

**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](#_Toc105968098)

[2.1 实验概述 1](#_Toc105968099)

[2.2 实验内容 2](#_Toc105968100)

[2.2.1 阶段1 破解<phase\_1> 3](#_Toc105968101)

[2.2.2 阶段2 破解<phase\_2> 6](#_Toc105968102)

[2.2.3 阶段3 破解<phase\_3> 9](#_Toc105968103)

[2.2.4 阶段4 破解<phase\_4> 12](#_Toc105968104)

[2.2.5 阶段5 破解<phase\_5> 16](#_Toc105968105)

[2.2.6 阶段6 破解<phase\_6> 19](#_Toc105968106)

[2.2.7 阶段7 寻找隐藏阶段 22](#_Toc105968107)

[2.3 实验小结 28](#_Toc105968108)

[**实验3：** **缓冲区溢出攻击** 29](#_Toc105968109)

[3.1 实验概述 29](#_Toc105968110)

[3.2 实验内容 30](#_Toc105968111)

[3.2.1 阶段0 smoke解题过程 32](#_Toc105968112)

[3.2.2 阶段1 fizz解题过程 34](#_Toc105968113)

[3.2.3 阶段2 bang解题过程 36](#_Toc105968114)

[3.2.4 阶段3 boom解题过程 40](#_Toc105968115)

[3.2.5 阶段4 nitro解题过程 43](#_Toc105968116)

[3.3实验小结 49](#_Toc105968117)

[**实验总结** 50](#_Toc105968118)

