

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： 计算机7班**

**学 号： U202215561**

**姓 名： 瞿明睿**

**指导教师： 张宇**

**报告日期： 2024年 5 月 21 日**

**计算机科学与技术学院**

**目录**

[实验内容及报告概述 1](#_Toc167432727)

[**1.1** **实验概述** 1](#_Toc167432728)

[实验2： Binary Bomb 实验 2](#_Toc167432729)

[2.1 实验概述 2](#_Toc167432730)

[2.1.1 实验目标 2](#_Toc167432731)

[2.1.2 实验要求和环境 2](#_Toc167432732)

[2.2 实验内容 2](#_Toc167432733)

[2.2.1 阶段1 字符串比较 3](#_Toc167432734)

[2.2.2 阶段2 循环 5](#_Toc167432735)

[2.2.3 阶段3 条件/分支 8](#_Toc167432736)

[2.2.4 阶段4递归调用和栈 12](#_Toc167432737)

[2.2.5 阶段5 指针 16](#_Toc167432738)

[2.2.6 阶段6 链表结构 19](#_Toc167432739)

[2.3 实验小结 22](#_Toc167432740)

[实验3：Buffer Overflow Attack 缓冲区溢出攻击 23](#_Toc167432741)

[3.1 实验概述 23](#_Toc167432742)

[3.2 实验内容 23](#_Toc167432743)

[3.2.1 阶段1 Smoke 23](#_Toc167432744)

[3.2.2 阶段2 fizz 25](#_Toc167432745)

[3.2.3 阶段3 Bang 27](#_Toc167432746)

[3.2.4 阶段4 boom 30](#_Toc167432747)

[3.2.5 阶段5 Nitro 33](#_Toc167432748)

[3.3实验小结 38](#_Toc167432749)

[实验总结 40](#_Toc167432750)

# 

# 实验内容及报告概述

本实验总共分为两个实验：Binary Bomb 实验和缓冲区溢出攻击。

第一个实验中，你要使用课程所学知识拆除一个“binary bombs”来增强对程序 的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。 一个“binary bombs”（二进制炸弹，下文将简称为炸弹）是一个Linux可执 行C程序，包含了6个阶段（phase1~phase6）。炸弹运行的每个阶段要求你输入 一个特定的字符串，若你的输入符合程序预期的输入，该阶段的炸弹就被“拆除”， 否则炸弹“爆炸”并打印输出 "BOOM!!!"字样。实验的目标是拆除尽可能多的炸弹层次。

第二个实验目的在于加深对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。

要求我们熟练使用gdb调试器和objdump：gdb调试器是我们主要的调试工具，帮助我们逐步跟踪和分析程序的执行过程。objdump用于查看程序的汇编代码，帮助我们理解程序的内部结构和逻辑。单步跟踪调试每一阶段的机器代码：通过gdb的单步调试功能，可以逐条执行汇编指令，观察程序的行为和状态变化。理解汇编语言代码的行为或作用，理解每条汇编指令的功能及其对程序状态的影响，是拆除炸弹的关键。“推断”拆除炸弹所需的目标字符串：通过分析和推断程序逻辑，找到每个阶段正确的输入字符串。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：在关键位置设置断点，有助于我们在程序执行到特定位置时暂停，以便仔细分析程序状态。

实验环境： C语言，at&t汇编语言，64位Ubuntu 22.03.4LTS (Windows虚拟机)，GCC，GDB

# Binary Bomb 实验

## 2.1 实验概述

### 2.1.1 实验目标

增强对程序机器级表示、汇编语言、调试器和逆向工程等理解。

### 2.1.2 实验要求和环境

熟练使用gdb调试器和objdump：gdb调试器是我们主要的调试工具，帮助我们逐步跟踪和分析程序的执行过程。objdump用于查看程序的汇编代码，帮助我们理解程序的内部结构和逻辑。单步跟踪调试每一阶段的机器代码：通过gdb的单步调试功能，可以逐条执行汇编指令，观察程序的行为和状态变化。

理解汇编语言代码的行为或作用：理解每条汇编指令的功能及其对程序状态的影响，是拆除炸弹的关键。“推断”拆除炸弹所需的目标字符串：通过分析和推断程序逻辑，找到每个阶段正确的输入字符串。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：在关键位置设置断点，有助于我们在程序执行到特定位置时暂停，以便仔细分析程序状态。

实验环境： C语言，at&t汇编语言，64位Ubuntu 22.03.4LTS (Windows虚拟机)

# 2.2 实验内容

本次实验提供了一个名为“Binary Bombs”（二进制炸弹，简称炸弹）的Linux可执行C程序。该程序包含从phase1到phase6共六个阶段。每个阶段要求输入一个特定的字符串，如果输入符合程序的预期，该阶段的炸弹就被“拆除”；否则，炸弹就会“爆炸”。实验的目标是拆除尽可能多的炸弹。各个阶段分别考察了机器级语言程序的不同方面，难度逐渐递增：

阶段1：字符串比较

该阶段要求输入一个特定的字符串，并通过字符串比较来验证输入是否正确。通过分析汇编代码，可以找到正确的字符串，从而顺利拆除炸弹。

阶段2：循环

这一阶段涉及到循环结构。输入的字符串需要经过特定的循环处理和验证。理解循环的终止条件和内部逻辑是拆除这个阶段炸弹的关键。

阶段3：条件/分支

在这个阶段，程序使用了条件语句和分支结构，包括switch语句。通过分析汇编代码，可以确定每个分支的逻辑，从而找到正确的输入。

阶段4：递归调用和栈

该阶段考察了递归调用和栈的使用。理解递归函数的调用过程和栈帧结构是顺利拆除这个阶段炸弹的关键。

阶段5：指针

这一阶段涉及到指针的操作。通过分析指针的使用和操作，可以找到正确的输入字符串。

阶段6：链表/指针/结构最后一个阶段综合考察了链表、指针和结构体。通过理解链表的结构和指针操作，能够找到正确的输入，拆除最终阶段的炸弹。

此外，在第4阶段之后，还存在一个隐藏阶段。在输入特定字符串后，该隐藏阶段会被激活，需要进一步拆除。

### 2.2.1 阶段1 字符串比较

1.任务描述

要破解bomb程序并拆除炸弹，首先需要分析其phase\_1函数 ，phase\_1函数对输入字符串进行某种验证。使用Ghidra或其他工具的反汇编/反编译功能来查看phase\_1的逻辑。使用找到的答案字符串作为输入，验证并通过phase\_1

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码，找到答案字符串的存储地址；进一步根据地址，使用gdb调试工具，从程序对应地址中找到答案字符串，从而完成任务。

3.实验过程

（1）使用objdump进行反汇编，通过调用objdump –d bomb > 561.txt命令，对bomb程序进行反汇编，并将汇编代码输出到561.txt中。并且更名为561.S方便使用Vscode分析。

（2）查看汇编源代码561.S文件，在main函数中找到调用phase\_1函数的位置，从而获取到phase\_1函数的地址为0x8048a47，如图2.1所示。

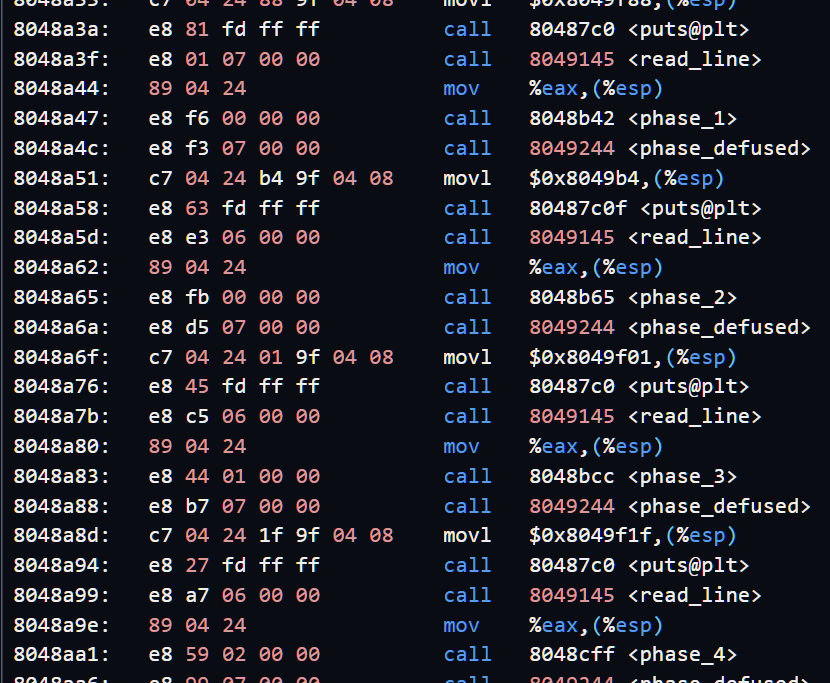


图2.1 main函数缩略图

（3）通过地址我们找到phase\_1函数，我们需要分析其结构找到对应的存储字符串的地址。

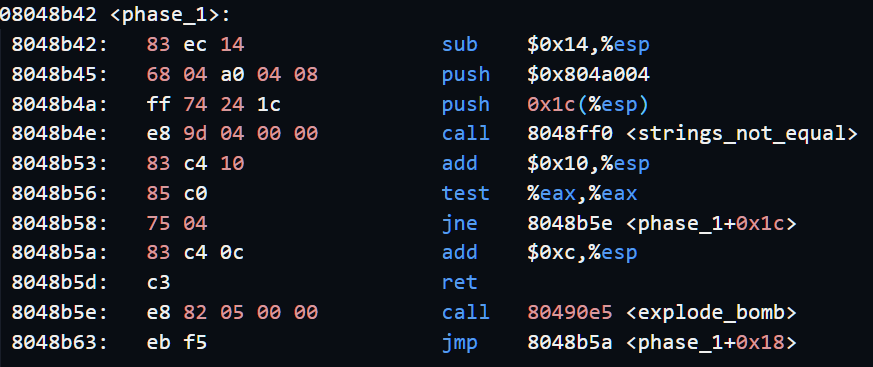


图2.2 phase\_1函数

（4）通过阅读，我们可以了解到：这是一个简单的函数，它的主要目的是比较两个字符串是否相等。如果不相等，它将调用explode\_bomb函数。将地址0x804a004压入栈中。这个地址可能是一个字符串的地址将栈上的另一个值压入栈中。这个值可能是另一个字符串的地址调用strings\_not\_equal函数，该函数可能会比较前面压入栈中的两个字符串。如果两个字符串不相等，调用 explode\_bomb 函数。如果相等则返回正常值，继续主函数。

（5）使用gdb调试工具使用gdb调试工具，通过执行gdb bomb命令，对bomb程序进行调试，并查找程序中0x804a004地址所存储的字符串内容，如图所示

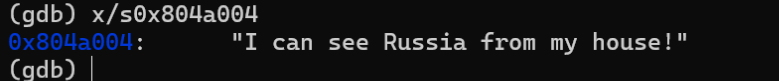


图2.3 phase\_1字符串

4.实验结果：

在bomb程序的输入获取到的字符串即可成功解决阶段一炸弹，如图2.4

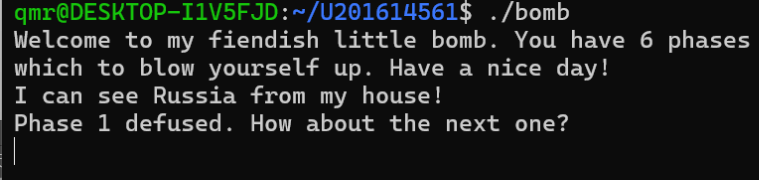


图2.4 bomb阶段一成功解除

5.反编译工具分析

借助IDA Pro可以对汇编程序进行相分析，结构如下图



图2.5 反编译分析

可以发现这个函数结构非常简单，只是一个简单的比较。如果比对时一致的，那么应该是返回主函数，如果比较失败，那么应该引爆炸弹。

### 2.2.2 阶段2 循环

1.任务描述

要破解bomb程序并拆除炸弹，首先需要分析其phase\_2函数 ，phase\_2函数对输入字符串进行某种验证。使用Ghidra或其他工具的反汇编/反编译功能来查看phase\_2的逻辑。使用找到的答案字符串作为输入，验证并通过phase\_2。

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码，找到答案字符串的存储地址；进一步根据地址，使用gdb调试工具，从程序对应地址中找到答案字符串，从而完成任务。

3.实验过程：

（1）查看汇编源代码561.S文件，在main函数中找到调用phase\_2函数的位置，从而获取到phase\_2函数的地址，并通过其找到phase\_2函数，查看phase\_2函数的内容，如图2.6所示。



图2.6 phase\_2函数代码

（2）通过观察phase\_2函数内容，发现其首先调用了read\_six\_numbers函数，内容如图2.6所示。通过阅读代码发现，该函数读取用户输入的最多6个整数，并将读入的数字个数存入esp指向内容的最低位，再从esp所指内容由低至高存入用户输入的整数。

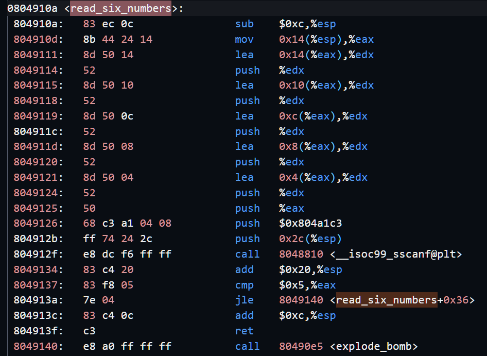


图2.7 read\_six\_numbers函数

而phase\_2函数调用read\_six\_numbers函数后，先通过指令cmpl $0x0,0x4(%esp)判断esp所指向内容最低位即第一个输入数字是否大于0，如果不是则引爆炸弹，反之则继续执行程序，后进入循环程序。

