网络安全实验报告

## 实验目录

[网络安全实验报告 1](#_Toc69760850)

[1. 实验目录 1](#_Toc69760851)

[2. 实验背景 3](#_Toc69760852)

[实验目的 3](#_Toc69760853)

[实验内容 3](#_Toc69760854)

[实验结果 3](#_Toc69760855)

[实验环境 3](#_Toc69760856)

[操作系统 3](#_Toc69760857)

[软件版本 3](#_Toc69760858)

[3. 实验原理 3](#_Toc69760859)

[4. 实验步骤 4](#_Toc69760860)

[4.1实验准备 4](#_Toc69760861)

[4.1.1工具下载 4](#_Toc69760862)

[4.1.2关闭保护机制 4](#_Toc69760863)

[4.1.3安装漏洞程序 5](#_Toc69760864)

[4.1.4构造shellcode 5](#_Toc69760865)

[4.2漏洞程序一 8](#_Toc69760866)

[4.2.1漏洞分析 8](#_Toc69760867)

[4.2.2攻击原理 8](#_Toc69760868)

[4.2.3构造payload 9](#_Toc69760869)

[4.3漏洞程序二 11](#_Toc69760870)

[4.3.1漏洞分析 11](#_Toc69760871)

[4.3.2攻击原理 12](#_Toc69760872)

[4.3.3构造payload 13](#_Toc69760873)

[4.4漏洞程序三 16](#_Toc69760874)

[4.4.1漏洞分析 16](#_Toc69760875)

[4.4.2攻击原理 17](#_Toc69760876)

[4.4.3构造payload 17](#_Toc69760877)

[4.5漏洞程序四 19](#_Toc69760878)

[4.5.1漏洞分析 19](#_Toc69760879)

[4.5.2攻击原理 20](#_Toc69760880)

[4.5.3构造payload 25](#_Toc69760881)

[4.6漏洞程序五 27](#_Toc69760882)

[4.6.1漏洞分析 27](#_Toc69760883)

[4.6.2攻击原理 27](#_Toc69760884)

[4.6.3构造payload 28](#_Toc69760885)

[4.7漏洞程序六 31](#_Toc69760886)

[4.7.1漏洞分析 31](#_Toc69760887)

[4.7.2攻击原理 32](#_Toc69760888)

[4.7.3构造payload 32](#_Toc69760889)

## 实验背景

### 实验目的

buffer overflow漏洞利用实践。

### 实验内容

编写exploits攻击漏洞程序

### 实验结果

获取具有root权限的shell

### 实验环境

#### 操作系统

Ubuntu16.04，64位wsl2，内核版本如下图所示



内核版本

#### 软件版本

gcc：5.4.0

make：4.1

## 实验原理

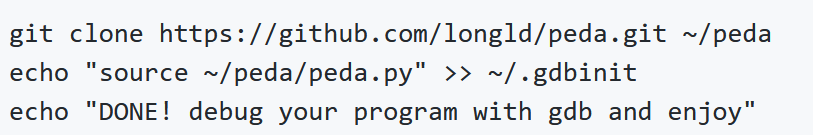
本实验一共有6个具有缓冲区溢出漏洞的程序，我们需要通过编写对应的exploit程序对其进行工具，并获得具有root权限的shell

## 实验步骤

### 4.1实验准备

#### 4.1.1工具下载

根据官网说明，下载gdb-peda



下载gdp-peda

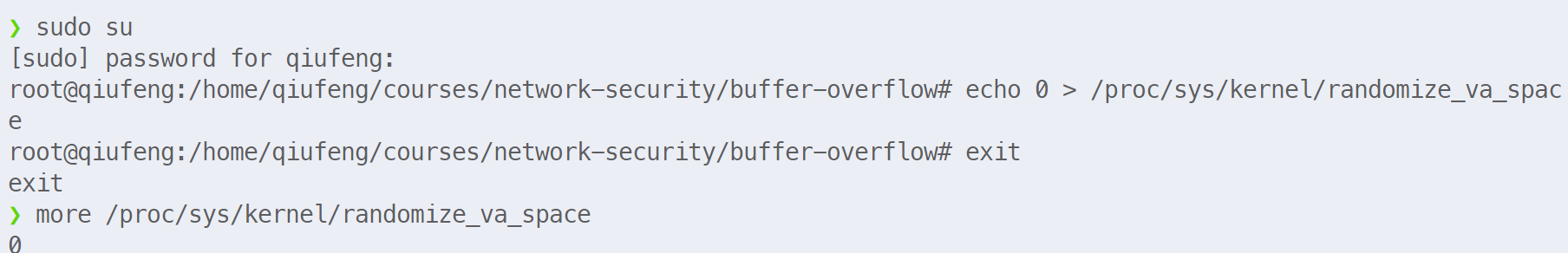
由于我们的gcc是32为版本的，因此还需要下载编译32位程序所需的链接库。执行如下命令



