北京郵電大學

实验报告

题目：拆解二进制炸弹

班　　级：2022211320

学　　号：2022211683

姓　　名：张晨阳

学　　院：计算机学院（国家示范性软件学院）

2023 年 10 月 31 日

一、实验目的

1. 理解 C 语言程序的机器级表示。
2. 初步掌握 GDB 调试器的用法。
3. 阅读 C 编译器生成的 x 86-64 机器代码，理解不同控制结构生成的基本指令模式，过程的实现。

二、实验环境

1. Windows PowerShell（10.120.11.12）
2. Linux
3. Objdump命令反汇编
4. GDB 调试工具
5. Visual Studio Code 1.83.1

三、实验内容

登录 bupt1服务器，在 home 目录下可以找到 Evil 博士专门为你量身定制的一个 bomb，当运行时，它会要求你输入一个字符串，如果正确，则进入下一关，继续要求你输入下一个字符串；否则，炸弹就会爆炸，输出一行提示信息并向计分服务器提交扣分信息。因此，本实验要求你必须通过反汇编和逆向工程对 bomb 执行文件进行分析，找到正确的字符串来解除这个的炸弹。

本实验通过要求使用课程所学知识拆除一个“binary bombs”来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。“binary bombs”是一个 Linux 可执行程序，包含了5个阶段（或关卡）。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；否则炸弹“爆炸”，打印输出 “BOOM!!!”。炸弹的每个阶段考察了机器级程序语言的一个不同方面，难度逐级递增。

为完成二进制炸弹拆除任务，需要使用 gdb 调试器和 objdump 来反汇编 bomb 文件，可以单步跟踪调试每一阶段的机器代码，也可以阅读反汇编代码，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。实验2的具体内容见实验2说明。

四、实验步骤及实验分析

准备工作

- 首先通过 ls 命令查看文件，找到炸弹文件包 bomb 516. tar；
- 然后通过命令 `tar -xvf bomb516.tar` 解压得到三个文件；

```
2022211683@bupt1:~$ ls
bomb516.tar  Lab1  Lab1.c  Lab1.s
2022211683@bupt1:~$ tar -xvf bomb516.tar
bomb516/README
bomb516/bomb.c
bomb516/bomb
```

图1-获取炸弹

第一阶段

- 使用 gdb 运行bomb；
- 设置断点在函数 `phase_1` 处，并开始运行，输入测试字符；
- 显示汇编代码开始分析；

```
(gdb) break phase_1
Breakpoint 1 at 0x400f2d
(gdb) run
Starting program: /students/2022211683/bomb516/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
12345

Breakpoint 1, 0x0000000000400f2d in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000400f2d <+0>:     sub    $0x8,%rsp
   0x0000000000400f31 <+4>:     mov    $0x402720,%esi
   0x0000000000400f36 <+9>:     callq  0x401475 <strings_not_equal>
   0x0000000000400f3b <+14>:    test   %eax,%eax
   0x0000000000400f3d <+16>:    je     0x400f44 <phase_1+23>
   0x0000000000400f3f <+18>:    callq  0x401749 <explode_bomb>
   0x0000000000400f44 <+23>:    add    $0x8,%rsp
   0x0000000000400f48 <+27>:    retq
End of assembler dump.
```

图2-设置断点开始分析

- 发现此时 gdb 已经进入 `phase_1` 函数中，并观察到调用了一个 `strings_not_equal` 函数，猜测是比较输入字符和正确字符的函数；
- 在 `strings_not_equal` 函数处设置断点，继续运行至进入该函数；
- 显示该函数内部汇编代码；

```
(gdb) b strings_not_equal
Breakpoint 2 at 0x401475
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000401475 in strings_not_equal ()
(gdb) disas
Dump of assembler code for function strings_not_equal:
=> 0x0000000000401475 <+0>:     push   %r12
   0x0000000000401477 <+2>:     push   %rbp
   0x0000000000401478 <+3>:     push   %rbx
   0x0000000000401479 <+4>:     mov    %rdi,%rbx
   0x000000000040147c <+7>:     mov    %rsi,%rbp
   0x000000000040147f <+10>:    callq  0x401457 <string_length>
   0x0000000000401484 <+15>:    mov    %eax,%r12d
   0x0000000000401487 <+18>:    mov    %rbp,%rdi
   0x000000000040148a <+21>:    callq  0x401457 <string_length>
   0x000000000040148f <+26>:    mov    $0x1,%edx
   0x0000000000401494 <+31>:    cmp    %eax,%r12d
   0x0000000000401497 <+34>:    jne    0x4014d5 <strings_not_equal+96>
   0x0000000000401499 <+36>:    movzbl (%rbx),%eax
   0x000000000040149c <+39>:    test   %al,%al
   0x000000000040149e <+41>:    je     0x4014c2 <strings_not_equal+77>
   0x00000000004014a0 <+43>:    cmp    0x0(%rbp),%al
   0x00000000004014a3 <+46>:    je     0x4014ac <strings_not_equal+55>
   0x00000000004014a5 <+48>:    jmp    0x4014c9 <strings_not_equal+84>
   0x00000000004014a7 <+50>:    cmp    0x0(%rbp),%al
   0x00000000004014aa <+53>:    jne    0x4014d0 <strings_not_equal+91>
   0x00000000004014ac <+55>:    add    $0x1,%rbx
   0x00000000004014b0 <+59>:    add    $0x1,%rbp
--Type <RET> for more, q to quit, c to continue without paging--
```

