《计算机系统》

BombLab 实验报告

班级: GitHub

学号: 78268851

姓名: AnicoderAndy

目录

1 实验项目

1.1 项目名称

BombLab 拆弹实验。

1.2 实验目的

- 熟悉 gdb 调试工具的使用，掌握基本调试命令。
- 理解 x86 汇编代码的执行逻辑，掌握函数调用约定、寄存器使用等底层知识。
- 熟悉逆向工程的基本方法，掌握逆向工程的基本技巧，培养逆向工程和二进制分析的能力。

1.3 实验资源

`bomb.tar` 压缩包，包含了 `bomb` 二进制文件、`bomb.c` 和 `README` 文件。

运行 `bomb` 发现需要输入 6 个阶段的密码以破解炸弹。使用 `objdump -D bomb > dumpfile.s` 得到反汇编文件。每次输入密码后会调用当前阶段的函数以验证密码。下面通过分析这些函数的逻辑得到每个阶段的密码。

2 实验任务

2.1 Phase 1

通过 objdump 得到 `phase_1` 函数的汇编代码如下[1]：

```
1  sub    $0x14,%esp
2  push   $0x8049fa4
3  push   0x1c(%esp)
4  call   8048f90 <strings_not_equal>
5  add    $0x10,%esp
6  test   %eax,%eax
7  je     8048b50 <phase_1+0x1d>        ; line 9
8  call   8049087 <explode_bomb>
9  add    $0xc,%esp
10 ret
```

该函数传入一个字符串参数，调用 `strings_not_equal` 函数比较传入字符串与内存地址 `0x8049fa4` 处的字符串是否相等，如果不相等函数 `strings_not_equal` 会返回 1，无法执行第 7 行的跳转命令，从而调用 `explode_bomb` 函数引爆炸弹。

运行 `gdb -q bomb`，输入 `x/s 0x8049fa4` 查看内存地址 `0x8049fa4` 处的字符串为“`The moon unit will be divided into two divisions.`”，因此输入该字符串即可通过第一关。

2.2 Phase 2

通过 objdump 得到 `phase_2` 函数的汇编代码。

```
1  push   %esi
2  push   %ebx
3  sub    $0x2c,%esp
4  mov    %gs:0x14,%eax
5  mov    %eax,0x24(%esp)
6  xor    %eax,%eax
7  lea    0xc(%esp),%eax
8  push   %eax                          ; address of the first number
9  push   0x3c(%esp)
10 call   80490ac <read_six_numbers>
11 add    $0x10,%esp
12 cmpl   $0x0,0x4(%esp)
13 jne    8048b84 <phase_2+0x30>        ; line 16
14 cmpl   $0x1,0x8(%esp)
15 je     8048b89 <phase_2+0x35>        ; line 17
16 call   8049087 <explode_bomb>
```

[1]本报告给出的所有汇编代码中，跳转指令后注释的数字表示该指令跳转目标位置代码中的行号。

以上的逻辑通过 `read_six_numbers` 函数读入 6 个整数（分析 `read_six_numbers` 部分的汇编代码得到该函数第一个参数传入输入字符串，第二个参数传入存储 6 个整数的首地址）。之后比较输入的第一个和第二个整数是否为 `0` 和 `1`，如果不是则引爆炸弹。

```
lea    0x4(%esp),%ebx
lea    0x14(%esp),%esi
mov    0x4(%ebx),%eax
add    (%ebx),%eax
cmp    %eax,0x8(%ebx)
je     8048ba0 <phase_2+0x4c>        ; line 24
call   8049087 <explode_bomb>
add    $0x4,%ebx
cmp    %esi,%ebx
jne    8048b91 <phase_2+0x3d>        ; line 19
mov    0x1c(%esp),%eax
xor    %gs:0x14,%eax
je     8048bb9 <phase_2+0x65>
call   8048790 <__stack_chk_fail@plt>
add    $0x24,%esp
pop    %ebx
pop    %esi
ret
```

分析可知以上部分的逻辑通过循环检查 6 个整数。每次循环检查当前位置的整数与下一整数位置的和是否等于下下一整数位置的值，如果不等则引爆炸弹，大致可以用以下 C 代码表示：

```
for (int* p = numbers; p < numbers + 4; p++)
    if (*p + *(p + 1) != *(p + 2))
        explode_bomb();
```