安装所需库

#### 4.1.2关闭保护机制

取消地址随机化



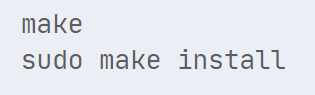
取消地址随机化

在编译漏洞程序时设置相应的参数

* ***-fno-stack-protector***：禁用堆栈溢出保护
* ***-z execstack***：关闭数据溢出保护

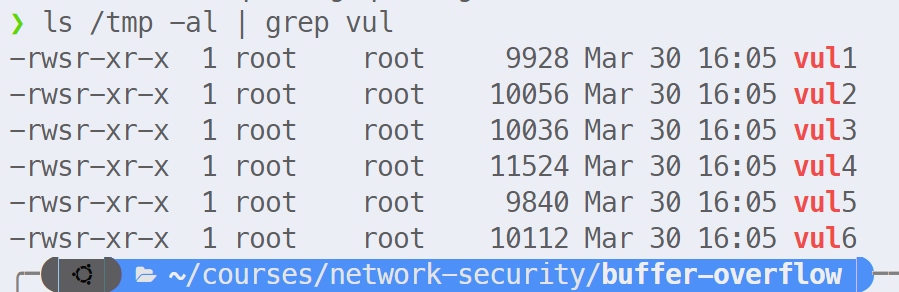
#### 4.1.3安装漏洞程序

根据**vulnerable**文件夹下的Makefile，我们执行下述命令编译漏洞程序，并将其复制到**/tmp**文件夹下



编译并安装漏洞程序

使用***ls /tmp -al | grep vul***查看安装的漏洞程序



查看安装的漏洞程序

#### 4.1.4构造shellcode

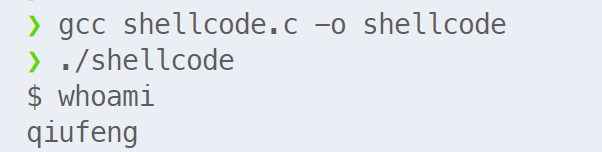
以下内容参考自文章[shellcode构造](https://www.jianshu.com/p/5d1b1eafca21)

