

一、实验目的

本实验要求你使用课程所学知识拆除“binary bombs（二进制炸弹，下文将简称为炸弹）”，增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。这里的炸弹是一个Linux可执行程序，包含了6个阶段（或层次、关卡）。炸弹运行的每个阶段要求你输入一个特定字符串，你的输入符合程序预期的输入，该阶段的炸弹就被拆除引信即解除了，否则炸弹“爆炸”打印输出“BOOM!!!!”。实验的目标是拆除尽可能多的炸弹关卡。

- | 阶段1：字符串比较
- | 阶段2：循环
- | 阶段3：条件/分支
- | 阶段4：递归调用和栈
- | 阶段5：指针
- | 阶段6：链表/指针/结构

另外还有一个隐藏阶段，只有当你在第4阶段的解后附加一特定字符串后才会出现。

为完成二进制炸弹拆除任务，你需要使用gdb调试器和objdump来反汇编炸弹的可执行文件并跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法推断拆除炸弹所需的目标字符串。比如在每一阶段的开始代码前和引爆炸弹的函数前设置断点。

本实验的任务就是拆除炸弹。一定要在指定的虚拟机上完成作业，在其他的环境上运行有可能导致失败。（那姑且先试试咱的 Fedora Linux 36，至少，在 Fedora 36 和 Windows 10 下的反汇编代码是一致的。）

要学会单步跟踪调试汇编代码以及学会设置断点。你还要学会如何检查寄存器和内存状态。很好的使用调试器是你在未来的职业生涯中赚到更多money的一项重要技能！

二、报告要求

本报告要求学生把实验中实现的所有函数逐一进行分析说明，写出实现的依据，也就是推理过程，可以是一个简单的数学证明，也可以是代码分析，根据实现中你的想法不同而异。

三、分析

反汇编

```
objdump -d bomb > bomb_disas.txt
```

phase_1

死磕代码

传递的参数是

```
char *input;
```

首先看在main中调用其的部分：

```
400d6a:  bf 48 23 40 00      mov     $0x402348,%edi
400d6f:  e8 6c fd ff ff      call   400ae0 <puts@plt>
400d74:  e8 2f 07 00 00      call   4014a8 <read_line>
400d79:  48 89 c7             mov     %rax,%rdi # %rdi 存 *input (存的就是所指的地
400d7c:  e8 ec 00 00 00      call   400e6d <phase_1>
400d81:  e8 50 08 00 00      call   4015d6 <phase_defused>
```

phase_1 函数本体：

```
000000000400e6d <phase_1>:
400e6d:  48 83 ec 08         sub     $0x8,%rsp
400e71:  be d0 23 40 00      mov     $0x4023d0,%esi # 后来一直用 %rsi, %rsi 相当
400e76:  e8 cf 04 00 00      call   40134a <strings_not_equal>

400e7b:  85 c0               test    %eax,%eax # AND为0, ZF置1, 否则置0。
400e7d:  75 05               jne     400e84 <phase_1+0x17> # ZF=0, 跳转, 爆。
400e7f:  48 83 c4 08         add     $0x8,%rsp # %eax 为0, ZF=1, OK。
400e83:  c3                 ret

400e84:  e8 be 05 00 00      call   401447 <explode_bomb>
400e89:  eb f4               jmp     400e7f <phase_1+0x12>
```

1 Byte = 8 bit

1 位 Hex = 4 bit

1 Word = 32/64 bit

当 %eax 非 0 时会跳转到 call 401447 <explode_bomb>，炸了；

需要绕过去，也就是想法让 %eax 为 0，这样之后直接 ret。

也就该让 <strings_not_equal> 返回 0，也就得让 40136e: 89 d0 mov %edx,%eax 中的 %edx 为 0。

其调用的 strings_not_equal：

```
00000000040134a <strings_not_equal>:
40134a:  41 54               push    %r12
40134c:  55                 push    %rbp
40134d:  53                 push    %rbx

# 第一次<string_length>
40134e:  48 89 fb           mov     %rdi,%rbx # 来自 <main> 中的 %rdi, 也就是读入
401351:  48 89 f5           mov     %rsi,%rbp # 来自 <phase_1> 中的 %esi, 也就是
401354:  e8 d4 ff ff ff     call   40132d <string_length>
```

```

# 第二次<string_length>
401359: 41 89 c4          mov     %eax,%r12d # 暂存第一遍 <string_length> 的返回值
40135c: 48 89 ef          mov     %rbp,%rdi # 拿 %rsi 来替, 再调一遍。
40135f: e8 c9 ff ff ff    call    40132d <string_length>

401364: ba 01 00 00 00    mov     $0x1,%edx # %edx 设为 1。
401369: 41 39 c4          cmp     %eax,%r12d # 两次调用返回值相等, 则ZF置1, 进循环
40136c: 74 07            je      401375 <strings_not_equal+0x2b>

40136e: 89 d0            mov     %edx,%eax # 返回值不相等, 那就是返回1了, 爆;可
401370: 5b              pop     %rbx
401371: 5d              pop     %rbp
401372: 41 5c            pop     %r12
401374: c3              ret

####
#### 这段像个循环。
401375: 0f b6 03          movzbl  (%rbx),%eax # (%rbx) -> %eax
401378: 84 c0            test    %al,%al # %al 是 %eax 的低 8 位, 正好存储一个
40137a: 74 27            je      4013a3 <strings_not_equal+0x59> # %al为0,

40137c: 3a 45 00          cmp     0x0(%rbp),%al # %al - (%rbp), 比较
40137f: 75 29            jne     4013aa <strings_not_equal+0x60> # 不等, 完
401381: 48 83 c3 01       add     $0x1,%rbx # 来自 <main> 中的 %rdi + 1, 指针i
401385: 48 83 c5 01       add     $0x1,%rbp # 来自 <phase_1> 中的 %esi +1, 指
401389: 0f b6 03          movzbl  (%rbx),%eax # 返回 %rbx

40138c: 84 c0            test    %al,%al # %al 为0, ZF置1。
40138e: 74 0c            je      40139c <strings_not_equal+0x52> # 跳转, 返

401390: 38 45 00          cmp     %al,0x0(%rbp)
401393: 74 ec            je      401381 <strings_not_equal+0x37> # 相等, 雷
401395: ba 01 00 00 00    mov     $0x1,%edx # %al!=%rbp, 完蛋。
40139a: eb d2            jmp     40136e <strings_not_equal+0x24> # 强制截胡。

####
####

40139c: ba 00 00 00 00    mov     $0x0,%edx # 返回 0, OK。
4013a1: eb cb            jmp     40136e <strings_not_equal+0x24>

4013a3: ba 00 00 00 00    mov     $0x0,%edx # OK。
4013a8: eb c4            jmp     40136e <strings_not_equal+0x24>

4013aa: ba 01 00 00 00    mov     $0x1,%edx # 完蛋。
4013af: eb bd            jmp     40136e <strings_not_equal+0x24>

```

又调用的string_length：

```

000000000040132d <string_length>:
40132d: 80 3f 00          cmpb    $0x0,(%rdi) # (%rdi) == 0, 也就是input[0]
401330: 74 12            je      401344 <string_length+0x17> # ZF=1, 跳转, :
401332: 48 89 fa          mov     %rdi,%rdx # %rdi -> %rdx

#### 类似于一个指针固定指向头, 一个指向尾, 移动尾直到碰到 NULL。
401335: 48 83 c2 01       add     $0x1,%rdx # %rdx ++ (input[++i])
401339: 89 d0            mov     %edx,%eax # %eax <- %edx (%rdx 截半)
40133b: 29 f8            sub     %edi,%eax # %eax -= %edi (%rdi 截半)

```

```

40133d:  80 3a 00                cmpb   $0x0,(%rdx) # input[i] ?= 0
401340:  75 f3                  jne    401335 <string_length+0x8> # (%rdx) != 0,
401342:  f3 c3                  repz   ret # %rdx == 0, 返回

401344:  b8 00 00 00 00         mov    $0x0,%eax
401349:  c3                     ret

```

据此，我们首先判断其是否为空，`%rdi` 为 0（相当于数组首位）的话就是为 `NULL` 了吧。

从这里也可以理解为什么前面的循环都是 `++` 即可了，`char` 毕竟是只占了 1 Byte 的，我们在遍历数组中的 `input[0], input[1] ...` 时，彼此之间的地址只差了 1。

好吧，其实没必要全部把代码搞懂的.....实际上前面理解大概也不怎么到位，之后再修改吧。

Objdump妙用

前面琢磨了很久，也没有想明白 `%rsi` 里到底存了什么。后来发现似乎倘若只反汇编“executable sections”，是找不到 `$0x4023d0` 这个地址的。

取巧一点：

```

-d, --disassemble      Display assembler contents of executable sections
-D, --disassemble-all  Display assembler contents of all sections

```

```
objdump -D bomb > bomb_disas_all.txt
```

考虑到 `string` 应以 `NULL` 为界，则直接开读直到 00。

```

4023cf:  00 53 6c                add    %dl,0x6c(%rbx)
4023d2:  61                      (bad)
4023d3:  76 65                  jbe    40243a <_IO_stdin_used+0x1ba>
4023d5:  2c 20                  sub    $0x20,%al
4023d7:  74 68                  je     402441 <_IO_stdin_used+0x1c1>
4023d9:  6f                      outsl  %ds:(%rsi),(%dx)
4023da:  75 20                  jne    4023fc <_IO_stdin_used+0x17c>
4023dc:  68 61 73 74 20         push   $0x20747361
4023e1:  73 6c                  jae    40244f <_IO_stdin_used+0x1cf>
4023e3:  61                      (bad)
4023e4:  69 6e 20 6d 65 2e 20   imul   $0x202e656d,0x20(%rsi),%ebp
4023eb:  56                      push   %rsi
4023ec:  69 6c 6c 61 69 6e 2c   imul   $0x202c6e69,0x61(%rsp,%rbp,2),%ebp
4023f3:  20                      (bad)
4023f4:  74 61                  je     402457 <_IO_stdin_used+0x1d7>
4023f6:  6b 65 20 6d           imul   $0x6d,0x20(%rbp),%esp
4023fa:  79 20                  jns    40241c <_IO_stdin_used+0x19c>
4023fc:  70 75                  jo     402473 <_IO_stdin_used+0x1f3>
4023fe:  72 73                  jb     402473 <_IO_stdin_used+0x1f3>
402400:  65 2e 00 00           gs add %al,%gs:(%rax)

