能使用指针操作。

```
long func1(long x){
    return x^0xff00;
}
```

使用异或将 x 的第 2 个字节（第 8~15 位）取反。