

目录

实验 2:	Binary Bomb	1
2.1 实验概述.....		1
2.2 实验内容.....		2
2.2.1 阶段 1 字符串比较		2
2.2.2 阶段 2 循环		3
2.2.3 阶段 3 条件/分支		4
2.2.4 阶段 4 递归调用和栈		7
2.2.5 阶段 5 指针		9
2.2.6 阶段 6 链表/指针/结构		10
2.2.7 阶段 7 隐藏		13
2.3 实验小结.....		18
实验总结.....		19

实验 2: Binary Bomb

2.1 实验概述

本实验中，我们要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。

一个“binary bombs”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，包含了 6 个阶段（phase1~phase6）。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。实验的目标是拆除尽可能多的炸弹层次。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
- * 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，我们需要使用 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，从中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。这可能需要我们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 AT&T 汇编

实验环境：linux

2.2 实验内容

使用 `objdump -t,objdump -d` 分别生成 bomb 符号表以及 bomb 反汇编文件。
根据反汇编文件进行逆向工程，拆除炸弹。

2.2.1 阶段 1 字符串比较

1. 任务描述：根据反汇编文件找出需要输入的字符串的位置，再从其中获取字符串信息即可。
2. 实验设计：首先查看反汇编文件，可以在 `main` 函数里找到 `phase_1`，然后找到其对应位置，找到所需要输入的字符串地址，然后使用 `gdb` 查看内存中的数据。
3. 实验过程：

(1) 在 `main` 函数中找到了 `phase_1`，然后查找 `phase_1` 的地址，定位到其反汇编代码如下图：

```
08048b42 <phase_1>:
8048b42: 83 ec 14          sub    $0x14,%esp
8048b45: 68 04 a0 04 08    push  $0x804a004
8048b4a: ff 74 24 1c       pushl 0x1c(%esp)
8048b4e: e8 9b 04 00 00    call  8048fee <strings_not_equal>
8048b53: 83 c4 10          add    $0x10,%esp
8048b56: 85 c0             test   %eax,%eax
8048b58: 75 04             jne    8048b5e <phase_1+0x1c>
8048b5a: 83 c4 0c          add    $0xc,%esp
8048b5d: c3               ret
8048b5e: e8 80 05 00 00    call  80490e3 <explode_bomb>
8048b63: eb f5             jmp    8048b5a <phase_1+0x18>
```

图 2-1 phase_1 反汇编代码

(2) 由上面可以看出，当两个字符串不相同时，会调用 `<explode_bomb>`，导致拆弹不成功。这是观察到前面有 `push $0x804a004`，很容易推断出此地址即为比较字符串的地址，使用 `gdb` 观察此地址中存放的数据，得出：

```
(gdb) x/s 0x804a004
0x804a004: "There are rumors on the internets."
(gdb)
```

图 2-2 待比较字符串

4. 实验结果：

输入上面得出的字符串，结果如下图：

```
Welcome to my fiendish little bomb. You have 6 phases with  
which to blow yourself up. Have a nice day!  
There are rumors on the internets.  
Phase 1 defused. How about the next one?
```

图 2-3 phase_1 结果

2.2.2 阶段 2 循环

1. 任务描述：根据反汇编文件找出需要输入的条件，再从其中获取条件信息。
2. 实验设计：首先查看反汇编文件，找到 phase_2 对应位置，观察语句逻辑，然后测试答案是否正确。
3. 实验过程：

（1）在 main 函数中找到了 phase_2，然后查找 phase_2 的地址，定位到其反汇编代码如下图：

8048b7e:	e8 85 05 00 00	call	8049108 <read_six_numbers>
8048b83:	83 c4 10	add	\$0x10,%esp
8048b86:	83 7c 24 04 00	cmpl	\$0x0,0x4(%esp)
8048b8b:	78 07	js	8048b94 <phase_2+0x2f>

图 2-4 phase_2 反汇编代码

（2）由上面可以看出，<read_six_number> 初步推测是读 6 个数字，后来找到其汇编语句，发现果然是这个功能。然后将 esp+4 与 0 比较，如果小于 0 则引爆炸弹。由堆栈结构可知，esp+4 指向第一个输入的数字。于是现在假定第一个数字为 1。

（3）继续观察反汇编代码，如下图：

8048b8d:	bb 01 00 00 00	mov	\$0x1,%ebx
8048b92:	eb 0f	jmp	8048ba3 <phase_2+0x3e>
8048b94:	e8 4a 05 00 00	call	80490e3 <explode_bomb>
8048b99:	eb f2	jmp	8048b8d <phase_2+0x28>
8048b9b:	83 c3 01	add	\$0x1,%ebx
8048b9e:	83 fb 06	cmp	\$0x6,%ebx
8048ba1:	74 12	je	8048bb5 <phase_2+0x50>
8048ba3:	89 d8	mov	%ebx,%eax
8048ba5:	03 04 9c	add	(%esp,%ebx,4),%eax
8048ba8:	39 44 9c 04	cmp	%eax,0x4(%esp,%ebx,4)
8048bac:	74 ed	je	8048b9b <phase_2+0x36>
8048bae:	e8 30 05 00 00	call	80490e3 <explode_bomb>