# 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。

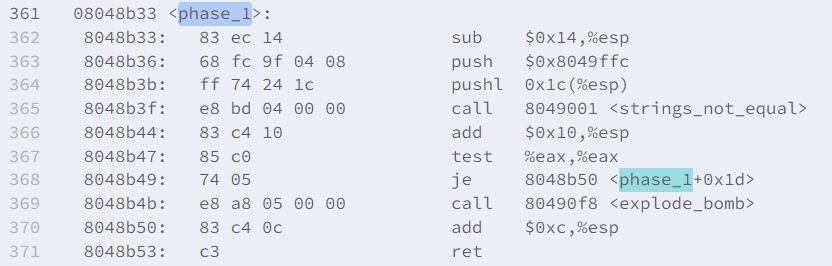


图 2. 1 在反汇编代码中定位到phase\_1函数

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。

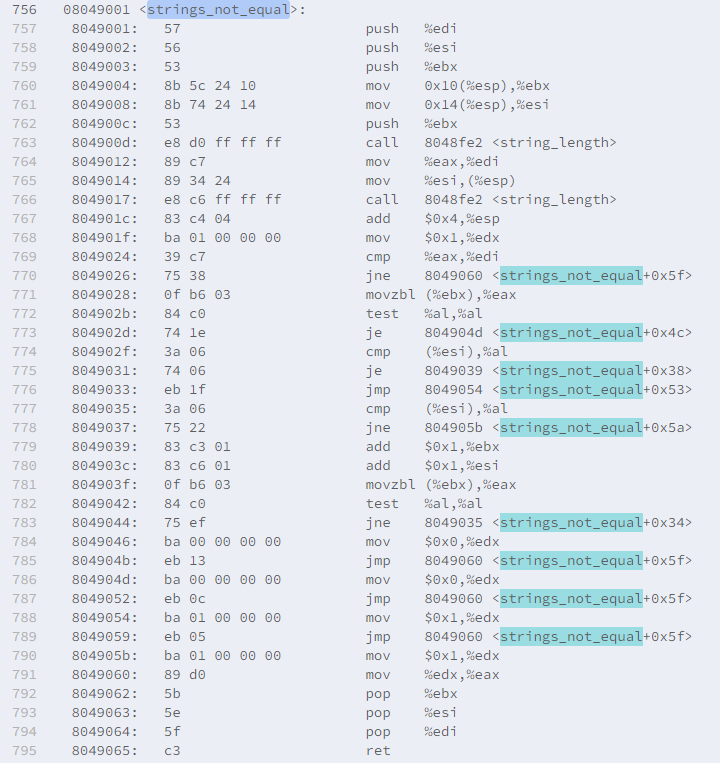


图 2. 2 在反汇编代码中定位到strings\_not\_equal函数

继续查找string\_length函数，如下图所示。

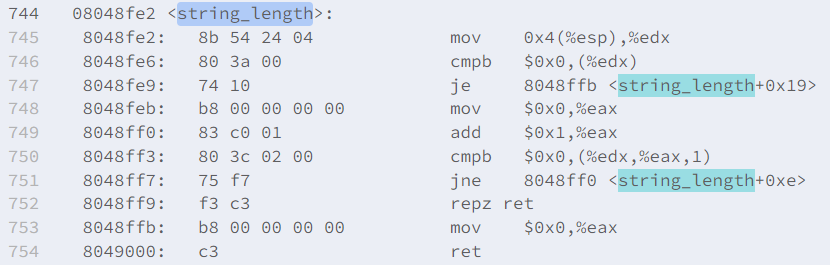


图 2. 3 在反汇编代码中定位到string\_length函数

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。

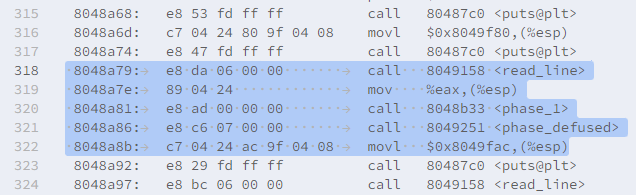


图 2. 4 main函数中调用phase\_1函数上下文

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。



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。

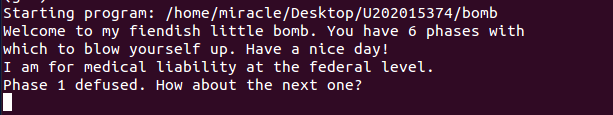


图 2. 6 测试第一个密码串

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。

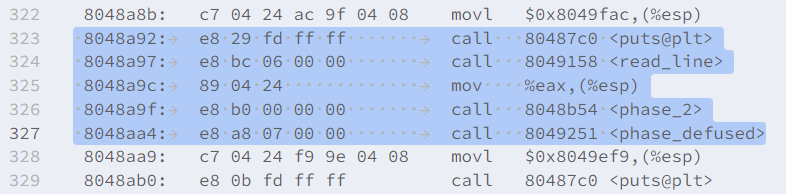


图 2. 7 main函数中调用phase\_2函数上下文

转到phase\_2函数，如下图所示。

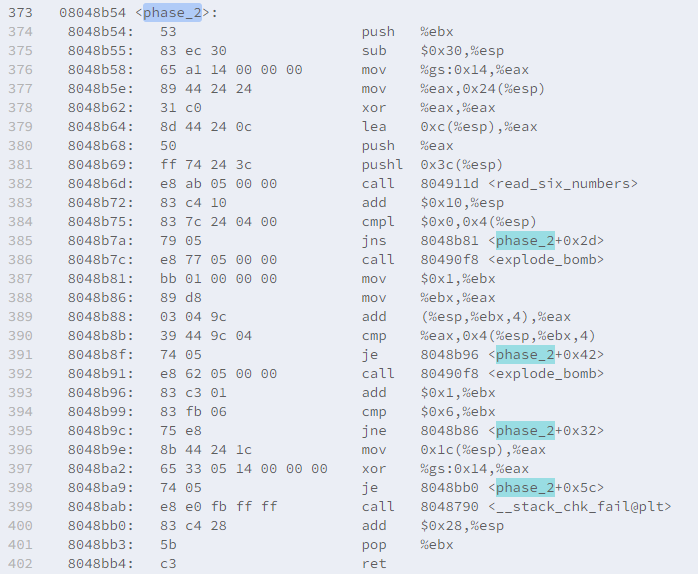


图 2. 8 在反汇编代码中定位到phase\_2函数

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。

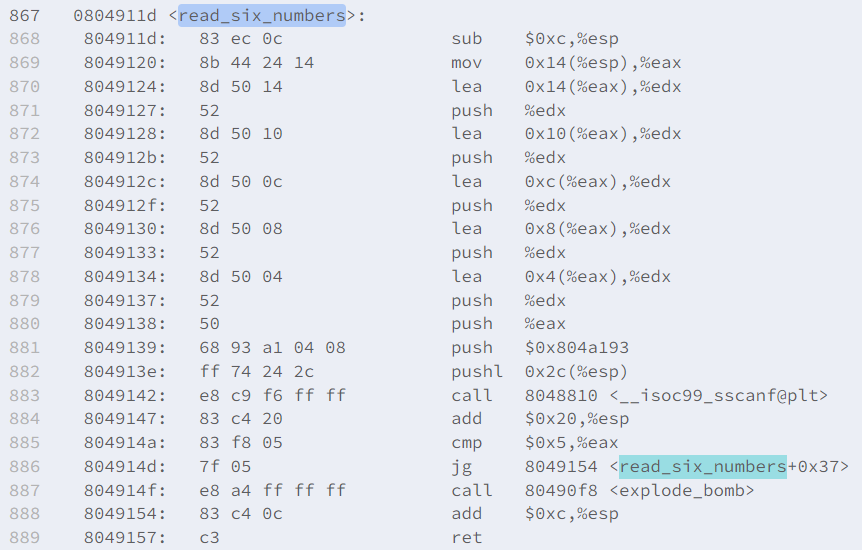


图 2. 9 在反汇编代码中定位到read\_six\_numbers函数

使用gdb查看Line 881中0x804a193地址处的值，如下图所示。

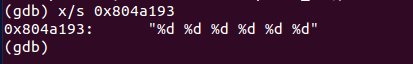