首先编写shellcode的C代码，这里我们主要使用了***execve***函数



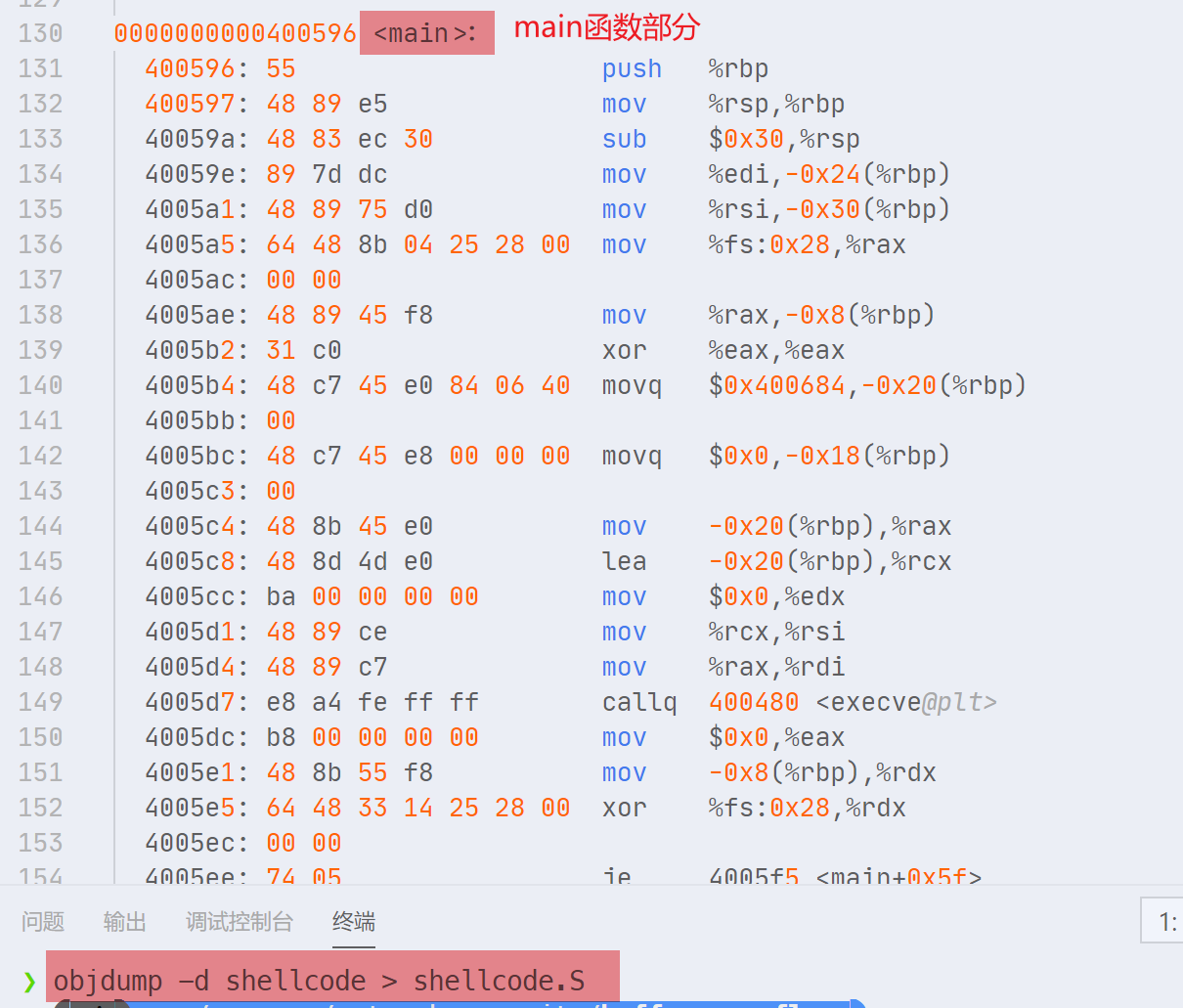
shellcode的C代码

用gcc编译并测试，发现能够弹出shell



gcc编译测试

使用objdump对编译出的二进制程序进行反汇编



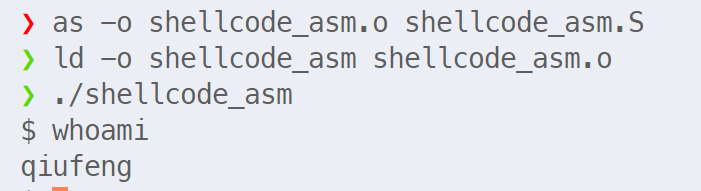
使用objdump得到汇编代码

根据编译得到的汇编代码，手工重写汇编代码如下



手工重写汇编代码

对重写的汇编代码进行编译、链接以及测试，发现成功弹出shell



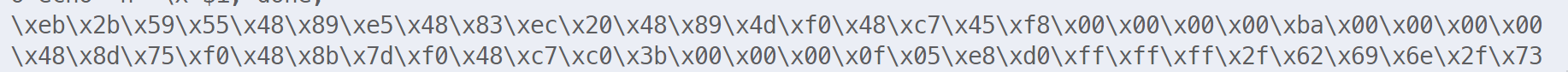
编译、链接、测试

使用如下命令获取shellcode的机器码



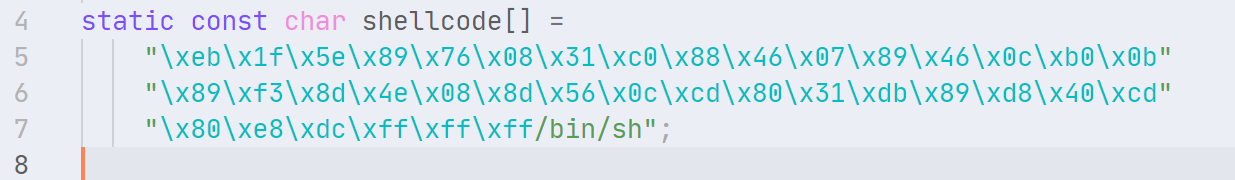
获取shellcode机器码

得到的机器码如下



shellcode机器码

实施上，除了自己构造以外，我们还可以采用其他人已经构造好的shellcode，由于实验需要使用到32位的shellcode，而我们的机器是64位的，因此这里直接使用**Aleph One**构造的shellcode(参考自**Berkeley**大学的讲义***[Smashing The Stack For Fun And Profit](https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf)***)。



### 4.2漏洞程序一

#### 4.2.1漏洞分析

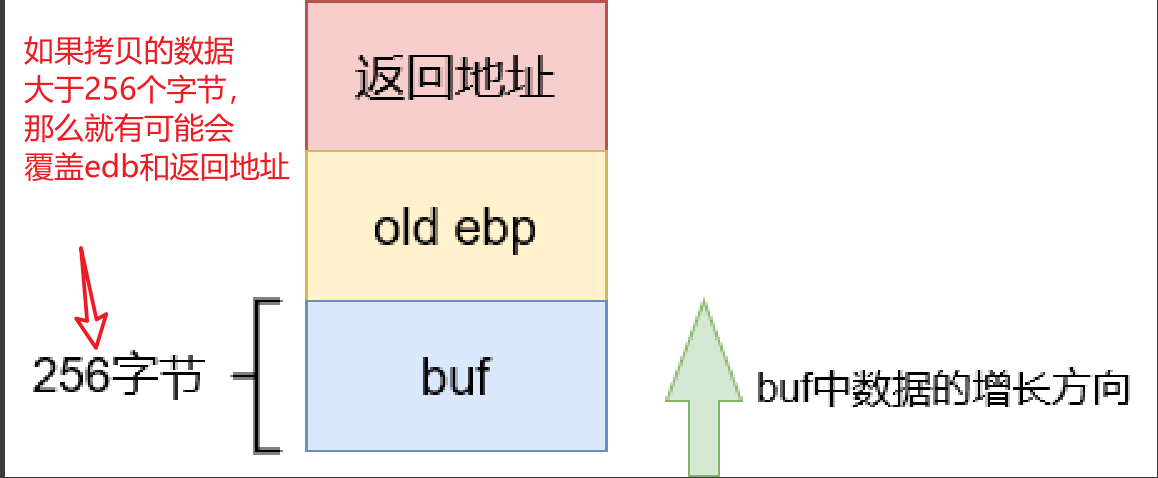


vul1.c

分析漏洞程序vul1，我们可以发现在bar函数中将输入的参数argv[1]拷贝到缓冲区buf中，由于**没有进行数据长度的限制**，因此我们可以通过溢出buf来**覆盖foo函数的返回地址**。