图 2-5 phase_2 反汇编代码

由上面代码可知，ebx 作为计数器，指向输入的不同变量。进入比较环节后，首先将 ebx 的值赋给 eax，然后让 eax 加上第 ebx 个变量的值，然后将 eax 与第 (ebx+1) 个变量的值比较，如果两者相同则继续，否则引爆炸弹，直到比完。由这段代码可知，第二个变量必须由第一个变量加 1，第三个变量必须由第二个变量加 2，以此类推……

得出答案为 1 2 4 7 11 16（多种答案）

4. 实验结果：

输入上面得出的字符串，结果如下图：

```
Phase 1 delayed, now about the next one!
1 2 4 7 11 16
That's number 2. Keep going!
```

图 2-6 phase_2 结果

2.2.3 阶段 3 条件/分支

1. 任务描述：根据反汇编文件找出需要输入的字符串。
2. 实验设计：首先查看反汇编文件，可以在 main 函数里找到 phase_3，然后找到其对应位置，根据语句的逻辑推出函数相应的功能，然后使用 gdb 协助破解。
3. 实验过程：

(1) 在 main 函数中找到了 phase_3，然后查找 phase_3 的地址，定位到其反汇编代码如下图：

8048be5:	68 cf a1 04 08	push	\$0x804a1cf
8048bea:	ff 74 24 2c	pushl	0x2c(%esp)
8048bee:	e8 1d fc ff ff	call	8048810 <__isoc99_sscanf@plt>

图 2-7 phase_3 反汇编代码

由上图看出，push \$0x804a1cf 语句后调用了输入函数，使用 gdb 查看此地址中存放的信息，如下图：

```
(gdb) x/s 0x804a1cf
0x804a1cf: "%d %d"
```

图 2-8 gdb 查看内存中数据

可知需要输入两个整型变量。

(2)

8048bfb:	83 7c 24 04 07	cmpl	\$0x7,0x4(%esp)
8048c00:	0f 87 8e 00 00 00	ja	8048c94 <phase_3+0xc8>
8048c06:	8b 44 24 04	mov	0x4(%esp),%eax
8048c0a:	ff 24 85 60 a0 04 08	jmp	*0x804a060(,%eax,4)
8048c11:	e8 cd 04 00 00	call	80490e3 <explode_bomb>
8048c16:	eb e3	jmp	8048bfb <phase_3+0x2f>

图 2-9 phase_3 反汇编代码

由上图可知，输入的第一个变量需要小于等于 7，否则会拆弹失败。由 jmp *0x804a060(,%eax,4) 可知，这很有可能是一个跳转表，在 gdb 中查看此内存中的数据，可以得到：

```
(gdb) p/x *0x804a060
$3 = 0x8048c18
(gdb)
```

图 2-10 gdb 查看内存中数据

此地址对应 phase_3 中的语句地址。

(3)

8048c18:	b8 68 01 00 00	mov	\$0x168,%eax
8048c1d:	eb 05	jmp	8048c24 <phase_3+0x58>
8048c1f:	b8 00 00 00 00	mov	\$0x0,%eax
8048c24:	2d e4 03 00 00	sub	\$0x3e4,%eax
8048c29:	05 e8 01 00 00	add	\$0x1e8,%eax
8048c2e:	2d 90 00 00 00	sub	\$0x90,%eax
8048c33:	05 90 00 00 00	add	\$0x90,%eax
8048c38:	2d 90 00 00 00	sub	\$0x90,%eax
8048c3d:	05 90 00 00 00	add	\$0x90,%eax
8048c42:	2d 90 00 00 00	sub	\$0x90,%eax
8048c47:	83 7c 24 04 05	cmpl	\$0x5,0x4(%esp)
8048c4c:	7f 06	jg	8048c54 <phase_3+0x88>
8048c4e:	3b 44 24 08	cmp	0x8(%esp),%eax
8048c52:	74 05	je	8048c59 <phase_3+0x8d>
8048c54:	e8 8a 04 00 00	call	80490e3 <explode_bomb>
8048c59:	8b 44 24 0c	mov	0xc(%esp),%eax
8048c5d:	65 33 05 14 00 00 00	xor	%gs:0x14,%eax
8048c64:	75 3a	jne	8048ca0 <phase_3+0xd4>
8048c66:	83 c4 1c	add	\$0x1c,%esp
8048c69:	c3	ret	
8048c6a:	b8 00 00 00 00	mov	\$0x0,%eax
8048c6f:	eb b8	jmp	8048c29 <phase_3+0x5d>
8048c71:	b8 00 00 00 00	mov	\$0x0,%eax
8048c76:	eb b6	jmp	8048c2e <phase_3+0x62>
8048c78:	b8 00 00 00 00	mov	\$0x0,%eax
8048c7d:	eb b4	jmp	8048c33 <phase_3+0x67>
8048c7f:	b8 00 00 00 00	mov	\$0x0,%eax
8048c84:	eb b2	jmp	8048c38 <phase_3+0x6c>
8048c86:	b8 00 00 00 00	mov	\$0x0,%eax
8048c8b:	eb b0	jmp	8048c3d <phase_3+0x71>
8048c8d:	b8 00 00 00 00	mov	\$0x0,%eax
8048c92:	eb ae	jmp	8048c42 <phase_3+0x76>
8048c94:	e8 4a 04 00 00	call	80490e3 <explode_bomb>