图 2. 10 gdb查看0x804a193处内容

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”。

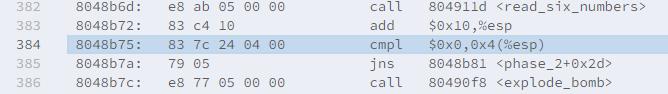


图 2. 11 Line 384局部反汇编代码

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）需要满足以下条件：

①≥0；

②；

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。

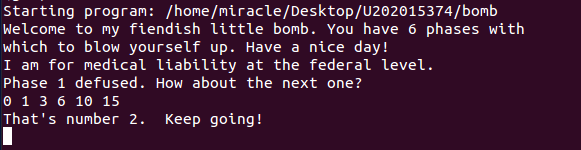


图 2. 12 测试第二个密码串

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”。

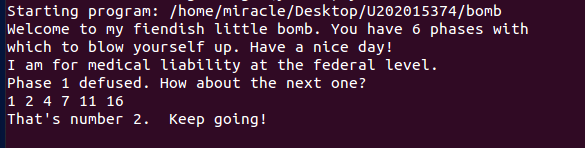


图 2. 13 测试第二个密码串的另一种解答

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。

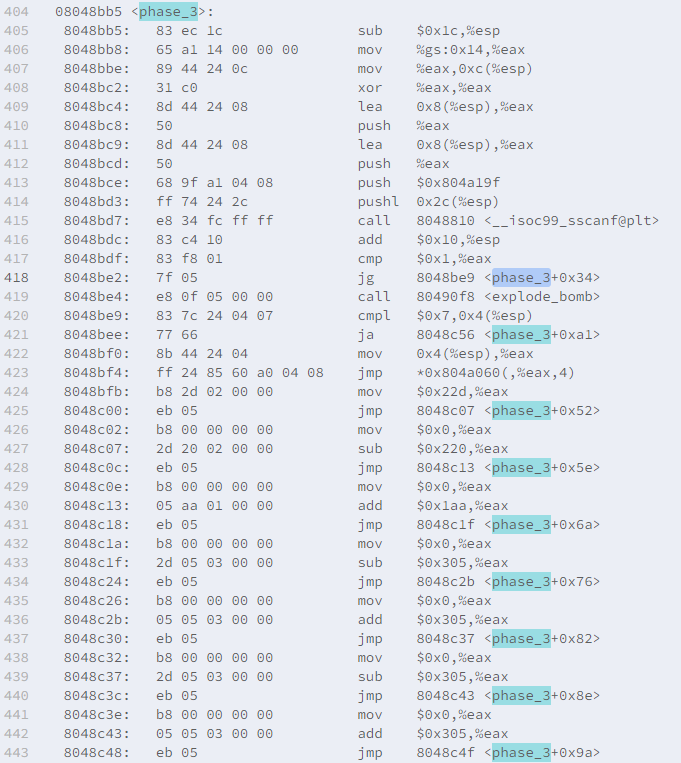




图 2. 14 在反汇编代码中定位到phase\_3函数

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)。

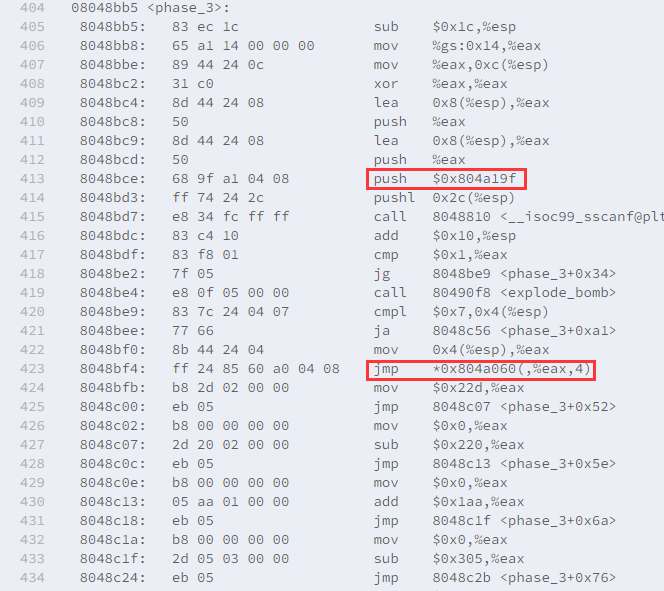


图 2. 15 phase\_3反汇编代码中值得注意的两个地方

使用gdb查看Line 413中0x804a19f地址处的值，如下图所示。



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）



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。

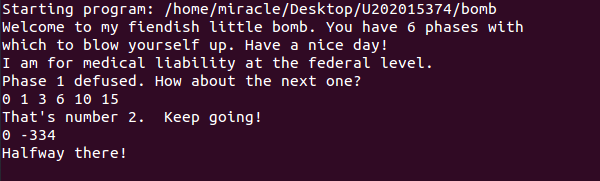


图 2. 18 测试第三个密码串

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。

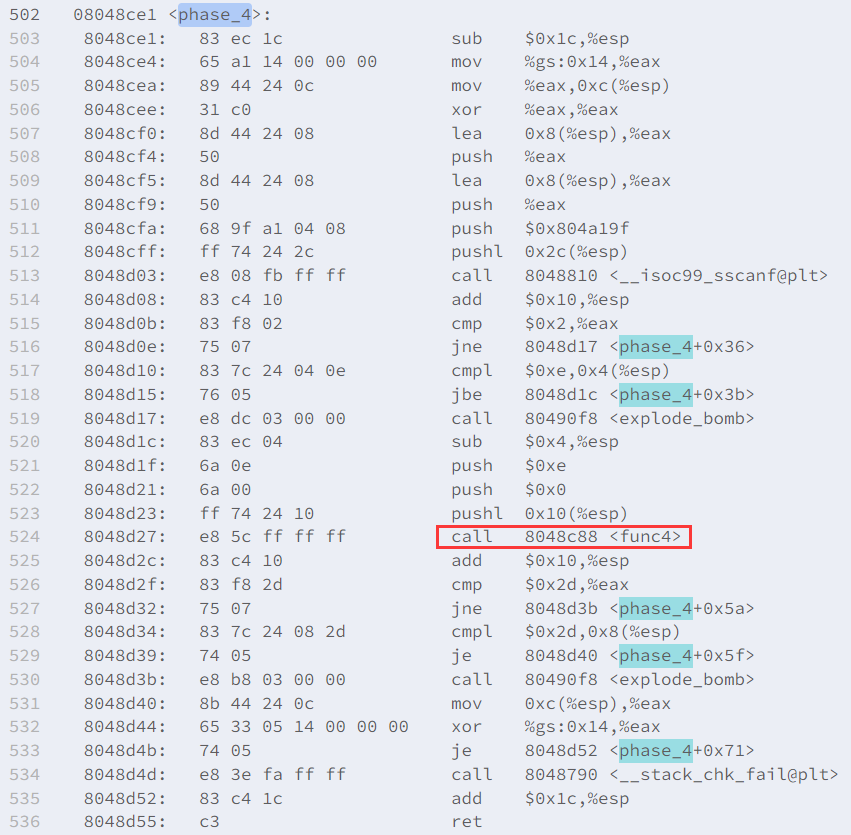


图 2. 19 在反汇编代码中定位到phase\_4函数

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。