图3-进入strings_not_equal函数分析

- 发现又调用了 `string_length` 函数，猜测是获取输入字符数量的函数；
- 在 `string_length` 函数处设置断点，继续运行至进入该函数；
- 显示该函数内部汇编代码；

```
(gdb) b string_length
Breakpoint 3 at 0x401457
(gdb) c
Continuing.

Breakpoint 3, 0x0000000000401457 in string_length ()
(gdb) disas
Dump of assembler code for function string_length:
=> 0x0000000000401457 <+0>:     cmpb   $0x0,(%rdi)
   0x000000000040145a <+3>:     je     0x40146f <string_length+24>
   0x000000000040145c <+5>:     mov    $0x0,%eax
   0x0000000000401461 <+10>:    add    $0x1,%rdi
   0x0000000000401465 <+14>:    add    $0x1,%eax
   0x0000000000401468 <+17>:    cmpb   $0x0,(%rdi)
   0x000000000040146b <+20>:    jne    0x401461 <string_length+10>
   0x000000000040146d <+22>:    repz retq
   0x000000000040146f <+24>:    mov    $0x0,%eax
   0x0000000000401474 <+29>:    retq
End of assembler dump.
```

图4-进入string_length函数分析

- 查看 %rdi 存储的字符串，发现存储的确实是输入的字符串；
- 回到 strings_not_equal 函数，分析其先比较字符串长度，二者不相等则在 %eax 中存 1 返回；若相等，则比较具体内容，若不相等则在 %eax 中存 1 返回；

```
Dump of assembler code for function strings_not_equal:
   0x0000000000401475 <+0>:     push   %r12
   0x0000000000401477 <+2>:     push   %rbp
   0x0000000000401478 <+3>:     push   %rbx
   0x0000000000401479 <+4>:     mov    %rdi,%rbx
   0x000000000040147c <+7>:     mov    %rsi,%rbp
   0x000000000040147f <+10>:    callq  0x401457 <string_length>
   0x0000000000401484 <+15>:    mov    %eax,%r12d
   0x0000000000401487 <+18>:    mov    %rbp,%rdi
   0x000000000040148a <+21>:    callq  0x401457 <string_length>
=> 0x000000000040148f <+26>:    mov    $0x1,%edx
   0x0000000000401494 <+31>:    cmp    %eax,%r12d
   0x0000000000401497 <+34>:    jne    0x4014d5 <strings_not_equal+96>
   0x0000000000401499 <+36>:    movzbl (%rbx),%eax
   0x000000000040149c <+39>:    test   %al,%al
   0x000000000040149e <+41>:    je     0x4014c2 <strings_not_equal+77>
   0x00000000004014a0 <+43>:    cmp    0x0(%rbp),%al
   0x00000000004014a3 <+46>:    je     0x4014ac <strings_not_equal+55>
   0x00000000004014a5 <+48>:    jmp    0x4014c9 <strings_not_equal+84>
   0x00000000004014a7 <+50>:    cmp    0x0(%rbp),%al
   0x00000000004014aa <+53>:    jne    0x4014d0 <strings_not_equal+91>
   0x00000000004014ac <+55>:    add    $0x1,%rbx
   0x00000000004014b0 <+59>:    add    $0x1,%rbp
--Type <RET> for more, q to quit, c to continue without paging--c
   0x00000000004014b4 <+63>:    movzbl (%rbx),%eax
   0x00000000004014b7 <+66>:    test   %al,%al
   0x00000000004014b9 <+68>:    jne    0x4014a7 <strings_not_equal+50>
   0x00000000004014bb <+70>:    mov    $0x0,%edx
   0x00000000004014c0 <+75>:    jmp    0x4014d5 <strings_not_equal+96>
   0x00000000004014c2 <+77>:    mov    $0x0,%edx
   0x00000000004014c7 <+82>:    jmp    0x4014d5 <strings_not_equal+96>
   0x00000000004014c9 <+84>:    mov    $0x1,%edx
   0x00000000004014ce <+89>:    jmp    0x4014d5 <strings_not_equal+96>
   0x00000000004014d0 <+91>:    mov    $0x1,%edx
   0x00000000004014d5 <+96>:    mov    %edx,%eax
   0x00000000004014d7 <+98>:    pop    %rbx
   0x00000000004014d8 <+99>:    pop    %rbp
   0x00000000004014d9 <+100>:   pop    %r12
   0x00000000004014db <+102>:   retq
End of assembler dump.
```

图5-调用完string_length的strings_not_equal函数

- 观察 cmp 的使用，不难发现 %rbp 存储的是正确密码，使用 x 指令查看其存储字符串；

```
(gdb) x /s $rbx
0x6047c0 <input_strings>:       "12345"
(gdb) x /s $rbp
0x402720:       "A binary bomb is a program provided to students as an object-code file."
```