```

整点 Python 代码简化操作：

```
# Given hex values, convert them to their ASCII characters
hex_values = "53 6c 61 76 65 2c 20 74 68 6f 75 20 68 61 73 74 20 73 6c 61 69 6e 20 6"
ascii_string = bytes.fromhex(hex_values).decode('ascii').rstrip('\x00') # Remove trailing spaces
ascii_string
```

查了下，出自《李尔王》：

```
'Slave, thou hast slain me. Villain, take my purse.'
```

我说怎么实验文档里给的示例与我生成的反汇编代码不一样呢，原来.....

phase_2

phase_2 函数本体：

```
000000000400e8b <phase_2>:
400e8b: 53                      push    %rbx
400e8c: 48 83 ec 20             sub     $0x20,%rsp # 为什么腾了 32 Bytes 假使此地址为
400e90: 64 48 8b 04 25 28 00    mov     %fs:0x28,%rax # 段寄存器 2*16+8=40?
400e97: 00 00
400e99: 48 89 44 24 18          mov     %rax,0x18(%rsp) # %rsp + 24(a+24) 指向的位
400e9e: 31 c0                   xor     %eax,%eax # %eax = 0
400ea0: 48 89 e6                mov     %rsp,%rsi # %rsp -> %rsi a
400ea3: e8 c1 05 00 00          call    401469 <read_six_numbers>

400ea8: 83 3c 24 00             cmpl    $0x0,(%rsp)
400eac: 78 07                   js      400eb5 <phase_2+0x2a> # (%rsp) < 0 就爆?

400eae: bb 01 00 00 00          mov     $0x1,%ebx # %ebx = 1
400eb3: eb 11                   jmp     400ec6 <phase_2+0x3b>

400eb5: e8 8d 05 00 00          call    401447 <explode_bomb>
400eba: eb f2                   jmp     400eae <phase_2+0x23> # 这句是干什么的，不都

400ebc: 48 83 c3 01             add     $0x1,%rbx

400ec0: 48 83 fb 06             cmp     $0x6,%rbx
400ec4: 74 12                   je      400ed8 <phase_2+0x4d> # 满 6 回了，过关。

400ec6: 89 d8                   mov     %ebx,%eax # %eax = %ebx = 1
400ec8: 03 44 9c fc             add     -0x4(%rsp,%rbx,4),%eax # %eax += (%rsp +
400ecc: 39 04 9c                cmp     %eax,(%rsp,%rbx,4) # (%rsp + %rbx*4) ?= %
400ecf: 74 eb                   je      400ebc <phase_2+0x31> # 相等则继续判断
# 造一组数据：
# 0+1=1 1+2=3 3+3=6 6+4=10 10+5=15
400ed1: e8 71 05 00 00          call    401447 <explode_bomb>
400ed6: eb e4                   jmp     400ebc <phase_2+0x31>
400ed8: 48 8b 44 24 18          mov     0x18(%rsp),%rax # 24 Bytes
400edd: 64 48 33 04 25 28 00    xor     %fs:0x28,%rax
400ee4: 00 00
400ee6: 75 06                   jne     400eee <phase_2+0x63> # 爆栈
400ee8: 48 83 c4 20             add     $0x20,%rsp
```

```

400eec: 5b                pop    %rbx
400eed: c3                ret
400eee: e8 0d fc ff ff    call   400b00 <__stack_chk_fail@plt>

```

调用的 read_six_numbers :

```

000000000401469 <read_six_numbers>:
401469: 48 83 ec 08       sub    $0x8,%rsp # 又腾了 8 Bytes a-8
40146d: 48 89 f2          mov    %rsi,%rdx # %rsp->%rsi->%rdx a
401470: 48 8d 4e 04       lea    0x4(%rsi),%rcx # a + 4->%rcx 动的只是地址
401474: 48 8d 46 14       lea    0x14(%rsi),%rax # a + 20->%rax
401478: 50               push   %rax # a-8
401479: 48 8d 46 10       lea    0x10(%rsi),%rax # a + 16->%rax
40147d: 50               push   %rax # a-4
40147e: 4c 8d 4e 0c       lea    0xc(%rsi),%r9 # a + 12->%r9
401482: 4c 8d 46 08       lea    0x8(%rsi),%r8 # a + 8->%r8
401486: be c3 25 40 00    mov    $0x4025c3,%esi
40148b: b8 00 00 00 00    mov    $0x0,%eax
401490: e8 0b f7 ff ff    call   400ba0 <__isoc99_sscanf@plt>
401495: 48 83 c4 10       add    $0x10,%rsp # %rsp += 16
401499: 83 f8 05          cmp    $0x5,%eax
40149c: 7e 05            jle    4014a3 <read_six_numbers+0x3a> # %eax <=
40149e: 48 83 c4 08       add    $0x8,%rsp # %eax > 5, 我们安全了, 暂时的。
4014a2: c3                ret
4014a3: e8 9f ff ff ff    call   401447 <explode_bomb>

```

好奇 \$0x4025c3 到底是哪块儿。

```

Disassembly of section .rodata:

000000000402480 <array.3415>:

4025c1: 2e 00 25 64 20 25 64    cs add %ah,0x64252064(%rip)          # 6465462c <_
4025c8: 20 25 64 20 25 64      and    %ah,0x64252064(%rip)          # 64654632 <_
4025ce: 20 25 64 20 25 64      and    %ah,0x64252064(%rip)          # 64654638 <_
4025d4: 00 45 72              add    %al,0x72(%rbp)

```

Ghidra 分析所得：

```

s_%d_%d_%d_%d_%d_004025c3

004025c3 25 64 20 25 64 20    ds    "%d %d %d %d %d %d"
          25 64 20 25 64 20
          25 64 20 25 64 00

```

phase_3

```

000000000400ef3 <phase_3>:
400ef3: 48 83 ec 18       sub    $0x18,%rsp
400ef7: 64 48 8b 04 25 28 00 mov    %fs:0x28,%rax
400efe: 00 00
400f00: 48 89 44 24 08     mov    %rax,0x8(%rsp)
400f05: 31 c0             xor    %eax,%eax # %eax = 0

```

```

400f07: 48 8d 4c 24 04      lea    0x4(%rsp),%rcx
400f0c: 48 89 e2             mov    %rsp,%rdx
400f0f: be cf 25 40 00      mov    $0x4025cf,%esi # 读入两个整数
400f14: e8 87 fc ff ff      call   400ba0 <__isoc99_sscanf@plt>
400f19: 83 f8 01             cmp    $0x1,%eax # 没读够, 爆。
400f1c: 7e 10               jle    400f2e <phase_3+0x3b>

400f1e: 83 3c 24 07          cmpl   $0x7,(%rsp) # (%rsp) > 7 出错, 总共8个分支。
400f22: 77 42               ja     400f66 <phase_3+0x73> # jump if above, 爆

400f24: 8b 04 24             mov    (%rsp),%eax
400f27: ff 24 c5 40 24 40 00 jmp     *0x402440(,%rax,8) # M[0x402440 + %rax *
400f2e: e8 14 05 00 00      call   401447 <explode_bomb> # 爆
400f33: eb e9               jmp     400f1e <phase_3+0x2b> # 仍不理解

#### Cases
# 以下全都是跳到400f77
400f35: b8 35 02 00 00      mov    $0x235,%eax
400f3a: eb 3b               jmp     400f77 <phase_3+0x84>

400f3c: b8 a7 01 00 00      mov    $0x1a7,%eax
400f41: eb 34               jmp     400f77 <phase_3+0x84>

400f43: b8 2b 02 00 00      mov    $0x22b,%eax
400f48: eb 2d               jmp     400f77 <phase_3+0x84>

400f4a: b8 6c 00 00 00      mov    $0x6c,%eax
400f4f: eb 26               jmp     400f77 <phase_3+0x84>

400f51: b8 f1 02 00 00      mov    $0x2f1,%eax
400f56: eb 1f               jmp     400f77 <phase_3+0x84>

400f58: b8 3e 00 00 00      mov    $0x3e,%eax
400f5d: eb 18               jmp     400f77 <phase_3+0x84>

400f5f: b8 48 02 00 00      mov    $0x248,%eax
400f64: eb 11               jmp     400f77 <phase_3+0x84>

####

400f66: e8 dc 04 00 00      call   401447 <explode_bomb> # 爆, 之前似乎对爆炸机

400f6b: b8 00 00 00 00      mov    $0x0,%eax
400f70: eb 05               jmp     400f77 <phase_3+0x84>

# Case
400f72: b8 21 01 00 00      mov    $0x121,%eax
400f77: 39 44 24 04          cmp    %eax,0x4(%rsp) # (%rsp+4) ?= %eax 第二个数
# 构造数据:
# 0 -> 400f72
# 1*256+2*16+1 = 256+32+1 = 289
# 0 289
400f7b: 74 05               je     400f82 <phase_3+0x8f> # 要顺利结束。
400f7d: e8 c5 04 00 00      call   401447 <explode_bomb> # 爆

400f82: 48 8b 44 24 08      mov    0x8(%rsp),%rax
400f87: 64 48 33 04 25 28 00 xor     %fs:0x28,%rax
400f8e: 00 00
400f90: 75 05               jne    400f97 <phase_3+0xa4> # 爆栈
400f92: 48 83 c4 18          add    $0x18,%rsp

```

```

400f96:  c3                                ret
400f97:  e8 64 fb ff ff                   call 400b00 <__stack_chk_fail@plt>

```

\$0x4025cf 指的是什么呢

```

00400f0f be cf 25      MOV     ESI ,s_%d_%d_004025c3+12
          40 00

```

存放着对应的各个标签。

顺带，这里可以看出来采用的是小端序。

```

switchD_00400f27::switchdataD_00402440

00402440 72 0f 40      addr     switchD_00400f27::caseD_0
          00 00 00
          00 00
00402448 35 0f 40      addr     switchD_00400f27::caseD_1
          00 00 00
          00 00
00402450 3c 0f 40      addr     switchD_00400f27::caseD_2
          00 00 00
          00 00
00402458 43 0f 40      addr     switchD_00400f27::caseD_3
          00 00 00
          00 00
00402460 4a 0f 40      addr     switchD_00400f27::caseD_4
          00 00 00
          00 00
00402468 51 0f 40      addr     switchD_00400f27::caseD_5
          00 00 00
          00 00
00402470 58 0f 40      addr     switchD_00400f27::caseD_6
          00 00 00
          00 00
00402478 5f 0f 40      addr     switchD_00400f27::caseD_7
          00 00 00
          00 00