#### 4.2.2攻击原理

该漏洞属于**栈溢出**。造成该漏洞的主要原因为：**栈从高地址向低地址增长，缓冲区恰好相反**。漏洞机理可以用下图来表示



漏洞机理

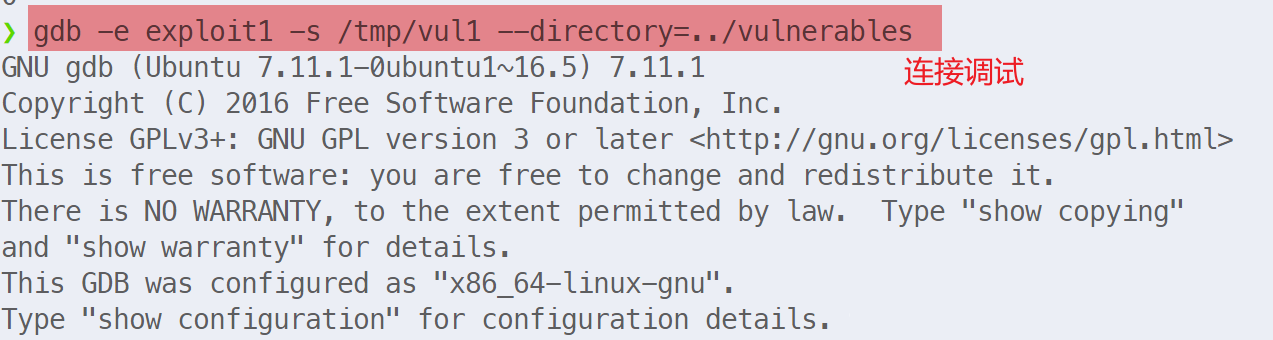
#### 4.2.3构造payload

构造payload的步骤如下

1. 使用gdb调试

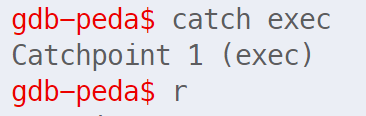
**这里只在实验一对gdb的使用方法进行详细说明。**

为了让调试时能够反映exploit执行时真实的内存情况，我们需要**将vul1和exploit1链接起来进行调试**



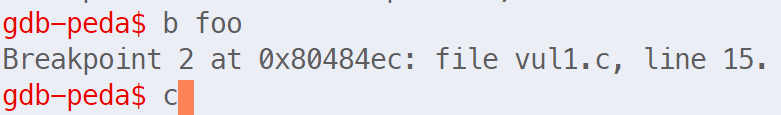
链接vul1和exploit1

由于在exploit1中使用了系统调用exec，我们在调试时需要使用命令***catch exec***设置catchpoint



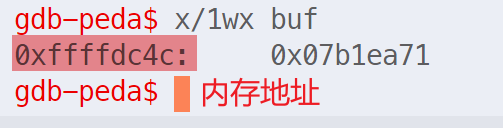
设置catchpoint

使用命令***b foo***为foo函数设置断点，并使用命令***c***步进到断点处



设置断点

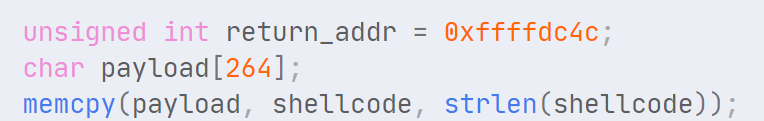
使用命令***x/1wx buf***查看缓冲区buf的起始内存地址为**0xffffdc4c**



buf起始内存地址

1. 确定shellcode地址

为了简便，这里选择直接将shellcode拷贝到buf的开始处。

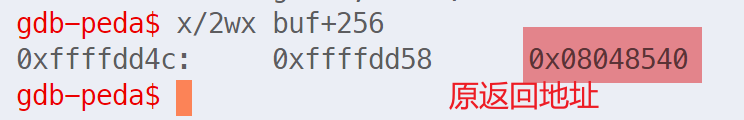


拷贝shellcode

于是我们只需将foo函数的返回地址覆盖为buf的起始地址，即**0xffffdc4c**处

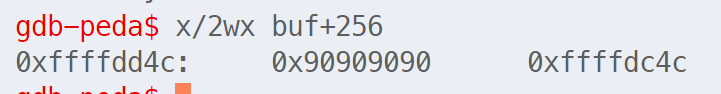
1. 覆盖返回地址

使用命令***x/2wx buf+256***，可以看到原来的返回地址为**0x08048540**



原返回地址

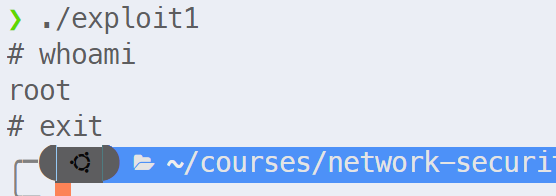
当执行完bar函数之后，我们可以看到返回地址被修改为**0xffffdc4c**，即shellcode的起始地址