（3）程序将mov $0x1,%ebx 设置一个循环计数器i为 1。add (%esp,%ebx,4),%eax和cmp %eax,0x4(%esp,%ebx,4)是一个循环，检查每个数字是否等于前一个数字加上循环计数器。如果不是，它将调用explode\_bomb。add $0x1,%ebx和cmp $0x6,%ebx是循环的结束部分，它将循环计数器增加 1，并检查是否已经处理了所有六个数字。因此，我们输入的数字的逻辑是，我输入第一个数字n，下一个数字应该是n+i，这里i就是循环计数器，也就是第i个数字。作为例子，如果我输入的第一个数字是1，那么我第二应该是，1+1（第一个数+第一个数的第几个位数）相应的，第三个应该是（1+1）+2（1+1为第二个数，2为其位数）

4.实验结果：

根据上述分析，使用1 2 4 7 11 16作为bomb程序phase\_2的输入，成功解除炸弹，如图2.8所示。

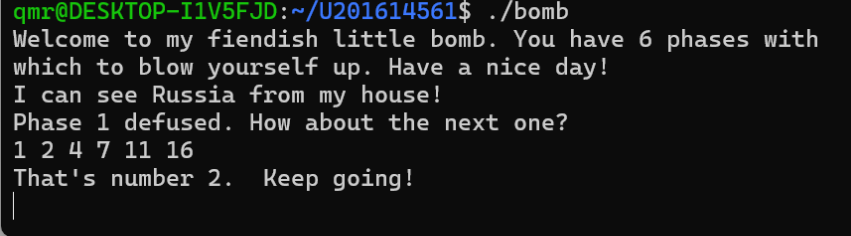


图2.8 bomb二阶段解除

5. 反编译工具分析

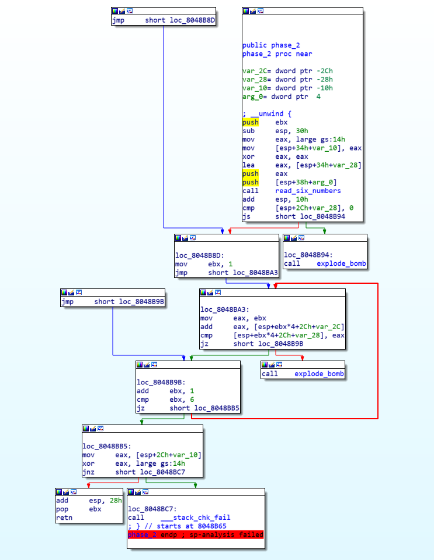


图2.9 phase\_2反编译分析

可以看到在这里ida pro给出了这个函数的关键，即指回loc\_8048BA3:的红色箭头，非常明显的循环结构。

### 2.2.3 阶段3 条件/分支

1.任务描述

要破解bomb程序并拆除炸弹，首先需要分析其phase\_3函数 ，phase\_3函数对输入字符串进行某种验证。使用Ghidra或其他工具的反汇编/反编译功能来查看phase\_3的逻辑。使用找到的答案字符串作为输入，验证并通过phase\_3。

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码，找到答案字符串的存储地址；进一步根据地址，使用gdb调试工具，从程序对应地址中找到答案字符串，从而完成任务。

3.实验过程：

（1）查看汇编源代码561.S文件，在main函数中找到调用phase\_3函数的位置，从而获取到phase\_3函数的地址，并通过其找到phase\_3函数，查看phase\_3函数的内容，如图2.9所示。

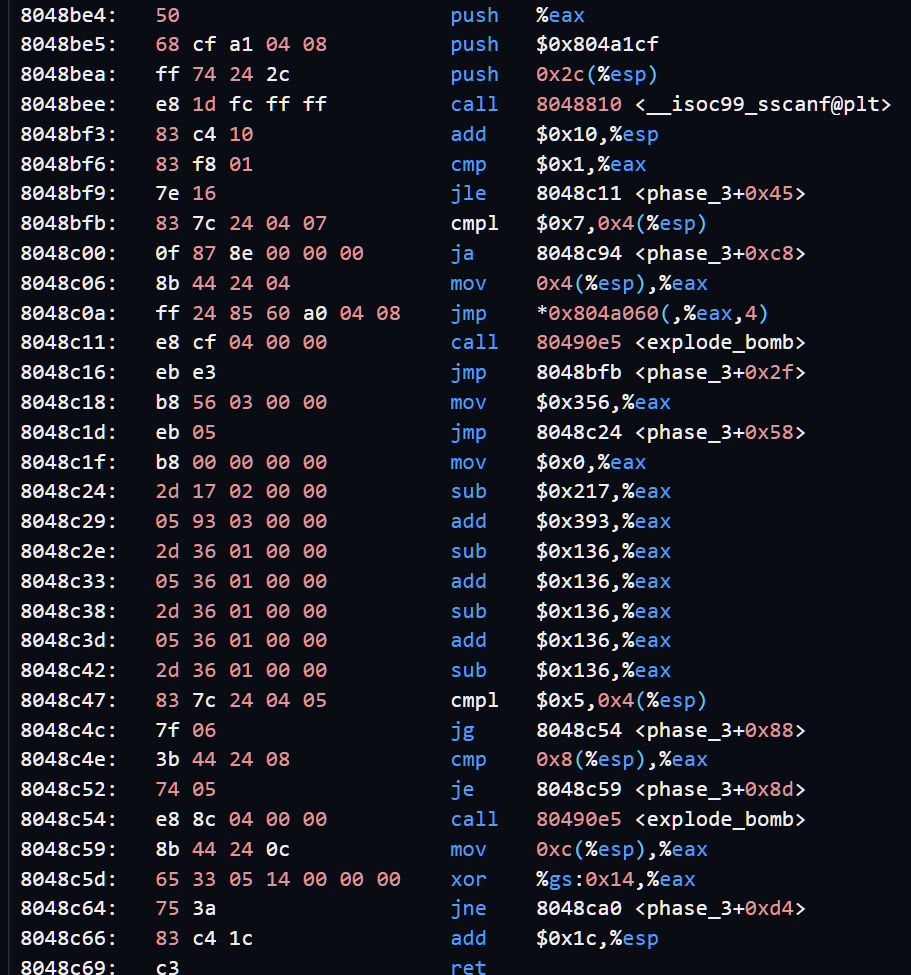


图2.9 phase\_3函数代码（部分）

（2）观察到phase\_3函数调用了scanf函数进行输入的读取，传递给scanf函数的数据地址为0x804a1ef，使用gdb调试工具获取到对应地址存储信息内容为"%d %d"，如图2.10所示，



图2.10 0x804a1ef地址内容示意图

即scanf读取了两个整数输入，并判断读入的整数个数是否为2，若不是，则调用explode\_bomb函数引爆炸弹。在读取输入后， phase\_3函数执行了cmpl $0x7,0x4(%esp)来比较第一个输入数字是否大于7，若大于则执行引爆炸弹程序。这里我们可知道第一个数字有非常明显的范围要求。

（3）phase\_3函数将读入的第一个整数a存入eax后，调用jmp \*0x804a080(,%eax,4)，跳转到0x804a080+a\*4处，利用gdb查看到0x804a060，如图2.11所示。

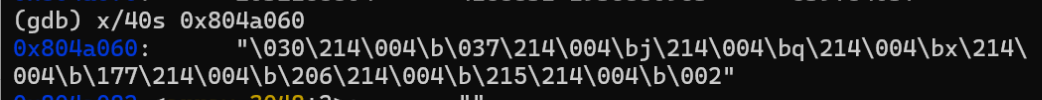


图2.11 地址内容解析

可以发现读取出的地址都为小端存储，我们需要转化才能正常读懂，经过转化这个字节序列表示的地址如下表2.12

|  |  |
| --- | --- |
| 地址内容 | 具体地址 |
| \030\214\004\b  \037\214\004\b  j\214\004\b  q\214\004\b  x\214\004\b  \177\214\004\b  \206\214\004\b  \215\214\004\b | 0x08048c18  0x08048c1f  0x08048c6a  0x08048C71  0x08048c78  0x08048c7f  0x08048c86  0x08048c8d |

表2.12 小端地址转化

（4）通过观察发现，phase\_3函数此后代码执行switch语句功能，按照读入的第一个整数跳转指定位置，而越前位置的代码执行量也越多，都是对寄存器eax进行一系列加减操作。为了简化计算，我们假设第一个输入值为0，程序会跳转到8048c18，然后执行以下操作:1.mov $0x356,%eax:将%eax寄存器的值设置为 0x356。2.sub $ox217,%eax:从%eax寄存器的值中减去 0x217。结果是0x356-0x217=0x13f。3.add $ox393,%eax:向%eax寄存器的值中加上0x393。结果是0x13f+0x393=0x4d2。4.sub $ox136,%eax:从%eax寄存器的值中减去 0x136。结果是0x4d2-0x136=0x39c。5.add $0x136,%eax:向%eax寄存器的值中加上 0x136。结果是0x39c+0x136=0x4d2。6.sub $0x136,%eax:从%eax寄存器的值中减去 0x136。结果是0x4d2-0x136=0x39c。7. add $0x136,%eax:向%eax寄存器的值中加上0x136。结果是0x39c+0x136=0x4d2。8.sub $0x136,%eax :从%eax寄存器的值中减去 0x136。结果是0x4d2-0x136=0x39c(进制的924)然后，程序会检查你的第二个输入值是否等于%eax的值(也就是924)。如果不等，程序会调用 explode bomb.因此，当我们你的第一个输入值是0，那么第二个输入值应该是924，才能避免触发explode bomb。

4.实验结果：

由上述分析可得，输入“0 924”即可解除phase\_3阶段的炸弹，如图2.13所示。（通过分析我们可以知道第一个数字必须在0和7之间，所以这个炸弹答案并不唯一）

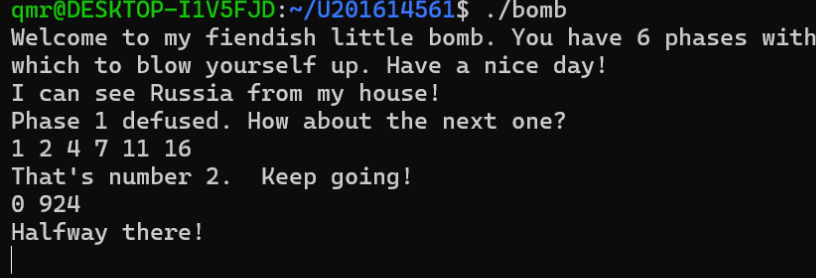


图2.13 bomb三阶段解除

5. 反编译工具分析

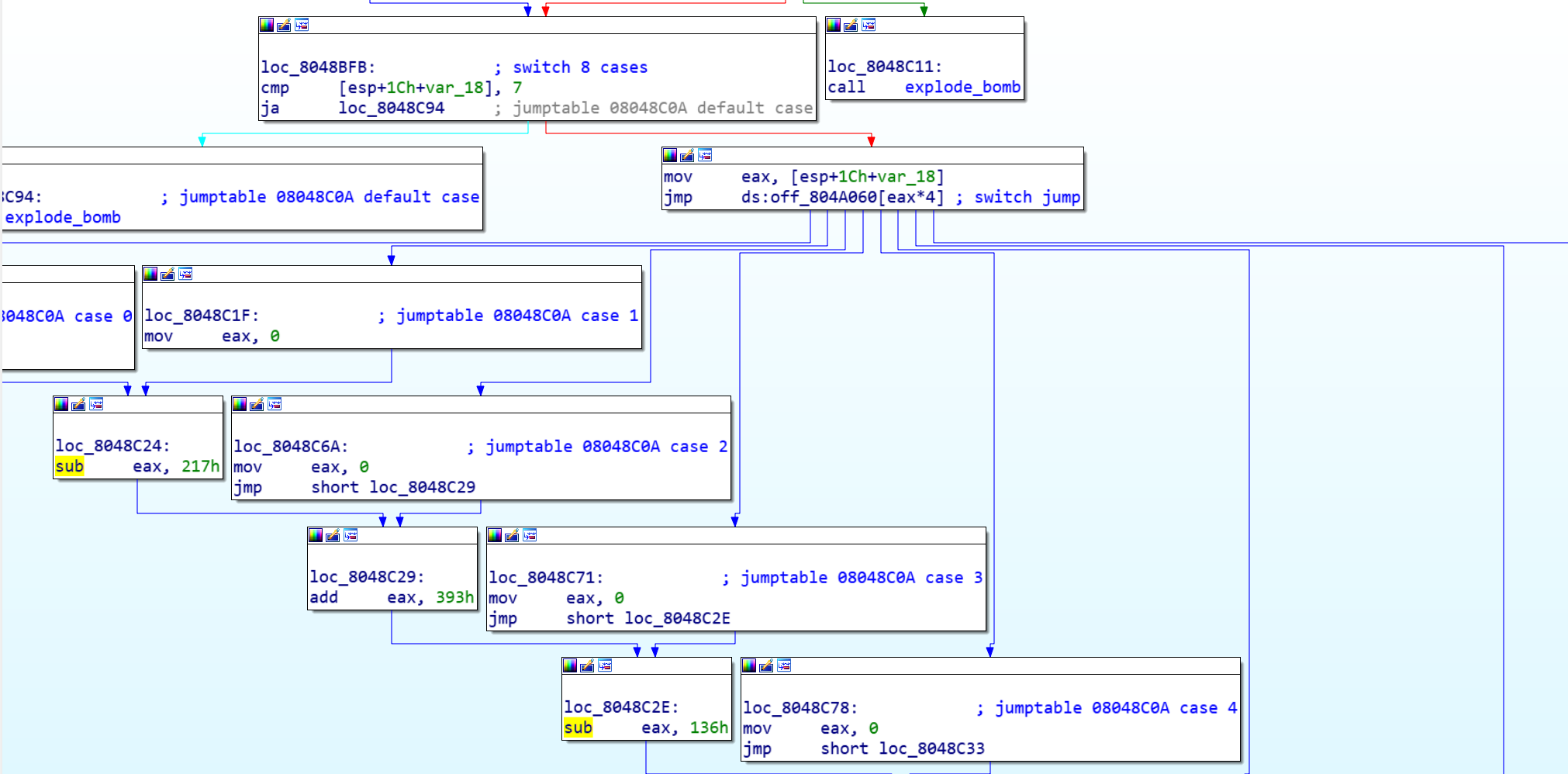


图2.12 反编译工具分析

从结构的角度可以看到，这个函数是非常明显的具有分支的结构，既符合switch函数的功能，跳转到不同的地址处进行计算，这也是为什么我说不同的答案，如果我第一个输入的数字为1，那么就应该以第二个解析地址开始计算，计算步数会减少