图6-阶段1正确密码

- 复制保存并退出当前调试。

第二阶段

- 重新进入调试模式；
- 由阶段一发现炸弹爆炸函数 explode_bomb，在该函数处设置断点，阻止炸弹爆炸；
- 在 phase_2 入口处设置断点，运行程序，输入阶段一答案，以及测试密码，进入阶段 2；
- 显示汇编代码开始阶段 2的分析；

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401749
(gdb) b phase_2
Breakpoint 2 at 0x400f49
(gdb) r
Starting program: /students/2022211683/bomb516/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
A binary bomb is a program provided to students as an object-code file.
Phase 1 defused. How about the next one?
123456

Breakpoint 2, 0x0000000000400f49 in phase_2 ()
(gdb) disas
Dump of assembler code for function phase_2:
=> 0x0000000000400f49 <+0>:     push   %rbp
   0x0000000000400f4a <+1>:     push   %rbx
   0x0000000000400f4b <+2>:     sub    $0x28,%rsp
   0x0000000000400f4f <+6>:     mov    %fs:0x28,%rax
   0x0000000000400f58 <+15>:    mov    %rax,0x18(%rsp)
   0x0000000000400f5d <+20>:    xor    %eax,%eax
   0x0000000000400f5f <+22>:    mov    %rsp,%rsi
   0x0000000000400f62 <+25>:    callq  0x40177f <read_six_numbers>
   0x0000000000400f67 <+30>:    cmpl   $0x0,(%rsp)
   0x0000000000400f6b <+34>:    jns    0x400f72 <phase_2+41>
   0x0000000000400f6d <+36>:    callq  0x401749 <explode_bomb>
   0x0000000000400f72 <+41>:    mov    %rsp,%rbp
   0x0000000000400f75 <+44>:    mov    $0x1,%ebx
   0x0000000000400f7a <+49>:    mov    %ebx,%eax
   0x0000000000400f7c <+51>:    add    0x0(%rbp),%eax
   0x0000000000400f7f <+54>:    cmp    %eax,0x4(%rbp)
   0x0000000000400f82 <+57>:    je     0x400f89 <phase_2+64>
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000400f84 <+59>:    callq  0x401749 <explode_bomb>
   0x0000000000400f89 <+64>:    add    $0x1,%ebx
   0x0000000000400f8c <+67>:    add    $0x4,%rbp
   0x0000000000400f90 <+71>:    cmp    $0x6,%ebx
   0x0000000000400f93 <+74>:    jne    0x400f7a <phase_2+49>
   0x0000000000400f95 <+76>:    mov    0x18(%rsp),%rax
   0x0000000000400f9a <+81>:    xor    %fs:0x28,%rax
   0x0000000000400fa3 <+90>:    je     0x400faa <phase_2+97>
   0x0000000000400fa5 <+92>:    callq  0x400b90 <__stack_chk_fail@plt>
   0x0000000000400faa <+97>:    add    $0x28,%rsp
   0x0000000000400fae <+101>:   pop    %rbx
   0x0000000000400faf <+102>:   pop    %rbp
   0x0000000000400fb0 <+103>:   retq
End of assembler dump.
```

图7-阶段2开始

- 发现调用了一个 `read_six_numbers` 函数，表明此阶段需要输入 6 个数字；
- 在 `read_six_numbers` 函数处设置断点，继续运行至进入该函数；
- 显示该函数内部汇编代码；

```
(gdb) b read_six_numbers
Breakpoint 3 at 0x40177f
(gdb) c
Continuing.

Breakpoint 3, 0x000000000040177f in read_six_numbers ()
(gdb) disas
Dump of assembler code for function read_six_numbers:
=> 0x000000000040177f <+0>:     sub    $0x8,%rsp
   0x0000000000401783 <+4>:     mov    %rsi,%rdx
   0x0000000000401786 <+7>:     lea    0x4(%rsi),%rcx
   0x000000000040178a <+11>:    lea    0x14(%rsi),%rax
   0x000000000040178e <+15>:    push   %rax
   0x000000000040178f <+16>:    lea    0x10(%rsi),%rax
   0x0000000000401793 <+20>:    push   %rax
   0x0000000000401794 <+21>:    lea    0xc(%rsi),%r9
   0x0000000000401798 <+25>:    lea    0x8(%rsi),%r8
   0x000000000040179c <+29>:    mov    $0x402a31,%esi
   0x00000000004017a1 <+34>:    mov    $0x0,%eax
   0x00000000004017a6 <+39>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x00000000004017ab <+44>:    add    $0x10,%rsp
   0x00000000004017af <+48>:    cmp    $0x5,%eax
   0x00000000004017b2 <+51>:    jg     0x4017b9 <read_six_numbers+58>
   0x00000000004017b4 <+53>:    callq  0x401749 <explode_bomb>
   0x00000000004017b9 <+58>:    add    $0x8,%rsp
--Type <RET> for more, q to quit, c to continue without paging--c
   0x00000000004017bd <+62>:    retq
End of assembler dump.
```

图8-read_six_numbers函数

- 发现一个内存地址 `0x402a31`，尝试打印其内容；