图 2-11 phase_3 反汇编代码

由上图可知，输入的第一个数要小于等于 7，再观察下面的 0x8048c6a 到 0x8048c92 可知，跳转表应该有一部分设置到了此区域。再由上面得到的 jmp *0x804a060(,%eax,4)中的前面地址的数据，带入 eax 的值测试数据。发现当输入第一个变量的值为4时,可以使得运算变得最简单(连续两个加0x90,减0x90)，最后使得输入的第二个变量为0即可。

```
(gdb) p/x *0x804a070
$1 = 0x8048c78
```

图 2-12 gdb 查看内存中数据

得出密码为 4 0（最简单的一组）

4. 实验结果：

输入上面得出的字符串，结果如下图：

```
Halfway there!
4 0
```

图 2-13 phase_3 结果

2.2.4 阶段 4 递归调用和栈

1. 任务描述：根据反汇编文件寻找需要输入的字符串。
2. 实验设计：首先查看反汇编文件，可以在 main 函数里找到 phase_4，然后找到其对应位置，分析代码。
3. 实验过程：

(1) 在 main 函数中找到了 phase_4，然后查找 phase_4 的地址，定位到其反汇编代码如下图：

8048d12:	83 f8 02	cmp	\$0x2,%eax
8048d15:	74 32	je	8048d49 <phase_4+0x61>
8048d17:	e8 c7 03 00 00	call	80490e3 <explode_bomb>
8048d1c:	83 ec 08	sub	\$0x8,%esp
8048d1f:	ff 74 24 0c	pushl	0xc(%esp)
8048d23:	6a 07	push	\$0x7
8048d25:	e8 7b ff ff ff	call	8048ca5 <func4>
8048d2a:	83 c4 10	add	\$0x10,%esp
8048d2d:	3b 44 24 08	cmp	0x8(%esp),%eax
8048d31:	74 05	je	8048d38 <phase_4+0x50>
8048d33:	e8 ab 03 00 00	call	80490e3 <explode_bomb>
8048d38:	8b 44 24 0c	mov	0xc(%esp),%eax
8048d3c:	65 33 05 14 00 00 00	xor	%gs:0x14,%eax
8048d43:	75 12	jne	8048d57 <phase_4+0x6f>
8048d45:	83 c4 1c	add	\$0x1c,%esp
8048d48:	c3	ret	
8048d49:	8b 44 24 04	mov	0x4(%esp),%eax
8048d4d:	83 e8 02	sub	\$0x2,%eax
8048d50:	83 f8 02	cmp	\$0x2,%eax
8048d53:	76 c7	jbe	8048d1c <phase_4+0x34>
8048d55:	eb c0	jmp	8048d17 <phase_4+0x2f>

图 2-14 phase_4 反汇编代码

由上图可知，push \$0x804a1cf 语句给出要输入的字符串，使用 gdb 查看内存中的内容，发现需要输入两个整型变量。

```
(gdb) x/s 0x804a1cf
0x804a1cf: "%d %d"
(gdb)
```


图 2-15 gdb 查看内存中数据

(2) 由反汇编代码可知，输入的第二个变量需要小于等于 4，在这里直接取 4。语句 `call 8048ca5 <func4>` 前进行了两次 `push` 操作，应该是把参数传递给 `func4` 函数，分别是 7, 4。之后比较 `func4` 的返回值与输入的第一个变量的值，相等则拆弹成功。

(3) 观察 `func4` 函数汇编代码。

8048ca8:	8b 5c 24 10	mov	0x10(%esp),%ebx
8048cac:	8b 7c 24 14	mov	0x14(%esp),%edi
8048cb0:	85 db	test	%ebx,%ebx
8048cb2:	7e 2d	jle	8048ce1 <func4+0x3c>
8048cb4:	89 f8	mov	%edi,%eax
8048cb6:	83 fb 01	cmp	\$0x1,%ebx
8048cb9:	74 22	je	8048cdd <func4+0x38>

图 2-16 func4 反汇编代码

上面 `ebx` 的值为 7，`edi` 的值为输入的第二个变量 (4)。当 `ebx` 小于等于 0 时，则返回 0 结束程序。当 `ebx` 等于 1 时则结束程序。

(4) 之后将 6, 4 当做参数传入 `func4`，递归。由此看出递归次数由 `ebx` 决定。

8048cbb:	83 ec 08	sub	\$0x8,%esp
8048cbe:	57	push	%edi
8048cbf:	8d 43 ff	lea	-0x1(%ebx),%eax
8048cc2:	50	push	%eax
8048cc3:	e8 dd ff ff ff	call	8048ca5 <func4>

图 2-17 func4 反汇编代码

(5) 当时仔细观察了递归结构，然后反汇编写了 C 语言程序，运行结果为 132。

(6) 综合上述分析过程，得出密码为 132 4。

4. 实验结果：

由于自己将工具换成了虚拟机，于是测试环境会不一样（之前用的是 win10 子系统）。

输入上面得出的字符串，结果如下图：

```

Halfway there!
132 4
So you got that one. Try this one.