修改后的返回地址

1. 执行漏洞利用程序

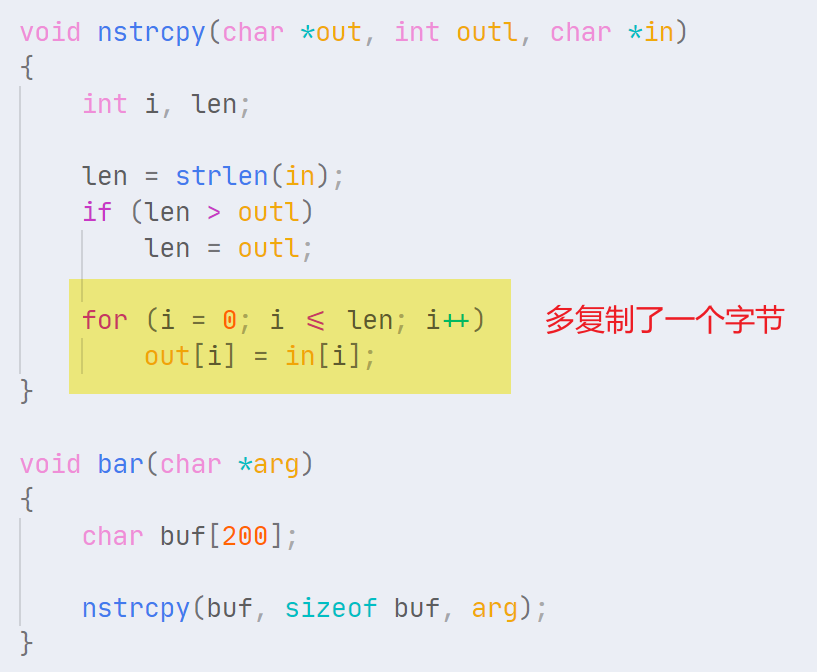
最后，执行编写的漏洞利用程序，可以发现成功获取具有root权限的shell



获取root权限shell

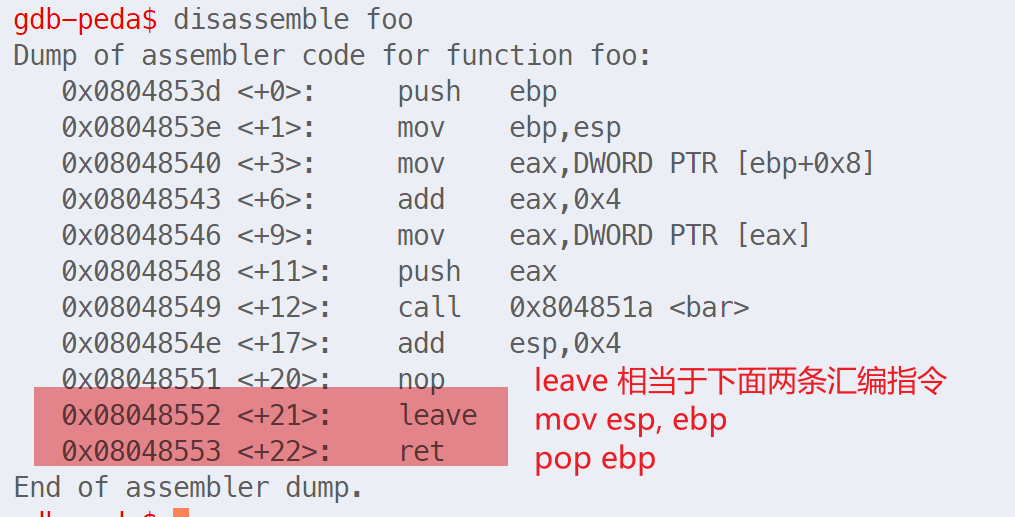
### 4.3漏洞程序二

#### 4.3.1漏洞分析



vul2.c

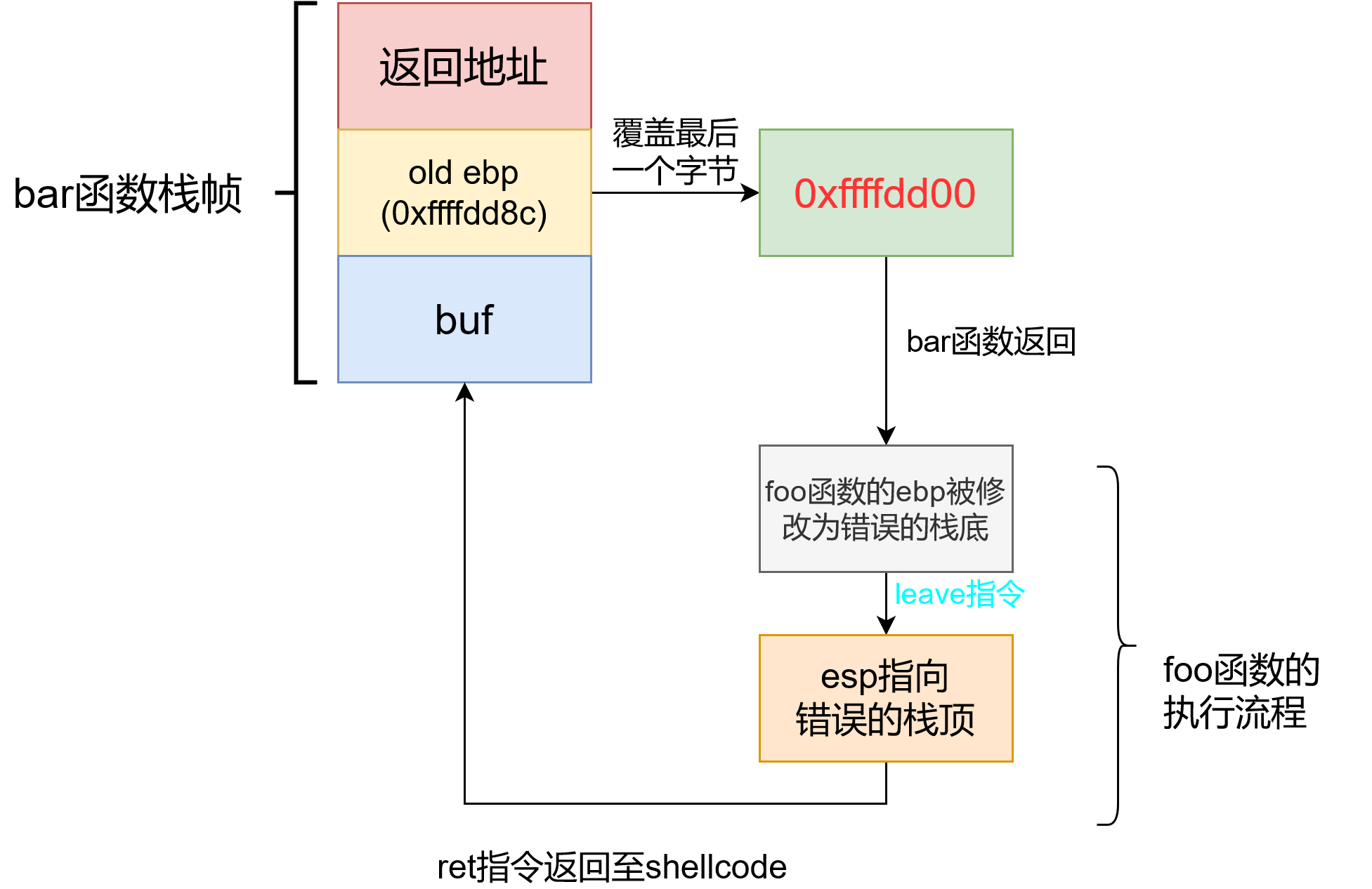
漏洞程序vul2和vul1非常相似，但它对拷贝时的字符串长度进行了限制。也就是说我们**不能够通过溢出缓冲区直接覆盖返回地址**。仔细观察函数***nstrcpy***，我们可以发现它在进行字符串拷贝的时候**多复制了一个字节**，这使得我们可以修改bar栈帧中的old ebp(实际上是调用函数foo的ebp)的最后一个字节。当foo函数返回时，会执行***leave***指令，将(被修改过的ebp)给到esp，如果esp+4指向shellcode的地址，那么当执行ret指令时，我们就能成功跳转到shellcode的位置



foo函数汇编代码

#### 4.3.2攻击原理

同vul1，该漏洞也属于**栈溢出**，只是我们不能够直接覆盖foo或者bar函数的返回地址。而是通过溢出修改foo的ebp寄存器的最后一个字节，使其在返回的过程中**esp指向错误的栈顶**，进而执行***ret***指令时跳转到shellcode地址。漏洞机理可以用下图来表示