即需保证输入序列是前两项为 0, 1 的斐波那契数列。因此输入 `0 1 1 2 3 5` 即可通过第二关。

2.3 Phase 3

使用 objdump 得到 `phase_3` 函数的汇编代码如下：

```
sub    $0x1c,%esp
mov    %gs:0x14,%eax
mov    %eax,0xc(%esp)
xor    %eax,%eax
lea    0x8(%esp),%eax                ; address of input2
push   %eax
lea    0x8(%esp),%eax                ; address of input1
push   %eax
push   $0x804a16f                    ; "%d %d"
```

```
10  push   0x2c(%esp)
11  call   8048810 <__isoc99_sscanf@plt>
12  add    $0x10,%esp
13  cmp    $0x1,%eax
14  jg     8048bf3 <phase_3+0x34>          ; line 16
15  call   8049087 <explode_bomb>
16  cmpl   $0x7,0x4(%esp)
17  ja     8048c36 <phase_3+0x77>          ; line 40
18  mov    0x4(%esp),%eax
19  jmp    *0x804a000(,%eax,4)
20  mov    $0x2db,%eax                     ; 1 jumps here
21  jmp    8048c47 <phase_3+0x88>          ; line 38
22  mov    $0x107,%eax                     ; 2 jumps here
23  jmp    8048c47 <phase_3+0x88>          ; line 38
24  mov    $0x156,%eax                     ; 3 jumps here
25  jmp    8048c47 <phase_3+0x88>          ; line 38
26  mov    $0x1f8,%eax                     ; 4 jumps here
27  jmp    8048c47 <phase_3+0x88>          ; line 38
28  mov    $0x15d,%eax                     ; 5 jumps here
29  jmp    8048c47 <phase_3+0x88>          ; line 38
30  mov    $0x150,%eax                     ; 6 jumps here
31  jmp    8048c47 <phase_3+0x88>          ; line 38
32  mov    $0x27f,%eax                     ; 7 jumps here
33  jmp    8048c47 <phase_3+0x88>          ; line 38
34  call   8049087 <explode_bomb>
35  mov    $0x0,%eax
36  jmp    8048c47 <phase_3+0x88>          ; line 38
37  mov    $0x2ae,%eax                     ; 0 jumps here
38  cmp    0x8(%esp),%eax
39  je     8048c52 <phase_3+0x93>
40  call   8049087 <explode_bomb>
41  mov    0xc(%esp),%eax
42  xor    %gs:0x14,%eax
43  je     8048c64 <phase_3+0xa5>
44  call   8048790 <__stack_chk_fail@plt>
45  add    $0x1c,%esp
46  ret
```

这段代码首先用 `sscanf` 函数读入两个整数，并且要求第二个输入数在区间 $[0, 7]$ 以内（第 5–17 行）。观察第 18–36 行汇编代码，发现这是一个典型的 `switch` 语句，根据输入的第一个整数的值跳转到不同的位置。根据第 19 行给出的地址 `0x804a000`，在 gdb 中找到跳转地址表，根据跳转地址表找到不同输入值跳转到的位置（已在注释中标注）。该结构的每个 `case` 都会将一个特定值移入寄存器 `eax`，然后与第二个输入值比较，如果相等则通过。从 8 个可能的组合中任选一个（例如选择 `0 686`）即可通过第三关。

2.4 Phase 4

2.4.1 校验函数

使用 objdump 得到 `phase_4` 函数的汇编代码如下所示：

```
1  sub    $0x1c,%esp
2  mov    %gs:0x14,%eax
3  mov    %eax,0xc(%esp)
4  xor    %eax,%eax
5  lea    0x4(%esp),%eax                    ; address of input2
6  push   %eax
7  lea    0xc(%esp),%eax                    ; address of input1
8  push   %eax
9  push   $0x804a16f                        ; "%d %d"
10 push   0x2c(%esp)
11 call   8048810 <__isoc99_sscanf@plt>
12 add    $0x10,%esp
13 cmp    $0x2,%eax
14 jne    8048ce6 <phase_4+0x3b>           ; line 19
15 mov    0x4(%esp),%eax
16 sub    $0x2,%eax
17 cmp    $0x2,%eax
18 jbe    8048ceb <phase_4+0x40>           ; line 20
19 call   8049087 <explode_bomb>
20 sub    $0x8,%esp
21 push   0xc(%esp)
22 push   $0x9
23 call   8048c68 <func4>
24 add    $0x10,%esp
25 cmp    0x8(%esp),%eax
26 je     8048d07 <phase_4+0x5c>
27 call   8049087 <explode_bomb>
28 mov    0xc(%esp),%eax
29 xor    %gs:0x14,%eax
30 je     8048d19 <phase_4+0x6e>
31 call   8048790 <__stack_chk_fail@plt>
32 add    $0x1c,%esp
33 ret
```