### 2.2.4 阶段4递归调用和栈

1.任务描述

要破解bomb程序并拆除炸弹，首先需要分析其phase\_4函数 ，phase\_4函数对输入字符串进行某种验证。使用Ghidra或其他工具的反汇编/反编译功能来查看phase\_4的逻辑。使用找到的答案字符串作为输入，验证并通过phase\_4。

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码，找到答案字符串的存储地址；进一步根据地址，使用gdb调试工具，从程序对应地址中找到答案字符串，从而完成任务。

3.实验过程：

（1）查看汇编源代码561.S文件，在main函数中找到调用phase\_24函数的位置，从而获取到phase\_4函数的地址，并通过其找到phase\_4函数，查看phase\_4函数的内容，如图2.14所示。

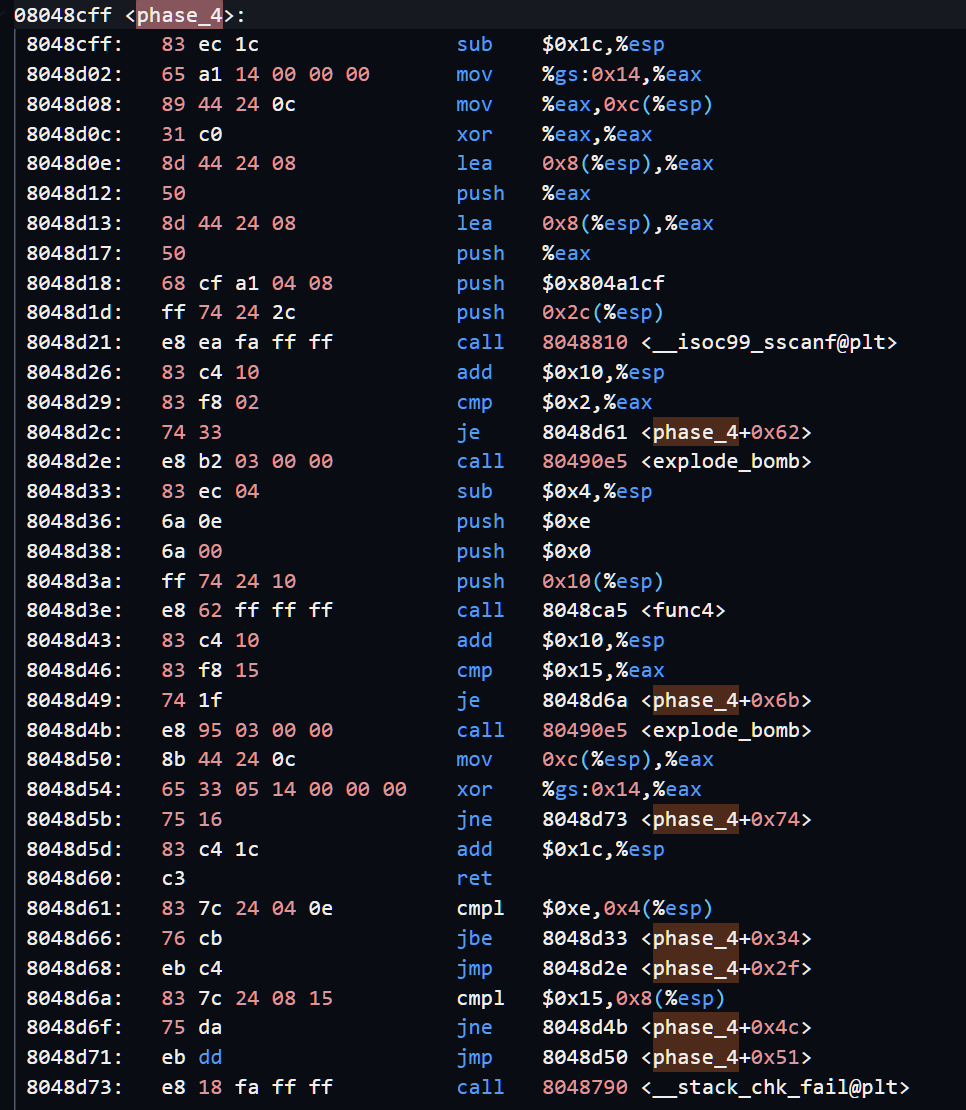


图2.14 phase\_4函数图

（2）观察到phase\_4函数同样调用了scanf函数，也读取了输入的两个整数，再执行cmp $0x2,%eax检查是否读取了两个数字。如果没有，它将调用explode\_bomb，随后即比较读入的第一个整数与14的大小，这里则是要求我们输入的第一个数字小于等于14。随后程序准备好三个参数调用func4函数，查看到func4函数内容如图2.14所示。通过分析程序，我们可以知道phase\_4先后将0xe，0x0，读入的第一个整数a这三个参数压栈，传递给func4。在最后程序调用了cmpl $0x15,0x8(%esp)检查第二个数字是否不等于 21。如果是，它将调用explode\_bomb。



图2.15 func4函数（部分）

（3）从这段汇编代码中，我们可以看到 func4 函数是一个递归函数，它在 phase 4函数中被调用。在 phase\_4函数中，func4函数的参数是输入的第一个数，0和14。func4 函数的工作方式是，首先计算14和0的差值，然后将结果除以2(向下取整)，并将结果加到0上，得到的结果存储在 %ebx 中。然后，%ebx 和输入的第一个数进行比较。如果 %ebx 大于输入的第一个数，那么 func4 函数会递归调用自己，其中输入的第一个数不变，0不变，14变为 %ebx减1。如果%ebx小于输入的第一个数，那么 func4 函数会递归调用自己，其中输入的第一个数不变，0变为 %ebx 加1，14不变。如果 %ebx等于输入的第一个数，那么 func4 函数返回 %ebx。而且在phase\_4函数中容易发现func4 函数的返回值必须等于0x15。因此，我们需要找到一个在0到14之间的数，使得 func4 函数返回21。由于 func4 函数是个二分查找算法，我们可以通过二分查找的方式来找到这个数。我们可以从中间的数7开始，这里设计了一个python代码来寻找这个数，如图2.16所示

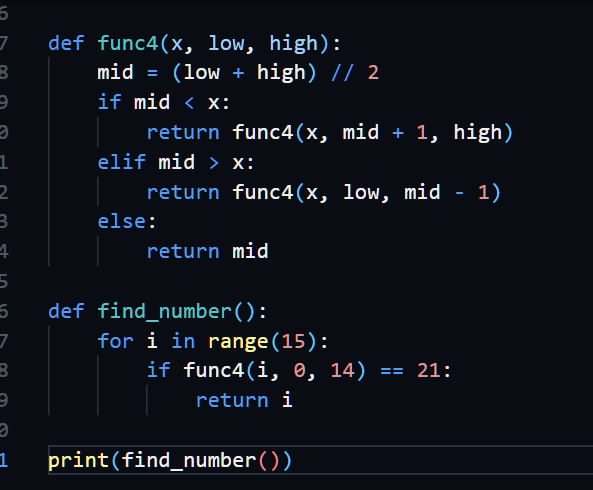


图2.16 暴力破解func4函数程序

通过计算可以发现，6是符合要求的，再加上已经了解第二个数必须为21，得出答案。

4.实验结果：

由上述分析，输入“6 21”即可解除phase\_4阶段的炸弹，如图2.17所示。

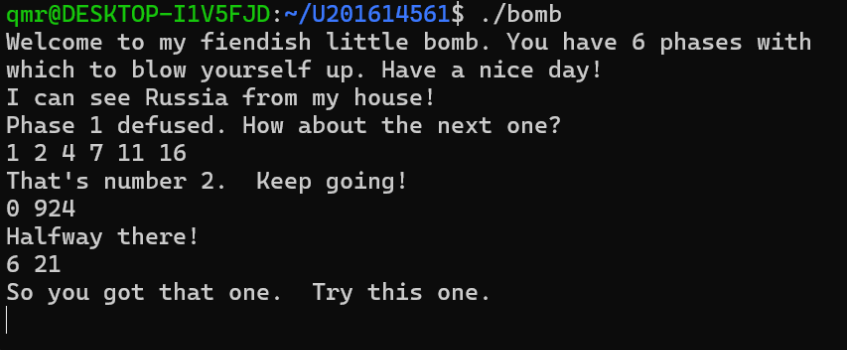


图2.17 bomb 四阶段解除

5. 反编译工具分析

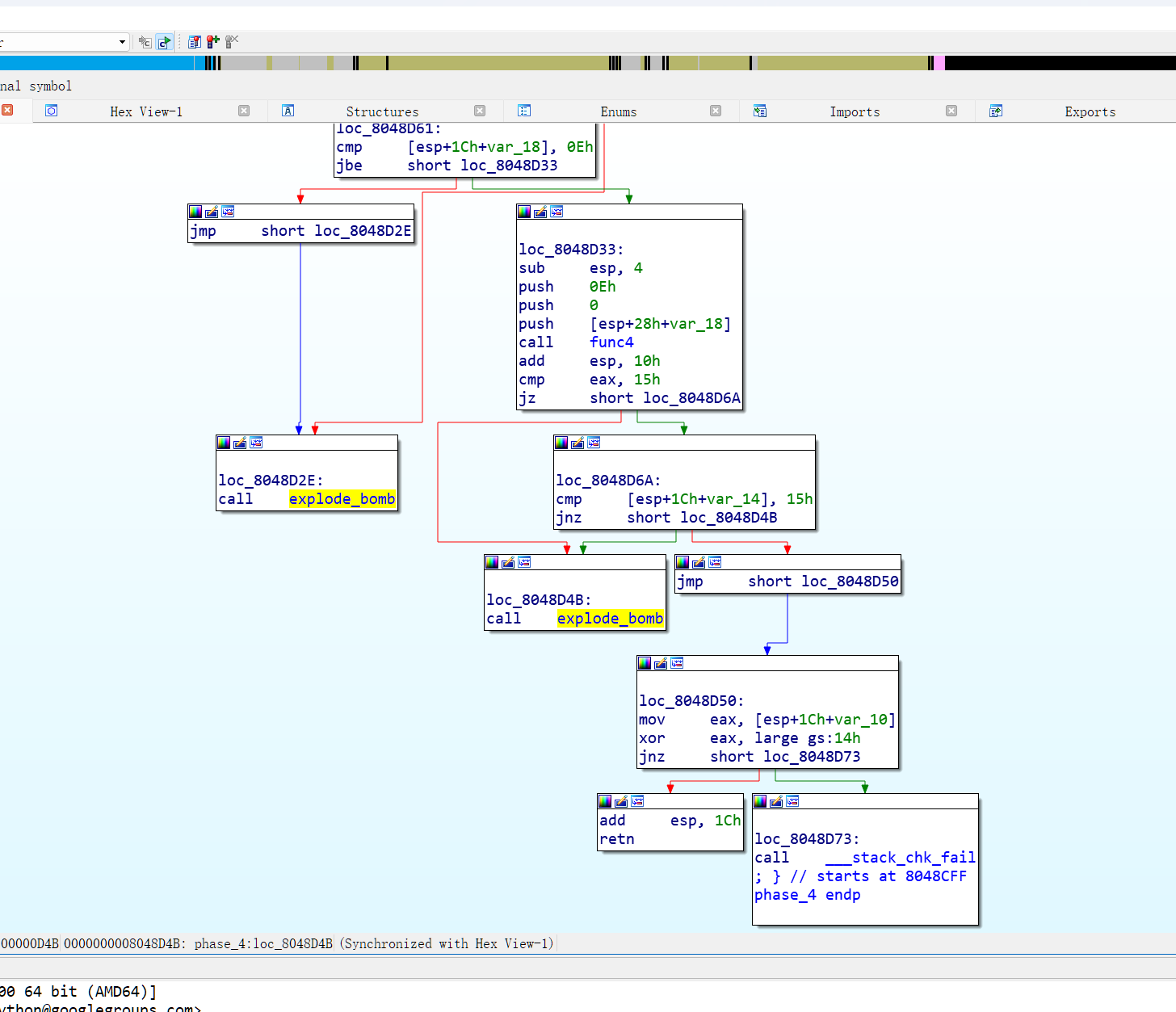


图2.18 反编译工具分析图

这里可以发现，反编译工具对函数的结构要求有很清楚的调用，这里主要分析了计算的栈中，首先scanf是否读取到值，返回的值是否符合要求，函数调用的数据最后的返回值是否等于21，最后就是第二个数值的要求。

### 2.2.5 阶段5 指针

1.任务描述

要破解bomb程序并拆除炸弹，首先需要分析其phase\_5函数 ，phase\_5函数对输入字符串进行某种验证。使用Ghidra或其他工具的反汇编/反编译功能来查看phase\_5的逻辑。使用找到的答案字符串作为输入，验证并通过phase\_5。