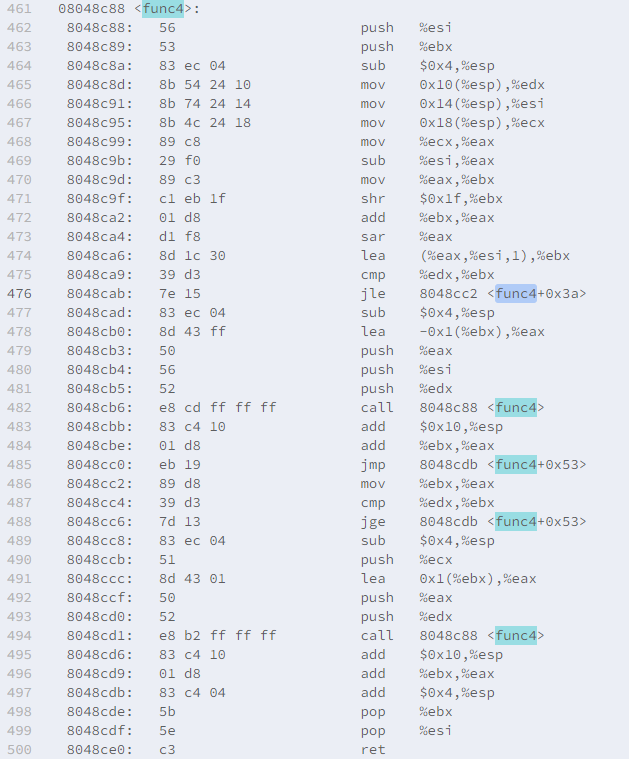


图 2. 20 在反汇编代码中定位到func4函数

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。



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。

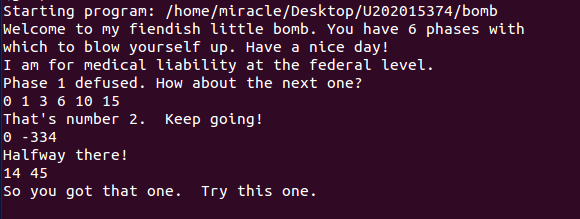


图 2. 22 测试第四个密码串

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。



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。

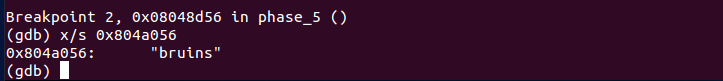


图 2. 24 gdb查看0x804a056处内容

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。

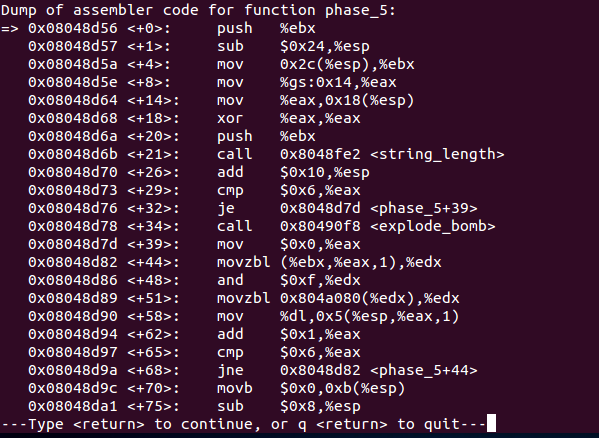


图 2. 25 使用disas反汇编phase\_5

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。

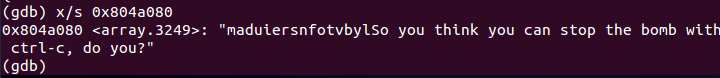


图 2. 26 gdb查看0x804a080处内容

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”。

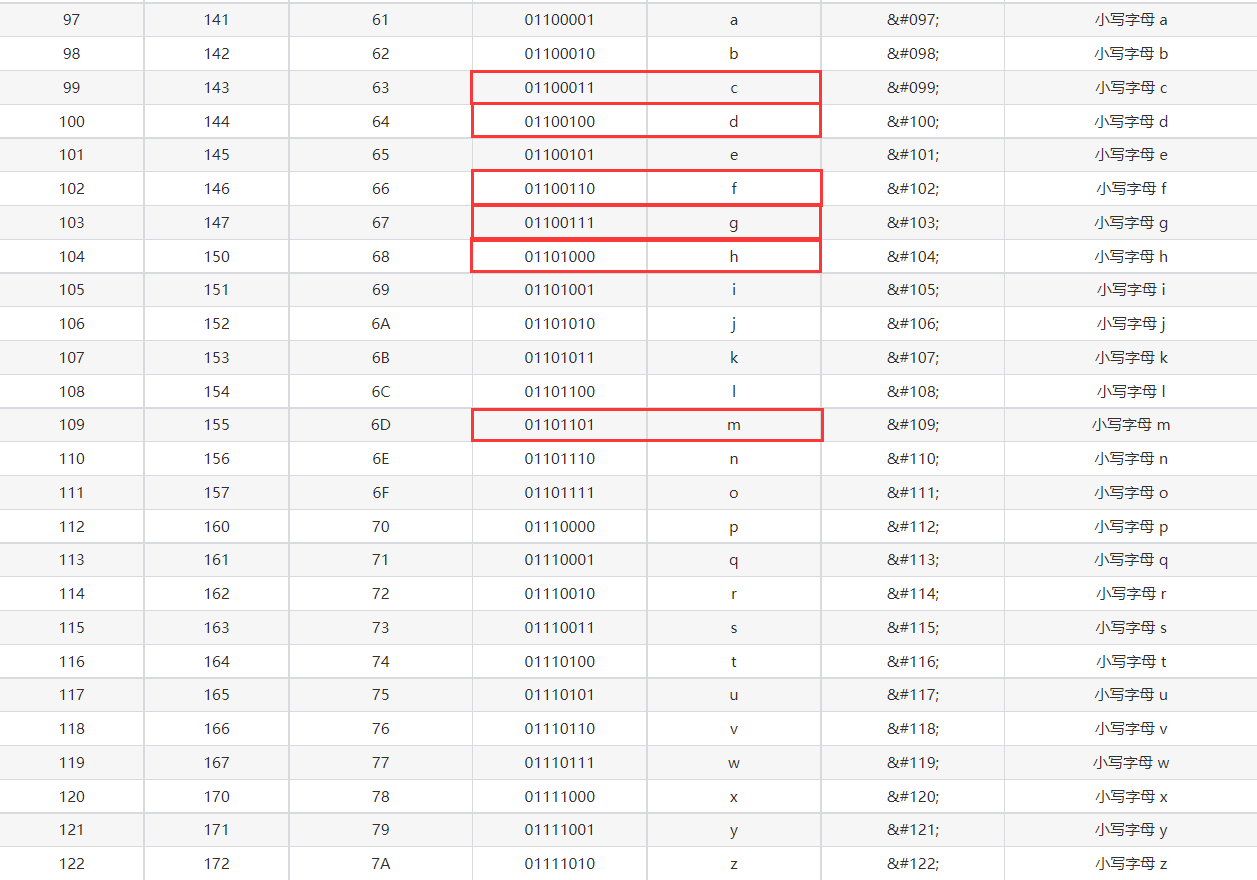


图 2. 27 在ASCII码表中选取字母

运行程序测试，发现第五个“炸弹”被成功拆除。

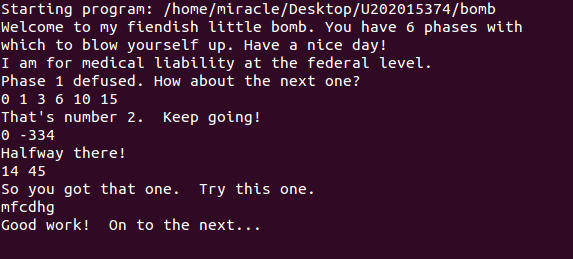


图 2. 28 测试第五个密码串

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

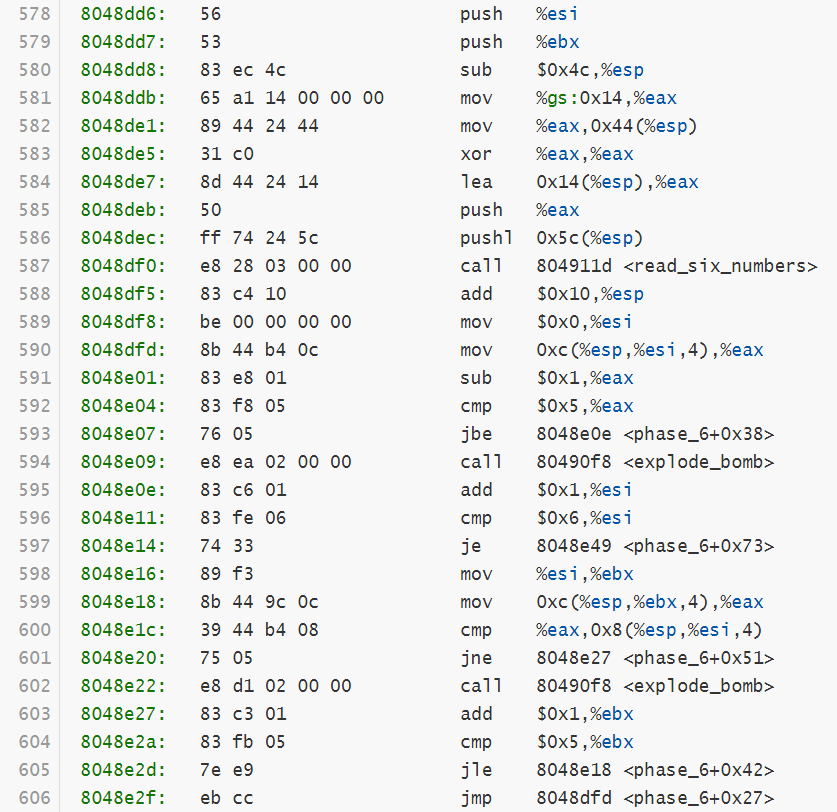


图 2. 29 phase\_6函数反汇编代码部分（I）

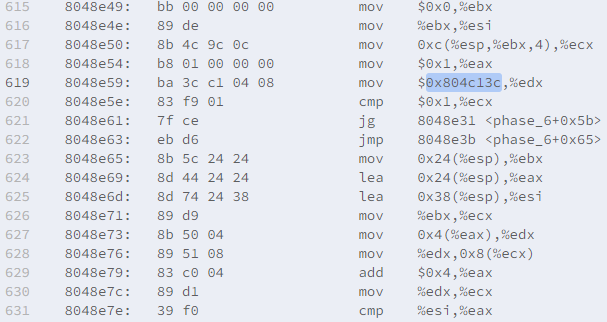
分析两个调用explode\_bomb函数语句的前几行（Line 593和Line 599 – Line 601），可以发现这一部分代码的两个功能：