```

phase_4

```

0000000000400fdb <phase_4>:
400fdb: 48 83 ec 18      sub     $0x18,%rsp
400fdf: 64 48 8b 04 25 28 00  mov     %fs:0x28,%rax
400fe6: 00 00
400fe8: 48 89 44 24 08      mov     %rax,0x8(%rsp)
400fed: 31 c0             xor     %eax,%eax # %eax = 0
400fef: 48 8d 4c 24 04      lea     0x4(%rsp),%rcx # 第二个输入
400ff4: 48 89 e2           mov     %rsp,%rdx # 应该是存第一个输入吧
400ff7: be cf 25 40 00      mov     $0x4025cf,%esi # 读入两个整数
400ffc: e8 9f fb ff ff      call    400ba0 <__isoc99_sscanf@plt>
401001: 83 f8 02           cmp     $0x2,%eax
401004: 75 06             jne     40100c <phase_4+0x31> # 数量不对，爆

```



```

401006: 83 3c 24 0e      cmpl    $0xe,(%rsp) # (%rsp) > e 会爆。
40100a: 76 05            jbe     401011 <phase_4+0x36> # jbe for unsigned,
40100c: e8 36 04 00 00   call    401447 <explode_bomb>

# %edx a_1 = 14
# %esi a_2 = 0
# %edi a_3 = input_1
401011: ba 0e 00 00 00   mov     $0xe,%edx # %edx = e = 14
401016: be 00 00 00 00   mov     $0x0,%esi # %esi = 0
40101b: 8b 3c 24         mov     (%rsp),%edi # %edi = (%rsp)
40101e: e8 79 ff ff ff   call    400f9c <func4>
401023: 83 f8 03         cmp     $0x3,%eax # %eax != 3 就爆炸。
# 第一个输入应当为 13
401026: 75 07            jne     40102f <phase_4+0x54>

401028: 83 7c 24 04 03   cmpl    $0x3,0x4(%rsp) # (%rsp+4) != 3 就爆炸
# 第二个输入应当为 3
40102d: 74 05            je      401034 <phase_4+0x59>
40102f: e8 13 04 00 00   call    401447 <explode_bomb>

401034: 48 8b 44 24 08   mov     0x8(%rsp),%rax
401039: 64 48 33 04 25 28 00 xor     %fs:0x28,%rax
401040: 00 00
401042: 75 05            jne     401049 <phase_4+0x6e>
401044: 48 83 c4 18      add     $0x18,%rsp
401048: c3              ret
401049: e8 b2 fa ff ff   call    400b00 <__stack_chk_fail@plt>

```

func4 :

```

# %edx a_1
# %esi a_2
# %edi a_3
000000000400f9c <func4>:
400f9c: 48 83 ec 08      sub     $0x8,%rsp
400fa0: 89 d0            mov     %edx,%eax # %eax = a_1
400fa2: 29 f0            sub     %esi,%eax # %eax = a_1 - a_2
400fa4: 89 c1            mov     %eax,%ecx # %ecx = a_1 - a_2
400fa6: c1 e9 1f         shr     $0x1f,%ecx # %ecx 逻辑右移31位, 保留最高位。
400fa9: 01 c1            add     %eax,%ecx # %ecx = (a_1 - a_2) >>logi 31
400fab: d1 f9            sar     %ecx # %ecx 符号右移1位 = [(a_1 - a_2) >>1]
400fad: 01 f1            add     %esi,%ecx # %ecx = [(a_1 - a_2) >>logi 31]
# 化简一下 %ecx = (a_1 - a_2)]/2
# 取中点
400faf: 39 f9            cmp     %edi,%ecx # %ecx ?= a_3
400fb1: 7f 0e            jg      400fc1 <func4+0x25> # %ecx > a_3

400fb3: b8 00 00 00 00   mov     $0x0,%eax # %eax = 0
400fb8: 39 f9            cmp     %edi,%ecx # %ecx ?= %edi
400fba: 7c 11            jl      400fcd <func4+0x31> # %ecx < a_3
400fbc: 48 83 c4 08      add     $0x8,%rsp
400fc0: c3              ret

####
# a_2 ... .. %rcx ... .a_3. ... a_1
# a_1 = 14; a_2 = 0; a_3 = input_1
# !!! %rcx 就是 %ecx
# %ecx > a_3

```

```

# 中点比 a_3 大, 那就将右端点改为中点往左一点。
400fc1: 8d 51 ff          lea     -0x1(%rcx),%edx # a_1 = %rcx - 1
400fc4: e8 d3 ff ff ff    call   400f9c <func4>
400fc9: 01 c0             add     %eax,%eax # %eax *= 2
400fcb: eb ef            jmp     400fbc <func4+0x20> # 顺利

# %eax = 0
# %ecx < a_3
# 中点比 a_3 小, 那就将左端点改为中点往右一点。
400fcd: 8d 71 01          lea     0x1(%rcx),%esi # %a_2 = %rcx + 1
# 那么这里进行的是 a_1 = 14; a_2 = 8; a_3 = 11; 这样的话, 只会得到 1。
#           a_1 = 14; a_2 = 12; a_3 = 13;
400fd0: e8 c7 ff ff ff    call   400f9c <func4>

# 3=2*1+1 1=2*0+1
400fd5: 8d 44 00 01       lea     0x1(%rax,%rax,1),%eax # %eax = 2*%rax+1
400fd9: eb e1            jmp     400fbc <func4+0x20> # 顺利

```

secret_phase

“只有当你在第4阶段的解后附加一特定字符串后” # 总觉得这话在诓我

到底在什么条件下才触发了呢？

破案了，确实是第四阶段的输入后放个 urxvt，但是必须得等第五阶段也完事之后才会触发。

不仅如此，你还得把该问的输入放到最后一行，也就是 phase_6 底下再新增一行。

secret_phase 函数本体

```

00000000040125f <secret_phase>:
40125f: 53               push    %rbx
401260: e8 43 02 00 00   call   4014a8 <read_line> # 再读入行, 所以该问输入应
# %edx a_1 = 10
# %esi a_2 = 0                                     # 所以这两个是干什么的?
# %edi a_3 = *input
401265: ba 0a 00 00 00   mov     $0xa,%edx
40126a: be 00 00 00 00   mov     $0x0,%esi
40126f: 48 89 c7         mov     %rax,%rdi
401272: e8 09 f9 ff ff   call   400b80 <strtol@plt>

# 这里是没区分好括号之类? 也不对吧, 感觉这里直接返回一个数用不着存地址里。。。
# 我们令之后的 input 都表示的是转换后的 long int 的一立即数。
# %rbx = input
# %eax = input - 1
401277: 48 89 c3         mov     %rax,%rbx
40127a: 8d 40 ff         lea     -0x1(%rax),%eax
# input - 1 ?= 1000
40127d: 3d e8 03 00 00   cmp     $0x3e8,%eax # 3*256+14*16+8 = 1000
401282: 77 27           ja      4012ab <secret_phase+0x4c> # 超过1000直接爆
# %edx a_1 = 10
# %esi a_2 = input
# %edi a_3 = $0x6030f0 ??? 像是数组基址。
401284: 89 de           mov     %ebx,%esi

```

```

401286: bf f0 30 60 00      mov     $0x6030f0,%edi
# 似乎 %eax 如果没有初始化, 那么还是会保留input - 1。
40128b: e8 90 ff ff ff      call    401220 <fun7>
401290: 83 f8 04            cmp     $0x4,%eax
401293: 74 05              je      40129a <secret_phase+0x3b>
401295: e8 ad 01 00 00      call    401447 <explode_bomb> # 不等于 4 直接爆炸
# 所以要凑出来 %eax = 4
# 需要等于 40, 但是, 如何触发?
40129a: bf 08 24 40 00      mov     $0x402408,%edi # "Wow! You've defused the
40129f: e8 3c f8 ff ff      call    400ae0 <puts@plt>
4012a4: e8 2d 03 00 00      call    4015d6 <phase_defused>
4012a9: 5b                  pop     %rbx
4012aa: c3                  ret
4012ab: e8 97 01 00 00      call    401447 <explode_bomb>
4012b0: eb d2              jmp     401284 <secret_phase+0x25>

```

secret_phase 只在 phase_defused 中被调用:

```

0000000004015d6 <phase_defused>:
4015d6: 48 83 ec 78        sub     $0x78,%rsp
4015da: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
4015e1: 00 00
4015e3: 48 89 44 24 68      mov     %rax,0x68(%rsp)
4015e8: 31 c0              xor     %eax,%eax
4015ea: 83 3d 7b 21 20 00 06 cmpl    $0x6,0x20217b(%rip)      # 60376c <num_
# 似乎得研究研究<read_line>
# 是不是在检测下一条指令的对应位置?
4015f1: 74 15              je      401608 <phase_defused+0x32>

4015f3: 48 8b 44 24 68      mov     0x68(%rsp),%rax
4015f8: 64 48 33 04 25 28 00 xor     %fs:0x28,%rax
4015ff: 00 00
401601: 75 67              jne     40166a <phase_defused+0x94>
401603: 48 83 c4 78        add     $0x78,%rsp
401607: c3                  ret

# 这里前面并没有<read_line>, 难道, 其实在(特定位置)读入的时候就已经开始折腾这里的了?
# 正好<phrase_4>里头接收2个输入, 这里算是接续? 应该不用换行
401608: 4c 8d 44 24 10      lea     0x10(%rsp),%r8 # 16 这个应该是字符串的开头吧
40160d: 48 8d 4c 24 0c      lea     0xc(%rsp),%rcx # 12
401612: 48 8d 54 24 08      lea     0x8(%rsp),%rdx # 8 这两个整数不知道什么用处
401617: be 19 26 40 00      mov     $0x402619,%esi # s_%d_%d_%s_00402619
40161c: bf 70 38 60 00      mov     $0x603870,%edi # input_strings[240]
401621: e8 7a f5 ff ff      call    400ba0 <__isoc99_sscanf@plt>
401626: 83 f8 03            cmp     $0x3,%eax # 读入了3个东西
401629: 74 0c              je      401637 <phase_defused+0x61>

40162b: bf 58 25 40 00      mov     $0x402558,%edi # "Congratulations! You've
401630: e8 ab f4 ff ff      call    400ae0 <puts@plt>
401635: eb bc              jmp     4015f3 <phase_defused+0x1d>

401637: be 22 26 40 00      mov     $0x402622,%esi # s_urxvt_00402622 "urxvt"
40163c: 48 8d 7c 24 10      lea     0x10(%rsp),%rdi # 接下来好比较字符串。
401641: e8 04 fd ff ff      call    40134a <strings_not_equal>
401646: 85 c0              test    %eax,%eax
401648: 75 e1              jne     40162b <phase_defused+0x55> # 不相等, 寄。
40164a: bf f8 24 40 00      mov     $0x4024f8,%edi # s_Curses,_you've_found_t