```
(gdb) x /s 0x402a31
0x402a31:       "%d %d %d %d %d %d"
```

图9-输入格式

- 联系需要输入六个数字，分析出此阶段需要输入六个整数，且用空格分开；
- 回到 `phase_2`，通过 `read_six_numbers` 函数后的汇编语句，分析第一个数需要非负数；

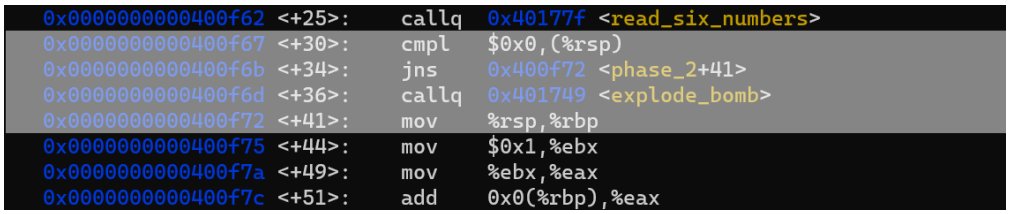


图10-分析第一个数

- 接下来一连串处理剩余数字的汇编指令，分析得第二个数需要等于第一个数加上 1；
- 同理分析得到，第三个数需要比第二个数增加 2，第四个数比第三个数增加 3，第五个数比第四个数增加 4，第六个数比第五个数增加 5；

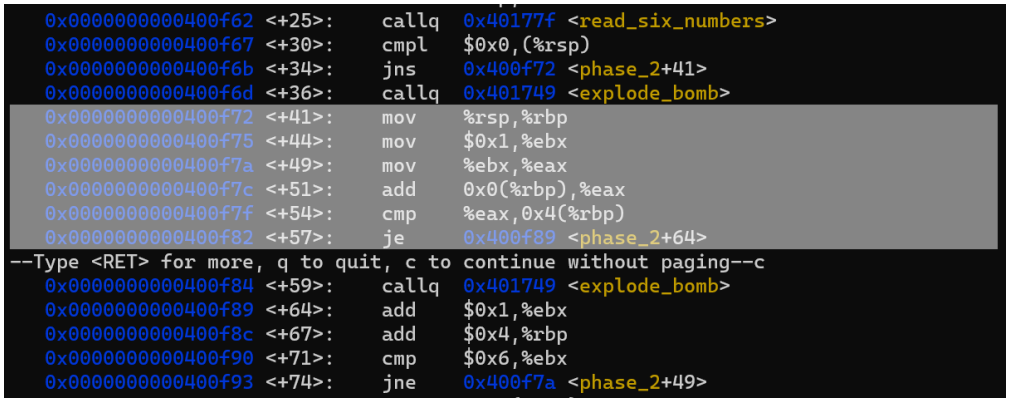
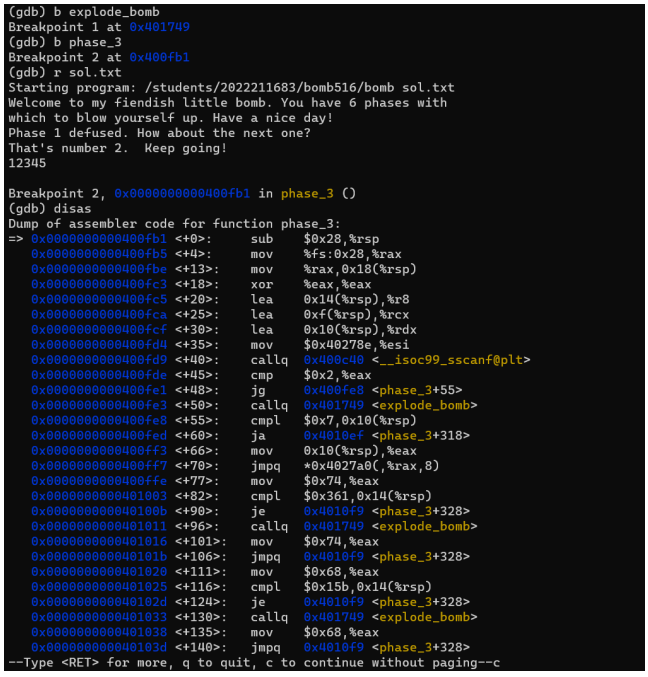


图11-分析剩余数字的判定

- 那么输入 `1 2 4 7 11 16`，答案正确，进入第三阶段。

第三阶段

- 重新进入调试模式；
- 在 `explode_bomb` 函数处设置断点，阻止炸弹爆炸；
- 在 `phase_3` 入口处设置断点，运行程序，输入阶段 1，2答案，以及测试密码，进入阶段 3；
- 显示汇编代码开始阶段 3的分析；



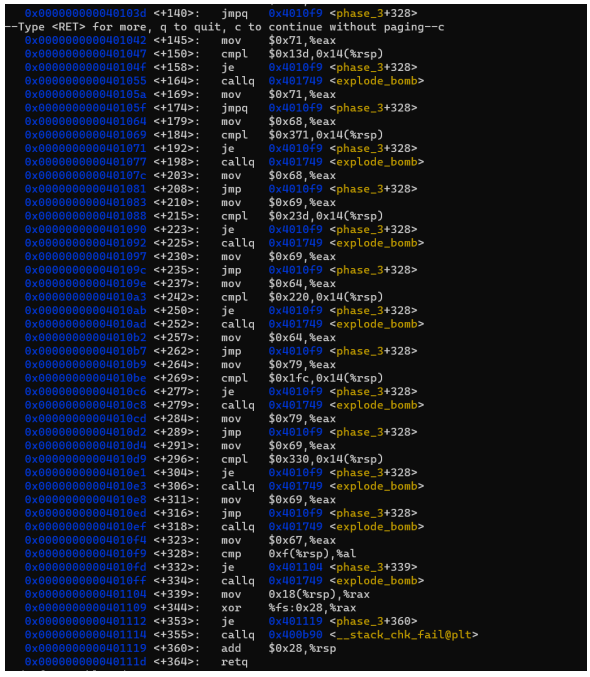


图12- 阶段3开始

- 在 `0x400fd4` 处发现一个内存地址 `0x40278e`，根据阶段 2 的经验，猜测与输入格式有关，用 `x` 指令查看；

