華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础实验

实验名称： 二进制程序分析

院　　系 ：计算机科学与技术

专业班级 ：CS2209

学　　号 ：U202210755

姓　　名 ：章宏宇

指导教师 ：李专

2024 年 4 月 2 日

一、实验目的与要求

通过逆向分析一个二进制程序（称为“二进制炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示各方面知识点的理解，增强反汇编、跟踪、分析、调试等能力。

实验环境：Ubuntu，GCC，GDB 等。

二、实验内容

作为实验目标的二进制炸弹（binary bombs）可执行程序由多个“关”组成。每一个“关”（阶段）要求输入一个特定字符串，如果输入满足程序代码的要求，该阶段即通过，否则程序输出失败。实验的目标是设法得到得出解除尽可能多阶段的字符串。

为了完成二进制炸弹的拆除任务，需要通过反汇编和分析跟踪程序每一阶段的机器代码，从中定位和理解程序的主要执行逻辑，包括关键指令、控制结构和相关数据变量等等，进而推断拆除炸弹所需要的目标字符串。

实验源程序及相关文件：

bomb.c　　主程序

phases.o　各个阶段的目标程序

support.c 完成辅助功能的目标程序

phases.h　support.h　公共头文件

阶段 1：　串比较 phase_1(char *input);

要求输出的字符串(input) 与程序中内置的某一特定字符串相同。提示：找到与 input 串相比较的特定串的地址，查看相应单元中的内容，从而确定 input 应输入的串。

阶段 2：循环 phase_2(char *input);

要求在一行上输入 6 个整数数据，与程序自动产生的 6 个数据进行比较，若一致，则过关。提示：将输入串 input 拆分成 6 个数据由函数 read_six_numbers(input, numbers) 完成。之后是各个数据与自动产生的数据的比较，在比较中使用了循环语句。

阶段 3：条件分支 phase_3(char *input);

要求输入一个整数数据，该数据与程序自动生成的 一个数据比较，相等则过关。提示：在自动生成数据时，使用了 switch … case 语句。

阶段 4：递归调用和栈 phase_4(char *input);

要求在一行中输入两个数，第一个数表示在一个有序的数组（或者 binary search tree）中需要搜索到的数，该数是在一定范围之内的；第二个数表示找到搜索数的路径（在树的左边搜索编码为二进制位 0，在树的有边搜索编码为二进制位 1）。

阶段 5：指针和数组访问 phase_5(char *input);

要求在一行中输入一个串，该串与程序自动生成的串相同。在生成串和比较串时，使用了数组和指针。

阶段 6：链表、结构、指针的访问 phase_6(char *input);

要求在一行中输入 6 个数，这 6 个数是一个链表中结点的顺序号（从 1 到 6）。按照输入

的顺序号，将对应链表结点中的值形成一个数组。若该数组是按照降序排列的，则过关。

三、实验记录及问题回答

（1）实验任务 1 的实验记录

1.phase_1(char *input):

思路：根据提示，可知程序中内置了一个特定字符串，猜测应为字符串常量，保存在读写数据段（静态数据区），所以只需要反汇编后找到对应的访问静态数据区指令，根据对应地址查看内存空间即可。

过程：首先，gdb 调试，命令行输入 gdb bomb 进入调试。然后用反汇编命令 disass /rs phase_1 查看 phase_1 的反汇编代码：

```
(gdb) disass /rs phase_1
Dump of assembler code for function phase_1:
   0x00005555555559fc <+0>:     f3 0f 1e fa     endbr64
   0x0000555555555a00 <+4>:     55      push   rbp
   0x0000555555555a01 <+5>:     48 89 e5        mov    rbp,rsp
   0x0000555555555a04 <+8>:     48 83 ec 10     sub    rsp,0x10
   0x0000555555555a08 <+12>:    48 89 7d f8     mov    QWORD PTR [rbp-0x8],rdi
   0x0000555555555a0c <+16>:    48 8b 45 f8     mov    rax,QWORD PTR [rbp-0x8]
   0x0000555555555a10 <+20>:    48 8d 35 01 19 00 00    lea    rsi,[rip+0x1901]        # 0x555555557318
   0x0000555555555a17 <+27>:    48 89 c7        mov    rdi,rax
   0x0000555555555a1a <+30>:    e8 e0 fb ff ff  call   0x5555555555ff <strings_not_equal>
   0x0000555555555a1f <+35>:    85 c0   test   eax,eax
   0x0000555555555a21 <+37>:    74 05   je     0x555555555a28 <phase_1+44>
   0x0000555555555a23 <+39>:    e8 d6 fe ff ff  call   0x5555555558fe <explode_bomb>
   0x0000555555555a28 <+44>:    90      nop
   0x0000555555555a29 <+45>:    c9      leave
   0x0000555555555a2a <+46>:    c3      ret
```