该函数第 5–14 行的指令通过 `sscanf` 函数读入两个整数。第 15–19 行检查输入 2 的值，要求其减去 2 后在区间 $[0, 2]$ 以内，也就是输入 2 的值只能是 2，3，4。第 20–23 行调用 `func4` 函数，依次将 9 和输入 2 作为参数传入。最后（第 25–27 行）比较 `func4` 的返回值与输入 1，如果二者不相等则引爆炸弹。

2.4.2 func4 函数

通过 objdump 得到 `func4` 函数的汇编代码如下：

```
1  push   %edi
2  push   %esi
3  push   %ebx
4  mov    0x10(%esp),%ebx
5  mov    0x14(%esp),%edi
6  test   %ebx,%ebx
7  jle    8048ca2 <func4+0x3a>      ; line 25
8  mov    %edi,%eax
9  cmp    $0x1,%ebx
10 je     8048ca7 <func4+0x3f>      ; line 26
11 sub    $0x8,%esp
12 push   %edi
13 lea    -0x1(%ebx),%eax
14 push   %eax
15 call   8048c68 <func4>
16 add    $0x8,%esp
17 lea    (%edi,%eax,1),%esi
18 push   %edi
19 sub    $0x2,%ebx
20 push   %ebx
21 call   8048c68 <func4>
22 add    $0x10,%esp
23 add    %esi,%eax
24 jmp    8048ca7 <func4+0x3f>      ; line 26
25 mov    $0x0,%eax
26 pop    %ebx
27 pop    %esi
28 pop    %edi
29 ret
```

该函数通过栈来传递两个参数，不妨记两个参数为 a, b。第 6–7 行检查 b 是否小于等于 0，如果是则返回 0。第 8–10 行检查 b 是否等于 1，如果是则返回 a。第 11–17 行递归调用函数自身，传入参数 $a-1, b$，将函数返回值加上 b 后暂存至寄存器 `esi`；第 18–24 行再次递归调用函数自身，传入参数 $a-2, b$，将函数返回值加上 `esi` 后返回。

总结以上行为，令该函数为 $f(a, b)$，则有：

$$f(a,b) = \begin{cases} 0, & b \le 0 \\ a, & b = 1 \\ f(a-1,b) + f(a-2,b) + b, & \text{otherwise} \end{cases}$$

2.4.3 小结

根据上面的分析，现需要输入两个数使得 $\text{input}_1 = f(9, \text{input}_2)$，同时 $2 \le \text{input}_2 \le 4$。编写 C 程序计算 $f(9,3) = 264$，不妨输入 `264 3` 通过第四关。

2.5 Phase 5

通过 objdump 得到 `phase_5` 函数的汇编代码如下：

```
1  push   %ebx
2  sub    $0x14,%esp
3  mov    0x1c(%esp),%ebx
4  push   %ebx                    ; address of input string
5  call   8048f71 <string_length>
6  add    $0x10,%esp
7  cmp    $0x6,%eax
8  je     8048d38 <phase_5+0x1b>  ; line 10
9  call   8049087 <explode_bomb>
10 mov    %ebx,%eax
11 add    $0x6,%ebx               ; address of the end of the string
12 mov    $0x0,%ecx               ; accumulator
13 movzbl (%eax),%edx
14 and    $0xf,%edx               ; get the low 4 bits of the character
15 add    0x804a020(,%edx,4),%ecx
16 add    $0x1,%eax
17 cmp    %ebx,%eax
18 jne    8048d42 <phase_5+0x25>  ; line 13
19 cmp    $0x21,%ecx
20 je     8048d60 <phase_5+0x43>  ; line 22
21 call   8049087 <explode_bomb>
22 add    $0x8,%esp
23 pop    %ebx
24 ret
```

