

天津大学

深入理解计算机系统课程实验

题目：实验二 bomb lab

学生姓名	<u>王雨朦</u>
学生学号	<u>2016229082</u>
学院名称	<u>国际工程师学院</u>
专业	<u>计算机</u>
时间	<u>2017/8/7</u>

目录

一、 实验目的	1
二、 实验要求和方法	1
• 实验要求	1
• 实验方法	1
三、 实验环境及设备	1
四、 实验步骤及函数思路	1
• 实验步骤	1
• 函数思路	2
A.Phase_1	2
B.Phase_2	2
C.Phase_3	3
D.Phase_4	4
E. Phase_5	6
F. Phase_6	7
五、 实验结果及截图	9

一、 实验目的

炸弹实验教实验者机器级程序的原则，以及一般调试器和反向工程等技能。

二、 实验要求和方法

- 实验要求

“二进制炸弹”是由六个 Linux 可执行 C 程序组成的。每个阶段期望依照 stdin 输入一个特定的字符串。如果输入预期的字符串，那么该阶段被“消除”；否则炸弹“爆炸”打印“BOOM !!!”。实验者的目标是尽可能多的阶段化解。

- 实验方法

为了消除炸弹，实验者必须使用调试器（通常是 gdb 或 ddd）来分解每个阶段的机器码的二进制和单步。即要了解每个汇编语句的作用，然后使用这个知识推断出消除的字符串。

三、 实验环境及设备

- 实验环境

VMware 虚拟机 Ubuntu16.04 (X64)

- 语言

C 语言

四、 实验步骤及函数思路

- 实验步骤

- 1、在 linux 系统中解压 bomb 并在控制台进入文件；
- 2、首先用 objdump 得到 bomb 反汇编代码：`# objdump -d bomb > bomb.s`；
- 3、打开 bomb.s 找到 phase_1, phase_2... phase_6,共 6 个函数；
- 4、接着对逐个函数进行 gdb 分析，设置断点调试，直到找到对应字符串为止。

- 函数思路

A. Phase_1

```
calmon@ubuntu: ~/lab/bomb
000000000400ee0 <phase_1>:
400ee0: 48 83 ec 08      sub    $0x8,%rsp
400ee4: be 00 24 40 00    mov    $0x402400,%esi
400ee9: e8 4a 04 00 00    callq 401338 <strings_not_equal>
400eee: 85 c0            test   %eax,%eax
400ef0: 74 05            je     400ef7 <phase_1+0x17>
400ef2: e8 43 05 00 00    callq 40143a <explode_bomb>
400ef7: 48 83 c4 08      add    $0x8,%rsp
400efb: c3              retq
```

➤ 分析

仔细分析第一个函数，发现这函数只是调用了 `strings_not_equal` 来判断输入的字符串是否与标准字符串相等，不等就爆炸。由上面的分析到标准字符串保存在 `0x402400` 位置处。

➤ 调试

用 GDB 调试： `gdb bomb`，再用 `x/s 402400` 命令查看字符串的值，得到 "Border relations with Canada have never been better." 这个字符串。

```
(gdb) x/s 0x402400
0x402400: "Border relations with Canada have never been better."
```

➤ Solution1: Border relations with Canada have never been better.

B. Phase_2

```
calmon@ubuntu: ~/lab/bomb
000000000400efc <phase_2>:
400efc: 55              push   %rbp
400efd: 53              push   %rbx
400efe: 48 83 ec 28     sub    $0x28,%rsp
400f02: 48 89 e6        mov    %rsp,%rsi
400f05: e8 52 05 00 00    callq 40145c <read_six_numbers>
400f0a: 83 3c 24 01     cmpl   $0x1,(%rsp)
400f0e: 74 20            je     400f30 <phase_2+0x34>
400f10: e8 25 05 00 00    callq 40143a <explode_bomb>
400f15: eb 19            jmp    400f30 <phase_2+0x34>
400f17: 8b 43 fc        mov    -0x4(%rbx),%eax
400f1a: 01 c0            add    %eax,%eax
400f1c: 39 03            cmp    %eax,(%rbx)
400f1e: 74 05            je     400f25 <phase_2+0x29>
400f20: e8 15 05 00 00    callq 40143a <explode_bomb>
400f25: 48 83 c3 04     add    $0x4,%rbx
400f29: 48 39 eb        cmp    %rbp,%rbx
400f2c: 75 e9            jne    400f17 <phase_2+0x1b>
400f2e: eb 0c            jmp    400f3c <phase_2+0x40>
400f30: 48 8d 5c 24 04    lea    0x4(%rsp),%rbx
400f35: 48 8d 6c 24 18    lea    0x18(%rsp),%rbp
400f3a: eb db            jmp    400f17 <phase_2+0x1b>
400f3c: 48 83 c4 28     add    $0x28,%rsp
400f40: 5b              pop    %rbx
400f41: 5d              pop    %rbp
400f42: c3              retq
```