观察代码，结合任务提示，发现〈phase_1+20〉中出现了静态数据区的地址。故用内存查看命令 x /40cb 0x555555557318 查看对应的内存，结果如下：

```
(gdb) x /40cb 0x555555557318
0x555555557318: 84 'T'  104 'h' 101 'e' 32 ' '  102 'f' 117 'u' 116 't' 117 'u'
0x555555557320: 114 'r' 101 'e' 32 ' '  119 'w' 105 'i' 108 'l' 108 'l' 32 ' '
0x555555557328: 98 'b'  101 'e' 32 ' '  98 'b'  101 'e' 116 't' 116 't' 101 'e'
0x555555557330: 114 'r' 32 ' '  116 't' 111 'o' 109 'm' 111 'o' 114 'r' 114 'r'
0x555555557338: 111 'o' 119 'w' 46 '.'  0 '\000'        37 '%'  100 'd' 32 ' '  37 '%'
```

故可得特定字符串为 “The future will be better tomorrow.”

2.phase_2(char *input):

思路：根据提示，先反汇编出机器指令，找到调用 read_six_numbers(input, numbers) 的指令，再找到 cmp 机器指令，打上断点。通过多次输入数据，找到规律。

过程：首先，gdb 调试，查看反汇编代码：

```
(gdb) disass /rs phase_2
Dump of assembler code for function phase_2:
   0x0000555555555a2b <+0>:     f3 0f 1e fa     endbr64
=> 0x0000555555555a2f <+4>:     55      push   rbp
   0x0000555555555a30 <+5>:     48 89 e5        mov    rbp,rsp
   0x0000555555555a33 <+8>:     48 83 ec 40     sub    rsp,0x40
   0x0000555555555a37 <+12>:    48 89 7d c8     mov    QWORD PTR [rbp-0x38],rdi
   0x0000555555555a3b <+16>:    64 48 8b 04 25 28 00 00 00      mov    rax,QWORD PTR fs:0x28
   0x0000555555555a44 <+25>:    48 89 45 f8     mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555a48 <+29>:    31 c0   xor    eax,eax
   0x0000555555555a4a <+31>:    48 8d 55 e0     lea    rdx,[rbp-0x20]
   0x0000555555555a4e <+35>:    48 8b 45 c8     mov    rax,QWORD PTR [rbp-0x38]
   0x0000555555555a52 <+39>:    48 89 d6        mov    rsi,rdx
   0x0000555555555a55 <+42>:    48 89 c7        mov    rdi,rax
   0x0000555555555a58 <+45>:    e8 fd fa ff ff  call   0x55555555555a <read_six_numbers>
   0x0000555555555a5d <+50>:    8b 45 e0        mov    eax,DWORD PTR [rbp-0x20]
   0x0000555555555a60 <+53>:    85 c0   test   eax,eax
   0x0000555555555a62 <+55>:    79 05   jns    0x555555555a69 <phase_2+62>
   0x0000555555555a64 <+57>:    e8 95 fe ff ff  call   0x5555555558fe <explode_bomb>
   0x0000555555555a69 <+62>:    c7 45 dc 01 00 00 00    mov    DWORD PTR [rbp-0x24],0x1
   0x0000555555555a70 <+69>:    eb 27   jmp    0x555555555a99 <phase_2+110>
   0x0000555555555a72 <+71>:    8b 45 dc        mov    eax,DWORD PTR [rbp-0x24]
   0x0000555555555a75 <+74>:    48 98   cdqe
   0x0000555555555a77 <+76>:    8b 54 85 e0     mov    edx,DWORD PTR [rbp+rax*4-0x20]
   0x0000555555555a7b <+80>:    8b 45 dc        mov    eax,DWORD PTR [rbp-0x24]
   0x0000555555555a7e <+83>:    83 e8 01        sub    eax,0x1
   0x0000555555555a81 <+86>:    48 98   cdqe
   0x0000555555555a83 <+88>:    8b 4c 85 e0     mov    ecx,DWORD PTR [rbp+rax*4-0x20]
   0x0000555555555a87 <+92>:    8b 45 dc        mov    eax,DWORD PTR [rbp-0x24]
   0x0000555555555a8a <+95>:    01 c8   add    eax,ecx
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000555555555a8c <+97>:    39 c2   cmp    edx,eax
   0x0000555555555a8e <+99>:    74 05   je     0x555555555a95 <phase_2+106>
   0x0000555555555a90 <+101>:   e8 69 fe ff ff  call   0x5555555558fe <explode_bomb>
   0x0000555555555a95 <+106>:   83 45 dc 01     add    DWORD PTR [rbp-0x24],0x1
   0x0000555555555a99 <+110>:   83 7d dc 05     cmp    DWORD PTR [rbp-0x24],0x5
   0x0000555555555a9d <+114>:   7e d3   jle    0x555555555a72 <phase_2+71>
   0x0000555555555a9f <+116>:   90      nop
   0x0000555555555aa0 <+117>:   48 8b 45 f8     mov    rax,QWORD PTR [rbp-0x8]
   0x0000555555555aa4 <+121>:   64 48 33 04 25 28 00 00 00      xor    rax,QWORD PTR fs:0x28
   0x0000555555555aad <+130>:   74 05   je     0x555555555ab4 <phase_2+137>
   0x0000555555555aaf <+132>:   e8 9c f6 ff ff  call   0x555555555150 <__stack_chk_fail@plt>
   0x0000555555555ab4 <+137>:   c9      leave
   0x0000555555555ab5 <+138>:   c3      ret
```