```
(gdb) x /s 0x40278e
0x40278e:       "%d %c %d"
```

图13-阶段3 输入格式

- 可以知道阶段 3 需要输入一个整数，空格，一个字符，空格，一个整数；
- 但在 `0x400fe8` 处的 `cmpl $0x7,0x10(%rsp)` 指令以及下一条指令 `ja 0x4010ef`，可以知道 `0x10(%rsp)` 处的值 ≤ `0x7`，暂记为 `0xT`；
- `0x400ff7` 处的指令 `jmpq *0x4027a0(,%rax,8)` 表示跳转到 `0x4027a0+T*8` 所存储的地址处；
- 使用 `x` 命令查看地址 `0x4027a0` 开始的 8 个地址值；

```
(gdb) x /64x 0x4027a0
0x4027a0:       0xfe    0x0f    0x40    0x00
0x4027a8:       0x20    0x10    0x40    0x00
0x4027b0:       0x42    0x10    0x40    0x00
0x4027b8:       0x64    0x10    0x40    0x00
0x4027c0:       0x83    0x10    0x40    0x00
0x4027c8:       0x9e    0x10    0x40    0x00
0x4027d0:       0xb9    0x10    0x40    0x00
0x4027d8:       0xd4    0x10    0x40    0x00
```

图14-查看地址值

- 到代码中查找分析这些 `mov`、`cmpl` 和跳转指令；

```
 (0)
400ffe <+77>:    mov    $0x74,%eax
401003 <+82>:    cmpl   $0x361,0x14(%rsp)
 (1)
401020 <+111>:   mov    $0x68,%eax
401025 <+116>:   cmpl   $0x15b,0x14(%rsp)
 (2)
401042 <+145>:   mov    $0x71,%eax
401047 <+150>:   cmpl   $0x13d,0x14(%rsp)
 (3)
401064 <+179>:   mov    $0x68,%eax
401069 <+184>:   cmpl   $0x371,0x14(%rsp)
 (4)
401083 <+210>:   mov    $0x69,%eax
401088 <+215>:   cmpl   $0x23d,0x14(%rsp)
 (5)
40109e <+237>:   mov    $0x64,%eax
4010a3 <+242>:   cmpl   $0x220,0x14(%rsp)
 (6)
4010b9 <+264>:   mov    $0x79,%eax
4010be <+269>:   cmpl   $0x1fc,0x14(%rsp)
 (7)
4010d4 <+291>:   mov    $0x69,%eax
4010d9 <+296>:   cmpl   $0x330,0x14(%rsp)
```

图15-switch语句对应

- 得到结论：先比较第二个数字，然后字符与 `%al`（即 `%eax` 低八位）进行比较，对应关系如下：

第一个数字	字符	第二个数字
0	t	865
1	h	347
2	q	317
3	h	881
4	i	573
5	d	544
6	y	508
7	i	816

- 输入其中任意一组作为阶段 3 答案即可，我们选择输入 `0 t 865`。
- 通过，进入第四阶段。

第四阶段

- 重新进入调试模式；
- 在 `explode_bomb` 函数处设置断点，阻止炸弹爆炸；
- 在 `phase_4` 入口处设置断点，运行程序，输入阶段 1，2，3 答案，以及测试密码，进入阶段 4；
- 显示汇编代码开始阶段 4 的分析；

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401749
(gdb) b phase_4
Breakpoint 2 at 0x401159
(gdb) r sol.txt
Starting program: /students/2022211683/bomb516/bomb sol.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
12345

Breakpoint 2, 0x0000000000401159 in phase_4 ()
(gdb) disas
Dump of assembler code for function phase_4:
=> 0x0000000000401159 <+0>:     sub    $0x18,%rsp
   0x000000000040115d <+4>:     mov    %fs:0x28,%rax
   0x0000000000401166 <+13>:    mov    %rax,0x8(%rsp)
   0x000000000040116b <+18>:    xor    %eax,%eax
   0x000000000040116d <+20>:    mov    %rsp,%rcx
   0x0000000000401170 <+23>:    lea    0x4(%rsp),%rdx
   0x0000000000401175 <+28>:    mov    $0x402a3d,%esi
   0x000000000040117a <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x000000000040117f <+38>:    cmp    $0x2,%eax
   0x0000000000401182 <+41>:    jne    0x40118f <phase_4+54>
   0x0000000000401184 <+43>:    mov    (%rsp),%eax
   0x0000000000401187 <+46>:    sub    $0x2,%eax
   0x000000000040118a <+49>:    cmp    $0x2,%eax
   0x000000000040118d <+52>:    jbe    0x401194 <phase_4+59>
   0x000000000040118f <+54>:    callq  0x401749 <explode_bomb>
   0x0000000000401194 <+59>:    mov    (%rsp),%esi
   0x0000000000401197 <+62>:    mov    $0x5,%edi
   0x000000000040119c <+67>:    callq  0x40111e <func4>
   0x00000000004011a1 <+72>:    cmp    0x4(%rsp),%eax
   0x00000000004011a5 <+76>:    je     0x4011ac <phase_4+83>
   0x00000000004011a7 <+78>:    callq  0x401749 <explode_bomb>
   0x00000000004011ac <+83>:    mov    0x8(%rsp),%rax
   0x00000000004011b1 <+88>:    xor    %fs:0x28,%rax
   0x00000000004011ba <+97>:    je     0x4011c1 <phase_4+104>
   0x00000000004011bc <+99>:    callq  0x400b90 <__stack_chk_fail@plt>
   0x00000000004011c1 <+104>:   add    $0x18,%rsp
   0x00000000004011c5 <+108>:   retq
End of assembler dump.
```