➤ 分析

调用 `read_six_numbers` 这个函数，读入 6 个数字。6 个数字满足关系， $a[n+1] = a[n]*2, n=0...5$ ，且 $a[0] = 1$ ，这一节主要就考察循环。

➤ 调试

用 GDB 单步调试可得到 `0x4025c3` 保存 `"%d %d %d %d %d %d"` 字符串，`0x400f0a` 中保存的是输入的 6 个数字的起始位置。

➤ Solution2: 1 2 4 8 16 32

C. Phase_3

```

calmon@ubuntu: ~/lab/bomb
000000000400f43 <phase_3>:
400f43: 48 83 ec 18      sub    $0x18,%rsp
400f47: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx
400f4c: 48 8d 54 24 08    lea    0x8(%rsp),%rdx
400f51: be cf 25 40 00    mov    $0x4025cf,%esi
400f56: b8 00 00 00 00    mov    $0x0,%eax
400f5b: e8 90 fc ff ff    callq 400bf0 <__isoc99_sscanf@plt>
400f60: 83 f8 01          cmp    $0x1,%eax
400f63: 7f 05            jg     400f6a <phase_3+0x27>
400f65: e8 d0 04 00 00    callq 40143a <explode_bomb>
400f6a: 83 7c 24 08 07    cmpl   $0x7,0x8(%rsp)
400f6f: 77 3c            ja     400fad <phase_3+0x6a>
400f71: 8b 44 24 08       mov    0x8(%rsp),%eax
400f75: ff 24 c5 70 24 40 00 jmpq   *0x402470(,%rax,8)
400f7c: b8 cf 00 00 00    mov    $0xcf,%eax
400f81: eb 3b            jmp     400fbe <phase_3+0x7b>
400f83: b8 c3 02 00 00    mov    $0x2c3,%eax
400f88: eb 34            jmp     400fbe <phase_3+0x7b>
400f8a: b8 00 01 00 00    mov    $0x100,%eax
400f8f: eb 2d            jmp     400fbe <phase_3+0x7b>
400f91: b8 85 01 00 00    mov    $0x185,%eax
400f96: eb 26            jmp     400fbe <phase_3+0x7b>
400f98: b8 ce 00 00 00    mov    $0xce,%eax
400f9d: eb 1f            jmp     400fbe <phase_3+0x7b>
400f9f: b8 aa 02 00 00    mov    $0x2aa,%eax
400fa4: eb 18            jmp     400fbe <phase_3+0x7b>
400fa6: b8 47 01 00 00    mov    $0x147,%eax
400fab: eb 11            jmp     400fbe <phase_3+0x7b>
400fad: e8 88 04 00 00    callq 40143a <explode_bomb>
400fb2: b8 00 00 00 00    mov    $0x0,%eax

400fb2: b8 00 00 00 00    mov    $0x0,%eax
400fb7: eb 05            jmp     400fbe <phase_3+0x7b>
400fb9: b8 37 01 00 00    mov    $0x137,%eax
400fbe: 3b 44 24 0c       cmp    0xc(%rsp),%eax
400fc2: 74 05            je     400fc9 <phase_3+0x86>
400fc4: e8 71 04 00 00    callq 40143a <explode_bomb>
400fc9: 48 83 c4 18       add    $0x18,%rsp
400fcd: c3              retq

```

➤ 分析

在 `400f6a` 得知第一个数不能超过 7，否则直接触雷，满足条件则将第一个数赋给 `rax`。`400f75` 这句对应为 `switch` 语句，指针的值 `0x402470+8*$rax`，`%rax` 中保存的是第一个数字，`%eax` 第二个字符，`(%rsp)-0x8` 第三个数字。这一关主要考察的是 `switch` 语句。0-7 为所有的情况。