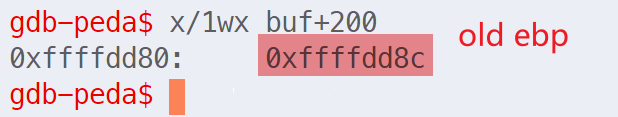


#### 4.3.3构造payload

构造payload的步骤如下

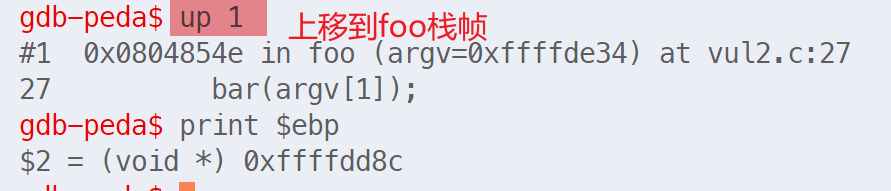
1. 覆盖bar函数栈帧中的old ebp

我们可以看到未覆盖之前bar栈帧中的old ebp如下图



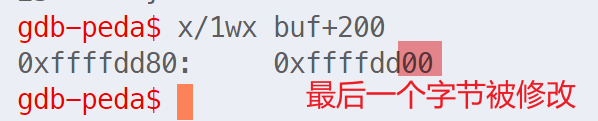
old ebp

我们使用命令***up 1***上移到foo函数的栈帧并查看ebp的值，可以发现和bar函数栈帧中old ebp的值是一样的



foo函数ebp

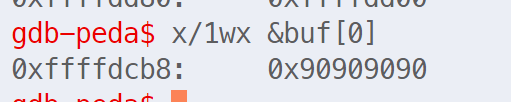
当执行完拷贝操作之后，我们可以发现bar函数栈帧中的old ebp最后一个字节被修改成了**0x00**



被修改后的old ebp

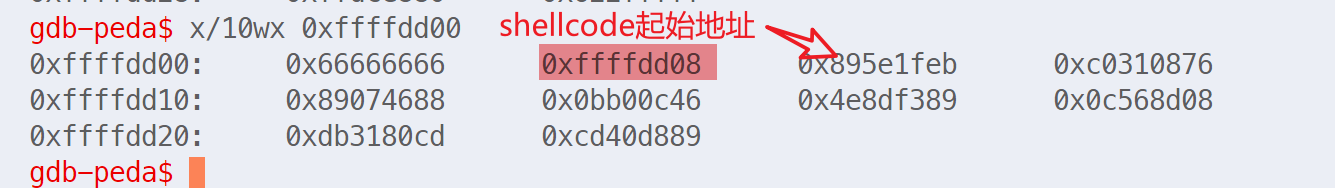
1. 确定shellcode的起始地址

我们首先获得buf的起始地址为**0xffffdcb8**



buf地址

根据修改后ebp的值**0xffffdd00**我们可以计算出其偏移量**offset=72**，于是**offset+4**的位置存放buf中shellcode的地址(它会被foo函数当作返回值)，这里我们直接将其设置为**ebp+8**，于是shellcode正好在其后面，偏移为**offset+8**



buf中存放的foo函数返回地址和shellcode

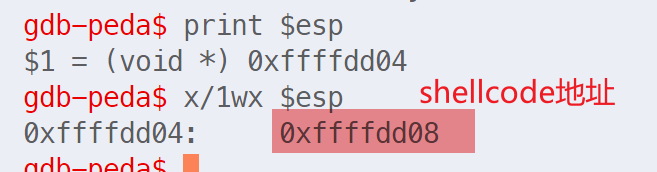
1. 使esp指向错误的栈顶

在foo函数执行***leave***指令之前打断点



leave之前打断点

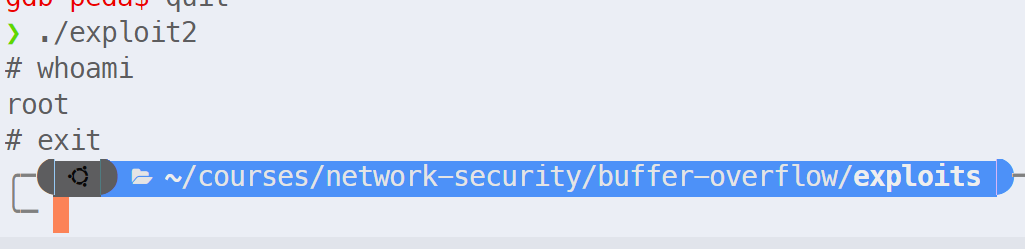
指令***leave***指令，发现esp的值为**0xffffdd04**，接着执行ret指令，跳转到shellcode的起始地址



esp寄存器的值

1. 执行漏洞利用代码

执行该漏洞利用程序，获取具有root权限的shell



获取具有root权限的shell

### 4.4漏洞程序三

#### 4.4.1漏洞分析



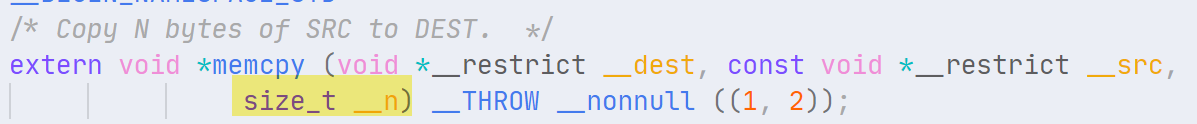
vul3

漏洞程序的主要流程如下

* 1. 将输入的前若干个字节转化成整数count，后若干个字节由字符指针in指向
  2. 调用foo函数
     1. 如果count小于MAX\_WIDGETS，那么拷贝in到缓冲区buf中

造成漏洞的原因主要有以下两点

* **foo函数中count的类型为int**
* **memcpy接受的内存长度类型为size\_t(32位程序中即代表unsigned int)**



memcpy函数原型

如果我们将**count构造成一个非常小的负数**，那么我们有可能绕过if判断并且拷贝大于缓冲区长度的数据

#### 4.4.2攻击原理

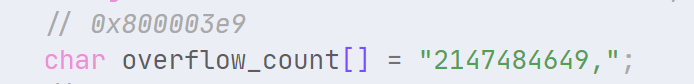
该漏洞属于**整数溢出漏洞**。通过将count构造成一个最高位为1的整数，能够绕过if判断，并且造成**栈溢出**。这里我们选择的整数为**0x800003e9**(十进制为2147484649)，因为0x3e9即十进制的1001，而缓冲区buf的最大长度为1000，因此多出来的20个字节(结构体widget\_t的大小)可以将foo函数的返回地址覆盖为shellcode的起始地址。