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码，找到答案字符串的存储地址；进一步根据地址，使用gdb调试工具，从程序对应地址中找到答案字符串，从而完成任务。

3.实验过程：

（1）查看汇编源代码561.S文件，在main函数中找到调用phase\_5函数的位置，从而获取到phase\_2函数的地址，并通过其找到phase\_5函数，查看phase\_5函数的内容，如图2.5所示。

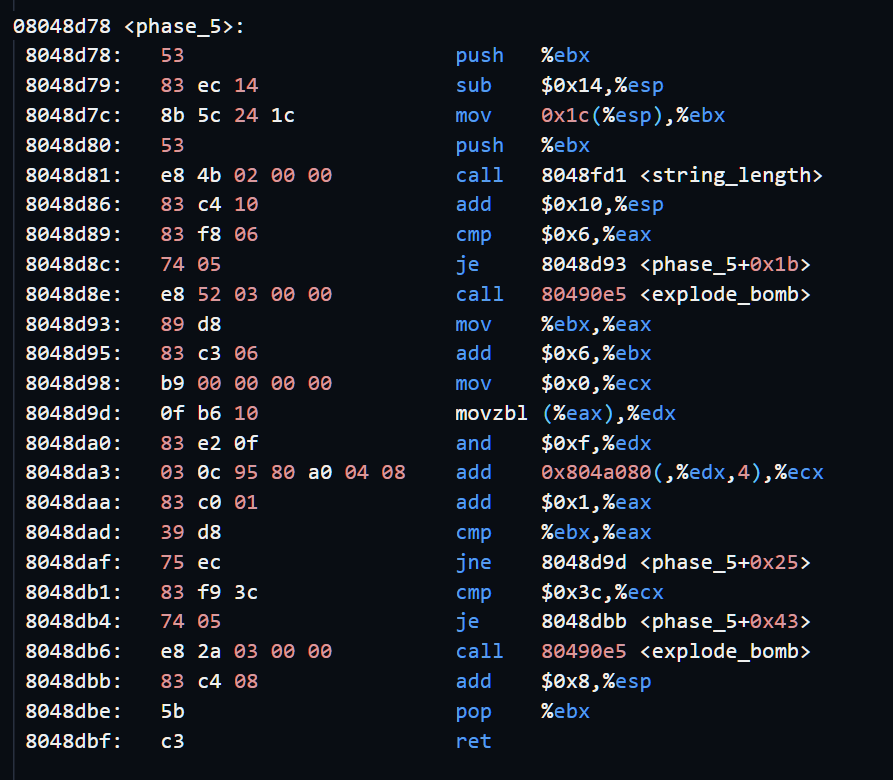


图2.19 phase\_5函数

（2）观察到phase\_5函数调用了string\_length函数，这个哈数通过寄存器设置0开始计数，通过读取栈上字符串的地址，逐个字符检测null，检测到null则返回寄存器的值



图2.20 string\_length函数

再回到主函数，这里函数使用了调用函数的返回值cmp $0x6,%eax，检查字符串的长度是否为6。如果不是，它将调用explode\_bomb。随后函数将对字符串进行循环处理将遍历字符串的每个字符。

mov $0x0,%ecx是初始化一个计数器，该计数器将用于计算字符串中每个字符的某个属性的总和。and $0xf,%edx指出，我们将遍历字符串的每个字符是为了计算每个字符的低4位，并将结果添加到计数器中。cmp $0x3c,%ecx检查计数器的值是否等于60。如果不是，它将调用explode\_bomb，考虑到低四位一共16位，所以我们在查询权值表的时候需要了解到16个值



图2.21 地址查询结果

我们在地址中找到了权值，那么我们需要找到这样的字符串符合要求，在这里我设计了一个python代码去寻找字母和数字的六位组合。

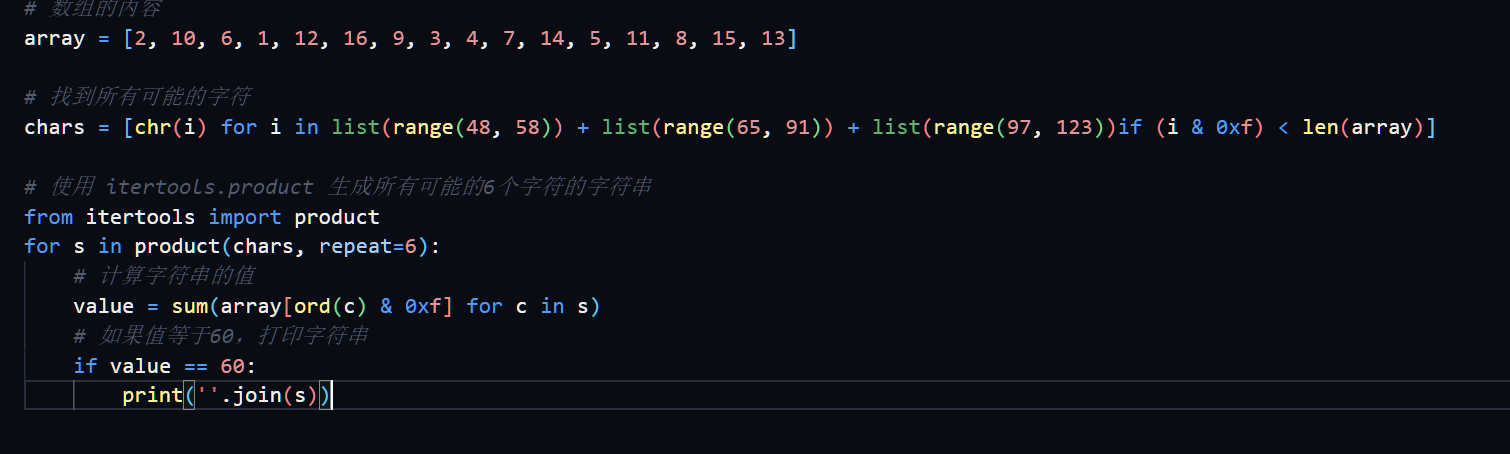


图2.22 python寻找字符串代码示意图

由程序的寻找结果可以知道，答案字符串不止一个。

4.实验结果：

由上述分析可得，输入“001EUZ”即可解除phase\_5阶段的炸弹，如图2.23所示。

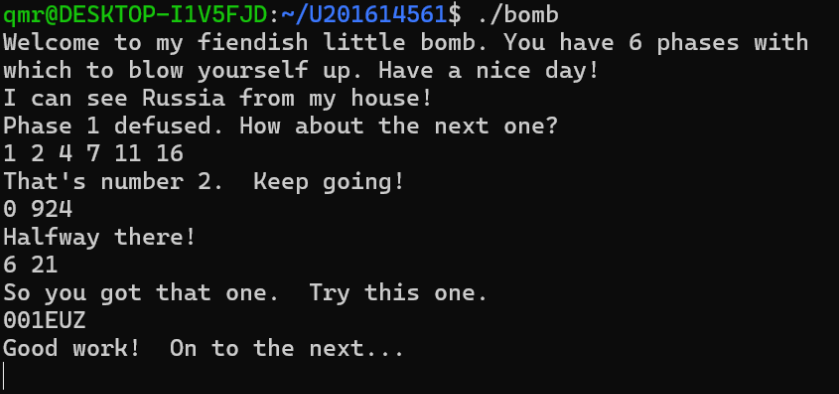


图2.23 阶段五解除

5. 反编译工具分析

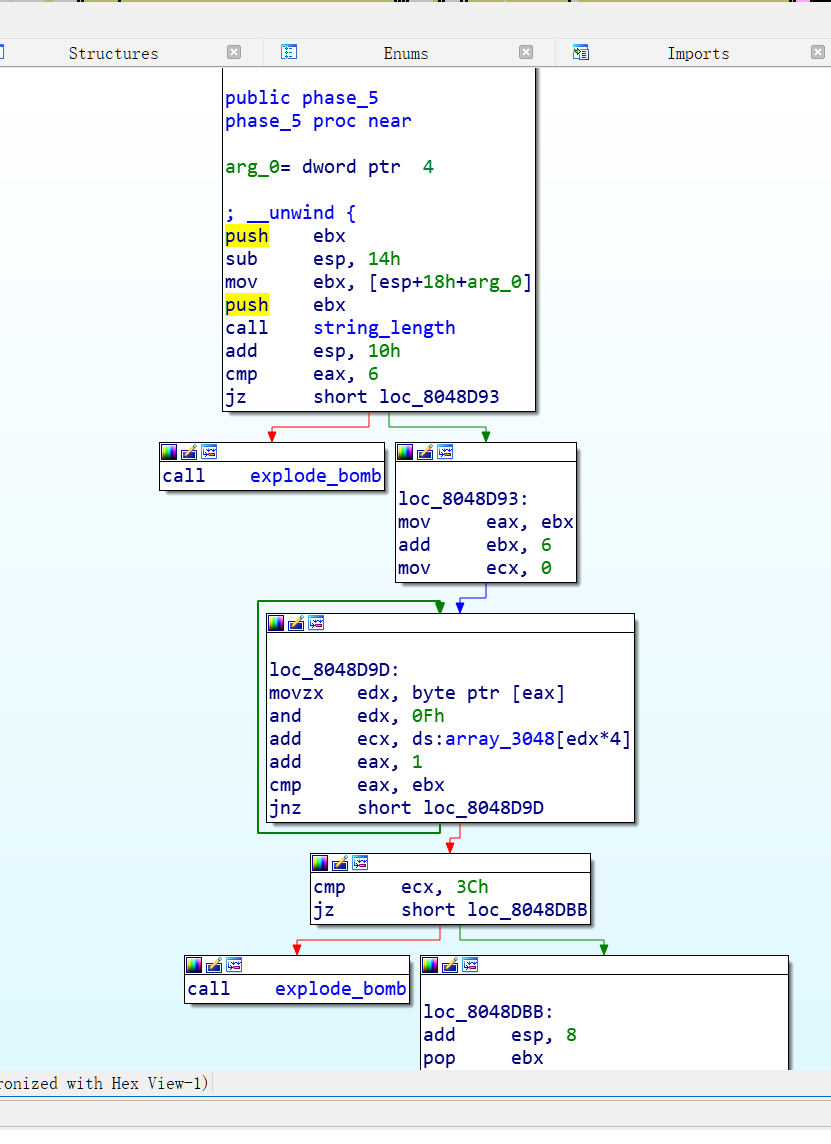


图2.24 反汇编分析

在这里很明显的，逆向工具指出了问题的关键在于loc-8048D9D处，add ecx, ds:array\_3048[edx\*4]，即我们需要的权值表最后在eax计数器到达制定的6后比较60，整体结构还是比较清楚的。

### 2.2.6 阶段6 链表结构

1.任务描述

要破解bomb程序并拆除炸弹，首先需要分析其phase\_6函数 ，phase\_6函数对输入字符串进行某种验证。使用Ghidra或其他工具的反汇编/反编译功能来查看phase\_6的逻辑。使用找到的答案字符串作为输入，验证并通过phase\_6

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码，找到答案字符串的存储地址；进一步根据地址，使用gdb调试工具，从程序对应地址中找到答案字符串，从而完成任务。

3.实验过程：

（1）查看汇编源代码561.S文件，在main函数中找到调用phase\_6函数的位置，从而获取到phase\_6函数的地址，并通过其找到phase\_6函数，查看phase\_6函数的内容，如图2.5所示。

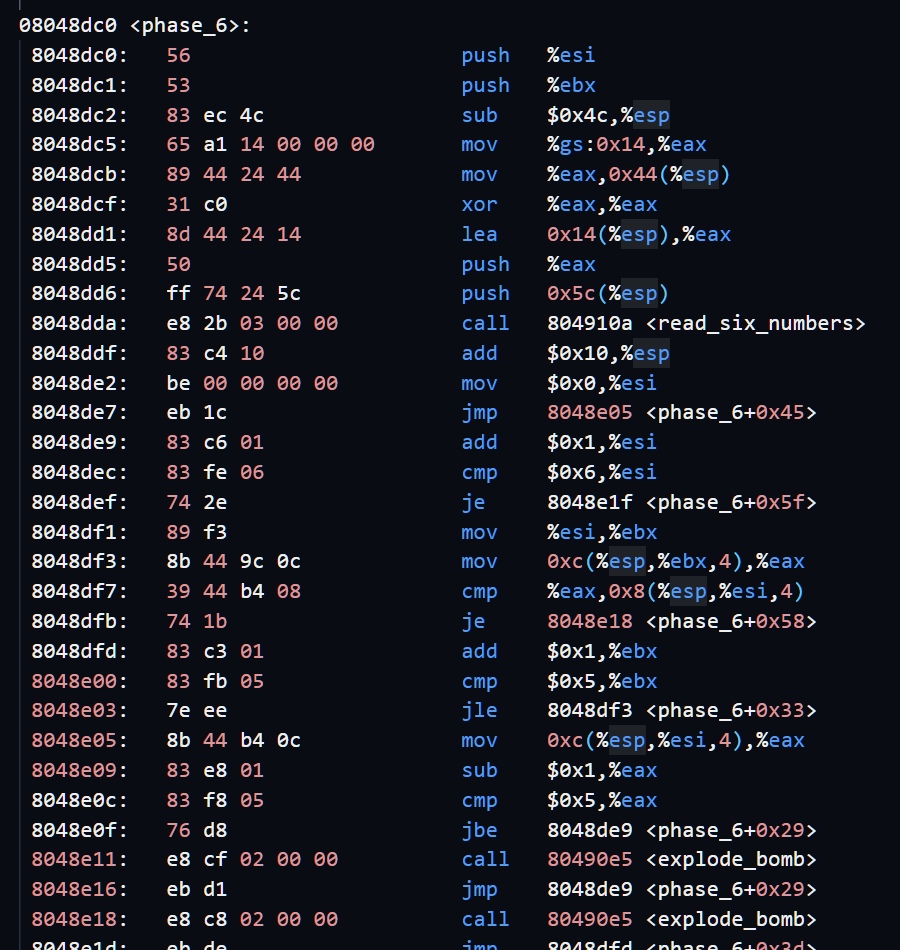


图2.25 phase\_6函数（部分）

（2）观察到phase\_6函数和phase\_2函数一样调用了read\_six\_numbers函数。在读入6个输入数字后，首先判断第1个输入数字减一后是否小于等于5，否则引爆炸弹，即第三个数字需小于等于6。