观察到<phase_2 + 97> 处出现了 cmp 指令，故打上断点。然后输入“0 10 20 30 40 50”，逐步调试。查看 edx 与 eax 存的值：

```
(gdb) i reg eax
eax            0x1                 1
(gdb) i reg edx
edx            0xa                 10
```

发现比较的是 numbers[0]+1 和 numbers[1]。同理，修改输入为“0 1 20 30 40 50”，用 continue 命令进入下次比较，查看寄存器中的值：

```
Breakpoint 3, 0x0000555555555a8c in phase_2 ()
(gdb) i reg eax
eax            0x3                 3
(gdb) i reg edx
edx            0x14                20
```

发现比较的是 numbers[1]+2 和 numbers[2]。故大胆猜测每次比较的都是 numbers[i]+i+1 和 numbers[i+1]。故修改输入为“0 1 3 6 10 15”，直接通过。

3.phase_3(char *input)：

思路：首先，根据提示可知，找到反汇编代码中的 switch 段，然后往后找到 cmp 指令，打上断点。不断尝试，看寄存器内容即可。

过程：先反汇编，找到 switch 对应的机器指令段，发现紧跟着一个 cmp 指令，故打上断点

```
   0x0000555555555b36 <+128>:   48 01 d0        add    rax,rdx
   0x0000555555555b39 <+131>:   3e ff e0        notrack jmp rax
   0x0000555555555b3c <+134>:   c7 45 f0 2f 03 00 00    mov    DWORD PTR [rbp-0x10],0x32f
   0x0000555555555b43 <+141>:   eb 44   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b45 <+143>:   c7 45 f0 30 01 00 00    mov    DWORD PTR [rbp-0x10],0x130
   0x0000555555555b4c <+150>:   eb 3b   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b4e <+152>:   c7 45 f0 84 01 00 00    mov    DWORD PTR [rbp-0x10],0x184
   0x0000555555555b55 <+159>:   eb 32   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b57 <+161>:   c7 45 f0 8e 02 00 00    mov    DWORD PTR [rbp-0x10],0x28e
   0x0000555555555b5e <+168>:   eb 29   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b60 <+170>:   c7 45 f0 1c 01 00 00    mov    DWORD PTR [rbp-0x10],0x11c
   0x0000555555555b67 <+177>:   eb 20   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b69 <+179>:   c7 45 f0 01 02 00 00    mov    DWORD PTR [rbp-0x10],0x201
   0x0000555555555b70 <+186>:   eb 17   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b72 <+188>:   c7 45 f0 a9 01 00 00    mov    DWORD PTR [rbp-0x10],0x1a9
   0x0000555555555b79 <+195>:   eb 0e   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b7b <+197>:   c7 45 f0 74 03 00 00    mov    DWORD PTR [rbp-0x10],0x374
   0x0000555555555b82 <+204>:   eb 05   jmp    0x555555555b89 <phase_3+211>
   0x0000555555555b84 <+206>:   e8 75 fd ff ff  call   0x5555555558fe <explode_bomb>
   0x0000555555555b89 <+211>:   8b 45 ec        mov    eax,DWORD PTR [rbp-0x14]
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000555555555b8c <+214>:   39 45 f0        cmp    DWORD PTR [rbp-0x10],eax
   0x0000555555555b8f <+217>:   74 05   je     0x555555555b96 <phase_3+224>
   0x0000555555555b91 <+219>:   e8 68 fd ff ff  call   0x5555555558fe <explode_bomb>
   0x0000555555555b96 <+224>:   90      nop
   0x0000555555555b97 <+225>:   48 8b 45 f8     mov    rax,QWORD PTR [rbp-0x8]
   0x0000555555555b9b <+229>:   64 48 33 04 25 28 00 00 00      xor    rax,QWORD PTR fs:0x28
   0x0000555555555ba4 <+238>:   74 05   je     0x555555555bab <phase_3+245>
   0x0000555555555ba6 <+240>:   e8 a5 f5 ff ff  call   0x555555555150 <__stack_chk_fail@plt>
   0x0000555555555bab <+245>:   c9      leave
   0x0000555555555bac <+246>:   c3      ret
End of assembler dump.
```