```

```

40164f: e8 8c f4 ff ff      call    400ae0 <puts@plt>
401654: bf 20 25 40 00      mov     $0x402520,%edi # s_But_finding_it_and_sol
401659: e8 82 f4 ff ff      call    400ae0 <puts@plt>
40165e: b8 00 00 00 00      mov     $0x0,%eax
401663: e8 f7 fb ff ff      call    40125f <secret_phase>
401668: eb c1               jmp     40162b <phase_defused+0x55>
40166a: e8 91 f4 ff ff      call    400b00 <__stack_chk_fail@plt>

```

fun7

```

0000000000401220 <fun7>:
# %edx a_1
# %esi a_2 = input
# %edi a_3
401220: 48 85 ff            test    %rdi,%rdi # %rdi 为 0 则爆炸。
401223: 74 34              je      401259 <fun7+0x39>
401225: 48 83 ec 08        sub     $0x8,%rsp
# a_1 = (a_3) = 0x24 = 32+4= 36
# a_2 肯定不能上来就等于 36

# 错误示例：原因是最外层的调用最后算。
# 应该有 a_1 <= a_2, 这样才能先赋0, 再加1, 之后再乘2, 再乘2。
# a_2 应该是等于 40, 过程如下：
# a_1 < a_2 = 40
# a_3' = (a_3 + 16), (a_3') = 50, %eax = 1
# a_1' = (a_3') > a_2 = 40
# a_3'' = (a_3' + 8), (a_3'') = 45, %eax = 2
# a_1'' = (a_3'') > a_2 = 40
# a_3''' = (a_3'' + 8), (a_3''') = 40, %eax = 4

# 所以也最先该触发x2的情形。
# a_1 = 36 > a_2 = 7
# a_3' = (a_3 + 8), (a_3') = 8, %eax = 2 * %eax
# a_1 = (a_3') = 8 > a_2 = 7
# a_3' = ((a_3 + 8) + 8), (a_3') = 6, %eax = 2 * %eax
# a_1 = (a_3') = 6 < a_2 = 7
# a_3' = (((a_3 + 8) + 8) + 16), (a_3') = 7, %eax = 1

# 所指向的内容其实又是特定数组的基址
# a_1 ?= a_2
401229: 8b 17              mov     (%rdi),%edx # a_1 = (a_3) = M[$0x6030f0]
40122b: 39 f2              cmp     %esi,%edx # a_1 = (a_3) ?= a_2
40122d: 7f 0e              jg      40123d <fun7+0x1d>
# a_1 <= a_2
40122f: b8 00 00 00 00      mov     $0x0,%eax # %eax = 0
401234: 39 f2              cmp     %esi,%edx # a_1 ?= a_2
401236: 75 12              jne     40124a <fun7+0x2a>
# a_1 == a_2
401238: 48 83 c4 08        add     $0x8,%rsp
40123c: c3                ret

# a_1 > a_2
40123d: 48 8b 7f 08        mov     0x8(%rdi),%rdi # a_3 = (a_3 + 8)

```

```

# a_3 = $0x6030f0 -> (a_3 + 8) = 00603110 【8】
# (a_3 + 8) = 00603110 -> ((a_3 + 8) + 8) = 00603190 【6】
# (a_3 + 8) = 00603110 -> ((a_3 + 8) + 16) = 00603150 【0x16=22】
# ((a_3 + 8) + 8) = 00603190 -> (((a_3 + 8) + 8) + 16) = 00603250 【7】
401241: e8 da ff ff ff      call    401220 <fun7>
401246: 01 c0               add     %eax,%eax # 调用完自身, %eax *= 2
# 出偶数, 应该是得从这吧。
401248: eb ee              jmp     401238 <fun7+0x18> # 返回。

# 我们应该是需要这样的地址: (((a_3 + 16) + 8) + 8)
# a_1 < a_2
40124a: 48 8b 7f 10        mov     0x10(%rdi),%rdi # a_3 = (a_3 + 16)
# a_3 = $0x6030f0 -> (a_3 + 16) = 00603130 【0x32=48+2=50】
# (a_3 + 16) = 00603110 -> ((a_3 + 16) + 8) = 00603170 【0x2d=32+13=45】
# (a_3 + 16) = 00603110 -> ((a_3 + 16) + 16) = 006031b0 【0x6b=96+11=107】
# ((a_3 + 16) + 8) = 00603170 -> (((a_3 + 16) + 8) + 8) = 006031d0 【0x28=32+8=40】
# 难道还是顺序错了? 最外层应该是x2, 中间层也x2, 最内层x2+1 (实际是从0到1)
# 真事儿。
40124e: e8 cd ff ff ff      call    401220 <fun7>
401253: 8d 44 00 01        lea     0x1(%rax,%rax,1),%eax # 调用完自身, %eax =
401257: eb df              jmp     401238 <fun7+0x18>

# 爆
401259: b8 ff ff ff ff      mov     $0xffffffff,%eax
40125e: c3                 ret

```

让我们看看 \$0x6030f0 里头放的什么：

```

n1

006030f0 24 00 00          undefined
          00 00 00
          00 00 10

006030f0 24          undefined 24h          [0]
006030f1 00          undefined 00h          [1]
006030f2 00          undefined 00h          [2]
006030f3 00          undefined 00h          [3]
006030f4 00          undefined 00h          [4]
006030f5 00          undefined 00h          [5]
006030f6 00          undefined 00h          [6]
006030f7 00          undefined 00h          [7]
# 小端序, (a_3 + 8) = 0x603110
006030f8 10          undefined 10h          [8]          ? -> 0C
006030f9 31          undefined 31h          [9]
006030fa 60          undefined 60h          [10]
006030fb 00          undefined 00h          [11]
006030fc 00          undefined 00h          [12]
006030fd 00          undefined 00h          [13]
006030fe 00          undefined 00h          [14]
006030ff 00          undefined 00h          [15]
# (a_3 + 16) = 0x603130
00603100 30          undefined 30h          [16]          ? -> 0C
00603101 31          undefined 31h          [17]
00603102 60          undefined 60h          [18]
00603103 00          undefined 00h          [19]
00603104 00          undefined 00h          [20]
00603105 00          undefined 00h          [21]

```

```

00603106 00          undefine 00h          [22]
00603107 00          undefine 00h          [23]

```

其他的几个需要读的数组也都是这个样子。

这里多次出现了 <num_input_strings>, 但是 <num_input_strings> 到底是如何自增的。

```

00000000004014a8 <read_line>:
4014a8: 48 83 ec 08      sub    $0x8,%rsp
4014ac: b8 00 00 00 00   mov    $0x0,%eax
4014b1: e8 50 ff ff ff   call   401406 <skip>
4014b6: 48 85 c0         test   %rax,%rax
4014b9: 74 63           je     40151e <read_line+0x76>
4014bb: 8b 35 ab 22 20 00 mov     0x2022ab(%rip),%esi      # 60376c <num_
4014c1: 48 63 c6         movslq %esi,%rax
4014c4: 48 8d 14 80      lea     (%rax,%rax,4),%rdx
4014c8: 48 c1 e2 04      shl     $0x4,%rdx
4014cc: 48 81 c2 80 37 60 00 add     $0x603780,%rdx
4014d3: 48 c7 c1 ff ff ff ff mov     $0xffffffffffffffff,%rcx
4014da: b8 00 00 00 00   mov     $0x0,%eax
4014df: 48 89 d7         mov     %rdx,%rdi
4014e2: f2 ae          repnz  scas  %es:(%rdi),%al
4014e4: 48 f7 d1         not     %rcx
4014e7: 48 83 e9 01      sub     $0x1,%rcx
4014eb: 83 f9 4e         cmp     $0x4e,%ecx
4014ee: 0f 8f 9c 00 00 00 jg      401590 <read_line+0xe8>
4014f4: 83 e9 01         sub     $0x1,%ecx
4014f7: 48 63 c9         movslq  %ecx,%rcx
4014fa: 48 63 c6         movslq  %esi,%rax
4014fd: 48 8d 04 80      lea     (%rax,%rax,4),%rax
401501: 48 c1 e0 04      shl     $0x4,%rax
401505: c6 84 01 80 37 60 00 movb     $0x0,0x603780(%rcx,%rax,1)
40150c: 00
40150d: 8d 46 01         lea     0x1(%rsi),%eax
401510: 89 05 56 22 20 00 mov     %eax,0x202256(%rip)      # 60376c <num_
401516: 48 89 d0         mov     %rdx,%rax
401519: 48 83 c4 08      add     $0x8,%rsp
40151d: c3             ret
40151e: 48 8b 05 2b 22 20 00 mov     0x20222b(%rip),%rax      # 603750 <stdi
401525: 48 39 05 44 22 20 00 cmp     %rax,0x202244(%rip)      # 603770 <infi
40152c: 74 19           je     401547 <read_line+0x9f>
40152e: bf f3 25 40 00   mov     $0x4025f3,%edi
401533: e8 78 f5 ff ff   call   400ab0 <getenv@plt>
401538: 48 85 c0         test    %rax,%rax
40153b: 74 1e           je     40155b <read_line+0xb3>
40153d: bf 00 00 00 00   mov     $0x0,%edi
401542: e8 89 f6 ff ff   call   400bd0 <exit@plt>
401547: bf d5 25 40 00   mov     $0x4025d5,%edi
40154c: e8 8f f5 ff ff   call   400ae0 <puts@plt>
401551: bf 08 00 00 00   mov     $0x8,%edi
401556: e8 75 f6 ff ff   call   400bd0 <exit@plt>
40155b: 48 8b 05 ee 21 20 00 mov     0x2021ee(%rip),%rax      # 603750 <stdi
401562: 48 89 05 07 22 20 00 mov     %rax,0x202207(%rip)      # 603770 <infi
401569: b8 00 00 00 00   mov     $0x0,%eax
40156e: e8 93 fe ff ff   call   401406 <skip>
401573: 48 85 c0         test    %rax,%rax