```

图 2-18 phase_4 结果

2.2.5 阶段 5 指针

1. 任务描述：根据反汇编文件寻找需要输入的字符串。
2. 实验设计：首先查看反汇编文件，可以在 main 函数里找到 phase_5，然后找到其对应位置，根据语句的逻辑推出函数相应的功能，然后使用 gdb 协助破解。
3. 实验过程：

(1) 在 main 函数中找到了 phase_5，然后查找 phase_5 的地址，定位到其反汇编代码如下图：

```

08048d5c <phase_5>:
8048d5c: 53                push    %ebx
8048d5d: 83 ec 14          sub     $0x14,%esp
8048d60: 8b 5c 24 1c       mov     0x1c(%esp),%ebx
8048d64: 53                push    %ebx
8048d65: e8 65 02 00 00    call   8048fcf <string_length>
8048d6a: 83 c4 10          add     $0x10,%esp
8048d6d: 83 f8 06          cmp     $0x6,%eax
8048d70: 74 05             je      8048d77 <phase_5+0x1b>
8048d72: e8 6c 03 00 00    call   80490e3 <explode_bomb>
8048d77: 89 d8             mov     %ebx,%eax
8048d79: 83 c3 06          add     $0x6,%ebx
8048d7c: b9 00 00 00 00    mov     $0x0,%ecx
8048d81: 0f b6 10          movzbl (%eax),%edx
8048d84: 83 e2 0f          and     $0xf,%edx
8048d87: 03 0c 95 80 a0 04 08 add     0x804a080(,%edx,4),%ecx
8048d8e: 83 c0 01          add     $0x1,%eax
8048d91: 39 d8             cmp     %ebx,%eax
8048d93: 75 ec             jne     8048d81 <phase_5+0x25>
8048d95: 83 f9 32          cmp     $0x32,%ecx
8048d98: 74 05             je      8048d9f <phase_5+0x43>
8048d9a: e8 44 03 00 00    call   80490e3 <explode_bomb>
8048d9f: 83 c4 08          add     $0x8,%esp
8048da2: 5b                pop     %ebx
8048da3: c3                ret

```

图 2-19 phase_5 反汇编代码

(2) 由 `cmp $0x6,%eax` 以及前面的`<string_length>`可知，应该是要输入 6 个字符。

(3) 后面的语句 `add 0x804a080(,%edx,4),%ecx` 有点奇怪，在 gdb 中查看了 0x804a080 及其以后单元的数据，如下图：

```
(gdb) x/20x 0x804a080
0x804a080 <array.3046>: 0x00000002      0x0000000a      0x00000006      0x00000001
0x804a090 <array.3046+16>: 0x0000000c      0x00000010      0x00000009      0x00000003
0x804a0a0 <array.3046+32>: 0x00000004      0x00000007      0x0000000e      0x00000005
0x804a0b0 <array.3046+48>: 0x0000000b      0x00000008      0x0000000f      0x0000000d
0x804a0c0: 0x79206f53      0x7420756f      0x6b6e6968      0x756f7920
(gdb) █
```

图 2-20 gdb 查看数据

发现前 16 个单元中存储的为 1-16。然后分析 `edx` 中存储的是输入的每个字符后 4 位数据。然后通过跳转表转移到上表中的数据区。后面的语句 `cmp $0x32,%ecx`，将 `ecx` 中的数据和 0x32 比较。而 `ecx` 中存储的是一个累加和。是输入的六个字符经过一定运算后生成的数据累加。

(4) 发现了这个规律后，我通过查找 `ascii` 码表，确定了几组密码。在确定密码的过程中，我发现有很多种组合。然后自己尝试了一段时间，最后确定了 2 个对自己比较有意义的密码。（CFADM!和@WDMAN）

4. 实验结果：

输入上面得出的字符串，结果如下图：

```
So you got that one. Try this one.
CFADM!
Good work! On to the next...
```

图 2-21 phase_5 结果

2.2.6 阶段 6 链表/指针/结构

1. 任务描述：根据反汇编文件分析出需要输入的字符串。

2. 实验设计：首先查看反汇编文件，可以在 `main` 函数里找到 `phase_6`，然后找到其对应位置，根据语句的逻辑推出函数相应的功能，然后使用 `gdb` 协助破解。