然后随便输入两个数，如“0 1”，continue 到断点处。查看 DWORD PTR [rbp-0x10]和 eax 对应的内容：

```
(gdb) i reg rbp
rbp            0x7fffffffe0c0      0x7fffffffe0c0
(gdb) x /16xb 0x7fffffffe0b0
0x7fffffffe0b0: 0x2f    0x03    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffe0b8: 0x00    0xa2    0x7f    0x06    0x38    0x27    0x05    0x07
(gdb) i reg eax
eax            0x1                 1
```

故 DWORD PTR [rbp-0x10]中存的是 0x0000032f=815，eax 中是 1。故修改输入为“0 815”。直接通关。

4. phase_4(char *input)：

过程：先反汇编看对应代码，发现特殊函数 func，结合提示大胆猜测 func 函数就是进行搜索的函数，返回值 eax 为搜索路径。

```
0x0000555555555c97 <+112>:   89 c7   mov    edi,eax
0x0000555555555c99 <+114>:   e8 0f ff ff ff  call   0x555555555bad <func4>
0x0000555555555c9e <+119>:   89 45 f4        mov    DWORD PTR [rbp-0xc],eax
0x0000555555555ca1 <+122>:   8b 45 f4        mov    eax,DWORD PTR [rbp-0xc]
0x0000555555555ca4 <+125>:   3b 45 f0        cmp    eax,DWORD PTR [rbp-0x10]
0x0000555555555ca7 <+128>:   75 08   jne    0x555555555cb1 <phase_4+138>
0x0000555555555ca9 <+130>:   8b 45 e8        mov    eax,DWORD PTR [rbp-0x18]
0x0000555555555cac <+133>:   39 45 f0        cmp    DWORD PTR [rbp-0x10],eax
0x0000555555555caf <+136>:   74 05   je     0x555555555cb6 <phase_4+143>
0x0000555555555cb1 <+138>:   e8 48 fc ff ff  call   0x5555555558fe <explode_bomb>
0x0000555555555cb6 <+143>:   90      nop
0x0000555555555cb7 <+144>:   48 8b 45 f8     mov    rax,QWORD PTR [rbp-0x8]
0x0000555555555cbb <+148>:   64 48 33 04 25 28 00 00 00      xor    rax,QWORD PTR fs:0x28
0x0000555555555cc4 <+157>:   74 05   je     0x555555555ccb <phase_4+164>
0x0000555555555cc6 <+159>:   e8 85 f4 ff ff  call   0x555555555150 <__stack_chk_fail@plt>
0x0000555555555ccb <+164>:   c9      leave
0x0000555555555ccc <+165>:   c3      ret
```

仔细查看反汇编代码，发现 cmp 出现了两次。第一次是比较 eax 与[rbp-0x10]。随便输入两个数“1 2”，在<phase_4+125>即 cmp 处打上断点。查看对应内存：

```
Breakpoint 3, 0x0000555555555c9e in phase_4 ()
(gdb) i reg rbp
rbp            0x7fffffffe0c0      0x7fffffffe0c0
(gdb) x /16xb 0x7fffffffe0b0
0x7fffffffe0b0: 0x07    0x00    0x00    0x00    0x04    0x00    0x00    0x00
0x7fffffffe0b8: 0x00    0xb4    0xc6    0xe3    0x9d    0x12    0x9d    0xaa
(gdb) i reg eax
eax            0x0                 0
```