这段逻辑中，第 4–9 行调用 `string_length` 函数计算输入字符串的长度，要求长度为 6。第 10–18 行是一个循环结构：用寄存器 `eax` 遍历输入字符串的每个字符，通过按位与 `0xf` 的方式取出字符 ASCII 码的低 4 位，将结果暂存至寄存器 `edx`；将内存地址 `0x804a020 + edx` 中的值累加到寄存器 `ecx` 中。第 19–21 行比较 `ecx` 中存放的累加结果是否为 $0x21$。

使用 gdb 输入命令 `x/16wd 0x804a020` 查看内存地址 `0x804a020` 处的值，发现是一个长度为 16 的整数数组

$$a = \{2, 10, 6, 1, 12, 16, 9, 3, 4, 7, 14, 5, 11, 8, 15, 13\}$$

因此只需输入一个长度为 6 的字符串，使得字符串中每个字符 ASCII 码低 4 位在数组中对应位置的值之和为 $(21)_{16}$ 即可。不妨构造字符串 `STICKY` 通过第五关。

2.6 Phase 6

通过 objdump 解析 `phase_6` 函数的汇编代码，分为以下阶段：

2.6.1 输入合规性检查

```
1  push   %esi
2  push   %ebx
3  sub    $0x4c,%esp
4  mov    %gs:0x14,%eax
5  mov    %eax,0x44(%esp)
6  xor    %eax,%eax
7  lea    0x14(%esp),%eax
8  push   %eax
9  push   0x5c(%esp)
10 call   80490ac <read_six_numbers>
11 add    $0x10,%esp
12 mov    $0x0,%esi                    ; i = 0
13 mov    0xc(%esp,%esi,4),%eax        ; eax = a[i]
14 sub    $0x1,%eax
15 cmp    $0x5,%eax
16 jbe    8048d9d <phase_6+0x38>       ; 0 <= a[i] - 1 <= 5
17 call   8049087 <explode_bomb>
18 add    $0x1,%esi                    ; increment i
19 cmp    $0x6,%esi                    ; outer loop condition
20 je     8048dd8 <phase_6+0x73>       ; line 39
21 mov    %esi,%ebx                    ; begin inner loop
22 mov    0xc(%esp,%ebx,4),%eax        ; eax = a[j]
23 cmp    %eax,0x8(%esp,%esi,4)
24 jne    8048db6 <phase_6+0x51>       ; a[j] != a[i]
25 call   8049087 <explode_bomb>
26 add    $0x1,%ebx
27 cmp    $0x5,%ebx
28 jle    8048da7 <phase_6+0x42>       ; line 22
29 jmp    8048d8c <phase_6+0x27>       ; line 13
```

这段逻辑在第 10 行读入 6 个整数以后，在 13–29 行开始遍历读入的数组 a，首先在第 14–17 行检查 $a[i]$ 是否在区间 $[1,6]$ 中，接着在第 18–28 行进入内层嵌套循环，检查 $a[i+1] \sim a[5]$ 是否不等于 $a[i]$。

用 C 代码表示上述逻辑如下：

```
1  read_six_numbers(input_string, a);
2  for (int i = 0; i < 6; i++) {
3      if (a[i] < 1 || a[i] > 6)
4          explode_bomb();
5      for (int j = i + 1; j < 6; j++)
6          if (a[j] == a[i])
7              explode_bomb();
8  }
```

综上所述，输入的六个数是 $1 \sim 6$ 的一个排列。

2.6.2 复制链表节点指针

值得注意的是，合规性检查完成循环后，会跳转到下面代码的第 39 行开始运行。

```
30  mov    0x8(%edx),%edx            ; move forward along the list
31  add    $0x1,%eax                 ; (inner loop):
32  cmp    %ecx,%eax                 ; (a[i] - 1) moves are needed
33  jne    8048dc0 <phase_6+0x5b>    ; line 30
34  mov    %edx,0x24(%esp,%esi,4)    ; copy pointer to current node to b[i]
35  add    $0x1,%ebx
36  cmp    $0x6,%ebx
37  jne    8048ddd <phase_6+0x78>    ; line 39 <Continue loop>
38  jmp    8048df4 <phase_6+0x8f>    ; line 47 <Terminate loop>
39  mov    $0x0,%ebx                 ; ENTRY
40  mov    %ebx,%esi
41  mov    0xc(%esp,%ebx,4),%ecx
42  mov    $0x1,%eax
43  mov    $0x804c13c,%edx
44  cmp    $0x1,%ecx
45  jg     8048dc0 <phase_6+0x5b>    ; line 30 <a[i] > 1>
46  jmp    8048dca <phase_6+0x65>    ; line 34 <a[i] == 1>
```