3. 实验过程：

(1) 在 `main` 函数中找到了 `phase_6`，然后查找 `phase_6` 的地址，定位到其反汇编代码如下图：

```

08048da4 <phase_6>:
8048da4: 56                push    %esi
8048da5: 53                push    %ebx
8048da6: 83 ec 4c          sub     $0x4c,%esp
8048da9: 65 a1 14 00 00 00 mov     %gs:0x14,%eax
8048daf: 89 44 24 44        mov     %eax,0x44(%esp)
8048db3: 31 c0             xor     %eax,%eax
8048db5: 8d 44 24 14        lea     0x14(%esp),%eax
8048db9: 50                push    %eax
8048dba: ff 74 24 5c        pushl   0x5c(%esp)
8048dbe: e8 45 03 00 00     call    8049108 <read_six_numbers>
8048dc3: 83 c4 10          add     $0x10,%esp
8048dc6: be 00 00 00 00     mov     $0x0,%esi
8048dcb: eb 1c             jmp     8048de9 <phase_6+0x45>
8048dcd: 83 c6 01          add     $0x1,%esi
8048dd0: 83 fe 06          cmp     $0x6,%esi
8048dd3: 74 2e             je      8048e03 <phase_6+0x5f>
8048dd5: 89 f3             mov     %esi,%ebx
8048dd7: 8b 44 9c 0c        mov     0xc(%esp,%ebx,4),%eax
8048ddb: 39 44 b4 08        cmp     %eax,0x8(%esp,%esi,4)
8048ddf: 74 1b             je      8048dfc <phase_6+0x58>
8048de1: 83 c3 01          add     $0x1,%ebx
8048de4: 83 fb 05          cmp     $0x5,%ebx
8048de7: 7e ee             jle     8048dd7 <phase_6+0x33>
8048de9: 8b 44 b4 0c        mov     0xc(%esp,%esi,4),%eax
8048ded: 83 e8 01          sub     $0x1,%eax
8048df0: 83 f8 05          cmp     $0x5,%eax
8048df3: 76 d8             jbe     8048dcd <phase_6+0x29>

```

图 2-22 phase_6 反汇编代码

(2) 由上面<read_six_numbers>可以知道需要输入六个数字，而后面的 sub \$0x1,%eax 和 cmp \$0x5,%eax 语句是判断输入的六个数要小于等于 6 才行。而之前的语句中有一句 cmp %eax,0x8(%esp,%esi,4)，分析其前后语句，发现是判断输入的变量两两不相等。

(3) 继续观察 phase_6 反汇编代码

```

8048e03: 8d 44 24 0c        lea     0xc(%esp),%eax
8048e07: 8d 5c 24 24        lea     0x24(%esp),%ebx
8048e0b: b9 07 00 00 00     mov     $0x7,%ecx
8048e10: 89 ca             mov     %ecx,%edx
8048e12: 2b 10             sub     (%eax),%edx
8048e14: 89 10             mov     %edx,(%eax)
8048e16: 83 c0 04          add     $0x4,%eax
8048e19: 39 c3             cmp     %eax,%ebx
8048e1b: 75 f3             jne     8048e10 <phase_6+0x6c>

```

图 2-23 phase_6 反汇编代码

上述语句实现的功能是用 7 减去输入的六个数，目前还不知道有什么作用。

(4) 继续观察

8048e1b:	75 f3	jne	8048e10 <phase_6+0x6c>
8048e1d:	bb 00 00 00 00	mov	\$0x0,%ebx
8048e22:	89 de	mov	%ebx,%esi
8048e24:	8b 4c 9c 0c	mov	0xc(%esp,%ebx,4),%ecx
8048e28:	b8 01 00 00 00	mov	\$0x1,%eax
8048e2d:	ba 3c c1 04 08	mov	\$0x804c13c,%edx
8048e32:	83 f9 01	cmp	\$0x1,%ecx
8048e35:	7e 0a	jle	8048e41 <phase_6+0x9d>
8048e37:	8b 52 08	mov	0x8(%edx),%edx
8048e3a:	83 c0 01	add	\$0x1,%eax
8048e3d:	39 c8	cmp	%ecx,%eax
8048e3f:	75 f6	jne	8048e37 <phase_6+0x93>
8048e41:	89 54 b4 24	mov	%edx,0x24(%esp,%esi,4)
8048e45:	83 c3 01	add	\$0x1,%ebx
8048e48:	83 fb 06	cmp	\$0x6,%ebx
8048e4b:	75 d5	jne	8048e22 <phase_6+0x7e>
8048e4d:	8b 5c 24 24	mov	0x24(%esp),%ebx
8048e51:	89 d9	mov	%ebx,%ecx
8048e53:	b8 01 00 00 00	mov	\$0x1,%eax
8048e58:	8b 54 84 24	mov	0x24(%esp,%eax,4),%edx
8048e5c:	89 51 08	mov	%edx,0x8(%ecx)
8048e5f:	83 c0 01	add	\$0x1,%eax
8048e62:	89 d1	mov	%edx,%ecx
8048e64:	83 f8 06	cmp	\$0x6,%eax
8048e67:	75 ef	jne	8048e58 <phase_6+0xb4>
8048e69:	c7 42 08 00 00 00 00	movl	\$0x0,0x8(%edx)
8048e70:	be 05 00 00 00	mov	\$0x5,%esi
8048e75:	eb 08	jmp	8048e7f <phase_6+0xdb>
8048e77:	8b 5b 08	mov	0x8(%ebx),%ebx
8048e7a:	83 ee 01	sub	\$0x1,%esi
8048e7d:	74 10	je	8048e8f <phase_6+0xeb>
8048e7f:	8b 43 08	mov	0x8(%ebx),%eax
8048e82:	8b 00	mov	(%eax),%eax