图16-阶段4开始

- 不难发现前面的结构与阶段 3 相似，查看 `0x401175` 处的内存地址 `0x402a3d`；

```
(gdb) x /s 0x402a3d
0x402a3d:       "%d %d"
```

图17-阶段4输入格式

- 由此可知，阶段 4 需要输入两个整数，中间用空格隔开；
- 发现该阶段调用了一个函数 `func4`，设置断点并进入，查看 `func4` 汇编指令；

```
(gdb) b func4
Breakpoint 3 at 0x40111e
(gdb) c
Continuing.

Breakpoint 3, 0x000000000040111e in func4 ()
(gdb) disas
Dump of assembler code for function func4:
=> 0x000000000040111e <+0>:     test   %edi,%edi
   0x0000000000401120 <+2>:     jle    0x40114d <func4+47>
   0x0000000000401122 <+4>:     mov    %esi,%eax
   0x0000000000401124 <+6>:     cmp    $0x1,%edi
   0x0000000000401127 <+9>:     je     0x401157 <func4+57>
   0x0000000000401129 <+11>:    push   %r12
   0x000000000040112b <+13>:    push   %rbp
   0x000000000040112c <+14>:    push   %rbx
   0x000000000040112d <+15>:    mov    %esi,%ebp
   0x000000000040112f <+17>:    mov    %edi,%ebx
   0x0000000000401131 <+19>:    lea    -0x1(%rdi),%edi
   0x0000000000401134 <+22>:    callq  0x40111e <func4>
   0x0000000000401139 <+27>:    lea    0x0(%rbp,%rax,1),%r12d
   0x000000000040113e <+32>:    lea    -0x2(%rbx),%edi
   0x0000000000401141 <+35>:    mov    %ebp,%esi
   0x0000000000401143 <+37>:    callq  0x40111e <func4>
   0x0000000000401148 <+42>:    add    %r12d,%eax
   0x000000000040114b <+45>:    jmp    0x401153 <func4+53>
   0x000000000040114d <+47>:    mov    $0x0,%eax
   0x0000000000401152 <+52>:    retq
   0x0000000000401153 <+53>:    pop    %rbx
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000401154 <+54>:    pop    %rbp
   0x0000000000401155 <+55>:    pop    %r12
   0x0000000000401157 <+57>:    repz retq
End of assembler dump.
```

图18-func4函数

- 分析 `func4` 前的指令发现，第二个输入需要 ≤ 4，否则会直接爆炸，且该函数传入两个参数，其中 `%esi` 为第二个输入，`%edi` 为 `0x5`；

- 分析 `func4` 后的指令发现，当 `cmp 0x4(%rsp),%eax` 相等时不爆炸，故 `func4` 需要让 `%eax` 等于输入的第一个数；
- 分析 `func4` 函数的汇编指令，发现这是一个递归函数，流程如下：