这段逻辑从 39 行开始进行循环，循环变量 i 从 0 开始，记录在寄存器 `ebx` 和 `esi` 中。每次循环开始都会将 $a[i]$ 存入寄存器 `ecx`，将一个特殊的地址 `0x804c13c` 存放在寄存器 `edx` 中。接着会连续 $a[i]-1$ 次将 `edx+8` 处的值存入 `edx`。由于 `edx` 的初值很像是一个地址且其被调用的行为也与地址相似，猜测 `edx+8` 处存放的内容也是地址，以 `0x804c13c` 为首地址处存放了每个单元长度为 12 字节的数据结构。

使用 gdb 输入 `x/3wx 0x804c13c` 查看该地址存放的内容：

```
1  (gdb) x/3wx 0x804c13c
2  0x804c13c <node1>:      0x000003c5      0x00000001      0x0804c148
3  (gdb)
4  0x804c148 <node2>:      0x000000ae      0x00000002      0x0804c154
5  (gdb)
6  0x804c154 <node3>:      0x000000f3      0x00000003      0x0804c160
7  (gdb)
8  0x804c160 <node4>:      0x000000bb      0x00000004      0x0804c16c
9  (gdb)
10 0x804c16c <node5>:      0x000000c7      0x00000005      0x0804c178
11 (gdb)
12 0x804c178 <node6>:      0x0000014b      0x00000006      0x00000000
```

发现先前的猜测正确，每个单元最后 4 字节都存放了互相关联的地址。分析这一数据结构，可将其抽象为链表。链表的每个节点依次存放了数据、编号、下一节点地址。

回到 `phase_6` 函数，第 30–33 行执行的逻辑是将当前指针沿着链表向前移动 $a[i]-1$ 个节点，第 34–36 行则将指向当前节点的指针拷贝到一个新开辟的数组 b 中。

2.6.3 重组链表和顺序检查

```
47  mov    0x24(%esp),%ebx                ; b[0]
48  lea    0x24(%esp),%eax                ; &b[1]
49  lea    0x38(%esp),%esi                ; &b[5]
50  mov    %ebx,%ecx                      ; b[0] <end of init.>
51  mov    0x4(%eax),%edx
52  mov    %edx,0x8(%ecx)                 ; b[i]->next = b[i + 1]
53  add    $0x4,%eax
54  mov    %edx,%ecx
55  cmp    %esi,%eax                      ; loop condition
56  jne    8048e02 <phase_6+0x9d>         ; line 51
57  movl   $0x0,0x8(%edx)                 ; b[5]->next = NULL
58  mov    $0x5,%esi
59  mov    0x8(%ebx),%eax                 ; nowP = lastP->next
60  mov    (%eax),%eax
61  cmp    %eax,(%ebx)                    ; lastP->data <= nowP->data
62  jle    8048e2b <phase_6+0xc6>         ; line 64
63  call   8049087 <explode_bomb>
64  mov    0x8(%ebx),%ebx
65  sub    $0x1,%esi                      ; loop control
66  jne    8048e1d <phase_6+0xb8>         ; line 59
67  mov    0x3c(%esp),%eax
68  xor    %gs:0x14,%eax
69  je     8048e45 <phase_6+0xe0>
70  call   8048790 <__stack_chk_fail@plt>
71  add    $0x44,%esp
72  pop    %ebx
73  pop    %esi
74  ret
```

这段代码在第 47–57 行将2.6.2节中提到的链表顺序按照数组 b 中的顺序重组。具体地，该逻辑将遍历数组 b，使 $b[i]$ 指向节点的下一节点指针指向 $b[i+1]$。遍历完成后使最后一个节点的下一节点指针置零。用 C 代码表示如下：

```
1  for (int* p = b; p != b + 5; p++)
2      p->next = *(p + 1);
3  b[5]->next = NULL;
```