图 2-24 phase_6 反汇编代码

这是我感觉到所有反汇编代码段中最复杂的一段代码，看了好久。后来发现原来这一段代码实现的功能是链表重排，其将 0x804c13c 开头的之后若干个数据进行降序排序。0x804c13c 开头的结构体数组，前 8 个字节存放了一个 int 型数据，后 8 个字节存放一个指针，该指针在指向结构体数组中的下一个元素。程序从 i=1 开始，将结构体数组中第 a[i] 个元素的首地址存放在 b[i] 中，循环 6 次。然后从 b[1] 开始，将 b[1] 对应的元素包含的地址指向 b[2]，依次循环，将 b[6] 所对应的元素包含的地址指向 0。然后从 b[1] 开始判断整个链表是否是降序排列

的，如果不是，则引爆 bomb。

(5) 使用 gdb 查看 0x804c13c 及其之后的数据。

```
(gdb) x/20x 0x804c13c
0x804c13c <node1>: 0x000000fc 0x00000001 0x0804c148 0x0000015f
0x804c14c <node2+4>: 0x00000002 0x0804c154 0x000002a6 0x00000003
0x804c15c <node3+8>: 0x0804c160 0x000001a5 0x00000004 0x0804c16c
0x804c16c <node5>: 0x000000d4 0x00000005 0x0804c178 0x000000f2
0x804c17c <node6+4>: 0x00000006 0x00000000 0x0c046821 0x00000000
(gdb)
```

图 2-25 0x804c13c 及其之后的数据

发现 node1 对应数据 0xfc，node2 对应数据 0x15f，node3 对应数据 0x2a6，node4 对应数据 0x1a5，node5 对应数据 0xd4，node6 对应数据 0xf2。按照降序排列，应该是 3 4 2 1 6 5，但这是用 7 减后的数据，原始输入数据应该是 4 3 5 6 1 2。

4. 实验结果：

输入上面得出的字符串，结果如下图：

```
4 3 5 6 1 2
Congratulations! You've defused the bomb!
may@may-VirtualBox: /VBShare/Lab2-bomb$
```

图 2-26 phase_6 结果

2.2.7 阶段 7 隐藏

1. 任务描述：根据反汇编文件找出进入隐藏关卡的方法。

2. 实验设计：首先查看反汇编文件，可以在 main 函数里找到 phase_defused，然后找到其对应位置，分析进入隐藏关卡的方法。

3. 实验过程：

(1) 查找 phase_defused 的地址，定位到其反汇编代码如下图：

```
08049242 <phase_defused>:
8049242: 83 ec 6c          sub    $0x6c,%esp
8049245: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
804924b: 89 44 24 5c       mov    %eax,0x5c(%esp)
804924f: 31 c0            xor    %eax,%eax
8049251: 83 3d cc c3 04 06 cmpl   $0x6,0x804c3cc
8049258: 74 11            je     804926b <phase_defused+0x29>
```

图 2-27 phase_defused 反汇编代码

由此可见输入的字符串数目必须是 6，意味着必须是在 phase_6 之后。

(2) 继续观察

804926b:	83 ec 0c	sub	\$0xc,%esp
804926e:	8d 44 24 18	lea	0x18(%esp),%eax
8049272:	50	push	%eax
8049273:	8d 44 24 18	lea	0x18(%esp),%eax
8049277:	50	push	%eax
8049278:	8d 44 24 18	lea	0x18(%esp),%eax
804927c:	50	push	%eax
804927d:	68 29 a2 04 08	push	\$0x804a229
8049282:	68 d0 c4 04 08	push	\$0x804c4d0
8049287:	e8 84 f5 ff ff	call	8048810 <__isoc99_sscanf@plt>
804928c:	83 c4 20	add	\$0x20,%esp
804928f:	83 f8 03	cmp	\$0x3,%eax
8049292:	74 12	je	80492a6 <phase_defused+0x64>
8049294:	83 ec 0c	sub	\$0xc,%esp
8049297:	68 58 a1 04 08	push	\$0x804a158
804929c:	e8 1f f5 ff ff	call	80487c0 <puts@plt>
80492a1:	83 c4 10	add	\$0x10,%esp
80492a4:	eb b4	jmp	804925a <phase_defused+0x18>
80492a6:	83 ec 08	sub	\$0x8,%esp
80492a9:	68 32 a2 04 08	push	\$0x804a232
80492ae:	8d 44 24 18	lea	0x18(%esp),%eax
80492b2:	50	push	%eax
80492b3:	e8 36 fd ff ff	call	8048fee <strings_not_equal>

图 2-28 phase_defused 反汇编代码

在 phase_4 读入两个数字和一个字符串，并且字符串需要和 0x804a232 处的字符串相同，使用 gdb 查看其中的数据。

```
(gdb) x/s 0x804a232
0x804a232: "DrEvil"
(gdb)
```