#### 4.4.3构造payload

构造payload的步骤如下

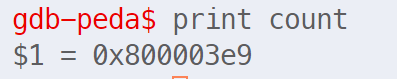
1. 构造count

根据在漏洞原理中的分析，我们构造的count为**0x800003e9**



构造count

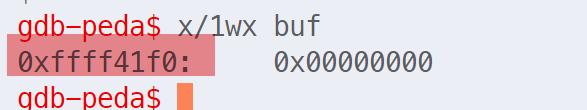
可以在gdb调试中确认count的值



count大小

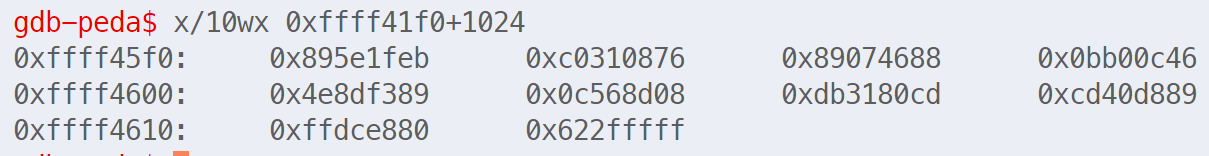
1. 确定shellcode地址

得到缓冲区buf的起始地址为**0xffff41f0**



buf起始地址

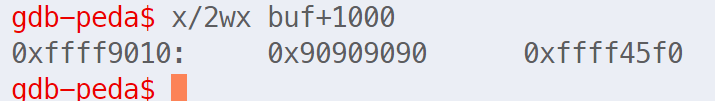
这里我们设定shellcode距离buf的偏移为**1024**，因此可以得到shellcode的起始地址为**0xffff45f0**，查看该内存地址的内容，发现shellcode已经写入



buf中shellcode位置

1. 覆盖返回地址

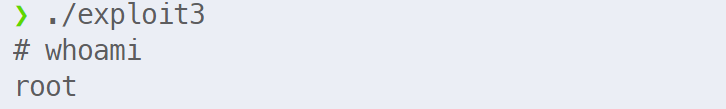
查看foo函数返回地址，可以发现已经背负覆盖成shellcode的起始地址**0xffff45f0**



foo函数返回地址

1. 执行漏洞利用程序

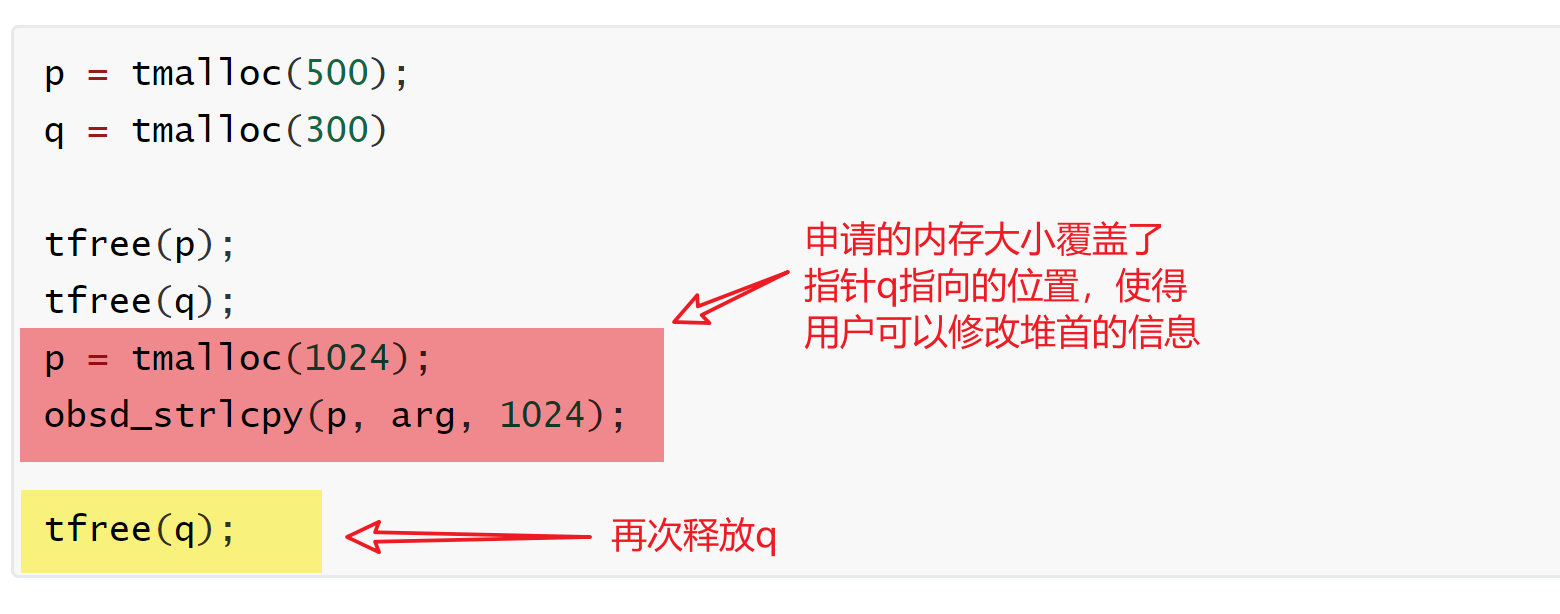
执行该漏洞利用程序，获取具有root权限的shell



获取具有root权限的shell

### 4.5漏洞程序四

#### 4.5.1漏洞分析



vul4

漏洞代码主要有以下几个步骤

* 1. 分别申请500字节和300字节的内存
  2. 释放这两篇内存区域
  3. 再次申请1024字节的内存
  4. 将用户的输入拷贝到这片区域
  5. 重复释放q指向的内存区域

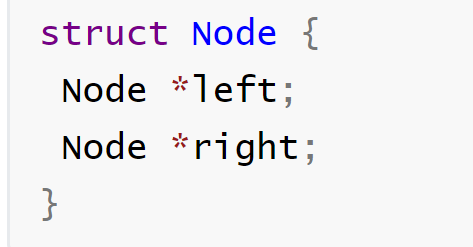
造成这段代码出现漏洞的原因主要有以下几点(在攻击原理中进行详细分析)