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



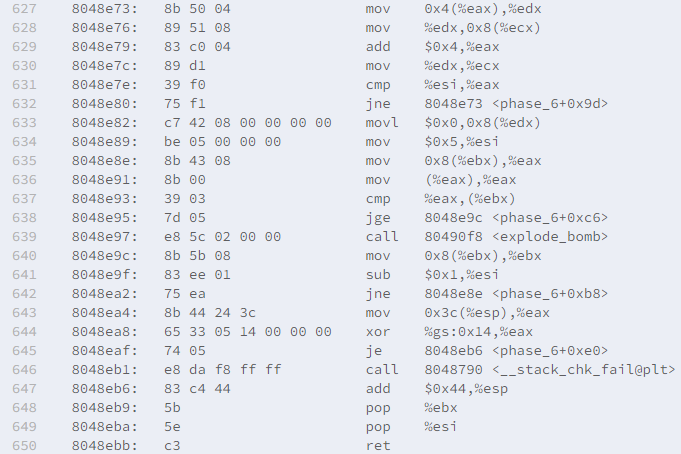


图 2. 30 phase\_6函数反汇编代码部分（II）

注意到这里有一个地址0x804c13c，使用gdb查看内容，如下图所示。

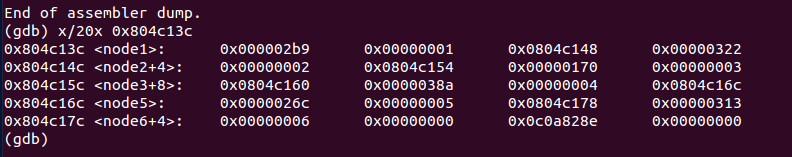


图 2. 31 gdb查看0x804c13c

这实际上是一个结构体，类似于

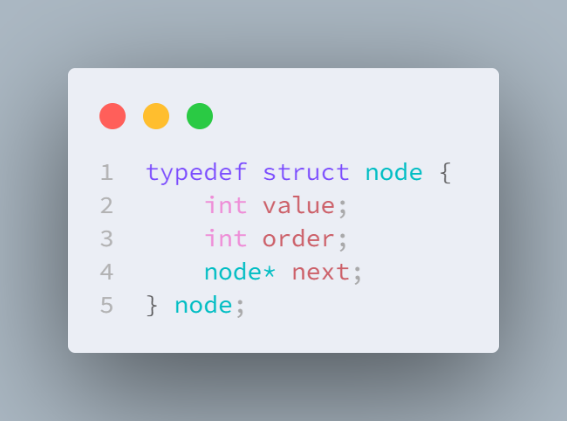


图 2. 32 反汇编结构体的C语言表示

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。

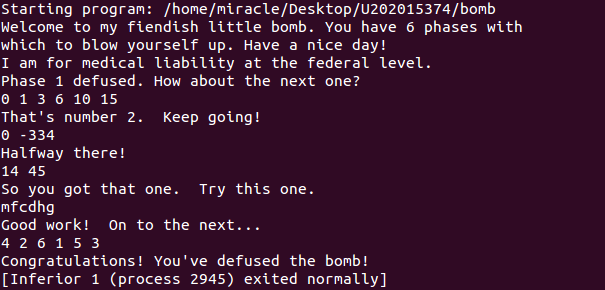


图 2. 33 测试第六个密码串

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。

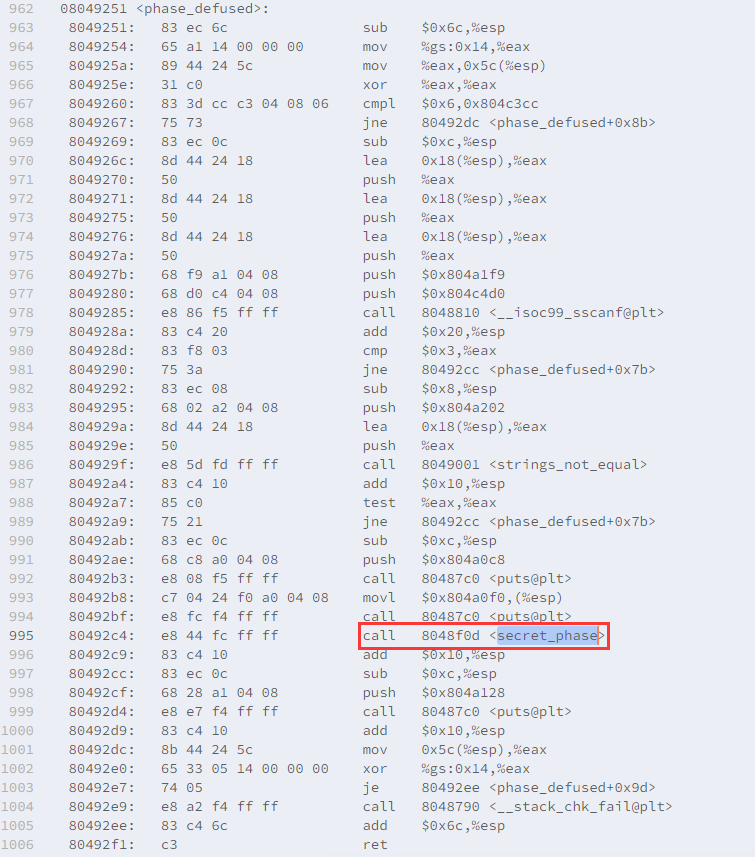


图 2. 34 在反汇编代码中定位到phase\_defused函数

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。

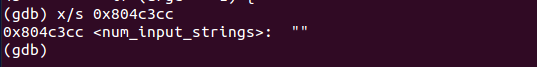


图 2. 35 gdb查看0x804c3cc处内容

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。

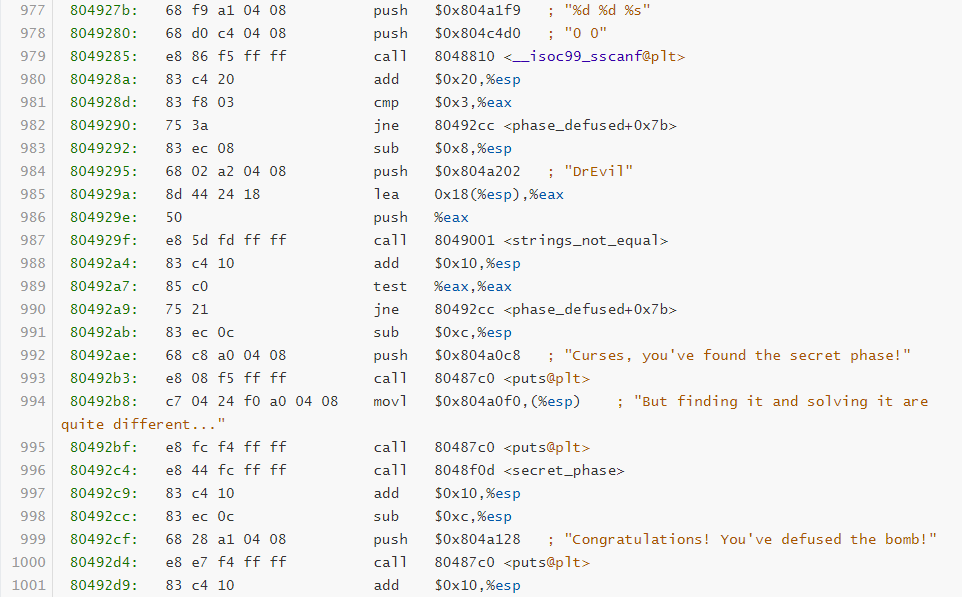


图 2. 36 含标注的phase\_defused反汇编代码

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。

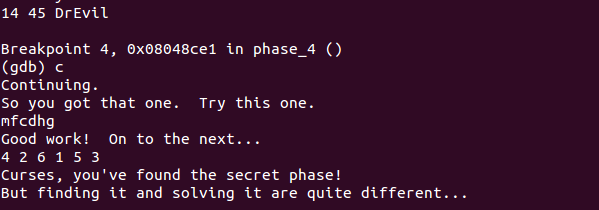


图 2. 37 附加字符串进入隐藏关卡