图 2-29 隐藏关卡字符串

需要在 phase_4 的密码后面加上一个空格和 DrEvil。

(3) 观察 secret_phase 代码


```

08048e+8 <secret_phase>:
8048ef8: 53                push    %ebx
8048ef9: 83 ec 08          sub     $0x8,%esp
8048efc: e8 42 02 00 00    call   8049143 <read_line>
8048f01: 83 ec 04          sub     $0x4,%esp
8048f04: 6a 0a            push    $0xa
8048f06: 6a 00            push    $0x0
8048f08: 50               push    %eax
8048f09: e8 72 f9 ff ff    call   8048880 <strtol@plt>
8048f0e: 89 c3            mov     %eax,%ebx
8048f10: 8d 40 ff          lea     -0x1(%eax),%eax
8048f13: 83 c4 10          add     $0x10,%esp
8048f16: 3d e8 03 00 00    cmp     $0x3e8,%eax
8048f1b: 77 32            ja      8048f4f <secret_phase+0x57>
8048f1d: 83 ec 08          sub     $0x8,%esp
8048f20: 53                push    %ebx
8048f21: 68 88 c0 04 08    push    $0x804c088
8048f26: e8 7c ff ff ff    call   8048ea7 <fun7>
8048f2b: 83 c4 10          add     $0x10,%esp
8048f2e: 83 f8 02          cmp     $0x2,%eax
8048f31: 74 05            je      8048f38 <secret_phase+0x40>
8048f33: e8 ab 01 00 00    call   80490e3 <explode_bomb>
8048f38: 83 ec 0c          sub     $0xc,%esp
8048f3b: 68 28 a0 04 08    push    $0x804a028
8048f40: e8 7b f8 ff ff    call   80487c0 <puts@plt>
8048f45: e8 f8 02 00 00    call   8049242 <phase_defused>
8048f4a: 83 c4 18          add     $0x18,%esp
8048f4d: 5b               pop     %ebx
8048f4e: c3               ret
8048f4f: e8 8f 01 00 00    call   80490e3 <explode_bomb>
8048f54: eb c7            jmp     8048f1d <secret_phase+0x25>

```

图 2-30 secret_phase 反汇编代码

Secret_phase 读取一个字符串并且将其转换成 long 型正数，将其与 0x3e8 比较，如果大于则引爆炸弹。否则调用 fun7，一个参数为 0x804c088，如果 fun7 的返回值为 2，则拆弹成功。

(4) 分析 fun7 反汇编代码


```

08048ea7 <fun7>:
8048ea7: 53                push    %ebx
8048ea8: 83 ec 08          sub     $0x8,%esp
8048eab: 8b 54 24 10       mov     0x10(%esp),%edx
8048eaf: 8b 4c 24 14       mov     0x14(%esp),%ecx
8048eb3: 85 d2            test    %edx,%edx
8048eb5: 74 3a            je      8048ef1 <fun7+0x4a>
8048eb7: 8b 1a            mov     (%edx),%ebx
8048eb9: 39 cb            cmp     %ecx,%ebx
8048ebb: 7f 21            jg      8048ede <fun7+0x37>
8048ebd: b8 00 00 00 00    mov     $0x0,%eax
8048ec2: 39 cb            cmp     %ecx,%ebx
8048ec4: 74 13            je      8048ed9 <fun7+0x32>
8048ec6: 83 ec 08          sub     $0x8,%esp
8048ec9: 51              push    %ecx
8048eca: ff 72 08          pushl   0x8(%edx)
8048ecd: e8 d5 ff ff ff    call    8048ea7 <fun7>
8048ed2: 83 c4 10          add     $0x10,%esp
8048ed5: 8d 44 00 01       lea     0x1(%eax,%eax,1),%eax
8048ed9: 83 c4 08          add     $0x8,%esp
8048edc: 5b              pop     %ebx
8048edd: c3              ret
8048ede: 83 ec 08          sub     $0x8,%esp
8048ee1: 51              push    %ecx
8048ee2: ff 72 04          pushl   0x4(%edx)
8048ee5: e8 bd ff ff ff    call    8048ea7 <fun7>
8048eea: 83 c4 10          add     $0x10,%esp
8048eed: 01 c0            add     %eax,%eax
8048eef: eb e8            jmp     8048ed9 <fun7+0x32>
8048ef1: b8 ff ff ff ff    mov     $0xffffffff,%eax
8048ef6: eb e1            jmp     8048ed9 <fun7+0x32>

```

图 2-31 func7 反汇编代码

Fun7 为一棵二叉搜索树，进行逆向工程：

```

struct tree {
    int value;
    struct tree * left;
    struct tree * right;
};

int fun7(struct tree * p, int num) {
    if (p == NULL) return -1;
    if (p->value == num) return 0;
    if (p->value < num)

```

```

{
    p = p->right;
    return 2 * fun7(p, num) + 1;
}
else
{
    p = p->left;
    return 2 * fun7(p, num);
}
}