（3）代码实际上是对读取的数据进行了检验。在读取了6个数字之后，这里设置了两重循 环，外层循环枚举每一个读取的数字，然后进行内层循环。内层循环枚举当前外层循环读取的数字的 20 前面的所有数字，如果发现了重复，则炸弹爆炸。同时，这里还要求所有的数字都在1~6之间。

函数会根据输入的数字序列构造一个链表，然后检查这个链表是否按照特定的顺序排序。这个顺序是由一个固定的链表决定的，这个链表的节点值是：1, 2, 3, 4, 5, 6。然后，函数会检查输入的数字序列构造的链表是否与这个固定链表的某个排列相同。如果不同，函数会调用explode\_bomb

（4）详细的来说，具体过程应该是首先将%ebx寄存器设置为0，然后进入一个循环，循环次数为6次，每次循环都会根据输入的数字序列中的一个数字构造链表的一个节点。在每次循环中，代码首先从栈上的一个位置读取一个数字，然后将这个数字与1进行比较。如果数字大于1，代码就会从一个固定的链表中取出一个节点，然后将这个节点的地址存储在栈上的一个位置。然后，%ebx寄存器加1，开始下一次循环。这个过程会构造一个新的链表，这个链表的每个节点都是从一个固定的链表中取出的，取出的顺序由输入的数字序列决定。所以我们应该如何找到这样的1-6的数字全排列呢？

我们注意到在你给出的汇编代码中，固定链表的起始地址被硬编码在了这行代码中：

8048e2f:    ba 3c c1 04 08          mov    $0x804c13c,%edx

这行代码将地址0x804c13c移动到%edx寄存器中，这个地址就是固定链表的起始地址。我们使用gdb工具来看。

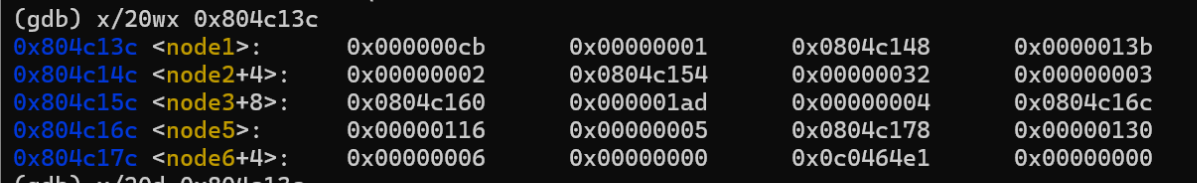


图2.26 地址查询结果

经过整理很容易得到下列表格

|  |  |  |  |
| --- | --- | --- | --- |
| 节点名称 | 数据值 | 下一个节点地址 | 索引值 |
| node1 | 0x000000cb | 0x0804c148 | 0x00000001 |
| node2 | 0x0000013b | 0x0804c154 | 0x00000002 |
| node3 | 0x00000032 | 0x0804c160 | 0x00000003 |
| node4 | 0x000001ad | 0x0804c16c | 0x00000004 |
| node5 | 0x00000116 | 0x0804c178 | 0x00000005 |
| node6 | 0x00000130 | N/A | 0x00000006 |

表2.27 链表节点表

在注意到我们取出的节点有明显要求：8048e88: 7e ef jle 8048e79 <phase\_6+0xb9>：如果当前节点的值小于或等于下一个节点的值，跳转到地址8048e79，否则继续执行。，意思几位我们需要找到升序节点，而我们节点中的值应该是升序的，在读取的角度来说应该是第一个输入的数字应该是最小的，那么通过对数据值排列，我们可以得到这样的序列3->1>5>6>2>4。

4.实验结果：

由上述分析可得，输入“3 1 5 6 2 4”即可解除phase\_6阶段的炸弹，如图2.28所示。

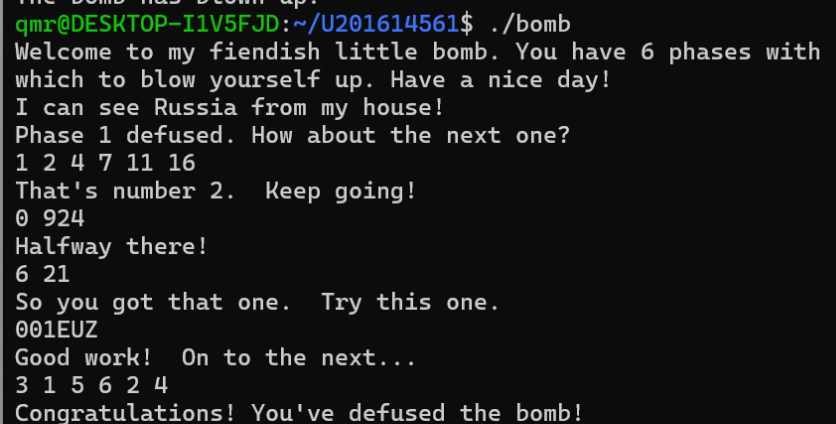


图2.28 阶段六解除

5. 反编译工具分析

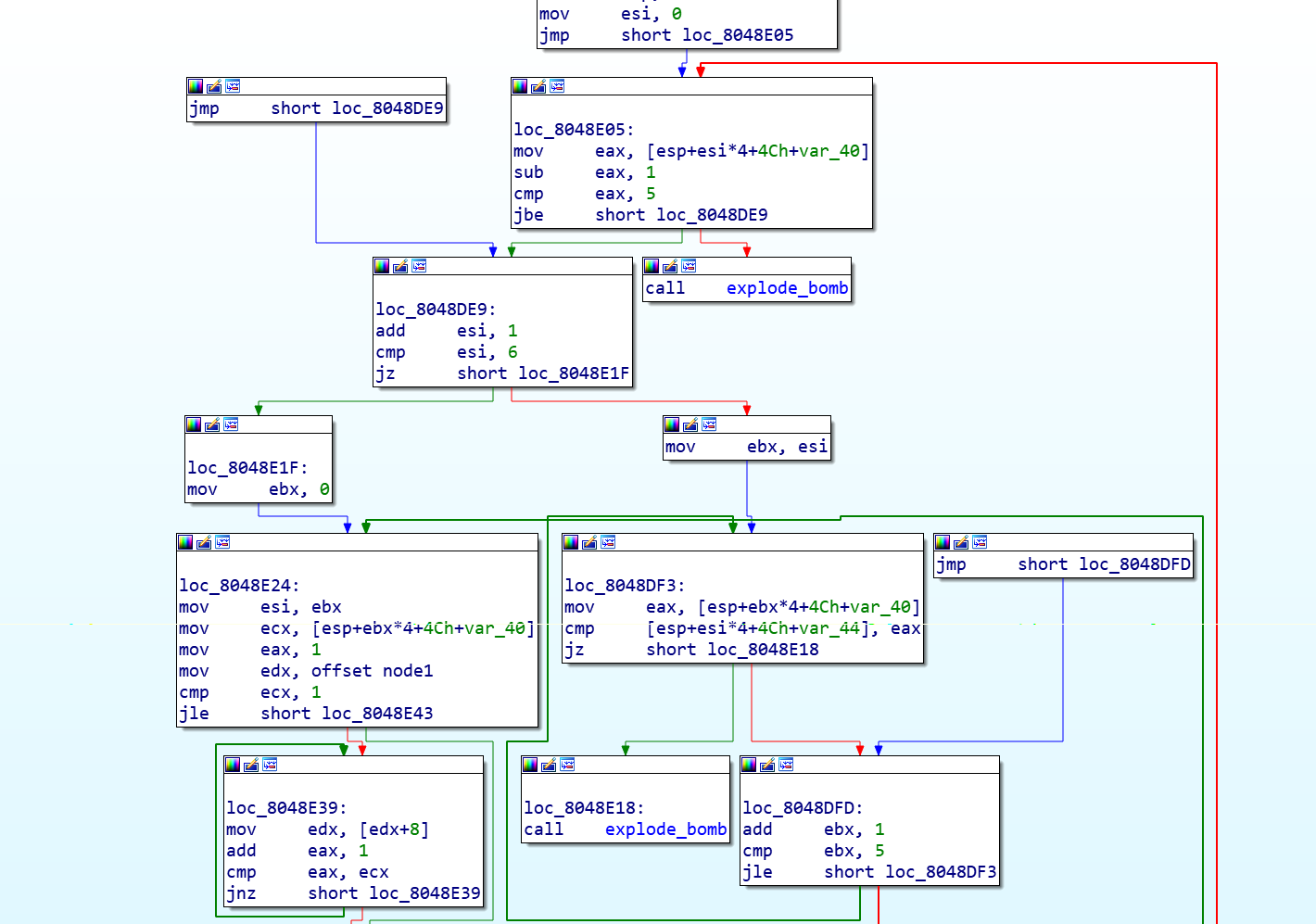


图2.12

首先就是循环部分读取我们的数据，可以看到有非常明显的比较数值部分即在loc\_8048DE9。此外就是我们的链表处理部分，可以发现每次读取值都需要寄存器累加重新确定地址。

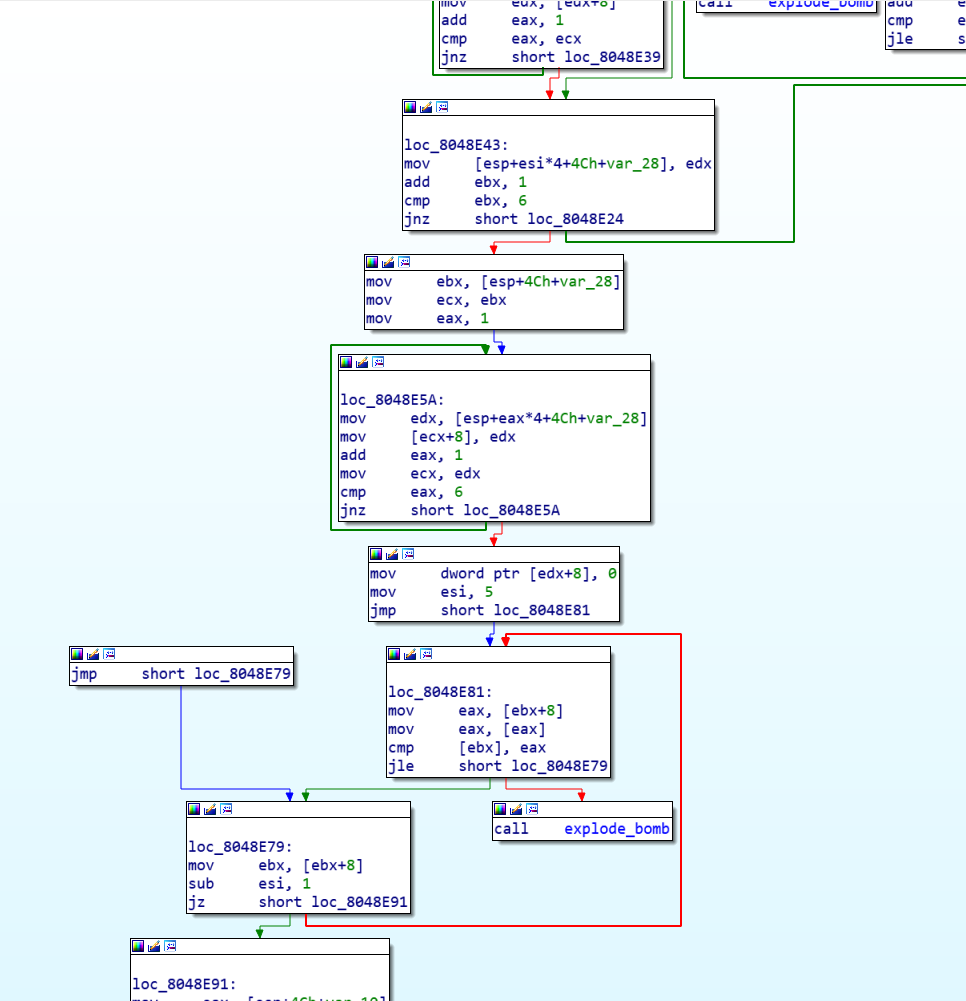


图2.12

类似的，从这样的结构中可以看出，这个链表读取时，应该是采用循环嵌套的方式，因为我们需要获得一个升序的链表而且需要读取我们获得的值，节点的数据值沿着链表逐渐上升，所以在第二张示意图中可以发现两个循环，依次寻找。

# 2.3 实验小结

本次实验旨在加深对程序的机器级表示、汇编语言、调试器和逆向工程等原理与技能的理解。实验内容是拆解一个名为“二进制炸弹”的Linux可执行C程序，其中包含六个阶段（phase1~phase6），每个阶段需要输入特定字符串以防止炸弹爆炸。实验目标是尽可能多地拆除炸弹的层次。