发现[rbp-0x10]存的是 0x00000007=7，故大胆猜测第二个数为 7。但之后就犯了难，尝试几次都 explode。于是往回看

```
0x0000555555555c66 <+63>:    89 45 ec        mov    DWORD PTR [rbp-0x14],eax
0x0000555555555c69 <+66>:    83 7d ec 02     cmp    DWORD PTR [rbp-0x14],0x2
0x0000555555555c6d <+70>:    75 0f   jne    0x555555555c7e <phase_4+87>
0x0000555555555c6f <+72>:    8b 45 e4        mov    eax,DWORD PTR [rbp-0x1c]
0x0000555555555c72 <+75>:    85 c0   test   eax,eax
0x0000555555555c74 <+77>:    78 08   js     0x555555555c7e <phase_4+87>
0x0000555555555c76 <+79>:    8b 45 e4        mov    eax,DWORD PTR [rbp-0x1c]
0x0000555555555c79 <+82>:    83 f8 0e        cmp    eax,0xe
0x0000555555555c7c <+85>:    7e 05   jle    0x555555555c83 <phase_4+92>
0x0000555555555c7e <+87>:    e8 7b fc ff ff  call   0x55555555558fe <explode_bomb>
0x0000555555555c83 <+92>:    c7 45 f0 07 00 00 00    mov    DWORD PTR [rbp-0x10],0x7
0x0000555555555c8a <+99>:    8b 45 e4        mov    eax,DWORD PTR [rbp-0x1c]
0x0000555555555c8d <+102>:   ba 0e 00 00 00  mov    edx,0xe
0x0000555555555c92 <+107>:   be 00 00 00 00  mov    esi,0x0
0x0000555555555c97 <+112>:   89 c7   mov    edi,eax
0x0000555555555c99 <+114>:   e8 0f ff ff ff  call   0x555555555bad <func4>
```

发现 func 之前也出现了多次 cmp，通过阅读反汇编代码可知，是比较[rbp-0x1c]即第一个数是否小于等于 14。故大胆猜测数组范围是 0 到 14。于是在 0 到 14 中多次选择一个当作第一个数输入，不断尝试，最终确定第一个数就是 14。故输入为“14 7”。

5.phase_5(char *input):

过程：观察反汇编代码，发现出现了多个静态数据区的地址

```
0x0000555555555d22 <+85>:    83 e0 0f        and    eax,0xf
0x0000555555555d25 <+88>:    48 98   cdqe
0x0000555555555d27 <+90>:    48 8d 15 12 38 00 00    lea    rdx,[rip+0x3812]        # 0x555555559540 <array.3096>
0x0000555555555d2e <+97>:    0f b6 14 10     movzx  edx,BYTE PTR [rax+rdx*1]
0x0000555555555d32 <+101>:   8b 45 e8        mov    eax,DWORD PTR [rbp-0x18]
0x0000555555555d35 <+104>:   48 98   cdqe
0x0000555555555d37 <+106>:   88 54 05 f1     mov    BYTE PTR [rbp+rax*1-0xf],dl
0x0000555555555d3b <+110>:   83 45 e8 01     add    DWORD PTR [rbp-0x18],0x1
0x0000555555555d3f <+114>:   83 7d e8 05     cmp    DWORD PTR [rbp-0x18],0x5
0x0000555555555d43 <+118>:   7e ca   jle    0x555555555d0f <phase_5+66>
0x0000555555555d45 <+120>:   c6 45 f7 00     mov    BYTE PTR [rbp-0x9],0x0
0x0000555555555d49 <+124>:   48 8d 45 f1     lea    rax,[rbp-0xf]
0x0000555555555d4d <+128>:   48 8d 35 10 16 00 00    lea    rsi,[rip+0x1610]        # 0x555555557364
0x0000555555555d54 <+135>:   48 89 c7        mov    rdi,rax
0x0000555555555d57 <+138>:   e8 a3 f8 ff ff  call   0x5555555555ff <strings_not_equal>
0x0000555555555d5c <+143>:   85 c0   test   eax,eax
```