➤ 调试

在 `400f75`，这是一个跳转命令，它要跳转到的地址与 `rax` 有关：目的地址是一个数，这个数存在一个地方，那个地方的地址是由基础地址 `0x402470` 加上

8*`rax` 得到的。用 `x/a` 读取这个基础地址，发现那里存着一个数字 `0x400f7c`，我们发现在这个地址处 `rax` 被赋予了值 `0xcf`，然后跳转 `400fbe` 将其与 `rcx` 比较，也就是我们输入的第二个数比较，如果相等就能跳过最后一个炸弹通过这个 phase，很显然我们如果输入第一个数 0 的话，第二个数就应该是 `0xcf` 了。考虑到第一个数目前只有小于等于 7 的限制，我们再用 `x/8a` 读取那个基础地址，得到：

```
0x402470: 0x400f7c <phase_3+57> 0x400fb9 <phase_3+118>
0x402480: 0x400f83 <phase_3+64> 0x400f8a <phase_3+71>
0x402490: 0x400f91 <phase_3+78> 0x400f98 <phase_3+85>
0x4024a0: 0x400f9f <phase_3+92> 0x400fa6 <phase_3+99>
```

➤ Solution3:

0 207

1 311

2 707

3 256

4 389

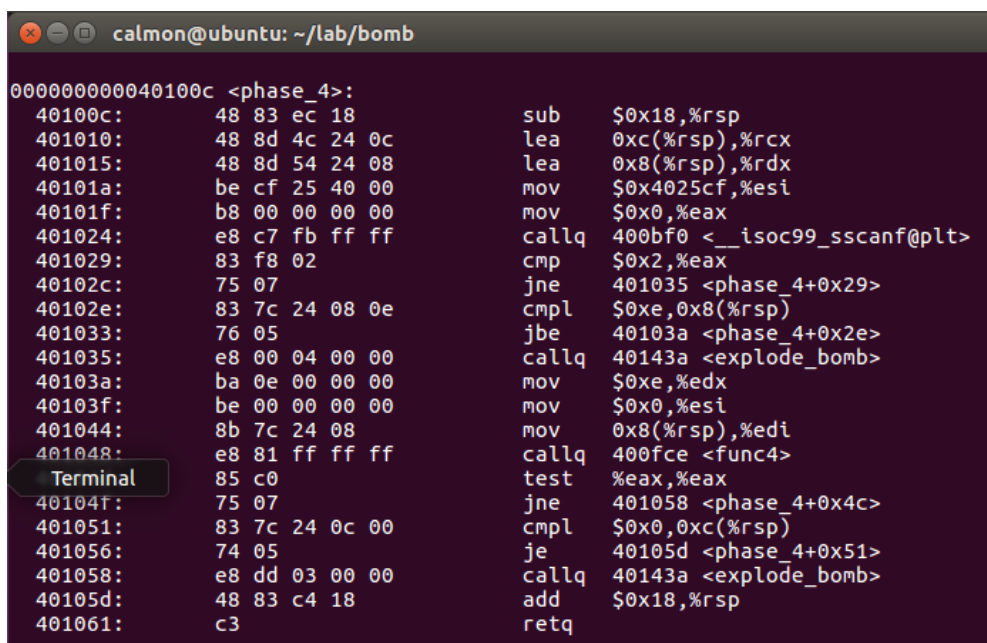
5 206

6 682

7 327

输入上述任意一个组合都能通过这个 phase。

D. Phase_4



```
calmon@ubuntu: ~/lab/bomb
00000000040100c <phase_4>:
40100c: 48 83 ec 18      sub    $0x18,%rsp
401010: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx
401015: 48 8d 54 24 08    lea    0x8(%rsp),%rdx
40101a: be cf 25 40 00    mov    $0x4025cf,%esi
40101f: b8 00 00 00 00    mov    $0x0,%eax
401024: e8 c7 fb ff ff    callq  400bf0 <__isoc99_sscanf@plt>
401029: 83 f8 02         cmp    $0x2,%eax
40102c: 75 07           jne    401035 <phase_4+0x29>
40102e: 83 7c 24 08 0e    cmpl   $0xe,0x8(%rsp)
401033: 76 05           jbe    40103a <phase_4+0x2e>
401035: e8 00 04 00 00    callq  40143a <explode_bomb>
40103a: ba 0e 00 00 00    mov    $0xe,%edx
40103f: be 00 00 00 00    mov    $0x0,%esi
401044: 8b 7c 24 08      mov    0x8(%rsp),%edi
401048: e8 81 ff ff ff    callq  400fce <func4>
Terminal 40104f: 85 c0          test   %eax,%eax
401051: 75 07           jne    401058 <phase_4+0x4c>
401056: 83 7c 24 0c 00    cmpl   $0x0,0xc(%rsp)