每个阶段考察机器级语言程序的不同方面，难度逐级递增，涉及字符串、循环、条件分支、递归与栈、指针和链表结构等知识点。完成任务需要使用gdb调试器和objdump来反汇编炸弹的可执行文件，逐步跟踪和调试每个阶段的机器代码，通过理解汇编语言代码的行为，推断出拆除炸弹所需的目标字符串。

我还使用的逆向工程中的常用工具IDA Pro对我们的反编译结果进行了初步分析，如果实验提供了完整的二进制文件甚至可以还原一部分c语言代码，这是一种非常特别的体验。

# 实验3：Buffer Overflow Attack 缓冲区溢出攻击

# 3.1 实验概述

### 3.1.1 实验目标

加深对IA-32函数调用规则和栈结构的具体理解。具体来说本次实验对一个可执行程序“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

### 3.1.2 实验要求与环境

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

### 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

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码并进行阅读理解，了解到程序的栈帧结构，从而获取到可以成功实现缓冲区溢出攻击并执行目标函数的输入内容，完成任务。

3.实验过程

（1）使用objdump进行反汇编，并将汇编代码输出到bufbomb.txt文件中。

（2）查看汇编源代码，找到getbuf函数，如图3.1所示，

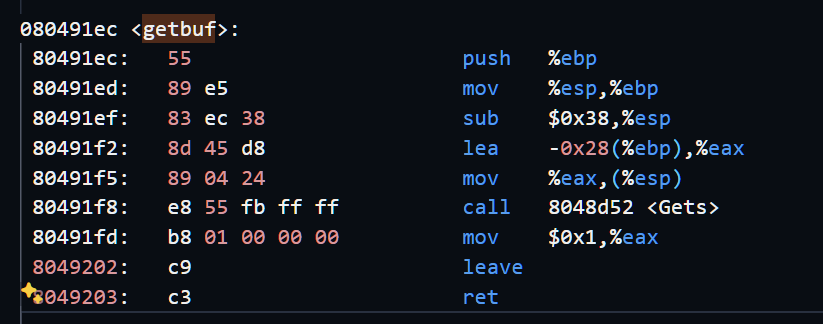


图3.1 getbuf函数

可以观察到，lea -0x28(%ebp),%eax给输入的字符串分配了0x28大小的空间，而返回地址是在%ebp+4处，而函数一开始将ebp压栈，占用了4字节，故输入字符串距离返回地址共48个字节。将最后4个字节设置为smoke函数的入口地址，将输入的字符串长度控制在48 字节，以覆盖原本的返回地址，并确保 eax 的值为 1，这样函数返回后会跳转到新的返回地址，覆盖原先的正常返回地址，造成缓冲区溢出攻击，使getbuf函数返回时转向执行smoke函数。

（3）在bufbomb.d文件中找到smoke函数，得到其地址为0x08048c90，如图3.2所示。我们需要提供48字节大小的输入字符串，并将其第45个字节至第48个字节设为smoke函数的入口地址0x08048c90即可，因为是小端存储，故设为“90 8c 04 08”，前44字节内容设为全0。

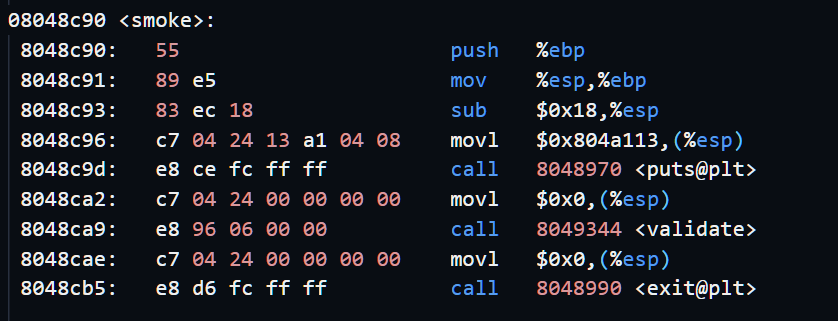


图3.2 smoke函数

（4）根据上述分析，最终的smoke.txt文件内容设置如图3.3所示。

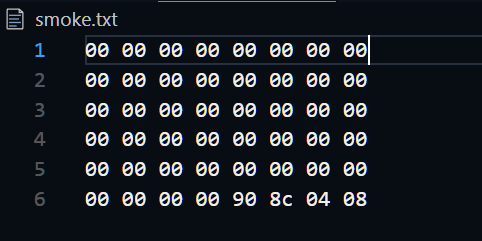


图3.3 smoke\_U201914858.txt文件内容

4.实验结果

将smoke.txt文件使用hex2raw工具将注释去掉，生成相应的raw攻击字符串文件，借助linux操作系统管道操作符和cat命令，成功调用smoke函数，如图3.4所示。

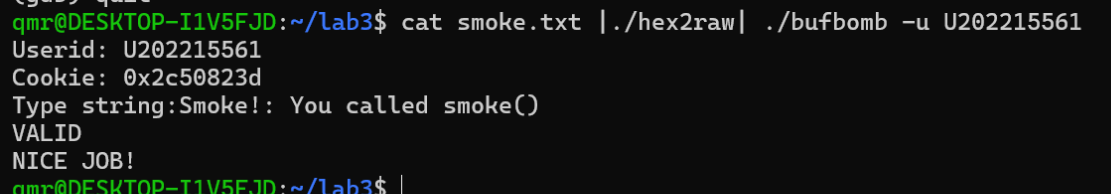


图3.4 成功调用smoke函数

### 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()。

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，通过观察汇编代码并进行阅读理解，了解到程序的栈帧结构，从而获取到可以成功实现缓冲区溢出攻击并执行目标函数的输入内容，完成任务。

与Smoke阶段不同的地方在于：这一次执行的是fizz函数，且需要我们传递一个参数。这个参数对每个同学来说是不同的，任务书中给出了获取方式。这里给出本人获得的：0x2c50823d。

3.实验过程

（1）在bufbomb.txt文件中找到fizz函数，得到其地址为0x08048cba。如图3.5

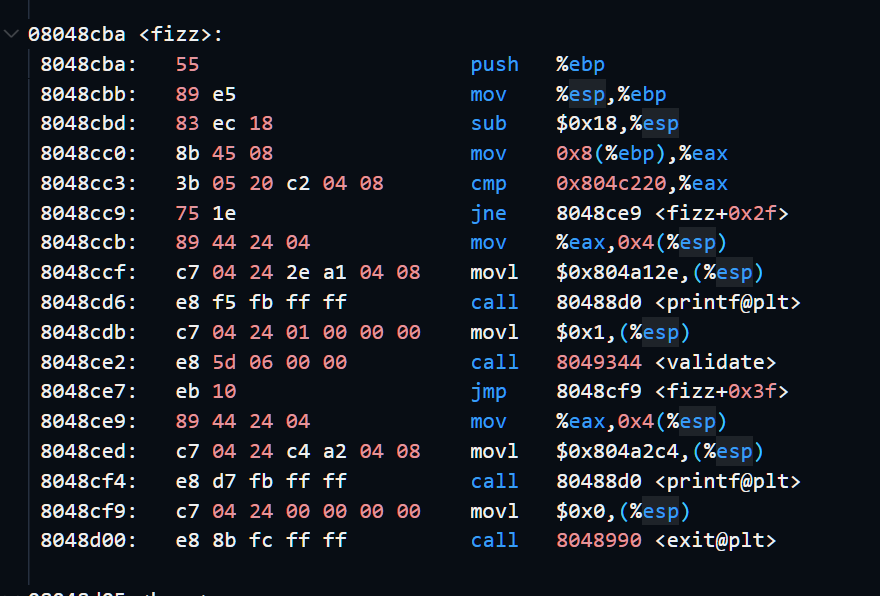


图3.5 fizz函数

（2）可以如同转移到smoke函数一样将前48个字节中的后四字节改为“ba 8c 04 08”，不同的是这一次需要传递参数，因此需要在这48个字节后面再次添加参数传递的指令，通过调用makecookie文件得到“U202215561”对应的cookie值为0x2c50823d，如图3.6所示。



图3.6 生成cookie

（3）后新增的8个字节中，后4个字节的内容应为该cookie值，在小端存储下，后4个字节应设为“3d 82 50 2c”，前4个字节可取任意不含换行符的ASCII码的内容，最终的文件内容设置如图3.7所示。

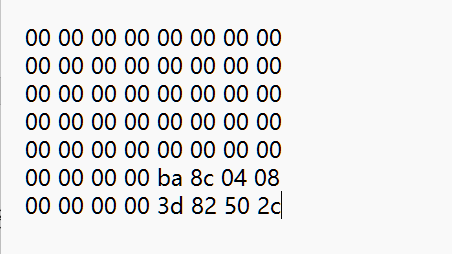


图3.7 fizz.txt文件内容

4.实验结果

将fizz.txt文件使用hex2raw工具将注释去掉，生成相应的raw攻击字符串文件，借助linux操作系统管道操作符和cat命令，成功调用fizz函数，如图3.8所示。

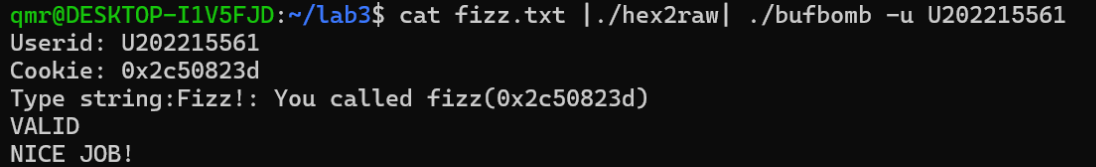


图3.8 成功调用fizz函数

### 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设置为学号对应的cookie值，然后转向执行bang()。

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，使用gdb对程序进行调试，通过观察汇编代码并进行阅读理解，了解到程序的栈帧结构，从而获取到可以成功实现缓冲区溢出攻击并执行目标函数的输入内容，完成任务。

3.实验过程

（1）在bufbomb.d文件中找到bang函数，得到其地址为0x08048d05，如图3.9所示。由mov 0x804c218,%eax命令可得，global\_value存放地址为0x804c218。

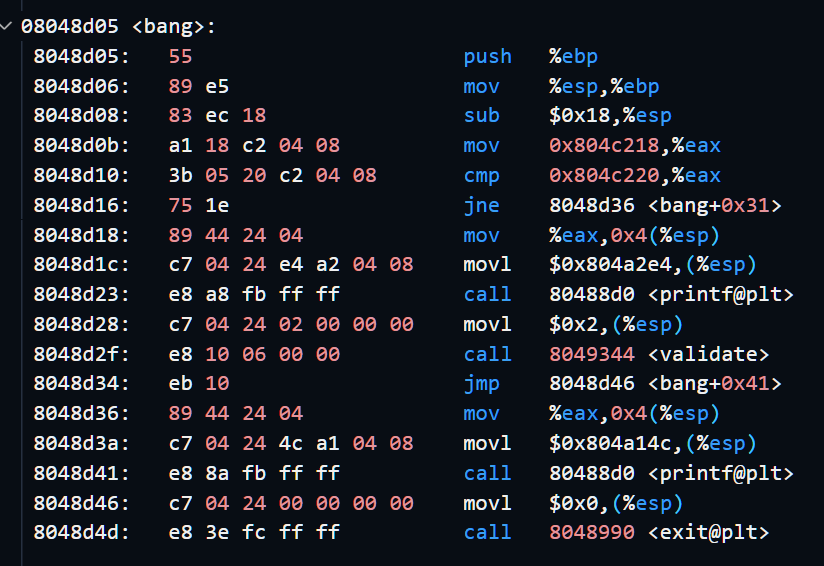


图3.9 bang函数

（2）注意mov 0x804c218,%eax和cmp 0x804c220,%eax这里比较了地址0x804c218与地址 0x804c220各自指向的内存单元的值。cookie的地址就是0x804c220，因此全局变量global\_value 的地址就是0x804c218。将全局变量global\_value的值修改为cookie的值，接下来还需要将程序跳转到函数bang的位置。因此在转换变量值之后，还需要一个ret指令来辅助转移。因此可编写攻击代码如图3.10所示。

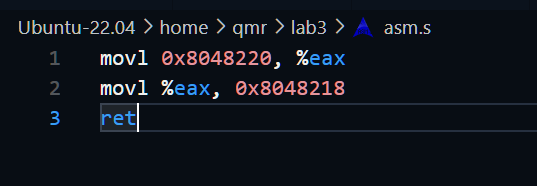


图3.10 攻击代码

（3）通过gcc将其编译成机器代码后，使用objdump将其反汇编获得需要的二进制机器指令字节序列如图3.11所示。

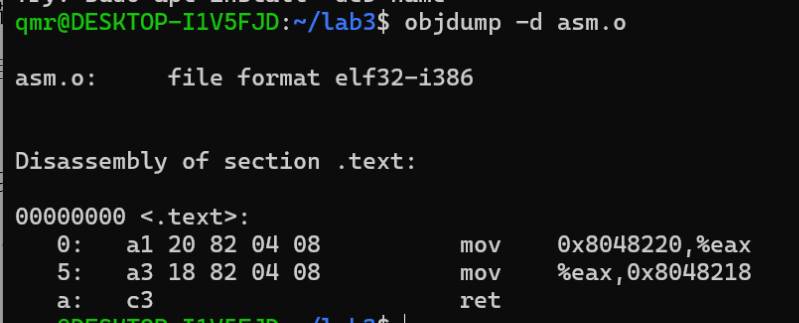


图3.11 攻击代码对应的二进制机器指令字节序列

（4）此外还需要在代码的辅助之下完成bang函数的跳转。返回地址需要被 buf 的首地址覆盖，因此如同前面一样，需要将45~48位设置为buf的首地址。buf的首地址可以在 gdb 调试过程中，通过getbuf函数正常运行期间查看寄存器eax的值来获取，在getbuf函数中，mov %eax,(%esp)在此将字符串首地址压入栈中，可得该行指令地址为0x80491f5。通过gdb得其地址为0x55683e18，如图3.12。