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。

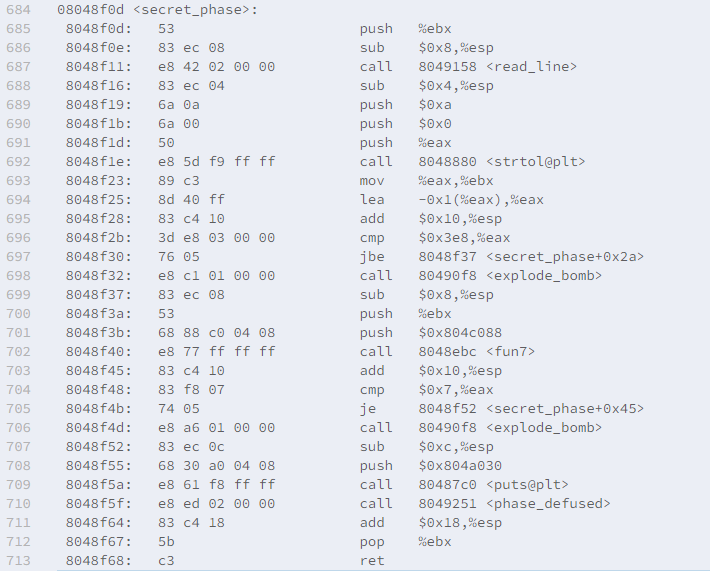


图 2. 38 在反汇编代码中定位到secret\_phase函数

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。

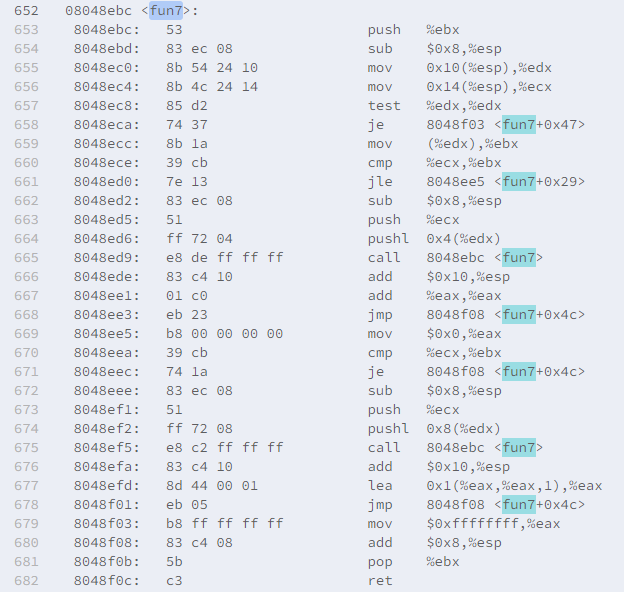


图 2. 39 在反汇编代码中定位到fun7函数

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。

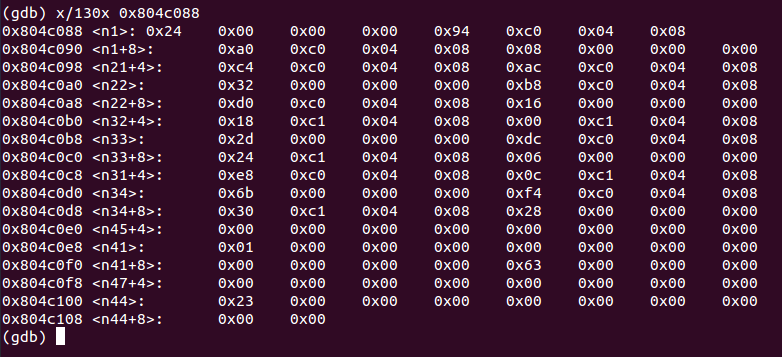


图 2. 40 gdb查看0x804c088处内容

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。

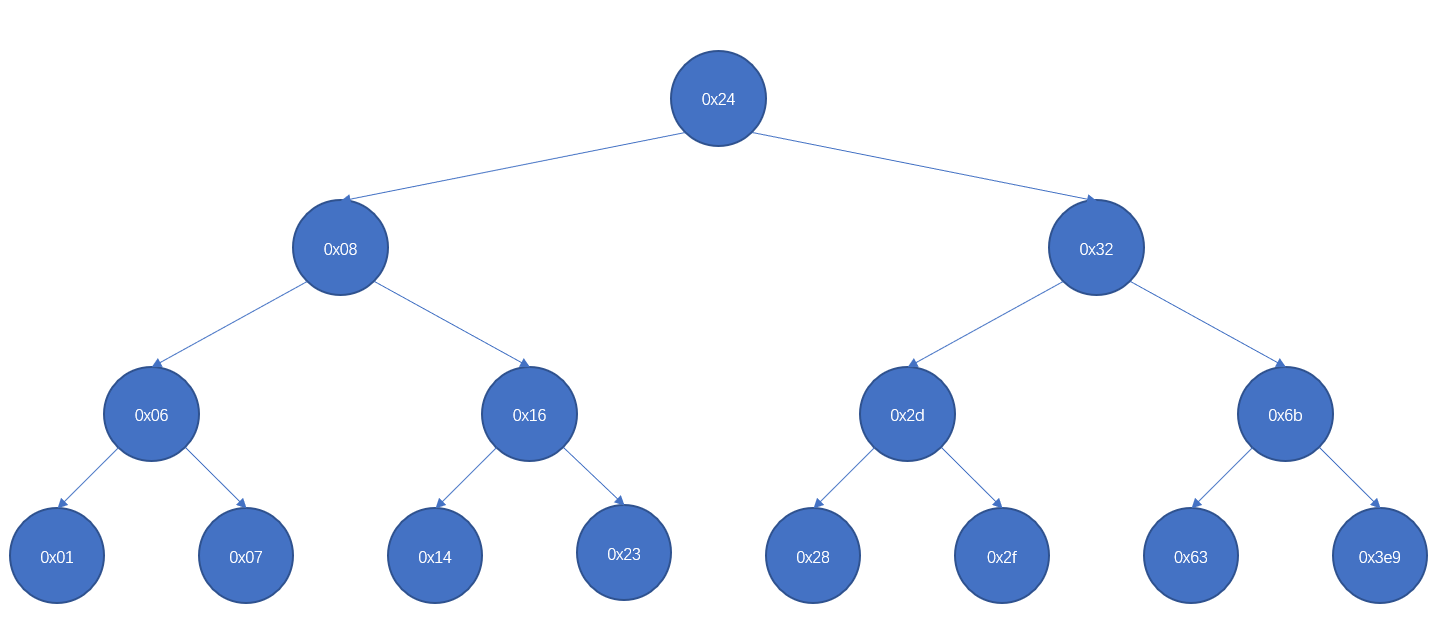


图 2. 41 结点对应的二叉树示意图

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 → 0x32 → 0x6b → 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。

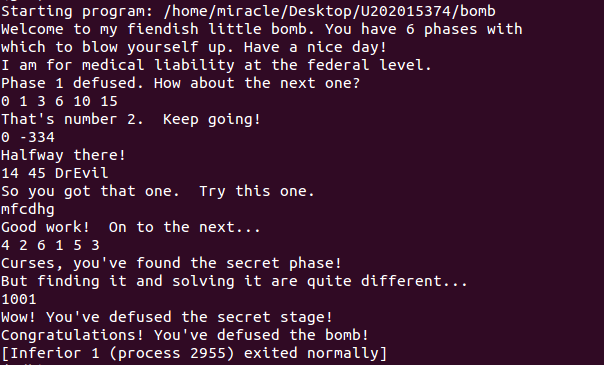


图 2. 44 测试隐藏关密码串

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。



图 3. 1 关键函数源码摘录

转到getbuf函数的反汇编代码。

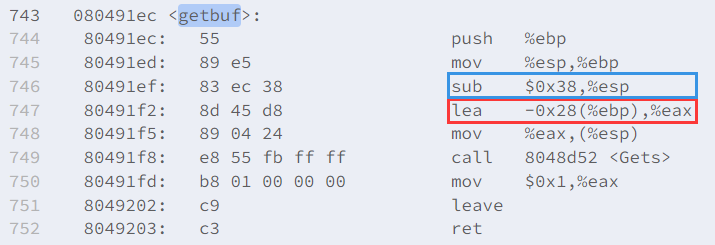


图 3. 2 在反汇编代码中定位到getbuf函数

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。

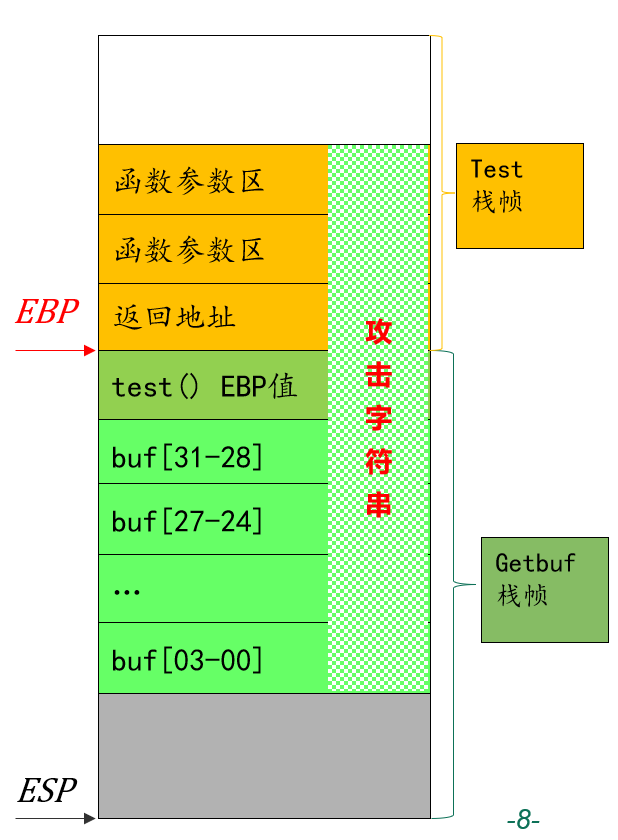
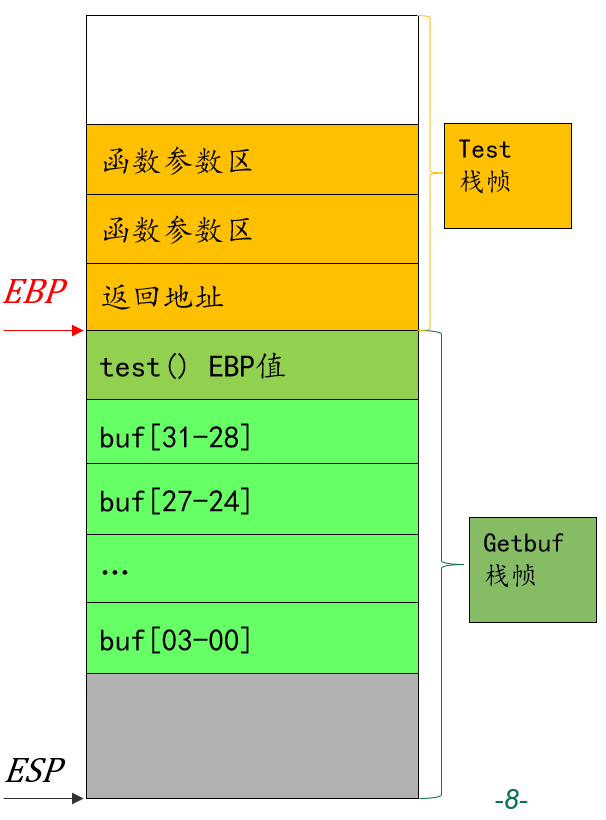