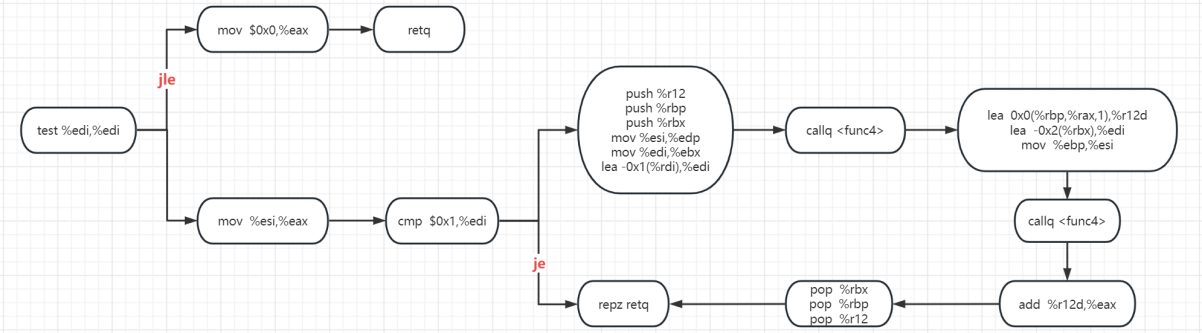


图19-递归流程

- 令第二个输入为 3，执行一遍该递归函数，求得函数返回的 `%eax` 的值为 36，则在第二个输入为 3 的前提下，第一个输入为 36；

```
(gdb) n
Single stepping until exit from function func4,
which has no line number information.
0x00000000004011a1 in phase_4 ()
(gdb) disas
Dump of assembler code for function phase_4:
   0x0000000000401159 <+0>:     sub    $0x18,%rsp
   0x000000000040115d <+4>:     mov    %fs:0x28,%rax
   0x0000000000401166 <+13>:    mov    %rax,0x8(%rsp)
   0x000000000040116b <+18>:    xor    %eax,%eax
   0x000000000040116d <+20>:    mov    %rsp,%rcx
   0x0000000000401170 <+23>:    lea    0x4(%rsp),%rdx
   0x0000000000401175 <+28>:    mov    $0x402a3d,%esi
   0x000000000040117a <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x000000000040117f <+38>:    cmp    $0x2,%eax
   0x0000000000401182 <+41>:    jne    0x40118f <phase_4+54>
   0x0000000000401184 <+43>:    mov    (%rsp),%eax
   0x0000000000401187 <+46>:    sub    $0x2,%eax
   0x000000000040118a <+49>:    cmp    $0x2,%eax
   0x000000000040118d <+52>:    jbe    0x401194 <phase_4+59>
   0x000000000040118f <+54>:    callq  0x401749 <explode_bomb>
   0x0000000000401194 <+59>:    mov    (%rsp),%esi
   0x0000000000401197 <+62>:    mov    $0x5,%edi
   0x000000000040119c <+67>:    callq  0x40111e <func4>
=> 0x00000000004011a1 <+72>:    cmp    0x4(%rsp),%eax
   0x00000000004011a5 <+76>:    je     0x4011ac <phase_4+83>
   0x00000000004011a7 <+78>:    callq  0x401749 <explode_bomb>
   0x00000000004011ac <+83>:    mov    0x8(%rsp),%rax
   0x00000000004011b1 <+88>:    xor    %fs:0x28,%rax
   0x00000000004011ba <+97>:    je     0x4011c1 <phase_4+104>
   0x00000000004011bc <+99>:    callq  0x400b90 <__stack_chk_fail@plt>
   0x00000000004011c1 <+104>:   add    $0x18,%rsp
   0x00000000004011c5 <+108>:   retq
End of assembler dump.
(gdb) i reg eax
eax            0x24                36
```

图20-递归求解%eax

- 输入 `36 3`，通过，进入第五阶段。

第五阶段

- 重新进入调试模式；
- 在 `explode_bomb` 函数处设置断点，阻止炸弹爆炸；
- 在 `phase_5` 入口处设置断点，运行程序，输入阶段 1，2，3，4答案，以及测试密码，进入阶段 5；
- 显示汇编代码开始阶段 5 的分析；

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401749
(gdb) b phase_5
Breakpoint 2 at 0x4011c6
(gdb) r sol.txt
Starting program: /students/2022211683/bomb516/bomb sol.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
12345

Breakpoint 2, 0x00000000004011c6 in phase_5 ()
(gdb) disas
Dump of assembler code for function phase_5:
=> 0x00000000004011c6 <+0>:     push   %rbx
   0x00000000004011c7 <+1>:     sub    $0x10,%rsp
   0x00000000004011cb <+5>:     mov    %rdi,%rbx
   0x00000000004011ce <+8>:     mov    %fs:0x28,%rax
   0x00000000004011d7 <+17>:    mov    %rax,0x8(%rsp)
   0x00000000004011dc <+22>:    xor    %eax,%eax
   0x00000000004011de <+24>:    callq  0x401457 <string_length>
   0x00000000004011e3 <+29>:    cmp    $0x6,%eax
   0x00000000004011e6 <+32>:    je     0x4011ed <phase_5+39>
   0x00000000004011e8 <+34>:    callq  0x401749 <explode_bomb>
   0x00000000004011ed <+39>:    mov    $0x0,%eax
   0x00000000004011f2 <+44>:    movzbl (%rbx,%rax,1),%edx
   0x00000000004011f6 <+48>:    and    $0xf,%edx
   0x00000000004011f9 <+51>:    movzbl 0x4027e0(%rdx),%edx
   0x0000000000401200 <+58>:    mov    %dl,(%rsp,%rax,1)
   0x0000000000401203 <+61>:    add    $0x1,%rax
   0x0000000000401207 <+65>:    cmp    $0x6,%rax
   0x000000000040120b <+69>:    jne    0x4011f2 <phase_5+44>
   0x000000000040120d <+71>:    movb   $0x0,0x6(%rsp)
   0x0000000000401212 <+76>:    mov    $0x402797,%esi
   0x0000000000401217 <+81>:    mov    %rsp,%rdi
   0x000000000040121a <+84>:    callq  0x401475 <strings_not_equal>
   0x000000000040121f <+89>:    test   %eax,%eax
   0x0000000000401221 <+91>:    je     0x401228 <phase_5+98>
   0x0000000000401223 <+93>:    callq  0x401749 <explode_bomb>
   0x0000000000401228 <+98>:    mov    0x8(%rsp),%rax
   0x000000000040122d <+103>:   xor    %fs:0x28,%rax
   0x0000000000401236 <+112>:   je     0x40123d <phase_5+119>
   0x0000000000401238 <+114>:   callq  0x400b90 <__stack_chk_fail@plt>
   0x000000000040123d <+119>:   add    $0x10,%rsp
   0x0000000000401241 <+123>:   pop    %rbx
   0x0000000000401242 <+124>:   retq
End of assembler dump.
```

图21-阶段5开始

- 与阶段 1 相似，同理分析 `string_length` 函数后的 `cmp`，不难猜到正确答案由六个字符组成；
- 由 `phase_1` 经验可知，在调用 `strings_not_equal` 时， `%rsi` 中存放的是正确答案字符串地址，`%rdi` 中存放的是输入字符串地址。发现指令 `mov $0x402797,%esi` 、`mov %rsp,%rdi` ，后面就调用了 `strings_not_equal` 函数，所以推出 `0x402797` 处的字符串就是正确答案字符串。查看 `0x402797` 处内容，可以知道最终字符串为 `flames`；