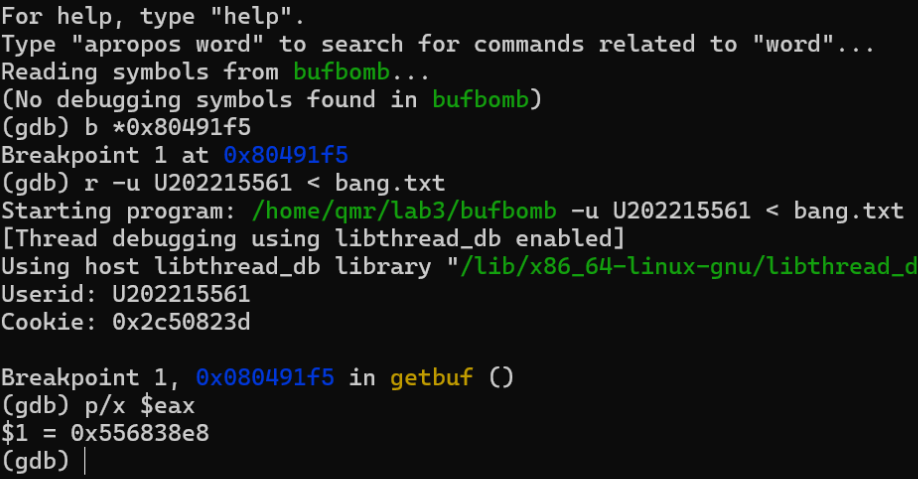


图3.12 获取字符串首址

最终的bang.txt文件内容设置如图3.13所示

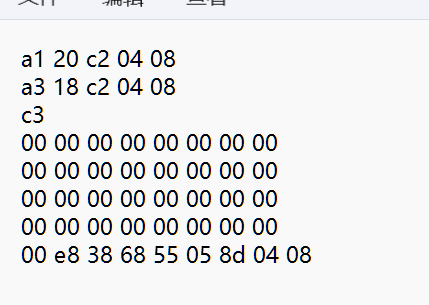


图3.13 bang\_U201914858.txt文件内容

4.实验结果

将bang.txt文件使用hex2raw工具去掉注释，生成相应的raw攻击字符串文件，借助linux操作系统管道操作符和cat命令，成功调用bang函数，如图3.14所示。

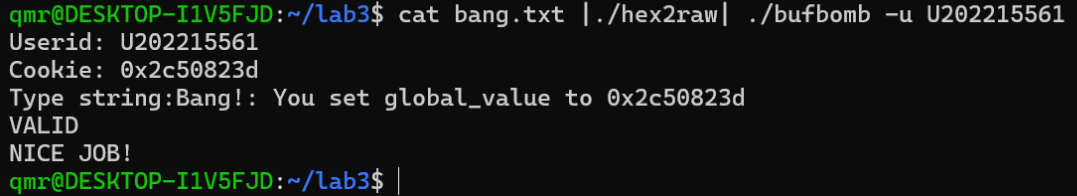


图3.14 成功调用bang函数

### 3.2.4 阶段4 boom

1.任务描述

构造一个攻击字符串，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

使用反汇编工具objdump对目标程序进行反汇编，使用gdb对程序进行调试，通过观察汇编代码并进行阅读理解，了解到程序的栈帧结构，从而获取到可以成功实现缓冲区溢出攻击并执行目标函数的输入内容，完成任务。

3.实验过程

（1）getbuf()函数在被调用时，程序的返回值被存储在%eax寄存器中。因此，要想返回学号对应的cookie值，只需将eax的值修改为目标cookie值即可。需要将getbuf后下一条指令地址压栈之后再ret，找到调用getbuf后下一条指令地址为0x8048e81，如图3.15。

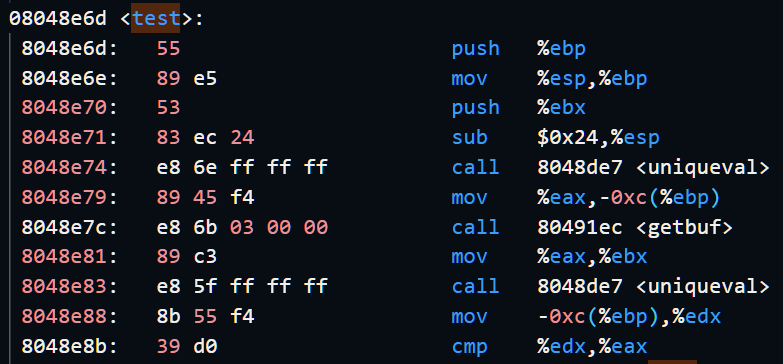


图3.15 调用getbuf的指令地址

（2）为了恢复%ebp，使用gdb调试，通过打断点得到ebp值为0x55683940，如图3.16所示。

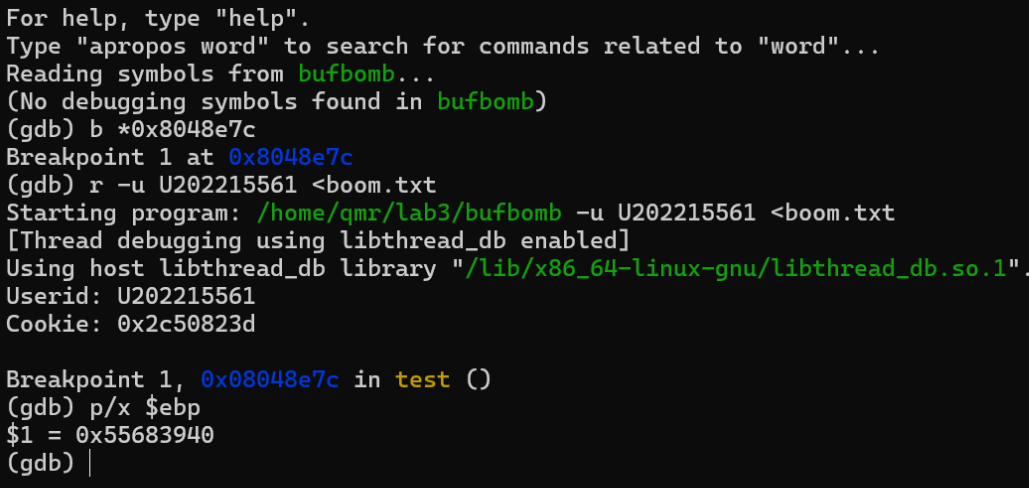


图3.16 观察ebp值

（3）故攻击代码编写如图3.17所示



图3.17 攻击代码

将攻击代码的汇编代码编译，再使用objdump将其反汇编获得的二进制机器指令字节序列如图3.18所示。

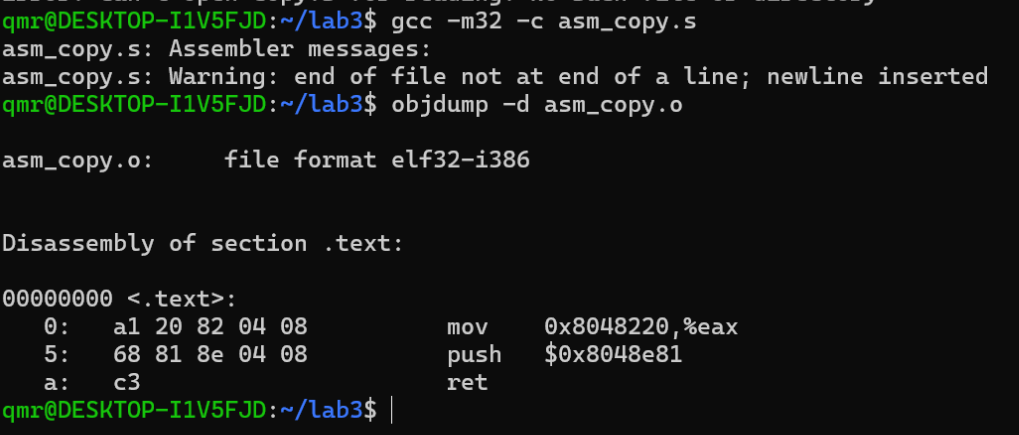


图3.18 攻击代码对应的二进制机器指令字节序列

（5）与bang阶段一样，输入字符串的第45个字节至第48个字节即为返回地址，应设置为输入字符串首址，在小端存储下即为“e8 38 68 55”。输入字符串的前部分直接填入攻击代码对应的二进制机器指令字节序列即可，其它部分置为全0。最终的boom.txt文件内容设置如图3.19所示。

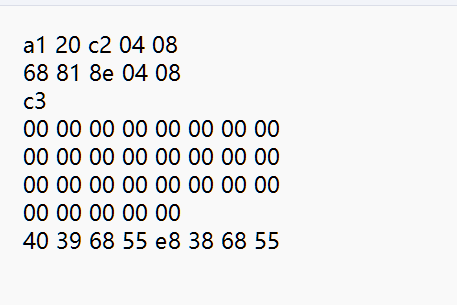


图3.19 boom.txt文件内容

4.实验结果

将boom.txt文件使用hex2raw工具去掉注释，生成相应的raw攻击字符串文件，借助linux操作系统管道操作符和cat命令，成功使getbuf返回指定cookie值，如图3.20所示。

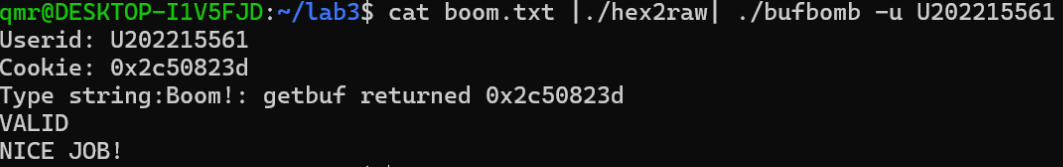


图3.20 成功使getbuf返回指定cookie值

### 3.2.5 阶段5 Nitro

1.任务描述

与阶段四类似，构造一攻击字符串使得 getbufn 函数（在 kaboom 阶段中，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。

2.实验设计

使用反汇编工具objdump对目标程序进行反汇编，使用gdb对程序进行调试，通过观察汇编代码并进行阅读理解，了解到程序的栈帧结构，从而获取到可以成功实现缓冲区溢出攻击并执行目标函数的输入内容，完成任务。

3.实验过程

 （1）由于栈上的机器代码是按地址由低向高顺序执行，要保证五次运行都能顺利执行有效机器代码，需要跳转地址位于有效机器代码入口地址之前的nop机器指令填充区。这要求尽可能增大nop填充区，尽可能使有效机器代码段往后挪。注意到ebp-0x208，可得输入字符串存储空间大小为520字节。

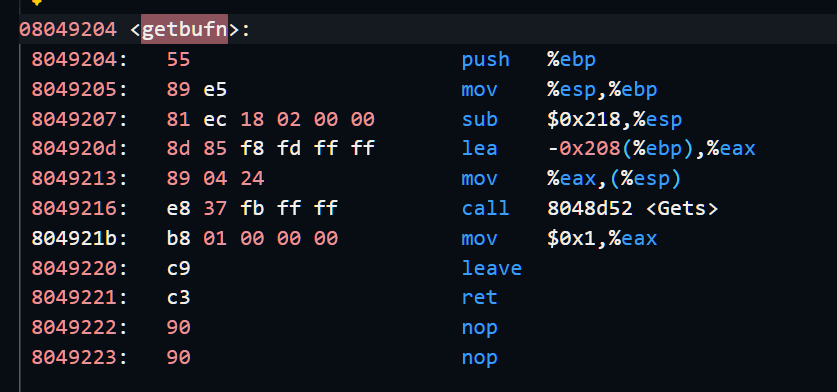


图3.21 getbufn函数

（2）这里找到testn函数，通过观察可以发现这次的，这一次的返回地址为0x8048e15

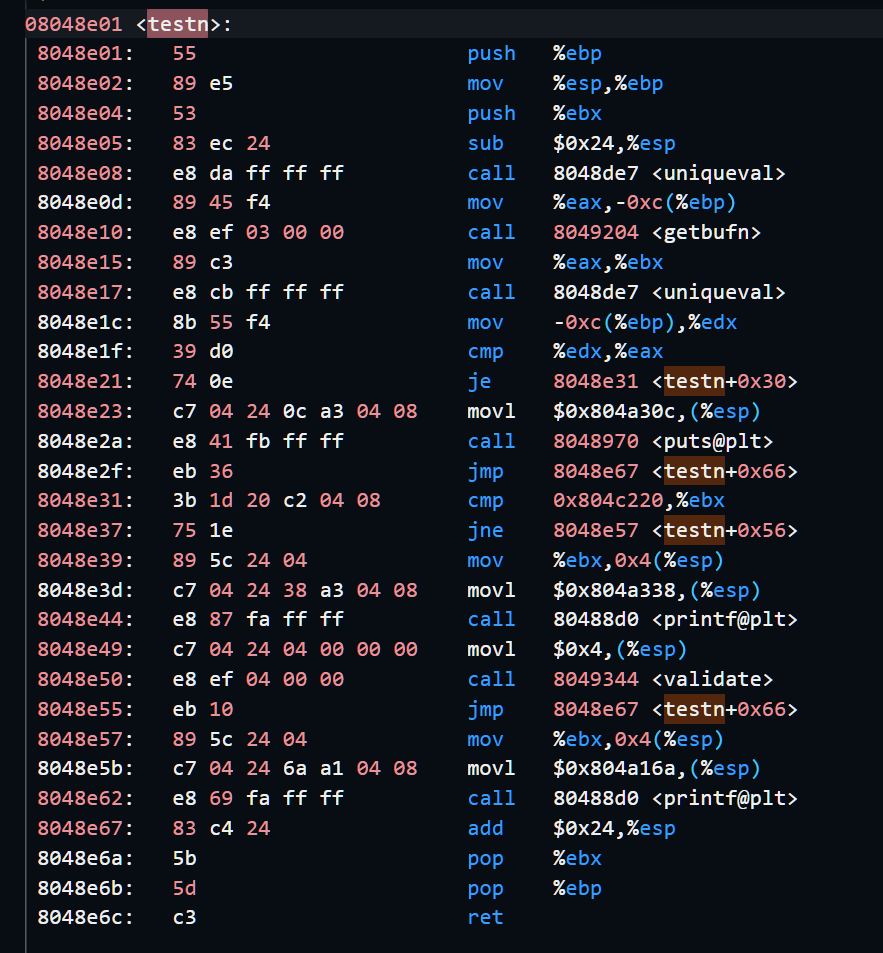


图3.22 testn函数

以getbufn中mov %esp,%ebp的下一条指令为断点（0x8049207），在Nitro模式下调试观察程序执行过程中ebp的值，如图3.23所示。现在需要一个返回地址，使得能够在buf首地址取到这里的每一个数值时， 都可以正常运行到buf内部。返回地址应该使得每一次被进入时都在buf的 内部。因此，这个返回地址的值应该大于等于buf首地址的最大值，也即0x55683980。