图 3. 3 栈帧结构和攻击字符串的摆放

### 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。



图 3. 4 smoke函数源码

转到smoke函数的反汇编代码。



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。

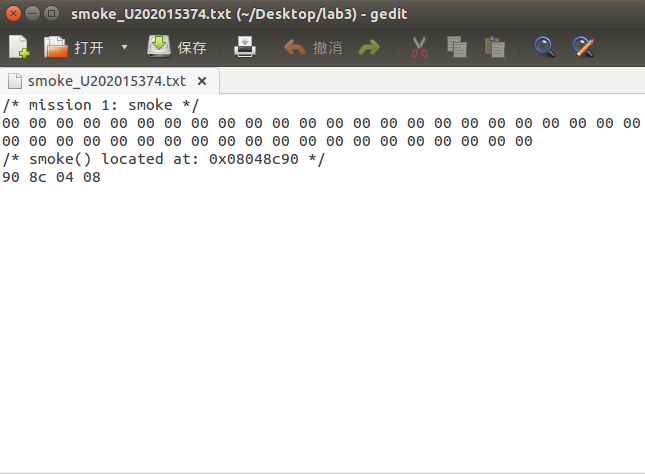


图 3. 6 smoke解决方案

运行程序测试，smoke攻击成功。

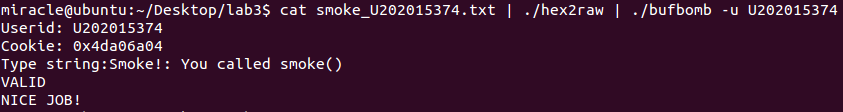


图 3. 7 实施smoke攻击

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。

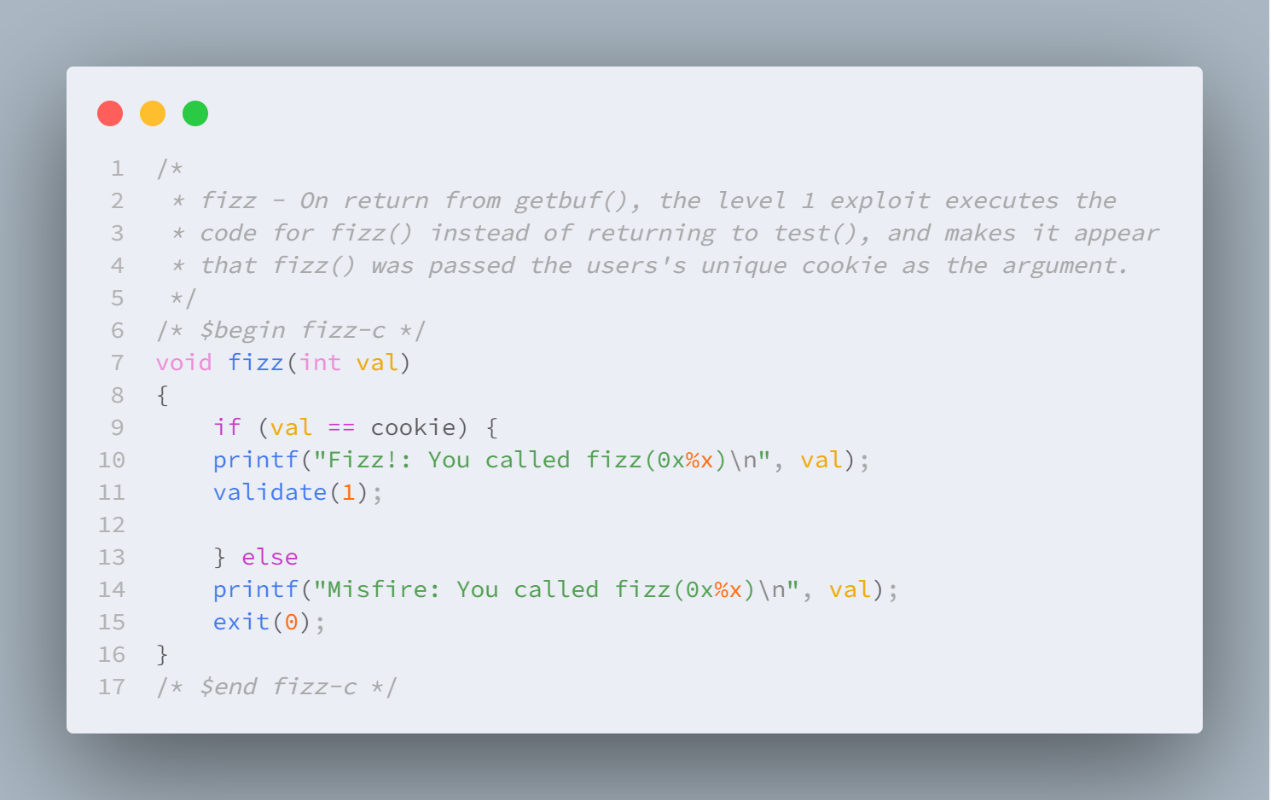


图 3. 8 fizz函数源码

转到fizz函数的反汇编代码。

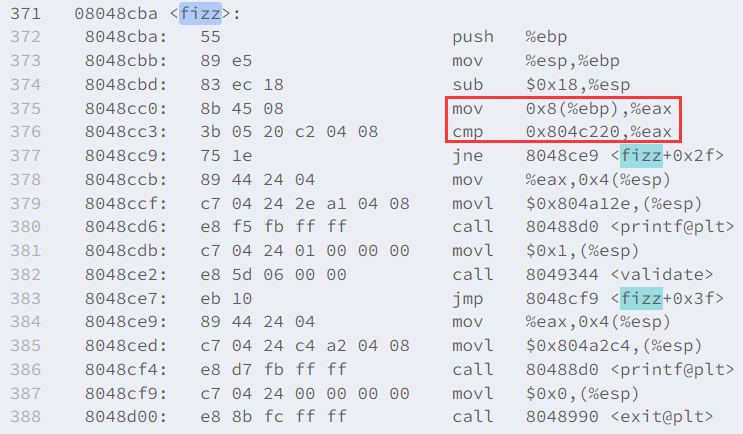


图 3. 9 在反汇编代码中定位到fizz函数

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。

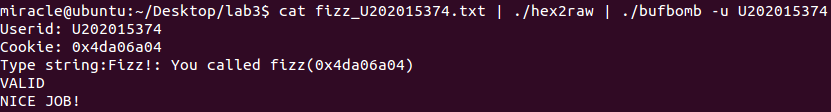


图 3. 11 实施fizz攻击

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。



图 3. 12 bang函数源码

转到bang函数反汇编代码。

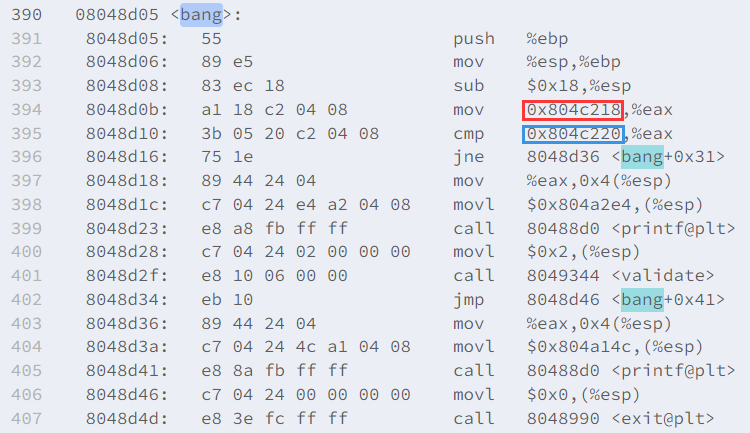


图 3. 13 在反汇编代码中定位到bang函数

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。



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。

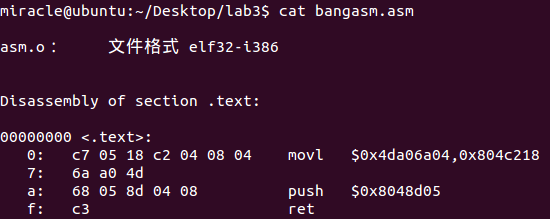


图 3. 15 cat查看恶意代码的机器码

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。

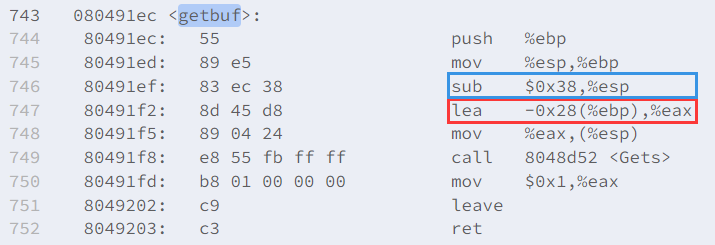


图 3. 16 getbuf函数反汇编代码

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。

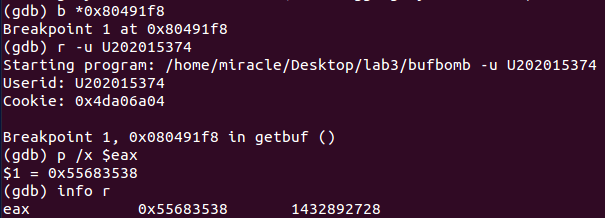


图 3. 17 gdb查看EAX寄存器

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。

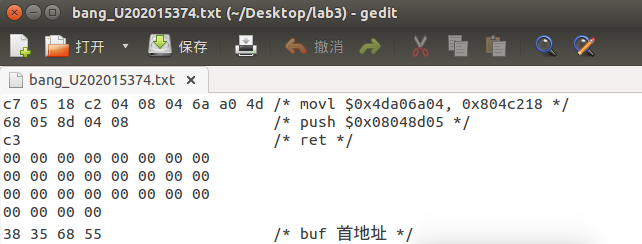


图 3. 18 bang解决方案

运行程序测试，bang攻击成功。

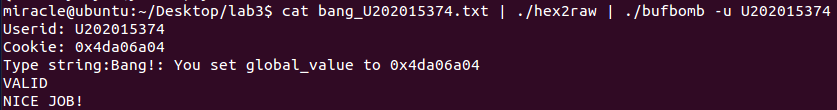


图 3. 19 实施bang攻击