401058: 74 05           je     40105d <phase_4+0x51>
40105d: e8 dd 03 00 00    callq  40143a <explode_bomb>
401061: 48 83 c4 18      add    $0x18,%rsp
c3           retq
```

```

calmon@ubuntu: ~/lab/bomb
0000000000400fce <func4>:
400fce:  48 83 ec 08      sub    $0x8,%rsp
400fd2:  89 d0            mov    %edx,%eax
400fd4:  29 f0            sub    %esi,%eax
400fd6:  89 c1            mov    %eax,%ecx
400fd8:  c1 e9 1f         shr    $0x1f,%ecx
400fdb:  01 c8            add    %ecx,%eax
400fdd:  d1 f8            sar    %eax
400fdf:  8d 0c 30         lea    (%rax,%rsi,1),%ecx
400fe2:  39 f9            cmp    %edi,%ecx
400fe4:  7e 0c            jle    400ff2 <func4+0x24>
400fe6:  8d 51 ff         lea    -0x1(%rcx),%edx
400fe9:  e8 e0 ff ff ff   callq  400fce <func4>
400fee:  01 c0            add    %eax,%eax
400ff0:  eb 15            jmp    401007 <func4+0x39>
400ff2:  b8 00 00 00 00   mov    $0x0,%eax
400ff7:  39 f9            cmp    %edi,%ecx
400ff9:  7d 0c            jge    401007 <func4+0x39>
400ffb:  8d 71 01         lea    0x1(%rcx),%esi
400ffe:  e8 cb ff ff ff   callq  400fce <func4>
401003:  8d 44 00 01         lea    0x1(%rax,%rax,1),%eax
401007:  48 83 c4 08      add    $0x8,%rsp
40100b:  c3              retq

```

➤ 分析

这个函数做的事情主要是递归函数 func4。如果我们仔细看 400fe2 和 400ff7 两处 cmp 命令及紧跟的 jle 和 jge，不难发现如果 edi 和 ecx 两个参数相等，就能直接跳过两次结束 func4 了，并且 400ff2 处 eax 也被赋予了 0。

可以看出，每次递归调用 func4 后如果经过处理的 ecx 能与我们输入的第一个数相等，就能返回正确结果。如果要让 ecx 比第一个数大，我们的数要小于 7 才行。

➤ 调试

40104d 中，需要 ZF 为 0 才能避免满足 jne 的条件触雷，也就是说 eax 必须为 0，其后 401051 可见，第二个数必须为 0，否则会在 401058 无法跳过触雷，这样我们就确定了一个解：0 0

每次递归调用 func4 后如果经过处理的 ecx 能与我们输入的第一个数相等，就能返回正确结果。

这样另外几个正确解就是：

1 0

3 0

7 0

➤ Solution4:

0 0

1 0

3 0

7 0

E. Phase_5

```

calmon@ubuntu: ~/lab/bomb
000000000401062 <phase_5>:
401062: 53                push    %rbx
401063: 48 83 ec 20       sub     $0x20,%rsp
401067: 48 89 fb         mov     %rdi,%rbx
40106a: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
401071: 00 00
401073: 48 89 44 24 18     mov     %rax,0x18(%rsp)
401078: 31 c0            xor     %eax,%eax
40107a: e8 9c 02 00 00     callq  40131b <string_length>
40107f: 83 f8 06         cmp     $0x6,%eax
401082: 74 4e            je      4010d2 <phase_5+0x70>
401084: e8 b1 03 00 00     callq  40143a <explode_bomb>
401089: eb 47            jmp     4010d2 <phase_5+0x70>
40108b: 0f b6 0c 03       movzbl  (%rbx,%rax,1),%ecx
40108f: 88 0c 24         mov     %cl,(%rsp)
401092: 48 8b 14 24       mov     (%rsp),%rdx
401096: 83 e2 0f         and     $0xf,%edx
Terminal 0f b6 92 b0 24 40 00 movzbl  0x4024b0(%rdx),%edx
4010a0: 88 54 04 10       mov     %dl,0x10(%rsp,%rax,1)
4010a4: 48 83 c0 01       add     $0x1,%rax
4010a8: 48 83 f8 06       cmp     $0x6,%rax
4010ac: 75 dd            jne     40108b <phase_5+0x29>
4010ae: c6 44 24 16 00     movb    $0x0,0x16(%rsp)
4010b3: be 5e 24 40 00     mov     $0x40245e,%esi
4010b8: 48 8d 7c 24 10     lea     0x10(%rsp),%rdi
4010bd: e8 76 02 00 00     callq  401338 <strings_not_equal>
4010c2: 85 c0            test    %eax,%eax
4010c4: 74 13            je      4010d9 <phase_5+0x77>
4010c6: e8 6f 03 00 00     callq  40143a <explode_bomb>
4010cb: 0f 1f 44 00 00     nopl    0x0(%rax,%rax,1)
4010cb: 0f 1f 44 00 00     nopl    0x0(%rax,%rax,1)
4010d0: eb 07            jmp     4010d9 <phase_5+0x77>
4010d2: b8 00 00 00 00     mov     $0x0,%eax
4010d7: eb b2            jmp     40108b <phase_5+0x29>
4010d9: 48 8b 44 24 18     mov     0x18(%rsp),%rax
4010de: 64 48 33 04 25 28 00 xor     %fs:0x28,%rax
4010e5: 00 00
4010e7: 74 05            je      4010ee <phase_5+0x8c>
4010e9: e8 42 fa ff ff     callq  400b30 <__stack_chk_fail@plt>
4010ee: 48 83 c4 20       add     $0x20,%rsp
4010f2: 5b              pop     %rbx
4010f3: c3              retq

```

➤ 分析

这一关考察指针。分析知程序设置了一个 table，地址 0x 4024b0，表中有 16 个字符，"maduiersnfotvbyl"。

输入的字符串为 6 个字符，6 个字符的低 4 位值对应为偏移量(0-f)，由偏移从表中得到新的字符串应该为"flyers"，存放位置 0x40245e。回正确结果。如果要比第一个数大，我们的数要小于 7 才行。

➤ 调试

按照 flyers 中字母在这个字符串中的位置一个一个找下来，我们需要的偏移量分别是：9 15 14 5 6 7

转换成 2 进制：1001 1111 1100 0101 0110 0111

查 ASCII 码表，不难得知很多字符的低 4 位都能满足需要，我们只需要找一个就行，比如："9?>567"，这样就通过了 phase_5

➤ Solution5:

9?>567

F. Phase_6

```

calmon@ubuntu: ~/lab/bomb
0000000004010f4 <phase_6>:
4010f4: 41 56          push    %r14
4010f6: 41 55          push    %r13
4010f8: 41 54          push    %r12
4010fa: 55            push    %rbp
4010fb: 53            push    %rbx
4010fc: 48 83 ec 50    sub     $0x50,%rsp
401100: 49 89 e5       mov     %rsp,%r13
401103: 48 89 e6       mov     %rsp,%rsi
401106: e8 51 03 00 00 callq   40145c <read_six_numbers>
40110b: 49 89 e6       mov     %rsp,%r14
40110e: 41 bc 00 00 00 00 mov     $0x0,%r12d
Backups 4c 89 ed       mov     %r13,%rbp
41 8b 45 00     mov     0x0(%r13),%eax
40111b: 83 e8 01       sub     $0x1,%eax
40111e: 83 f8 05       cmp     $0x5,%eax
401121: 76 05         jbe     401128 <phase_6+0x34>
401123: e8 12 03 00 00 callq   40143a <explode_bomb>
401128: 41 83 c4 01    add     $0x1,%r12d
40112c: 41 83 fc 06    cmp     $0x6,%r12d
401130: 74 21         je      401153 <phase_6+0x5f>
401132: 44 89 e3       mov     %r12d,%ebx
401135: 48 63 c3       movslq  %ebx,%rax
401138: 8b 04 84       mov     (%rsp,%rax,4),%eax
40113b: 39 45 00       cmp     %eax,0x0(%rbp)
40113e: 75 05         jne     401145 <phase_6+0x51>
401140: e8 f5 02 00 00 callq   40143a <explode_bomb>
401145: 83 c3 01       add     $0x1,%ebx
401148: 83 fb 05       cmp     $0x5,%ebx

40114b: 7e e8         jle     401135 <phase_6+0x41>
40114d: 49 83 c5 04    add     $0x4,%r13
401151: eb c1         jmp     401114 <phase_6+0x20>
401153: 48 8d 74 24 18 lea     0x18(%rsp),%rsi
401158: 4c 89 f0       mov     %r14,%rax
40115b: b9 07 00 00 00 mov     $0x7,%ecx
401160: 89 ca         mov     %ecx,%edx
401162: 2b 10         sub     (%rax),%edx
401164: 89 10         mov     %edx,(%rax)
401166: 48 83 c0 04    add     $0x4,%rax
40116a: 48 39 f0       cmp     %rsi,%rax
40116d: 75 f1         jne     401160 <phase_6+0x6c>
40116f: be 00 00 00 00 mov     $0x0,%esi
401174: eb 21         jmp     401197 <phase_6+0xa3>
401176: 48 8b 52 08    mov     0x8(%rdx),%rdx
40117a: 83 c0 01       add     $0x1,%eax
40117d: 39 c8         cmp     %ecx,%eax
40117f: 75 f5         jne     401176 <phase_6+0x82>
401181: eb 05         jmp     401188 <phase_6+0x94>
401183: ba d0 32 60 00 mov     $0x6032d0,%edx
401188: 48 89 54 74 20 mov     %rdx,0x20(%rsp,%rsi,2)
40118d: 48 83 c6 04    add     $0x4,%rsi
401191: 48 83 fe 18    cmp     $0x18,%rsi
401195: 74 14         je      4011ab <phase_6+0xb7>
401197: 8b 0c 34       mov     (%rsp,%rsi,1),%ecx
40119a: 83 f9 01       cmp     $0x1,%ecx
40119d: 7e e4         jle     401183 <phase_6+0x8f>
40119f: b8 01 00 00 00 mov     $0x1,%eax
4011a4: ba d0 32 60 00 mov     $0x6032d0,%edx
4011a9: eb cb         jmp     401176 <phase_6+0x82>

```

```

4011ab: 48 8b 5c 24 20    mov     0x20(%rsp),%rbx
4011b0: 48 8d 44 24 28    lea     0x28(%rsp),%rax
4011b5: 48 8d 74 24 50    lea     0x50(%rsp),%rsi
4011ba: 48 89 d9          mov     %rbx,%rcx
4011bd: 48 8b 10          mov     (%rax),%rdx
4011c0: 48 89 51 08       mov     %rdx,0x8(%rcx)
4011c4: 48 83 c0 08       add     $0x8,%rax
4011c8: 48 39 f0          cmp     %rsi,%rax
4011cb: 74 05            je      4011d2 <phase_6+0xde>
4011cd: 48 89 d1          mov     %rdx,%rcx
4011d0: eb eb            jmp     4011bd <phase_6+0xc9>
4011d2: 48 c7 42 08 00 00 00 movq    $0x0,0x8(%rdx)
4011d9: 00
4011da: bd 05 00 00 00    mov     $0x5,%ebp
4011df: 48 8b 43 08       mov     0x8(%rbx),%rax
4011e3: 8b 00             mov     (%rax),%eax
4011e5: 39 03            cmp     %eax,(%rbx)
4011e7: 7d 05            jge     4011ee <phase_6+0xfa>
4011e9: e8 4c 02 00 00    callq   40143a <explode_bomb>
4011ee: 48 8b 5b 08       mov     0x8(%rbx),%rbx
4011f2: 83 ed 01          sub     $0x1,%ebp
4011f5: 75 e8            jne     4011df <phase_6+0xeb>
4011f7: 48 83 c4 50       add     $0x50,%rsp
4011fb: 5b              pop     %rbx
4011fc: 5d              pop     %rbp
4011fd: 41 5c            pop     %r12
4011ff: 41 5d            pop     %r13
401201: 41 5e            pop     %r14
401203: c3              retq

```

➤ 分析

这是在对一个链表进行操作，node 是一个结构体，含有两个量，第一个是整数，第二个则是一个指针指向下一个 node。

```

struct node {
    unsigned int data;
    struct LElement *next;
}

```

0x0000014c 是一个已经有了的链表的第一个结点的位置。程序要做的是根据输入的 6 个不同大小的数字(1-6)来重新排列这个链表使其按降序排列。

\$ebp-0x3c 保存的是初始链表的头指针,链表的数据元素是个结构体,

%rdx 寄存器是用来传递指针的,把握住它的值这个程序就好读一些.