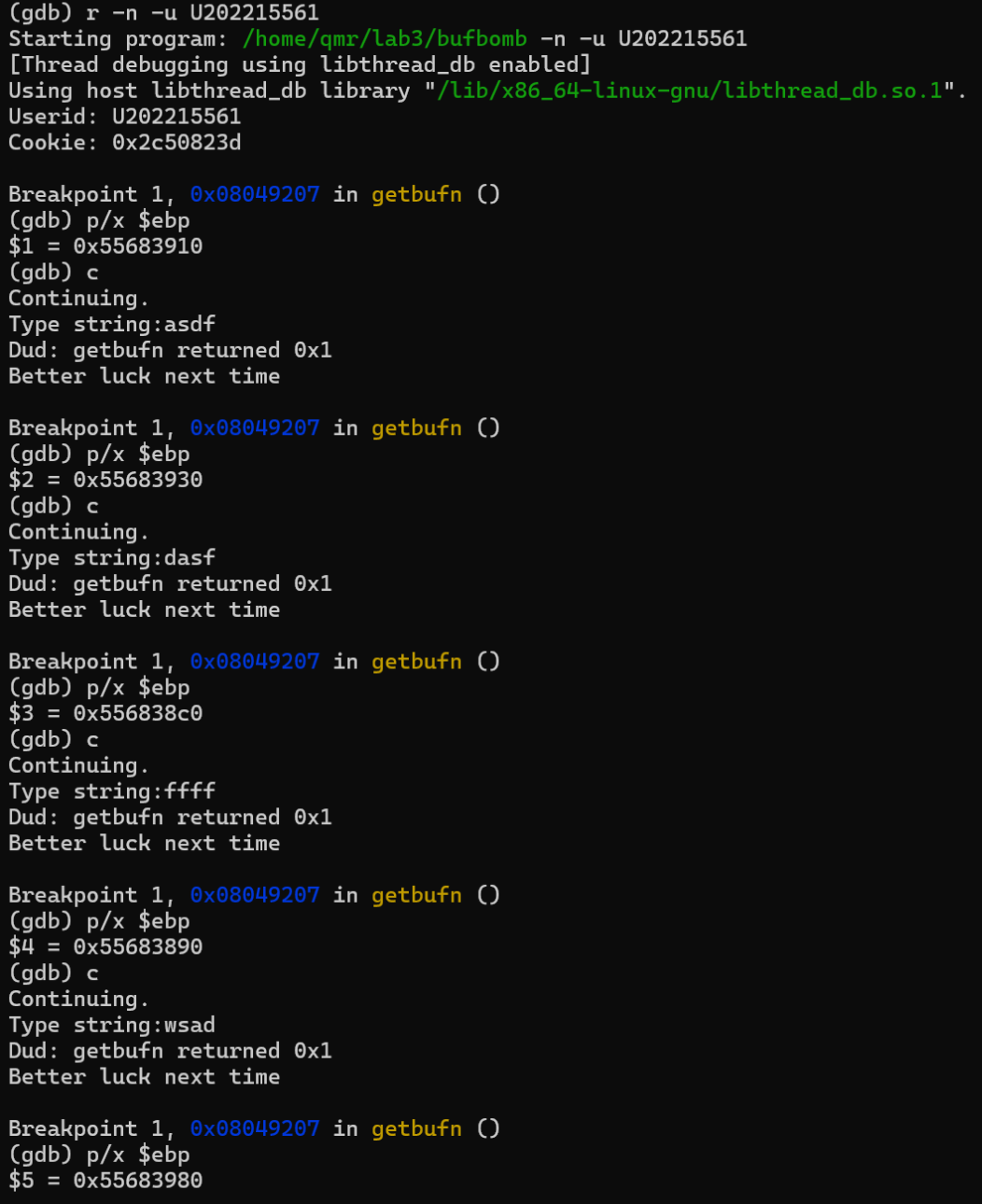


图3.23 调试观察ebp变化

（3）每一次的ebp的旧值都会因为栈帧位置的改变而改变，所以维护的ebp的旧值需要通过间接手段来获取。虽然每一次的位置都发生变化，但程序都执行了相同的函数和指令，因此ebp的旧值与新值的相对位置是不变的，也即ebp的旧值与寄存器esp的值的相对大小的不变的，不难发现其实ebp旧值 = R[esp] + 0x28.。

（4）结合上述分析，攻击代码需将ebp置为esp+0x28，再返回至调用getbufn的下一条指令地址，修改getbufn返回值为指定cookie值的方法与boom阶段相同，故编写得攻击代码如图3.24。

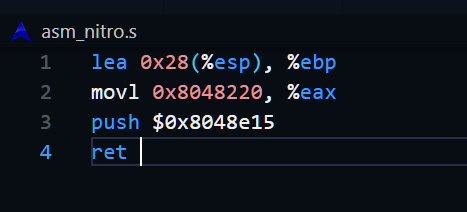


图3.24 攻击代码

通过gcc将攻击代码的汇编代码编译成机器代码后，使用objdump将其反汇编获得需要的二进制机器指令字节序列如图3.25所示。

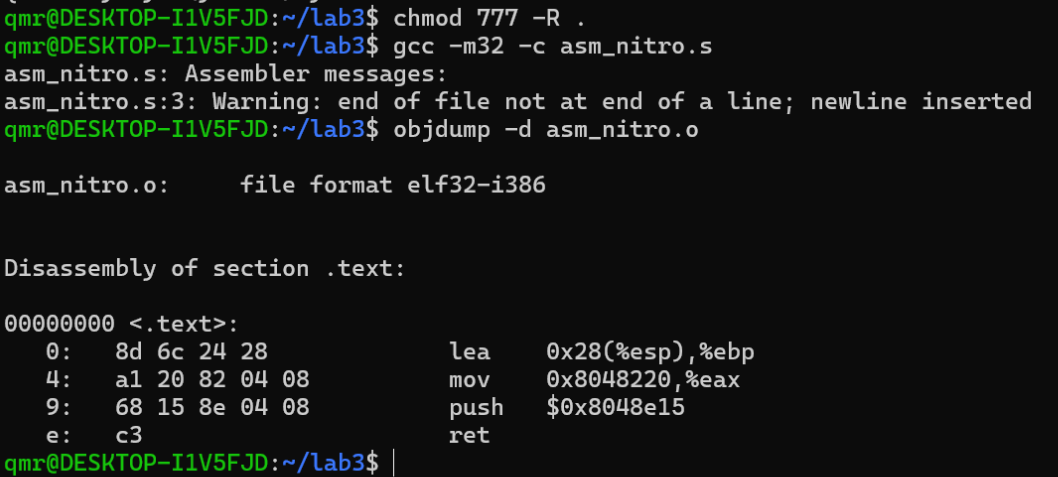


图3.25 攻击代码对应的二进制机器指令字节序列

（5）输入字符串共需528个字节，需要在其中填写nop代码、攻击代码和最终返回地址。根据前述分析，最后4字节的返回地址应为输入字符串的最大始地址，在小端存储下即为“80 39 68 55”，其前即为攻击代码对应的二进制机器指令字节序列，共15字节，其余的528-4-15=509字节全用nop填满。最终的nitro.txt文件内容设置如图3.26所示。

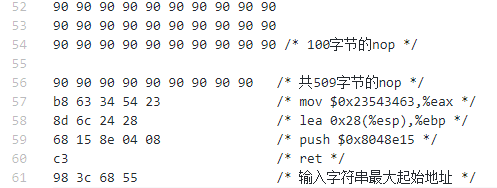


图3.27 nitro\_U201914858.txt文件内容（前509字节的nop代码部分省略）

4.实验结果

将nitro.txt文件使用hex2raw工具去掉注释，生成相应的raw攻击字符串文件，借助linux操作系统管道操作符和cat命令，成功攻击，如图3.28所示。

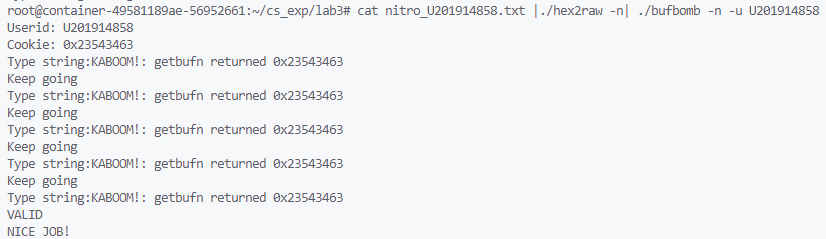


图3.28 成功攻击

# 3.3实验小结

本次实验是一个大型的反汇编实验，共设计了5个阶段，整个实验通过要求设计缓冲区溢出攻击这一攻击手段，总体上重点考察了对栈帧结构的具体理解。我通过使用反汇编工具与调试工具，观察目标程序经过反汇编后的汇编代码，并对其进行分析，推理得出能够完成任务目标、实现溢出攻击的正确输入字符串，从而逐个完成各个阶段的任务。

在本次实验中，通过阅读大量的汇编代码，加深了我对IA-32函数调用规则的具体理解；同时对于gdb、objdump、gcc等工具的频繁使用，使我熟练地掌握了这些工具的使用。为了完成实验目标，必须理解IA-32函数调用、返回、执行时，各个阶段下栈帧结构的变化。与实验二类似，本次实验设计了多个关卡，难度逐渐递增，虽然很考验逻辑思考、代码阅读能力和对IA-32函数调用规则和栈帧结构的理解，也花费了大量的时间和精力，但也非常有挑战性和趣味性，使我在这些方面的能力在很大程度上得到了提高。

# 实验总结

二进制炸弹实验的目的是通过逆向工程和调试，理解和拆解复杂的二进制程序，避免其在输入错误时“爆炸”。缓冲区溢出攻击实验的主要目的是理解和实践缓冲区溢出攻击的基本原理和方法。实验中，通过编写和运行存在缓冲区溢出漏洞的程序，利用该漏洞实现对程序控制流的劫持。

在编写C/C++程序时，需要特别注意内存管理，尤其是用户输入的处理，避免使用不安全的函数（如strcpy、gets等）。通过使用调试工具（如GDB）分析程序执行过程，定位漏洞并确认攻击的有效性。在实际开发中，应采用安全编程规范，如使用安全的库函数、对输入进行严格的边界检查和验证。

实验中，我学会了使用多种逆向工程和调试工具，分析未知的二进制代码，理解其功能和逻辑。破解每一关都需要细心分析程序的指令和数据流，提高了我的代码阅读和逻辑推理能力。面对复杂的二进制炸弹程序，需要有足够的耐心和细致的工作态度，逐步分析和解决问题。了解了二进制程序在输入处理上的一些常见漏洞，增强了我对程序安全性的理解和防范意识。

在遇到难题或变化时，可能会过于依赖已有的方法和思路，缺乏灵活的应变和创新解决方案的能力。这两个实验不仅强化了我对缓冲区溢出和二进制逆向工程的理解，还培养了我的调试和分析能力。通过实践，我体会到了安全编程的重要性和复杂程序分析的挑战性。这些技能和经验将为我未来的编程和安全研究工作提供重要的支持和指导。

本次实验使用了CMU在《CSAPP》一书中设计的实验内容，非常经典且实用，采用难度逐渐递增的关卡式设计，使我在挑战关卡的过程中由浅入深地掌握了计算机系统基础的一系列知识。尽管后面的实验较为困难，整体实验量也较大，但关卡设计和实验中的一些幽默彩蛋，使整个实验趣味盎然，使我在正反馈下完成了整个实验内容。

庞大的实验量使我大大增强了对程序机器级表示、汇编语言、调试器和逆向工程等的理解，加深了对字符串比较、循环、条件/分支、递归调用和栈、指针、链表、结构体的掌握，深刻理解了IA-32函数调用规则和栈帧结构，完美达成了实验目的。同时，我熟练掌握了GDB、objdump、gcc等常用的调试、反汇编和编译工具，熟悉了Linux系统下的各种基础操作。实验还考察了递归调用甚至是二叉排序等数据结构和算法知识，熟悉了反汇编程序攻击、缓冲区溢出攻击等手段，将知识考查融入这些具有现实意义且趣味盎然的攻击和侦测手段中。

在本次实验中，我的计算机系统基础素养和知识水平有了大幅提升，收获颇丰的同时也感到非常满足。非常感谢在我遇到困难时帮助过我的老师和同学。