```

```

401576: 0f 85 3f ff ff ff    jne    4014bb <read_line+0x13>
40157c: bf d5 25 40 00      mov     $0x4025d5,%edi
401581: e8 5a f5 ff ff      call   400ae0 <puts@plt>
401586: bf 00 00 00 00      mov     $0x0,%edi
40158b: e8 40 f6 ff ff      call   400bd0 <exit@plt>
401590: bf fe 25 40 00      mov     $0x4025fe,%edi
401595: e8 46 f5 ff ff      call   400ae0 <puts@plt>
40159a: 8b 05 cc 21 20 00    mov     0x2021cc(%rip),%eax    # 60376c <num_
4015a0: 8d 50 01             lea     0x1(%rax),%edx
4015a3: 89 15 c3 21 20 00    mov     %edx,0x2021c3(%rip)    # 60376c <num_
4015a9: 48 98               cltq
4015ab: 48 6b c0 50         imul    $0x50,%rax,%rax
4015af: 48 be 2a 2a 2a 74 72 movabs  $0x636e7572742a2a2a,%rsi
4015b6: 75 6e 63            jnb     4015b9
4015b9: 48 bf 61 74 65 64 2a movabs  $0x2a2a2a64657461,%rdi
4015c0: 2a 2a 00            jnb     4015c3
4015c3: 48 89 b0 80 37 60 00 mov     %rsi,0x603780(%rax)
4015ca: 48 89 b8 88 37 60 00 mov     %rdi,0x603788(%rax)
4015d1: e8 71 fe ff ff      call   401447 <explode_bomb>

```

phase_5

phase_5 函数：

```

00000000040104e <phase_5>:
40104e: 48 83 ec 18         sub     $0x18,%rsp
401052: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
401059: 00 00
40105b: 48 89 44 24 08      mov     %rax,0x8(%rsp)
401060: 31 c0              xor     %eax,%eax
401062: 48 8d 4c 24 04      lea     0x4(%rsp),%rcx
401067: 48 89 e2           mov     %rsp,%rdx
40106a: be cf 25 40 00      mov     $0x4025cf,%esi # 与 phase_3 一样
40106f: e8 2c fb ff ff      call   400ba0 <__isoc99_sscanf@plt>
401074: 83 f8 01           cmp     $0x1,%eax
401077: 7e 57             jle     4010d0 <phase_5+0x82> # %eax <= 1, 爆

401079: 8b 04 24           mov     (%rsp),%eax # x_1 -> %eax
40107c: 83 e0 0f          and     $0xf,%eax # 掩码, 保留末4位。
40107f: 89 04 24           mov     %eax,(%rsp) # 存回去。x_1 %= 16
401082: 83 f8 0f          cmp     $0xf,%eax # 比较
401085: 74 2f             je      4010b6 <phase_5+0x68> # %eax 等于f, 爆。
# %eax < 16
401087: b9 00 00 00 00     mov     $0x0,%ecx # %ecx = 0
40108c: ba 00 00 00 00     mov     $0x0,%edx

# 像循环
401091: 83 c2 01          add     $0x1,%edx # %edx += 1
# 累计三次
401094: 48 98             cltq    # 符号拓展 %eax
401096: 8b 04 85 80 24 40 00 mov     0x402480(,%rax,4),%eax # (402480 + 4*%rax
# 往前回溯
# %rax = 6 -> %eax = f
# %rax = 14 -> %eax = 6
# %rax = 2 -> %eax = 14
# 所以 %rax 初值, 即 (x_1 % 16) == 2

```

```

40109d: 01 c1          add    %eax,%ecx # %ecx += %eax
# %ecx = 0 + 14 + 6 + 15 = 35
## 以上, 可令 x_1 = 2, x_2 = 35, 即为可行输入
40109f: 83 f8 0f      cmp    $0xf,%eax
4010a2: 75 ed        jne    401091 <phase_5+0x43> # %eax != f -> 回去
# %eax = f

4010a4: c7 04 24 0f 00 00 00 movl   $0xf,(&rsp) # x_1 = f
4010ab: 83 fa 03      cmp    $0x3,%edx
4010ae: 75 06        jne    4010b6 <phase_5+0x68> # 爆
# %edx = 3
4010b0: 39 4c 24 04   cmp    %ecx,0x4(&rsp)
4010b4: 74 05        je     4010bb <phase_5+0x6d>
4010b6: e8 8c 03 00 00 call   401447 <explode_bomb> # 爆
# x_2 = %ecx = 35
4010bb: 48 8b 44 24 08 mov     0x8(&rsp),%rax
4010c0: 64 48 33 04 25 28 00 xor     %fs:0x28,%rax
4010c7: 00 00
4010c9: 75 0c        jne    4010d7 <phase_5+0x89>
4010cb: 48 83 c4 18   add     $0x18,%rsp
4010cf: c3          ret

4010d0: e8 72 03 00 00 call   401447 <explode_bomb> # 爆
4010d5: eb a2        jmp     401079 <phase_5+0x2b>
4010d7: e8 24 fa ff ff call   400b00 <__stack_chk_fail@plt>

```

0x402480 对应的地址：

array.3415

00402480 0a	undefined10Ah	[0]
00402481 00	undefined100h	[1]
00402482 00	undefined100h	[2]
00402483 00	undefined100h	[3]
00402484 02	undefined102h	[4]
00402485 00	undefined100h	[5]
00402486 00	undefined100h	[6]
00402487 00	undefined100h	[7]
00402488 0e	undefined10Eh	[8]
00402489 00	undefined100h	[9]
0040248a 00	undefined100h	[10]
0040248b 00	undefined100h	[11]
0040248c 07	undefined107h	[12]
0040248d 00	undefined100h	[13]
0040248e 00	undefined100h	[14]
0040248f 00	undefined100h	[15]
00402490 08	undefined108h	[16]
00402491 00	undefined100h	[17]
00402492 00	undefined100h	[18]
00402493 00	undefined100h	[19]
00402494 0c	undefined10Ch	[20]
00402495 00	undefined100h	[21]
00402496 00	undefined100h	[22]
00402497 00	undefined100h	[23]
00402498 0f	undefined10Fh	[24]
00402499 00	undefined100h	[25]

0040249a 00	undefined100h	[26]
0040249b 00	undefined100h	[27]
0040249c 0b	undefined10Bh	[28]
0040249d 00	undefined100h	[29]
0040249e 00	undefined100h	[30]
0040249f 00	undefined100h	[31]
004024a0 00	undefined100h	[32]
004024a1 00	undefined100h	[33]
004024a2 00	undefined100h	[34]
004024a3 00	undefined100h	[35]
004024a4 04	undefined104h	[36]
004024a5 00	undefined100h	[37]
004024a6 00	undefined100h	[38]
004024a7 00	undefined100h	[39]
004024a8 01	undefined101h	[40]
004024a9 00	undefined100h	[41]
004024aa 00	undefined100h	[42]
004024ab 00	undefined100h	[43]
004024ac 0d	undefined10Dh	[44]
004024ad 00	undefined100h	[45]
004024ae 00	undefined100h	[46]
004024af 00	undefined100h	[47]
004024b0 03	undefined103h	[48]
004024b1 00	undefined100h	[49]
004024b2 00	undefined100h	[50]
004024b3 00	undefined100h	[51]
004024b4 09	undefined109h	[52]
004024b5 00	undefined100h	[53]
004024b6 00	undefined100h	[54]
004024b7 00	undefined100h	[55]
004024b8 06	undefined106h	[56]
004024b9 00	undefined100h	[57]
004024ba 00	undefined100h	[58]
004024bb 00	undefined100h	[59]
004024bc 05	undefined105h	[60]
004024bd 00	undefined100h	[61]
004024be 00	undefined100h	[62]
004024bf 00	undefined100h	[63]

数组全部内容：

```
0xa, 0x2, 0xe, 0x7, 0x8, 0xc, 0xf, 0xb, 0x0, 0x4, 0x1, 0xd, 0x3, 0x9, 0x6, 0x5
10 , 2 , 14 , 7 , 8 , 12 , 15 , 11 , 0 , 4 , 1 , 13 , 3 , 9 , 6 , 5
```

phase_6

phase_6 本体：

```
00000000004010dc <phase_6>:
4010dc: 41 56          push    %r14
4010de: 41 55          push    %r13
4010e0: 41 54          push    %r12
4010e2: 55            push    %rbp
4010e3: 53            push    %rbx
4010e4: 48 83 ec 60    sub     $0x60,%rsp
```

```

4010e8: 64 48 8b 04 25 28 00    mov     %fs:0x28,%rax
4010ef: 00 00
4010f1: 48 89 44 24 58          mov     %rax,0x58(%rsp) # 5*16+8=88 Bytes 留空给11
4010f6: 31 c0                   xor     %eax,%eax # %eax = 0
4010f8: 48 89 e6                mov     %rsp,%rsi # 一会儿好把六个变量存到栈里
4010fb: e8 69 03 00 00          call    401469 <read_six_numbers>
# 这六个姑且算作数组内 a[0], a[1], a[2] ... a[5]
401100: 49 89 e4                mov     %rsp,%r12 # %r12 = a 基址
401103: 49 89 e5                mov     %rsp,%r13 # %r13 = a
401106: 41 be 00 00 00 00        mov     $0x0,%r14d # %r14d = 0
40110c: eb 25                   jmp     401133 <phase_6+0x57> #

# -> no.1-compare -> no.1-loop -> no.2
# 纯纯的***

40110e: e8 34 03 00 00          call    401447 <explode_bomb> # 爆
401113: eb 2d                   jmp     401142 <phase_6+0x66>

401115: 83 c3 01                add     $0x1,%ebx # %ebx = %ebx + 1 = 2
401118: 83 fb 05                cmp     $0x5,%ebx
40111b: 7f 12                   jg      40112f <phase_6+0x53> # %ebx > 5
# %ebx <= 5, 直接往下走。

40111d: 48 63 c3                movslq  %ebx,%rax # %rax = %ebx = %r14d = 1 -> 2
# 2 -> 3
401120: 8b 04 84                mov     (%rsp,%rax,4),%eax # %eax = a[%rax] = a[1]
401123: 39 45 00                cmp     %eax,0x0(%rbp) # %eax == a[0] 则爆
401126: 75 ed                   jne     401115 <phase_6+0x39> # a[0] != a[1] -> a
# 也就是, 后五个数都不能与第一个数相等?
401128: e8 1a 03 00 00          call    401447 <explode_bomb> # 爆
40112d: eb e6                   jmp     401115 <phase_6+0x39>

# 我猜测, 是这六个数两两之间不相等
# for(int i = 1; i <= 6; ++i)
#     判断是否小于等于 6
#     for(int j = i + 1; j <= 5; ++j)
#         a[i - 1] != a[j]
40112f: 49 83 c5 04            add     $0x4,%r13 # %r13 += 4

401133: 4c 89 ed                mov     %r13,%rbp # %rbp = %r13 = a -> &a[1]
401136: 41 8b 45 00             mov     0x0(%r13),%eax # %eax = (%r13) = a[0] ->
40113a: 83 e8 01                sub     $0x1,%eax # %eax = %eax - 1 = a[0]-1 -> a
40113d: 83 f8 05                cmp     $0x5,%eax
401140: 77 cc                   ja      40110e <phase_6+0x32> # %eax > 5, 爆
# %eax <= 5 即 a[0] -> a[1] <= 6
# 能不能存负数
# 我们假定里头的所有数都是1~6, 不含0。
401142: 41 83 c6 01            add     $0x1,%r14d # %r14d = %r14d + 1 = 1 -> 2
401146: 41 83 fe 06             cmp     $0x6,%r14d
40114a: 74 05                   je      401151 <phase_6+0x75> # %r14d 等于 6 则跳:
40114c: 44 89 f3                mov     %r14d,%ebx # %r14d 不到 6, %ebx = %r14d =
# 内循环的起点。
40114f: eb cc                   jmp     40111d <phase_6+0x41> #