输入的数字与对应的指针如下:

1 -> 0x6032d0 -> data = 14c

2 -> 0x6032e0 -> data = a8

3 -> 0x6032f0 -> data = 39c

4 -> 0x603300 -> data = 2b3

5 -> 0x603310 -> data = 1dd

6 -> 0x603320 -> data = 1bb

由最后的循环来看，新的链表中每个节点的值都要大于下一个的，是判断链表是否为降序排列。有 $*p[0] > *p[1] > *p[2] \dots > *p[5]$

则有 $2b3 > 39c > a8 > 14c > 1bb > 1dd$

对应数字顺序为: 4 3 2 1 6 5

➤ 调试

401183, edx 赋成 6032d0, 然后把 rdx 存到 rsp+32 的位置。6032d0 那里存着什么呢? 用 x/a 命令看, 那里存着一个数 0x0000014c, 并且看起来是一个 node; 401176 处, rdx 被赋值成 rdx+8 所指向的东西, 看看 0x6032d8, 那里存着

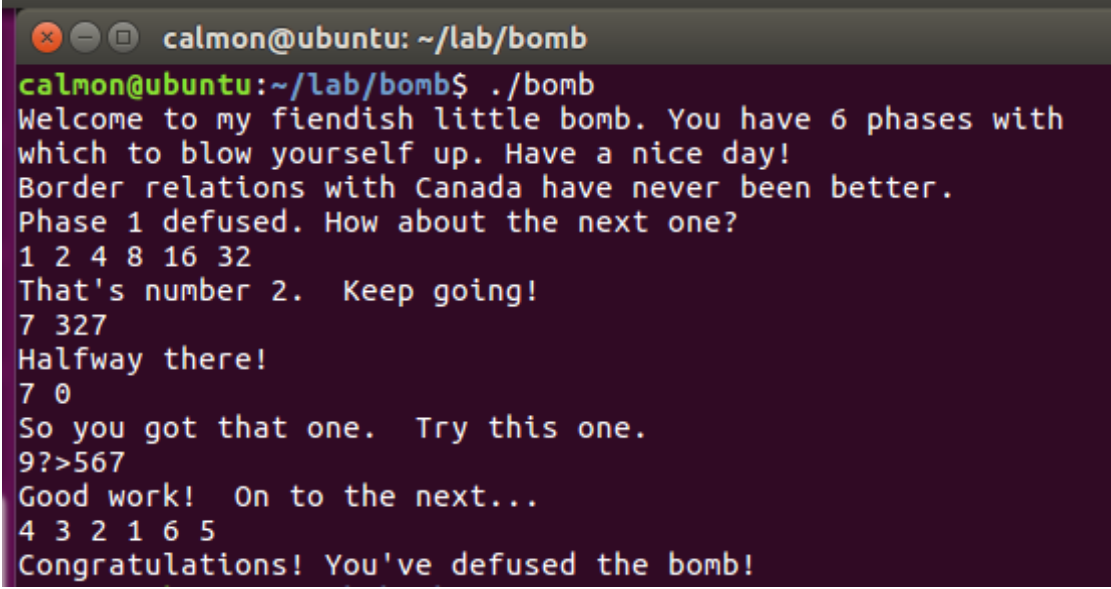
0x6032e0 (node2 的地址), 这样就能猜出来, node 是一个结构体。

➤ Solution6:

4 3 2 1 6 5

五、 实验结果及截图

🌀 Solutions:
Border relations with Canada have never been better.
1 2 4 8 16 32
7 327
7 0
9?>567
4 3 2 1 6 5



```
calmon@ubuntu: ~/lab/bomb
calmon@ubuntu:~/lab/bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
7 327
Halfway there!
7 0
So you got that one. Try this one.
9?>567
Good work! On to the next...
4 3 2 1 6 5
Congratulations! You've defused the bomb!
```