汇编代码在第 58–66 行遍历检查链表的数据是否服从递增顺序。具体地，寄存器 `ebx` 中存放上一个节点的地址，寄存器 `eax` 中存放当前节点的地址，初始时二者分别指向链表首节点和第二个节点。每次循环在第 60–63 行比较上一个节点的数据是否小于等于当前节点的数据，在第 64 行更新寄存器 `ebx` 的值，第 59 行更新寄存器 `eax` 的值。以上循环用寄存器 `esi` 控制循环次数为 5 次。用 C 代码表示如下：

```
int i = 5;
node* lastP = b[0];
while (i--) {
    if (lastP->data > lastP->next->data)
        explode_bomb();
    lastP = lastP->next;
}
```

2.6.4 小结

综上所述，phase_6 函数要求输入一个 1 ～ 6 的排列。其后会根据输入顺序重组2.6.2节中提到的链表，使得该链表节点数据按照递增顺序排列。

符合要求的答案为 2 4 5 3 6 1。

2.7 Secret Phase

2.7.1 寻找入口

在使用 objdump 得到的反汇编文件中，发现了一个 secret_phase 函数。查找该函数的被调用情况，发现其首先会被函数 phase_defused 函数调用。检查 phase_defused 的汇编代码，发现其会在 0x804c3cc 地址存放的变量等于 6 时，将 0x804c4d0 处的字符串传入 sscanf，执行一系列判断符合要求后调用 secret_phase 函数。

将先前 6 题的答案输入文件 key，接着在终端执行 gdb -q bomb，执行以下命令：

```
watch *0x804c3cc
watch *0x804c4d0
set disassemble-next-line on
r key
```

gdb 会在两个关心的地址发生变化时停止程序运行。0x804c3cc 存放的变量在每次调用 read_line 函数时会加一，其作用是记录读入的行数。0x804c4d0 会在 phase_4 被调用前被更改。使用 x 指令查看该地址的值，得到 264 3，正是先前输入的 phase_4 的答案。

根据上面的观察，可以知道在 6 个炸弹全部通过后，phase_defused 函数会将 phase_4 的答案传入 sscanf 函数，同时传入格式化字符串"%d %d %s"。也就是在 phase_4 的答案后还需要输入一个字符串，phase_defused 函数会将读入的字符串与 0x804a1d2 处的字符串（通过 gdb 查看内存得到"DrEvil"）比较，如果相等则调用 secret_phase 函数。

在先前 Phase 4 的答案后添加 DrEvil，其他答案不变，输入完全部 6 个密码后即可进入 Secret Phase。

2.7.2 寻找答案

`secret_phase` 函数将读入字符串传入 `strtol` 函数，将其转换为十进制数 x，检查 x 处在区间 $[1, 1001]$ 之后调用函数 `fun7(0x804c088, x)`。如果返回值不等于 7 则会引爆炸弹。下面分析 `fun7` 函数的汇编代码：

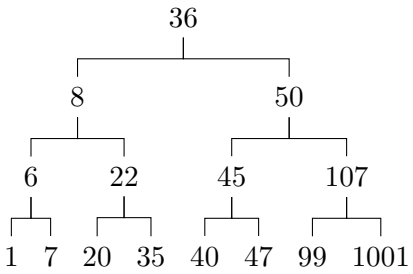
```
1  push   %ebx
2  sub    $0x8,%esp
3  mov    0x10(%esp),%edx          ; edx: param1
4  mov    0x14(%esp),%ecx          ; ecx: param2
5  test   %edx,%edx
6  je     8048e92 <fun7+0x47>      ; line 27
7  mov    (%edx),%ebx
8  cmp    %ecx,%ebx
9  jle    8048e74 <fun7+0x29>      ; line 17
10 sub    $0x8,%esp
11 push   %ecx                     ; param2: ecx
12 push   0x4(%edx)                ; param1: *(edx + 4)
13 call   8048e4b <fun7>
14 add    $0x10,%esp
15 add    %eax,%eax
16 jmp    8048e97 <fun7+0x4c>      ; line 28
17 mov    $0x0,%eax
18 cmp    %ecx,%ebx
19 je     8048e97 <fun7+0x4c>      ; line 28
20 sub    $0x8,%esp
21 push   %ecx                     ; param2: ecx
22 push   0x8(%edx)                ; param1: *(edx + 8)
23 call   8048e4b <fun7>
24 add    $0x10,%esp
25 lea    0x1(%eax,%eax,1),%eax
26 jmp    8048e97 <fun7+0x4c>
27 mov    $0xffffffff,%eax
28 add    $0x8,%esp
29 pop    %ebx
30 ret
```