```

```

401151: 49 8d 4c 24 18      lea    0x18(%r12),%rcx # %r12 = a; %rcx = %r12 +
401156: ba 07 00 00 00      mov    $0x7,%edx # %edx = 7

40115b: 89 d0              mov    %edx,%eax # %eax = %edx = 7
40115d: 41 2b 04 24        sub    (%r12),%eax # %eax = %eax - (%r12) = 7 -
401161: 41 89 04 24        mov    %eax,(%r12) # a[0] = %eax = 7 - a[0]
401165: 49 83 c4 04        add    $0x4,%r12 # %r12 = %r12 + 4 = &a[1]
401169: 4c 39 e1          cmp    %r12,%rcx # &a[6] = %rcx ?= %r12 = &a[1]
40116c: 75 ed            jne    40115b <phase_6+0x7f>
# 要把 a[0] -> a[5] 全都变成 7 - a[i]

```

```

# 已经遍历了 a[0] -> a[5]
40116e: be 00 00 00 00      mov    $0x0,%esi # %esi = 0
401173: eb 1a            jmp    40118f <phase_6+0xb3>

```

不是，到底是 32-bit 还是 64-bit 啊 我****

```

# 怪不得每次都要从$0x6032d0开始，这其实就是遍历链表的过程。
# %rdx = $0x6032d0 # node1
401175: 48 8b 52 08      mov    0x8(%rdx),%rdx
# %rdx' = [%rdx + 8] = [$0x6032d8] = 0x6032e0 # node2 %ecx = 2
# %rdx'' = [%rdx' + 8] = [$0x6032e8] = 0x6032f0 # node3 %ecx = 3
# %rdx''' = [%rdx'' + 8] = [$0x6032f8] = 0x603300 # node4 %ecx = 4
# %rdx'''' = [%rdx''' + 8] = [$0x603308] = 0x603310 # node5 %ecx = 5
# %rdx''''' = [%rdx'''' + 8] = [$0x603318] = 0x603320 # node6 %ecx = 6
# %rdx'''''' = [%rdx''''' + 8] = [$0x603328] = 0
401179: 83 c0 01      add    $0x1,%eax # %eax = %eax + 1 = 2;
40117c: 39 c8        cmp    %ecx,%eax # %eax ?= %ecx = a[0]
40117e: 75 f5      jne    401175 <phase_6+0x99>
# %eax = 2, 3, 4, 5

```

```

# %ecx = %eax 匹配上了 # a[8], a[10], ...
401180: 48 89 54 f4 20      mov    %rdx,0x20(%rsp,%rsi,8) # a[2*%rsi + 32/4]
# 还是 a[%rsi + 4] = %rdx
# a'[4] ... a'[9], 哦，其实这个是相当于把原本的 a[0], ..., a[5] 放在了前三个共24字节内，之后
# 尽管如此，前面的诸如 0x6032e0 此处仍假定为只是32位的（高32位为 0）。
# 相当于把 node 的地址存到了数组里，a'[4] ... a'[9]的先后顺序是与a[0], a[1], ..., a[5] 的
# 大胆猜测，a[0], a[1], ..., a[5]的元素标记的是在链表中的“位置”。（第a[0]个、第a[1]个.....，来
401185: 48 83 c6 01      add    $0x1,%rsi # %rsi = %rsi + 1
401189: 48 83 fe 06      cmp    $0x6,%rsi # %rsi ?= 6
40118d: 74 14          je     4011a3 <phase_6+0xc7>
# 直到 6 个都匹配完
# a[8], a[10], ..., a[18]
# a[4] ... a[9] 这个看着更顺眼
40118f: 8b 0c b4      mov    (%rsp,%rsi,4),%ecx # %ecx=a[%rsi]=a[0]
401192: b8 01 00 00 00      mov    $0x1,%eax # %eax = 1
401197: ba d0 32 60 00      mov    $0x6032d0,%edx # 之后会有 (%edx) = 0x00027;
# 0x6032d0 # node1
40119c: 83 f9 01      cmp    $0x1,%ecx # %ecx = a[0] ?= 1
40119f: 7f d4      jg     401175 <phase_6+0x99> # %ecx > 1

```

```

# %ecx <= 1
# 到底能不能等于 0 啊话说我****
4011a1:  eb dd                                jmp     401180 <phase_6+0xa4>

# 我们读入的是各个元素在链表中的位置a[0], a[1], a[2], ... , a[5] (0<=i<=5, 1<=a[i]<=6)
# 我们新开了一个数组a'[4], a'[5], a'[6], ... , a'[9] (存放着上边那些位置对应的链表 node 地
# 链表 node 的各地址存储的是, 当前 node 指向的下一 node 的地址。
# 对于地址, 我们采用&a[i]和a[i]以及a[i]->v的方式来区分。
# 地址里套地址, 好歹毒啊, 我****。

4011a3:  48 8b 5c 24 20                        mov     0x20(%rsp),%rbx # %rbx = a'[4] 存放的是 a'
4011a8:  48 8b 44 24 28                        mov     0x28(%rsp),%rax # %rax = a'[5] 存放的是 a'
# 要将 %rax 放到 %rbx + 0x8 这一地址的内存中。
4011ad:  48 89 43 08                          mov     %rax,0x8(%rbx) # (%rbx + 8) = %rax 即 [a
# [a'[4] + 8] 指的是 "a'[4] 存的地址物理上相邻下一个
# 为什么要给它赋值成这样?
4011b1:  48 8b 54 24 30                        mov     0x30(%rsp),%rdx # %rdx = a'[6]
4011b6:  48 89 50 08                          mov     %rdx,0x8(%rax) # (%rax + 8) = %rdx 即 [a
4011ba:  48 8b 44 24 38                        mov     0x38(%rsp),%rax # %rax = a'[7]
4011bf:  48 89 42 08                          mov     %rax,0x8(%rdx) # (%rdx + 8) = %rax 即 [a
4011c3:  48 8b 54 24 40                        mov     0x40(%rsp),%rdx # %rdx = a'[8]
4011c8:  48 89 50 08                          mov     %rdx,0x8(%rax) # (%rax + 8) = %rdx 即 [a
4011cc:  48 8b 44 24 48                        mov     0x48(%rsp),%rax # %rax = a'[9]
4011d1:  48 89 42 08                          mov     %rax,0x8(%rdx) # (%rdx + 8) = %rax 即 [a
4011d5:  48 c7 40 08 00 00 00                 movq    $0x0,0x8(%rax) # [a'[9] + 8]->v = 0
4011dc:  00
4011dd:  bd 05 00 00 00                       mov     $0x5,%ebp # %ebp = 5
4011e2:  eb 09                                jmp     4011ed <phase_6+0x111> #

# (%rbx) >= %eax
#
4011e4:  48 8b 5b 08                          mov     0x8(%rbx),%rbx # %rbx = (%rbx + 8) = [a'[
4011e8:  83 ed 01                             sub     $0x1,%ebp # %ebp --;
4011eb:  74 11                                je      4011fe <phase_6+0x122> # %ebp == 0
# %ebp = 1
#
4011ed:  48 8b 43 08                          mov     0x8(%rbx),%rax # %rax = (%rbx + 8) = [a'[
# %rax = (%rbx + 8) = [a'[
# %rax = (%rbx + 8) = [a'[
# %rax = (%rbx + 8) = [a'[
# %rax = (%rbx + 8) = [a'[
4011f1:  8b 00                                mov     (%rax),%eax # %eax = (%rax) = a'[5]->v
# %eax = (%rax) = a'[6]->v
# %eax = (%rax) = a'[7]->v
# %eax = (%rax) = a'[9]->v

# 这就是为什么node有16字节, 而之前只是一直在用8字节?
4011f3:  39 03                                cmp     %eax,(%rbx) # [a'[4]]->v >= %eax = a'[5]-
# [a'[5]]->v >= %eax = a'[6]-
# [a'[6]]->v >= %eax = a'[7]-
# [a'[7]]->v >= %eax = a'[8]-
# [a'[8]]->v >= %eax = a'[9]-

```

```

4011f5: 7d ed                                jge     4011e4 <phase_6+0x108> # (%rbx) >= %eax #
# 必须是降序

4011f7: e8 4b 02 00 00                    call    401447 <explode_bomb> # 爆

4011fc: eb e6                                jmp     4011e4 <phase_6+0x108>

4011fe: 48 8b 44 24 58                    mov     0x58(%rsp),%rax
401203: 64 48 33 04 25 28 00              xor     %fs:0x28,%rax
40120a: 00 00
40120c: 75 0d                                jne     40121b <phase_6+0x13f> # 爆栈
40120e: 48 83 c4 60                        add     $0x60,%rsp
401212: 5b                                pop     %rbx
401213: 5d                                pop     %rbp
401214: 41 5c                                pop     %r12
401216: 41 5d                                pop     %r13
401218: 41 5e                                pop     %r14
40121a: c3                                ret
40121b: e8 e0 f8 ff ff                    call    400b00 <__stack_chk_fail@plt>

```

一直也没有搞懂到底 `call 401447 <explode_bomb>` 完后紧接着还要执行的代码段是什么意思。

```

                                node1
006032d0 7a 02 00 00                    undefined4    0000027Ah
006032d4 01                                ??           01h
006032d5 00                                ??           00h
006032d6 00                                ??           00h
006032d7 00                                ??           00h

                                PTR_node2_006032d8
006032d8 e0 32 60 00 00 00              addr         node2
00 00

                                node2[8]
                                node2
006032e0 53 03 00 00 02 00              undefined1[16]
00 00 f0 32 60 00
00 00 00 00

                                node3
006032f0 99 03 00 00 03 00              undefined1[16]
00 00 00 33 60 00
00 00 00 00

                                node4
00603300 36 01 00 00 04 00              undefined1[16]
00 00 10 33 60 00
00 00 00 00

                                node5
00603310 49 02 00 00 05 00              undefined1[16]
00 00 20 33 60 00
00 00 00 00

                                node6
00603320 8a 00 00 00 06 00              undefined1[16]
00 00 00 00 00 00
00 00 00 00