通过内存查看命令查看内存如下：

```
(gdb) x /24cb 0x555555559540
0x555555559540 <array.3096>:    109 'm' 97 'a'  100 'd' 117 'u' 105 'i' 101 'e' 114 'r' 115 's'
0x555555559548 <array.3096+8>:  110 'n' 102 'f' 111 'o' 116 't' 118 'v' 98 'b'  121 'y' 108 'l'
0x555555559550 <node5>: 56 '8'  3 '\003'        0 '\000'        0 '\000'        5 '\005'        0 '\000'
(gdb) x /16cb 0x555555557364
0x555555557364: 98 'b'  114 'r' 117 'u' 105 'i' 110 'n' 115 's' 0 '\000'        0 '\000'
0x55555555736c: 0 '\000'        0 '\000'        0 '\000'        0 '\000'        87 'W'  111 'o' 119 'w' 33 '!'
```

此时还是一头雾水，故随便输入一个字符串“bomb”，一步步执行，发现在<phase_5+52>处 explode。仔细观察发现之前调用了 string_length 函数，之后把返回值与 6 进行比较。故可知要求输入的字符串长度为 6。

```
0x0000555555555cf0 <+35>:    48 89 c7        mov    rdi,rax
0x0000555555555cf3 <+38>:    e8 d1 f8 ff ff  call   0x5555555555c9 <string_length>
0x0000555555555cf8 <+43>:    89 45 ec        mov    DWORD PTR [rbp-0x14],eax
0x0000555555555cfb <+46>:    83 7d ec 06     cmp    DWORD PTR [rbp-0x14],0x6
0x0000555555555cff <+50>:    74 05   je     0x555555555d06 <phase_5+57>
0x0000555555555d01 <+52>:    e8 f8 fb ff ff  call   0x55555555558fe <explode_bomb>
0x0000555555555d06 <+57>:    c7 45 e8 00 00 00 00    mov    DWORD PTR [rbp-0x18],0x0
```

于是进行修改，这次我们输入“bombbo”，注意到在<phase_5+138>处出现了 strings_not_equal 函数，故大胆猜测这个函数的两个输入为输入串和比较串。查看内存：

```
Breakpoint 2, 0x0000555555555d57 in phase_5 ()
(gdb) i reg rdi
rdi            0x7fffffffe0b1      140737488347313
(gdb) i reg rsi
rsi            0x555555557364      93824992244580
```

可知 rsi 是比较串的地址，那么 rdi 就是生成串的地址。查看生成串内存：

```
(gdb) x /16cb 0x7fffffffe0b1
0x7fffffffe0b1: 100 'd' 108 'l' 98 'b'  100 'd' 100 'd' 108 'l' 0 '\000'       0 '\000'
0x7fffffffe0b9: 122 'z' 82 'R'  61 '='  -114 '\216'     48 '0'  -112 '\220'     102 'f' -16 '\360'
```

可知对于相同的字符，生成串也是相同的。那么思路来了，我们输入 26 个小写字母，依次查看对应的生成字符即可。下面是整理后的对应表

a	0001	a	m	1101	b
b	0010	d	n	1110	y
c	0011	u	o	1111	l
d	0100	i	p	0000	m
e	0101	e	q	0001	a
f	0110	r	r	0010	d
g	0111	s	s	0011	u
h	1000	n	t	0100	i
i	1001	f	u	0101	e
j	1010	o	v	0110	r
k	1011	t	w	0111	s
l	1100	v	x	1000	n
			y	1001	f
			z	1010	o