```

由上面的 secret_phase 可以知道由节点 0x804c088 开始。

```

(gdb) x/100 0x804c088
0x804c088 <n1>: 0x24    0x00    0x00    0x00    0x94    0xc0    0x04    0x08
0x804c090 <n1+8>:    0xa0    0xc0    0x04    0x08    0x08    0x00    0x00    0x00
0x804c098 <n21+4>:    0xc4    0xc0    0x04    0x08    0xac    0xc0    0x04    0x08
0x804c0a0 <n22>:      0x32    0x00    0x00    0x00    0xb8    0xc0    0x04    0x08
0x804c0a8 <n22+8>:    0xd0    0xc0    0x04    0x08    0x16    0x00    0x00    0x00
0x804c0b0 <n32+4>:    0x18    0xc1    0x04    0x08    0x00    0xc1    0x04    0x08
0x804c0b8 <n33>:      0x2d    0x00    0x00    0x00    0xdc    0xc0    0x04    0x08
0x804c0c0 <n33+8>:    0x24    0xc1    0x04    0x08    0x06    0x00    0x00    0x00
0x804c0c8 <n31+4>:    0xe8    0xc0    0x04    0x08    0x0c    0xc1    0x04    0x08
0x804c0d0 <n34>:      0x6b    0x00    0x00    0x00    0xf4    0xc0    0x04    0x08
0x804c0d8 <n34+8>:    0x30    0xc1    0x04    0x08    0x28    0x00    0x00    0x00
0x804c0e0 <n45+4>:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804c0e8 <n41>:      0x01    0x00    0x00    0x00
(gdb) print n1
$1 = 36
(gdb) print n21
$2 = 8
(gdb) print n22
$3 = 50
(gdb) print n31
$4 = 6
(gdb) print n32
$5 = 22
(gdb) print n33
$6 = 45
(gdb) print n34
$7 = 107
(gdb) print n41
$8 = 1
(gdb) print n42
$9 = 7
(gdb) print n43
$10 = 20
(gdb) print n44
$11 = 35
(gdb) print n45
$12 = 40
(gdb) print n46
$13 = 47
(gdb) print n47
$14 = 99
(gdb) print n48
$15 = 1001
(gdb)

```

图 2-32 gdb 查看数据

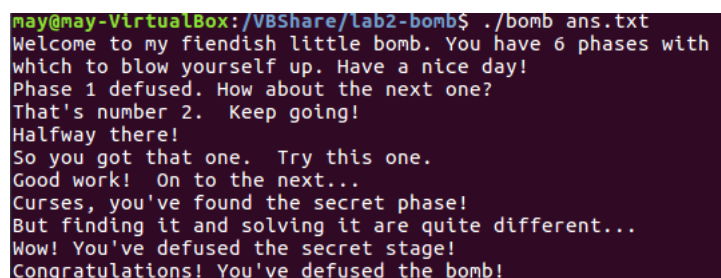
首先查看内存中数据，发现有 n1, n21, n22 等奇怪的标号。然后使用 print

打印这些标号后的数据，发现是一棵树，其数据及分布如上。

要使 fun7 返回值为 2，可以令第二个参数为 22。这样根据逆向工程得出的函数可以推出其返回值为 2。

4. 实验结果：

输入上面得出的密码 22，结果如下图：



```
may@may-VirtualBox:/VBShare/Lab2-bomb$ ./bomb ans.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...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

图 2-33 secret_phase 结果

2.3 实验小结

本次实验主要利用反汇编出来的文件进行分析，结合 gdb 工具查看内存或者寄存器中的值、进行单步调试等。

本次实验最重要的就是要看懂汇编代码，能够对汇编代码进行正确的分析，进行逆向工程得到高级语言代码。能够对系统将一个高级语言程序翻译成汇编语言的过程有深入了解，了解机器是如何读懂代码并进行实现的。

通过本次实验，我深入了解了栈的机制，并且知道函数的传参方式，大大提升了自己读汇编代码的能力，也有一定的能力能够将汇编代码进行逆向工程得到高级语言。

实验总结

在实验一中，进行的是 datalab，主要是将一些运算转换成位操作来实现。在这个过程中，我主要深入了解了数据的机器级表示是怎么在计算机中存储的，通过对数据的二进制码进行位操作能够实现一些基本的运算操作以及较复杂的操作。

本次实验的主要内容就是实验二和实验三，这两个都是需要实验者有很好的汇编语言基础，要看懂汇编代码，能够对汇编代码进行正确的分析，进行逆向工程得到高级语言代码。能够对系统将一个高级语言程序翻译成汇编语言的过程有深入了解，了解机器是如何实现的。

实验二，三主要利用反汇编出来的文件进行分析，结合 gdb 工具查看内存或者寄存器中的值、进行单步调试等。

实验二主要是能够进行破解密码，主要是对汇编语言的能力有所考察，考察了我们是否能对汇编代码进行很好地翻译，找到我们需要输入的密码。而实验三则主要是对栈的理解，缓冲区溢出攻击。我们要很好地对函数的返回机制，调用机制，栈的保存机制有很好地理解，对实验者这一方面的能力有了更高的要求，特别是缓冲区攻击的最后一个阶段，我通过查阅资料才能将这个阶段解决，属于比较难的一部分。

本次实验主要是获得了读汇编代码，熟练使用 gdb 工具的能力。最重要的是深入理解计算机系统是如何工作的。对自己的系统能力有一定的提升。