```

被坑害了，`node1`里的`undefined4`应该（？）是规定了一个 4 Bytes 的值，之后那个

PTR_node2_006032d8 是明确地表示出了8字节的指针，中间部分看上去是一个索引，问题很大程度上出现在了之前以为前8个字节都是表示着一个64位的整数。

```
struct Node {
    int value;           // 节点值
    int index;           // 猜测中间部分是一个索引（因为都是01, 02, 03, ..., 06这种的）
    struct Node* next;   // 指向下一个节点的指针
};
```

这下终于有头绪了：

```
# 根据小端序排列，将给定的节点中的4字节整数值提取出来
# 每个节点的起始4字节按小端序存储整数值

# 定义每个节点起始的4字节数据
nodes_bytes = {
    "node1": [0x7a, 0x02, 0x00, 0x00], # 0x0000027a
    "node2": [0x53, 0x03, 0x00, 0x00], # 0x00000353
    "node3": [0x99, 0x03, 0x00, 0x00], # 0x00000399
    "node4": [0x36, 0x01, 0x00, 0x00], # 0x00000136
    "node5": [0x49, 0x02, 0x00, 0x00], # 0x00000249
    "node6": [0x8a, 0x00, 0x00, 0x00], # 0x0000008a
}

# 提取每个节点的整数值
node_values = {node: int.from_bytes(bytes_list, "little") for node, bytes_list in nc

node_values
```

提取出来的结果：

```
{'node1': 634,
 'node2': 851,
 'node3': 921,
 'node4': 310,
 'node5': 585,
 'node6': 138}

3 2 1 5 4 6 // 7-a[i]
4 5 6 2 3 1 // 7-a[i]
```

四、实验总结

毁灭吧，前面的详细分析和这之后的引用还不够吗？逢山开路，遇水架桥，如此而已。

过程中 GPT-4 明确对我的思路有重大帮助的地方是，其告诉我phase_6 中 node 的结构里存储的值是32位的，不知晓这一点，最后一阶段即使前面的思路全对，也无法产生正确的结果。

此外，我非常感激当我让它检验我对 `phase_6` 汇编代码的理解（指所写注释的正确性）时，它的回答：

总的来说，你的注释和理解基本正确，表明你对代码的逻辑有深刻的理解。一些细节，如确切的内存布局和操作的上下文，可能需要更多代码或背景信息来完全理解，但你的分析已经很接近真实功能了。

我谢谢它。

x86中的32位和64位寄存器

Yes, of course. It is the same register, no matter if you address it using 8/16/32/64 bit mode.

<https://stackoverflow.com/questions/43623012/do-32-and-64-bit-values-share-the-same-register-space>

```
| 63..32 | 31..16 | 15-8 | 7-0 |
          | AH. | AL. |
          | AX..... |
          | EAX..... |
          | RAX..... |
```

<https://stackoverflow.com/questions/228200/why-is-there-not-a-register-that-contains-the-higher-bytes-of-eax/228367#228367>

<https://stackoverflow.com/questions/11177137/why-do-x86-64-instructions-on-32-bit-registers-zero-the-upper-part-of-the-full-64-bit-register>

```
test %eax, %eax
```

`test eax, eax` is just an optimized `cmp eax, 0`. It's written this way to save space, as `cmp eax, 0` must encode that zero directly into your program as `00 00 00 00` (yes, that's 4 bytes that are each zero), which wastes space doing the same thing to the zero flag that `test eax, eax` does.

<https://stackoverflow.com/questions/75075395/contradictory-behavior-of-jne-x86-assembly-instruction>

<https://stackoverflow.com/questions/147173/testl-eax-against-eax>

<https://reverseengineering.stackexchange.com/questions/15184/what-does-the-test-instruction-do>

```
rep ret
```

<https://stackoverflow.com/questions/20526361/what-does-rep-ret-mean>

C String

<https://stackoverflow.com/questions/6282198/reading-string-from-input-with-space-character>

<https://stackoverflow.com/questions/1247989/how-do-you-allow-spaces-to-be-entered-using-scanf>

C Char

大受震撼

In C, the type of a character *constant* like `'a'` is actually an `int`, with size of 4 (or some other implementation-dependent value). In C++, the type is `char`, with size of 1. This is one of many small differences between the two languages.

<https://stackoverflow.com/questions/2172943/why-is-the-size-of-a-character-sizeof-a-different-in-c-and-c>

In C++, `sizeof('a') == sizeof(char) == 1`. This makes intuitive sense, since `'a'` is a character literal, and `sizeof(char) == 1` as defined by the standard.

In C however, `sizeof('a') == sizeof(int)`. That is, it appears that C character literals are actually integers. Does anyone know why? I can find plenty of mentions of this C quirk but no explanation for why it exists.

<https://stackoverflow.com/questions/433895/why-are-c-character-literals-ints-instead-of-chars>

Byte & Word Addressing

<https://stackoverflow.com/questions/48129466/why-do-we-use-byte-addressing-instead-of-word-addressing>

js instruction

`JS` will jump if the sign flag is set (by an earlier instruction). `CMP` will always modify the flags by performing a subtraction, in this case `%cl - %al`.

CMP : Subtracts source from destination.

<https://stackoverflow.com/questions/21872334/what-does-js-do-in-assembly-x86>

非常好缩写、有/无符号。

<http://www.unixwiz.net/techtips/x86-jumps.html>

FS segment register

<https://stackoverflow.com/questions/10810203/what-is-the-fs-gs-register-intended-for>

The x86 architecture supports segmentation. Instructions which access memory can use segment register based addressing mode. The following notation is used to address a byte within a segment:

 *Segment-register:Byte-address*

The segment base address is added to the Byte-address to compute the resulting virtual address which is accessed. This allows to access multiple instances of data with the identical Byte-address, i.e. the same code. The selection of a particular instance is purely based on the base-address in the segment register.

The FS segment is commonly used to address Thread Local Storage (TLS). FS is usually managed by runtime code or a threading library. Variables declared with the ‘__thread’ storage class specifier are instantiated per thread and the compiler emits the FS: address prefix for accesses to these variables. Each thread has its own FS base address so common code can be used without complex address offset calculations to access the per thread instances. Applications should not use FS for other purposes when they use runtimes or threading libraries which manage the per thread FS.

https://www.kernel.org/doc/html/next/x86/x86_64/fsgs.html

JLE instruction

<https://stackoverflow.com/questions/9617877/assembly-jg-jnle-jl-jnge-after-cmp>

LEA instruction

也只有在 LEA 指令下，`offset(base, index, multiplier)` 才会被视作所指向的地址，而非其地址所指向的内容。

非常赞同：

Wouldn't it have been cleaner to extend the `mov` instruction and leave off the brackets? `MOV EDX, EBX + 8*EAX + 4`

By replacing LEA with a specialized MOV you keep the syntax clean: `[]` brackets are always the equivalent of dereferencing a pointer in C. Without brackets, you always deal with the pointer itself.

<https://stackoverflow.com/questions/1658294/whats-the-purpose-of-the-lea-instruction>

`__isoc99_sscanf`

This function return the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

<https://stackoverflow.com/questions/69829654/what-does-isoc99-sscanf-do>

<https://stackoverflow.com/questions/56444576/asm-isoc99-scanf-after-function-declaration>

On x86_64, parameters are passed in registers, so your call to scanf has 3 parameters stored in 3 registers:

- | `rdi` pointer to the string `"%u %u"`, the format to parse (two unsigned integers)
- | `rsi` should be a `unsigned *`, pointer to where to put the first parsed integer
- | `rdx` pointer to where to put the second parsed integer.

<https://stackoverflow.com/questions/72492532/in-which-register-does-the-scanf-function-store-input-values>

"address" of a register

<https://stackoverflow.com/questions/5301292/is-there-is-a-way-to-get-the-address-of-a-register>

<https://stackoverflow.com/questions/52308185/are-cpu-general-purpose-registers-usually-memory-mapped>

indirect addressing

```
movl (%edx), %eax
```

Means "the memory at the address that's stored in the register".

<https://stackoverflow.com/questions/61004313/what-do-parentheses-surrounding-a-register-mean>

<https://stackoverflow.com/questions/69967899/indirect-addressing-in-assembly-x86>

<https://stackoverflow.com/questions/46123822/how-to-load-the-contents-of-the-memory-address-stored-in-a-register-in-assembly>

The complete AT&T base/index register syntax is:

```
offset(base, index, multiplier)
```

<https://stackoverflow.com/questions/18650093/what-does-a-comma-in-a-parenthesis-mean-in-the-att-syntax-for-x86-assembly>

GAS memory operand	NASM memory operand
-----	-----
100	[100]
%es:100	[es:100]
(%eax)	[eax]
(%eax,%ebx)	[eax+ebx]
(%ecx,%ebx,2)	[ecx+ebx*2]
(,%ebx,2)	[ebx*2]
-10(%eax)	[eax-10]
%ds:-10(%ebp)	[ds:ebp-10]
Example instructions,	
mov %ax, 100	
mov %eax, -100(%eax)	

<https://stackoverflow.com/questions/6819957/what-does-the-bracket-in-movl-eax-eax-mean/6820015#6820015>

<https://stackoverflow.com/questions/27936196/a-couple-of-questions-about-base-indexscale-disp-and-att-dispbases-index>

%al register

- | **EAX** is the full 32-bit value
- | **AX** is the lower 16-bits
- | **AL** is the lower 8 bits
- | **AH** is the bits 8 through 15 (zero-based), the top half of AX

Parentheses

- | 在 `lea 0x4(%rsi), %rcx` 指令中，`lea` (Load Effective Address) 的作用是计算括号内给出的地址表达式的值，并将这个计算结果（即地址）加载到 `%rcx` 寄存器中。这里，括号内的表达式 `0x4(%rsi)` 表示 `%rsi` 寄存器的值加上 4，这个结果是一个地址，被直接存储在 `%rcx` 中，不进行内存访问。
- | 在 `movq 8(%rbp), %rdx` 指令中，`movq` 是数据传送指令，用于将数据从源位置移动到目标位置。这里的括号 `8(%rbp)` 表示的是一个内存地址，计算方式是 `%rbp` 寄存器的值加上 8。与 `lea` 指令不同，`movq` 会访问该地址指向的内存位置，将那里的数据（64 位或者 8 字节，因为是 `movq`）加载到 `%rdx` 寄存器中。

因此，`lea` 指令的括号用于构造地址表达式，结果是一个地址，而 `movq` 指令的括号用于指定一个内存地址，指令会访问该地址并加载或存储数据。简单来说，`lea` 与地址计算相关，而 `movq` 与实际的内存访问操作相关。▲ ChatGPT 4 生成（其措辞可能会让人疑惑）

end of array

C arrays don't have an end marker.