结合已知比较串，可知一个合法的输入串为“mfcdhg”。

6.phase_6(char *input)：

过程：结合题目可知，我们只需要知道链表结点中存的数值和顺序号即可。先反汇编出函数汇编代码：

```
(gdb) disass /rs phase_6
Dump of assembler code for function phase_6:
   0x0000555555555d7c <+0>:     f3 0f 1e fa     endbr64
   0x0000555555555d80 <+4>:     55      push   rbp
   0x0000555555555d81 <+5>:     48 89 e5        mov    rbp,rsp
   0x0000555555555d84 <+8>:     48 81 ec 90 00 00 00    sub    rsp,0x90
   0x0000555555555d8b <+15>:    48 89 bd 78 ff ff ff    mov    QWORD PTR [rbp-0x88],rdi
   0x0000555555555d92 <+22>:    64 48 8b 04 25 28 00 00 00      mov    rax,QWORD PTR fs:0x28
   0x0000555555555d9b <+31>:    48 89 45 f8     mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555d9f <+35>:    31 c0   xor    eax,eax
   0x0000555555555da1 <+37>:    48 8d 05 e8 37 00 00    lea    rax,[rip+0x37e8]        # 0x555555559590 <node1>
   0x0000555555555da8 <+44>:    48 89 45 98     mov    QWORD PTR [rbp-0x68],rax
   0x0000555555555dac <+48>:    48 8d 55 a0     lea    rdx,[rbp-0x60]
   0x0000555555555db0 <+52>:    48 8b 85 78 ff ff ff    mov    rax,QWORD PTR [rbp-0x88]
   0x0000555555555db7 <+59>:    48 89 d6        mov    rsi,rdx
   0x0000555555555dba <+62>:    48 89 c7        mov    rdi,rax
   0x0000555555555dbd <+65>:    e8 98 f7 ff ff  call   0x55555555555a <read_six_numbers>
   0x0000555555555dc2 <+70>:    c7 45 88 00 00 00 00    mov    DWORD PTR [rbp-0x78],0x0
   0x0000555555555dc9 <+77>:    eb 54   jmp    0x555555555e1f <phase_6+163>
   0x0000555555555dcb <+79>:    8b 45 88        mov    eax,DWORD PTR [rbp-0x78]
```

注意到在静态数据区出现了名为 node1 的数据，本人略懂一点英文，故大胆猜测这是链表

的第一个结点。查看对应内存：

```
(gdb) x /16xb 0x555555559590
0x555555559590 <node1>: 0x19    0x01    0x00    0x00    0x01    0x00    0x00    0x00
0x555555559598 <node1+8>:       0x80    0x95    0x55    0x55    0x55    0x55    0x00    0x00
```

注意到<node1+8>保存的和 node 的地址极为相似，结合对链表的理解，大胆猜测这就是 next 变量，保存下一个结点的地址。故查看对应内存：

```
(gdb) x /16xb 0x555555559580
0x555555559580 <node2>: 0x8b    0x03    0x00    0x00    0x02    0x00    0x00    0x00
0x555555559588 <node2+8>:       0x70    0x95    0x55    0x55    0x55    0x55    0x00    0x00
```

符合猜测，同时注意到<node2+4> 是 2，大胆猜测这就是顺序号，那么<node2>就是数值。所有六个结点内存如下：

```
1 0x555555559590  0x00000001 0x00000119
2 0x555555559580  0x00000002 0x0000038b
3 0x555555559570  0x00000003 0x00000142
4 0x555555559560  0x00000004 0x00000079
5 0x555555559550  0x00000005 0x00000338
6 0x555555559430  0x00000006 0x00000210
```

故顺序应为“2 5 6 3 1 4”。

四、体会

1.这次实验的过程异常引人入胜，从一开始就设定了拆弹的场景，分成了六个逐步深入的阶段，仿佛在进行一场扣人心弦的游戏。这种设置极大地激发了我的积极性，使我投入其中。

2.通过这次实验，我不仅学到了许多新知识，还深入理解了汇编代码。尽管报告中提到的“Attention is all you need”等概念可能显得轻描淡写，但这些概念都建立在对汇编代码的深刻理解之上。我并非指望能够在一开始就完全理解每一行代码，但至少要熟悉常见的指令，比如 cmp、mov、lea 等。因此，我选择将汇编代码逐行转换为 C 语言，尽管这种方法在别人看来可能效率较低，但我认为这是最好的入门方式。这样做可以确保在完成实验后，我对汇编语言有了更全面的认识，而不是仅仅靠猜测。

此外，我还学会了如何使用调试工具 gdb。在阅读反汇编代码时，由于我刚入门，很多语句都让我感到困惑，无法立即理解。因此，我需要依靠调试工具来验证我的猜想。通过这次实验，我已经掌握了许多常用的 gdb 命令，如 break、info、si、disass 等，这给了我很大的收获。

最后，这次实验还锻炼了我的思维能力。虽然我们不鼓励完全依赖猜测，但考虑到反汇编代码的复杂性以及任务书的提示，我们并不需要完全理解所有内容，只需找到关键指令即可。

总的来说，这次实验让我收获颇丰。通过挑战拆弹的过程，我不仅学到了新知识，还提升了对汇编代码的理解能力，并掌握了调试工具的使用技巧。同时，这次实验也让我锻炼了思维，学会了在复杂情境下找到解决问题的关键点。