该函数是递归函数，会在参数 1 等于 0 时返回 -1（第 5–6 行），在参数 1 指向地址等于参数 2 时返回 0（从第 9 行跳转至第 17–19 行），其他情况均会调用自身，传入的参数 2 均为原参数，参数 1 是原来参数 1 加 4 或加 8 内存处的值。由于参数 1 是一个地址，可以推断其加 4 和加 8 内存处也存放了地址，这可能是一个递归的数据结构。使用 gdb 查看初始传入参数 `0x804c088` 附近的值，检查该数据结构：

```
1  0x804c088 <n1>:    0x00000024    0x0804c094    0x0804c0a0
2  0x804c094 <n21>:   0x00000008    0x0804c0c4    0x0804c0ac
3  0x804c0a0 <n22>:   0x00000032    0x0804c0b8    0x0804c0d0
4  0x804c0ac <n32>:   0x00000016    0x0804c118    0x0804c100
```

```
(...)
```

不难发现，这是一个树形结构，每个节点依次存放了一个数据、两个子节点的地址。画出树形结构如下：



不难发现，这是一棵二叉搜索树。从而可以将 `fun7` 函数的逻辑用 C 代码表示如下：

```c
int fun7(node* p, int x) {
    if (p == NULL) return -1;
    if (p->data <= x) {
        if (p->data == x) return 0;
        return 2 * fun7(p->right, x) + 1;
    }
    return 2 * fun7(p->left, x);
}
```

该函数从树的根节点开始查找 `x`，每次向右的查找都会对答案产生 2^h 的贡献（其中 h 表示当前层数），向左的查找不会产生贡献。例如，查找数字 `1001`：从根节点开始向右查找移向 `50`，产生贡献 1；再向右移向 `107`，产生贡献 2；最后再向右移向 `1001`，产生贡献 4——总计产生 7 点贡献。

刚才的示例恰得到了题目所需的函数值，故输入 `1001` 即可通过 `Secret Phase`。至此，所有关卡均已通过，炸弹完全得到破解。

3 总结

3.1 实验中出现的问题

- 起初未能正确分析2.2节中取址和取值行为，导致未能正确理解逻辑意图，通过整理、标记不同行为，梳理思路后顺利解决问题。
- 起初未能正确计算2.4.1节中的输入变量存储的位置，未能找到输入 `1` 的用途。通过关注 `add`，`sub`，`push`，`pop` 指令的使用以系统化计算寄存器 `esp` 的变化，最终得到正确的输入变量行为。
- 在撰写本实验报告时，遇到了诸多 LaTeX 排版问题，查阅资料后使用 `sloppy` 命令解决了行末元素溢出问题，通过安装 Python 解决了 `minted` 宏包无法使用的问题……

3.2 心得体会

本次实验我收获颇丰。首先，该实验大幅加深了我对汇编语言的理解，尽管我最开始阅读汇编代码磕磕绊绊，但通过前几关内容循序渐进的强化记忆，我在阅读后几关以及 `secret_phase` 函数时能迅速理解代码的逻辑，且能在较短时间内还原出源代码的大致框架。第二，通过本次实验，我加强了对 gdb 工具使用的熟练程度，能够通过 gdb 快速定位问题，查看内存中的数据，从而更好理解程序内容。第三，该实验让我对基本数据结构在底层的实现有了更深刻的理解，实验中的链表和二叉搜索树结构的汇编代码让我更清晰地了解了基本数据结构在内存中的存储方式以及算法中的调用方式。第四，在撰写实验报告时遇到的诸多问题，让我对 LaTeX 排版有了更深入的了解，也让我学会了高效查阅资料解决问题的方法。总的来说，这次实验是我大学以来进行的最有趣、最令人激动的实验项目，为我提供了一段非常难忘的学习经历——行文至此，该实验的告一段落甚至让我觉得有些怅然若失。