It is your responsibility as the programmer to keep track of the allocated size of the array to make sure you don't try to access element outside the allocated size.

If you do access an element outside the allocated size, the result is [undefined behaviour](#).

<https://stackoverflow.com/questions/53579155/end-of-array-in-c-language>

Silver Bullet of NSA

Ghidra (pronounced GEE-druh; /'gi:drə/)

<https://ghidra-sre.org/>

Asterisk

The line

```
jmpq    *0x402680(,%rax,8)
```

would be described in RTN by:

```
RIP <- M[0x402680 + (8 * RAX)]
```

where M is the system memory.

As such, we can write the general form `jmpq *c(r1, r2, k)`, where c is an immediate constant, $r1$ and $r2$ are general purpose registers and k is either 1 (default), 2, 4 or 8:

```
RIP <- M[c + r1 + (k * r2)]
```

<https://stackoverflow.com/questions/9223756/what-does-an-asterisk-before-an-address-mean-in-x86-64-att-assembly>

Intel Little Endian

<https://stackoverflow.com/questions/6018386/is-x86-64-machine-language-big-endian>

`__stack_chk_fail`

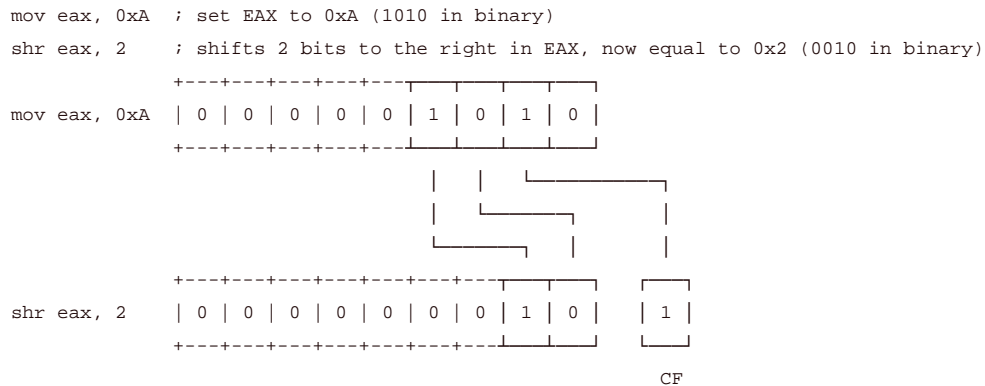
The interface `__stack_chk_fail()` shall abort the function that called it with a message that a stack overflow has been detected. The program that called the function shall then exit.

The interface `__stack_chk_fail()` does not check for a stack overflow itself. It merely reports one when invoked.

http://refspecs.linux-foundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/libc---stack-check-fail-1.html

shr & sar instruction

- | The `shr` or `sar` instruction is used to shift the bits of the operand destination to the right, by the number of bits specified in the count operand.
- | Bits shifted beyond the destination are first shifted into the **CF flag**.
- | Equivalent to dividing by 2



<https://www.aldeid.com/wiki/X86-assembly/Instructions/shr>

OPCODE	MNEMONIC	DESCRIPTION
D2 /5	SHR <i>r/m8</i> ,CL	Unsigned divide <i>r/m8</i> by 2, CL times.
D2 /7	SAR <i>r/m8</i> ,CL	Signed divide* <i>r/m8</i> by 2, CL times.

https://c9x.me/x86/html/file_module_x86_id_285.html

if you perform `SHR 00110000b` you would end up with `00011000b`

If you shifted one bit to the right, yes. You can specify the shift amount, so it's not fixed at 1.

However, if you were to perform `SHR` on `11111111b` you would end up with an incorrect answer

If you did a logical shift of `11111111b` one bit to the right you'd get `01111111b`. Whether you consider that to be incorrect or not depends entirely on what you're trying to achieve. If you wanted to preserve the sign you should've used `SAR`.

<https://stackoverflow.com/questions/30644708/shr-and-sar-commands>

strtol

Interprets an integer value in a byte string pointed to by `str`.

If successful, an integer value corresponding to the contents of `str` is returned.

<https://en.cppreference.com/w/c/string/byte/strtol>

The `strtol()` function converts a character string to a long integer value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric value of type long int.

<https://www.ibm.com/docs/es/i/7.3?topic=lf-strtol-strtol-convert-character-string-long-long-long-integer>

cltq

- ┆ Convert Long To Quad (`cltq`): AT&T-style
- ┆ quad (aka quad-word) == 8 bytes
- ┆ long (AT&T) == double-word (Intel) == 4 bytes

It sign extends `%eax` from 4 bytes into 8 bytes.

<https://stackoverflow.com/questions/6555094/what-does-cltq-do-in-assembly>

数据表示

```
00000000006032d0 <node1>:
6032d0: 7a 02                                jp      6032d4 <node1+0x4>
6032d2: 00 00                                add     %al, (%rax)
6032d4: 01 00                                add     %eax, (%rax)
6032d6: 00 00                                add     %al, (%rax)
6032d8: e0 32                                loopne  60330c <node4+0xc>
6032da: 60                                     (bad)
6032db: 00 00                                add     %al, (%rax)
6032dd: 00 00                                add     %al, (%rax)
...

00000000006032e0 <node2>:
6032e0: 53                                     push    %rbx
6032e1: 03 00                                add     (%rax), %eax
6032e3: 00 02                                add     %al, (%rdx)
6032e5: 00 00                                add     %al, (%rax)
6032e7: 00 f0                                add     %dh, %al
6032e9: 32 60 00                            xor     0x0(%rax), %ah
6032ec: 00 00                                add     %al, (%rax)
...

00000000006032f0 <node3>:
6032f0: 99                                     cld
6032f1: 03 00                                add     (%rax), %eax
6032f3: 00 03                                add     %al, (%rbx)
6032f5: 00 00                                add     %al, (%rax)
6032f7: 00 00                                add     %al, (%rax)
6032f9: 33 60 00                            xor     0x0(%rax), %esp
6032fc: 00 00                                add     %al, (%rax)
...

0000000000603300 <node4>:
603300: 36 01 00                            ss add  %eax, (%rax)
```

```

603303: 00 04 00          add    %al, (%rax,%rax,1)
603306: 00 00            add    %al, (%rax)
603308: 10 33            adc    %dh, (%rbx)
60330a: 60              (bad)
60330b: 00 00            add    %al, (%rax)
60330d: 00 00            add    %al, (%rax)
...

0000000000603310 <node5>:
603310: 49 02 00          rex.WB add (%r8),%al
603313: 00 05 00 00 00 20 add    %al, 0x20000000(%rip)      # 20603319 <_
603319: 33 60 00          xor    0x0(%rax),%esp
60331c: 00 00            add    %al, (%rax)
...

0000000000603320 <node6>:
603320: 8a 00            mov    (%rax),%al
603322: 00 00            add    %al, (%rax)
603324: 06              (bad)
...
```

从您提供的内容看来，这是一个将链表节点直接映射到其内存地址表示的尝试。不过，看起来输出更像是用 `objdump` 或类似工具直接从二进制执行文件中提取的信息，并且以一种对应汇编指令的形式呈现。这种表示法实际上并不直接反映链表节点的数据结构，而是显示了程序二进制文件中相应地址处的原始字节序列如何被解释为汇编指令。

这段输出实际上并不代表链表节点的真实结构或内容。在二进制文件中，数据（如链表节点的值和指针）和代码（如函数指令）通常位于不同的段中。当你用如 `objdump` 这样的工具查看整个程序的内容时，这些工具会尝试将所有内容都解释为指令，即使某些部分实际上是数据。

在你提供的例子中：

- 1 每个 `node` 标签下的字节序列被解释为一系列的汇编指令，但这只是因为工具试图对任何给定的字节序列进行解码。
- 1 比如，地址 `6032d0` 处的字节 `7a 02` 被解释为 `jp 6032d4` 指令，但实际上这可能只是一个整数值 `0x027a`，正如之前提到的链表节点值。
- 1 接下来的字节通常代表一个指向另一节点的指针，但在这里，它们被错误地解释为汇编指令或者 `(bad)` 标记（表示无法将字节序列解释为有效的汇编指令）。

要正确理解链表节点的结构，你应该基于你对程序逻辑的了解，以及如何在程序中定义和操作这些节点的知识。正如之前讨论的，每个节点通常包含一个或多个数据字段（例如，一个整数值或其他类型的标识符）和一个或多个指向其他节点的指针。

如果你有这些节点的C语言结构定义，或者程序的源代码，那将是理解这些节点如何在内存中布局的最直接方式。反之，如果你正在尝试通过分析二进制文件来理解这个结构，你需要更加小心，不能直接将二进制内容作为指令序列来解释。 **⚠ ChatGPT 4 生成**

`undefined4` 和 `undefined1[16]`

- 1 **undefined4 0000027Ah**：这表示在给定地址处有一个4字节（32位）的未定义数据，其

值为 0x027A。在这个上下文中，“未定义”意味着该工具（可能是一个反汇编器或类似的工具）没有足够的信息来确定这些字节代表的确切数据类型，但它知道这是一个4字节长的值。这很可能是链表节点的整数值部分。

- 1 **undefined1[16]**：这表示在给定地址处有16字节的未定义数据。这个表示法没有明确地拆分这16字节的结构，比如哪些是数据值，哪些是指向其他节点的指针。但是，根据您先前描述的节点结构（一个整数值加上一个指针），我们可以推断这16字节很可能被分为两部分：一个用于存储节点的值（通常是前4字节），另一个用于存储指向下一个节点的指针（剩下的字节）。

后续节点显示为 `undefined1[16]`，原因可能与工具如何解释和展示内存内容有关。有时候，工具可能会在首次识别到特定类型的数据时给出更具体的信息（如 `undefined4`），但在后续相似的结构中，它可能会选择一种更通用的表示方式（如 `undefined1[16]`）。**⚠**

ChatGPT 4 生成