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。

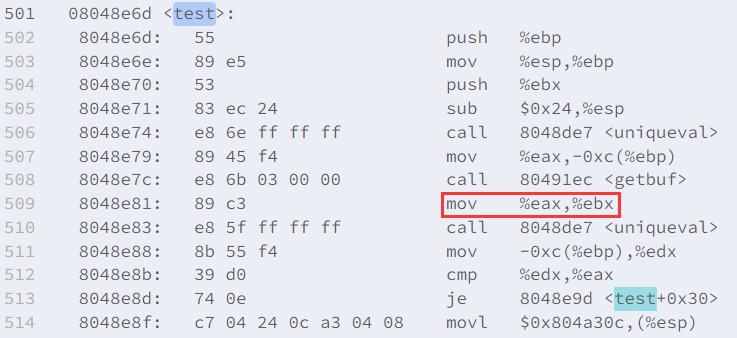


图 3. 20 找到test函数调用getbuf函数后的下一条指令地址

我们需要构造的汇编代码如下，存放在asm2.s文件中。

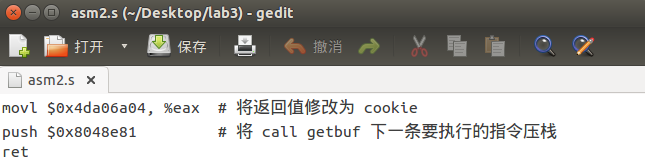


图 3. 21 恶意代码asm2.s

和构造bang攻击时相似，我们得到汇编代码对应的机器码。

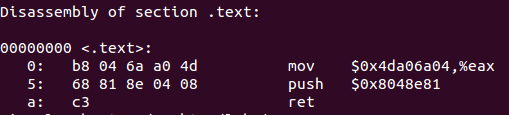


图 3. 22 cat查看恶意代码的机器码

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。

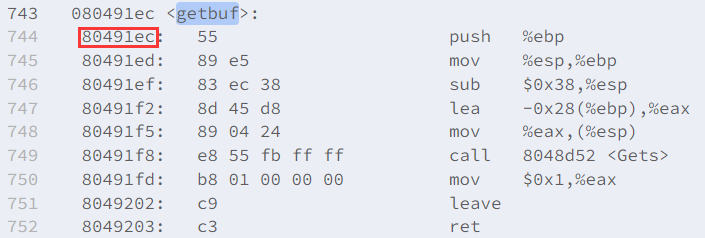


图 3. 23 找到设置断点的位置

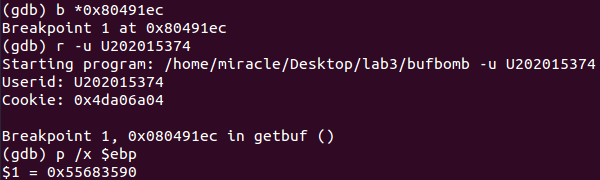


图 3. 24 gdb查看ebp寄存器

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。



图 3. 25 boom解决方案

运行程序测试，boom攻击成功。

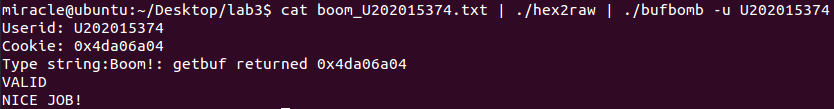


图 3. 26 实施boom攻击

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。



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。

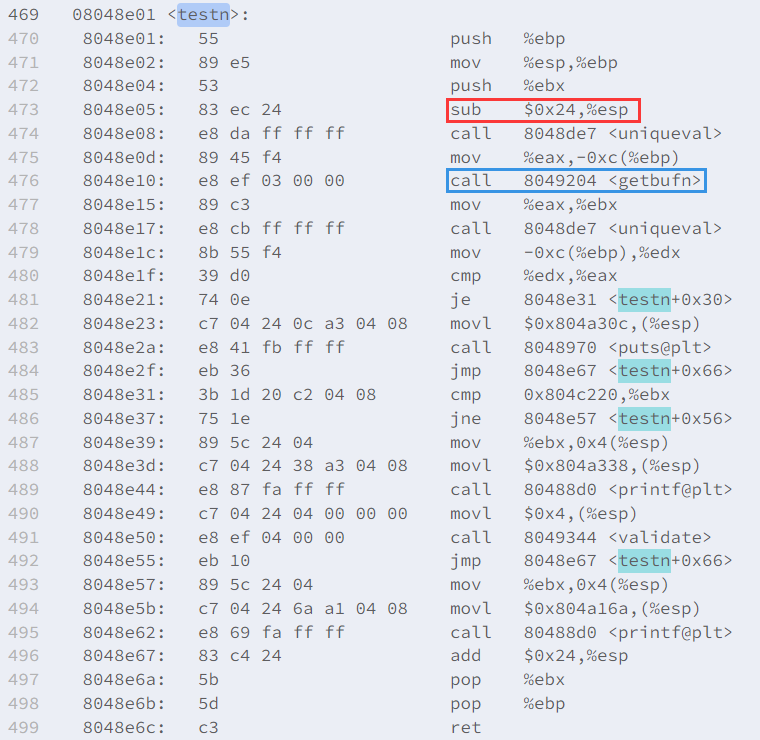


图 3. 28 在反汇编代码中定位到testn函数

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。

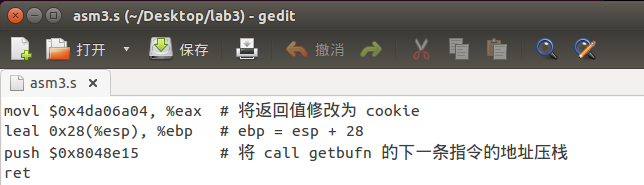


图 3. 29 恶意代码asm3.s

转换为机器码如下。

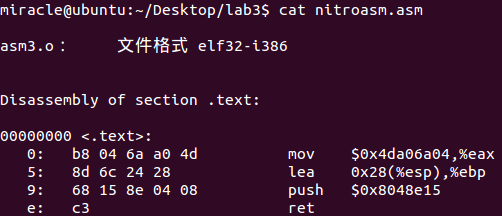


图 3. 30 cat查看恶意代码的机器码

回到getbufn函数分析。

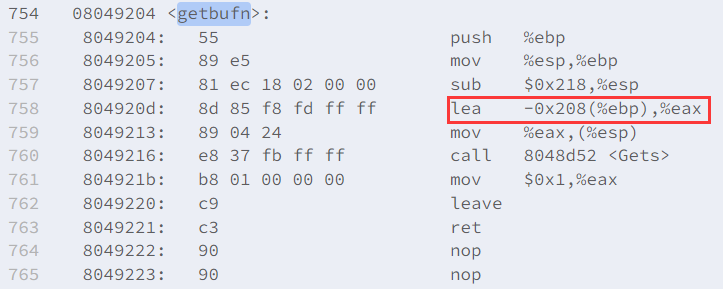


图 3. 31 getbufn函数反汇编代码

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。



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。



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）

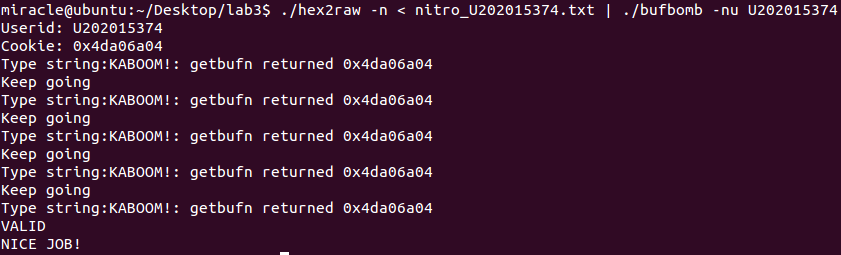


图 3. 34 实施nitro攻击

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。