```
(gdb) x /s 0x402797
0x402797:       "flames"
```

图22-最终判定的字符串

- 但很明显这个阶段没有那么简单，继续分析与输入字符串有关的指令，一开始将输入字符串 `mov` 到了 `%rbx` 中，然后通过指令 `movzbl (%rbx,%rax,1),%edx` 取出输入字符的第一位存到 `%edx` 中，对于这取出的字符，使用指令 `and $0xf,%edx` 将其低四位保留下来，其余位清零，接着又通过指令 `movzbl 0x4027e0(%rdx),%edx` 从 `0x4027e0 + %rdx` 处取出 1 字节数据并零扩展到 4 字节后存储到 `%edx` 中，随后将 `%dl` 中的数据存储在 `(%rsp,%rax,1)` 处；

```
0x00000000004011de <+24>:    callq  0x401457 <string_length>
0x00000000004011e3 <+29>:    cmp    $0x6,%eax
0x00000000004011e6 <+32>:    je     0x4011ed <phase_5+39>
0x00000000004011e8 <+34>:    callq  0x401749 <explode_bomb>
0x00000000004011ed <+39>:    mov    $0x0,%eax
0x00000000004011f2 <+44>:    movzbl (%rbx,%rax,1),%edx
0x00000000004011f6 <+48>:    and    $0xf,%edx
0x00000000004011f9 <+51>:    movzbl 0x4027e0(%rdx),%edx
0x0000000000401200 <+58>:    mov    %dl,(%rsp,%rax,1)
0x0000000000401203 <+61>:    add    $0x1,%rax
0x0000000000401207 <+65>:    cmp    $0x6,%rax
0x000000000040120b <+69>:    jne    0x4011f2 <phase_5+44>
```

图23-对输入字符串的操作

- 上述操作重复 6 次直到 `%rax` 等于 6；
- 查看一下 `0x4027e0` 处的内容；

```
(gdb) x /s 0x4027e0
0x4027e0 <array.3600>:  "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

图24-取数据字符串

- 由此推断出，该阶段并不是直接输入字符串，而是在 `0x4027e0` 处挑选第 `%rdx` 个字符，然后这些挑选出的字符组合成一个字符串，而这个字符串应该是 `flames` ；
- `flames` 对应的索引为 9，15，1，0，5，7，那么只需要输入的字符的低四位符合对应的索引即可；
- 在每个索引上加上 64（这样不会改变低四位），得到 73，79，65，64，69，71，对应字符串 `IOA@EG`；
- 输入 `IOA@EG`,通过。

答案汇总与通过截图

```
(gdb) r sol.txt
Starting program: /students/2022211683/bomb516/bomb sol.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
```

图25-5个阶段都通过

```
2022211683@bupt1:~/bomb516$ cat sol.txt
A binary bomb is a program provided to students as an object-code file.
1 2 4 7 11 16
0 t 865
36 3
IOA@EG
```

图26-全部答案

五、总结体会

- 本次实验耗费时间长，花费精力大，但是收获也非常多。
- 本次实验的第四阶段花费了我最多的时间，一开始我想要直接猜测出递归函数的最终结果，进行了很久无意义的操作尝试，最终还是放弃了这种想法，不仅低效，而且对理解汇编代码没有任何帮助。我沉下心来分析机器指令对应的实现功能，并结合一步一步的运行，直至完成整个递归函数，这样才解决了这一阶段。
- 在 gdb 调试指令方面，我得到了很多的练习，但是有些指令我并不熟悉，比如显示不同寄存器的值的指令，但通过查阅相关资料，我最终解决了这个问题。
- 总的来说，本次实验虽然触发爆炸，但花费时间过多，对指令还不够熟悉，且面对逻辑过程时还是不太愿意耐心地一步一步研究，希望之后的实验能够更加熟练的使用 gdb 指令，对于逻辑分析也更耐心细心。
- 对本次实验的建议：希望记录爆炸次数的网站也可以记录拆炸弹的时间，避免发生在错误的路上浪费过长的时间，导致做实验时心急，无法耐心分析。