* tfree后**堆块数据指针**q**指向的位置不变**
* 指针p第二次tmalloc申请的内存**覆盖了堆块q的块首信息**
* 用户可以在堆块p中写入任意数据
* 重复释放已经释放过的堆块指针q

#### 4.5.2攻击原理

前三个漏洞都是**栈溢出**类型，该漏洞属于**堆溢出**。在看具体的漏洞利用之前，首先来分析一个由**双向链表删除操作造成任意内存写入**的例子。

假定双向链表节点***Node***的定义如下图所示



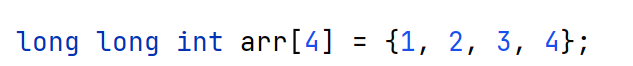
双向链表节点

当前链表存在p、q、r三个节点，且连接关系如图所示



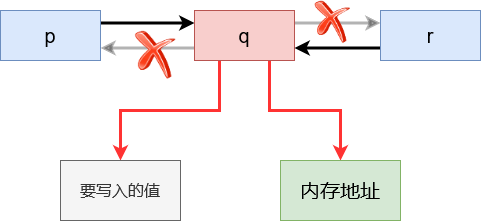
节点连接关系

此外，我们还有一个数组***arr***的内容如下所示



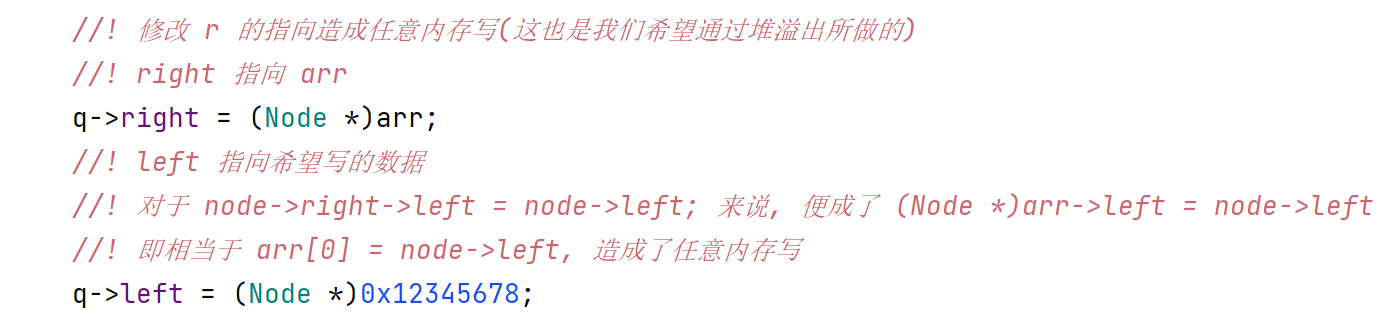
数组arr

我们希望将节点q从链表中删除。但是在进行删除操作之前我们对q的左右指针进行如下修改



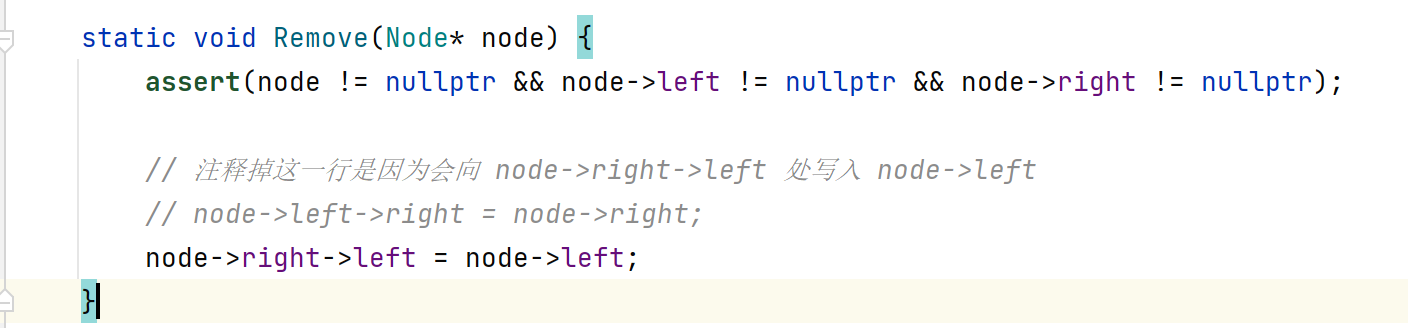
修改指针q

具体的，将q的右指针指向数组***arr***的地址，q的左指针**等于要写入的值**，我们设置为**0x12345678**



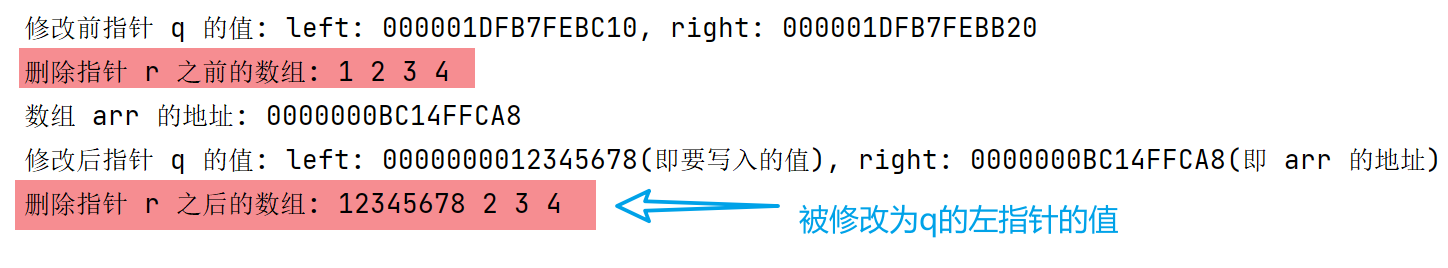
修改q的左右指针

接着我们将节点q从双向链表中删除，主要逻辑如下



删除q节点

当执行到语句***node->right->left = node->left***时会导致数组***arr[0]***的值被修改。实际上，我们可以将q的右指针指向任意的内存地址，这就造成了**任意内存写入**



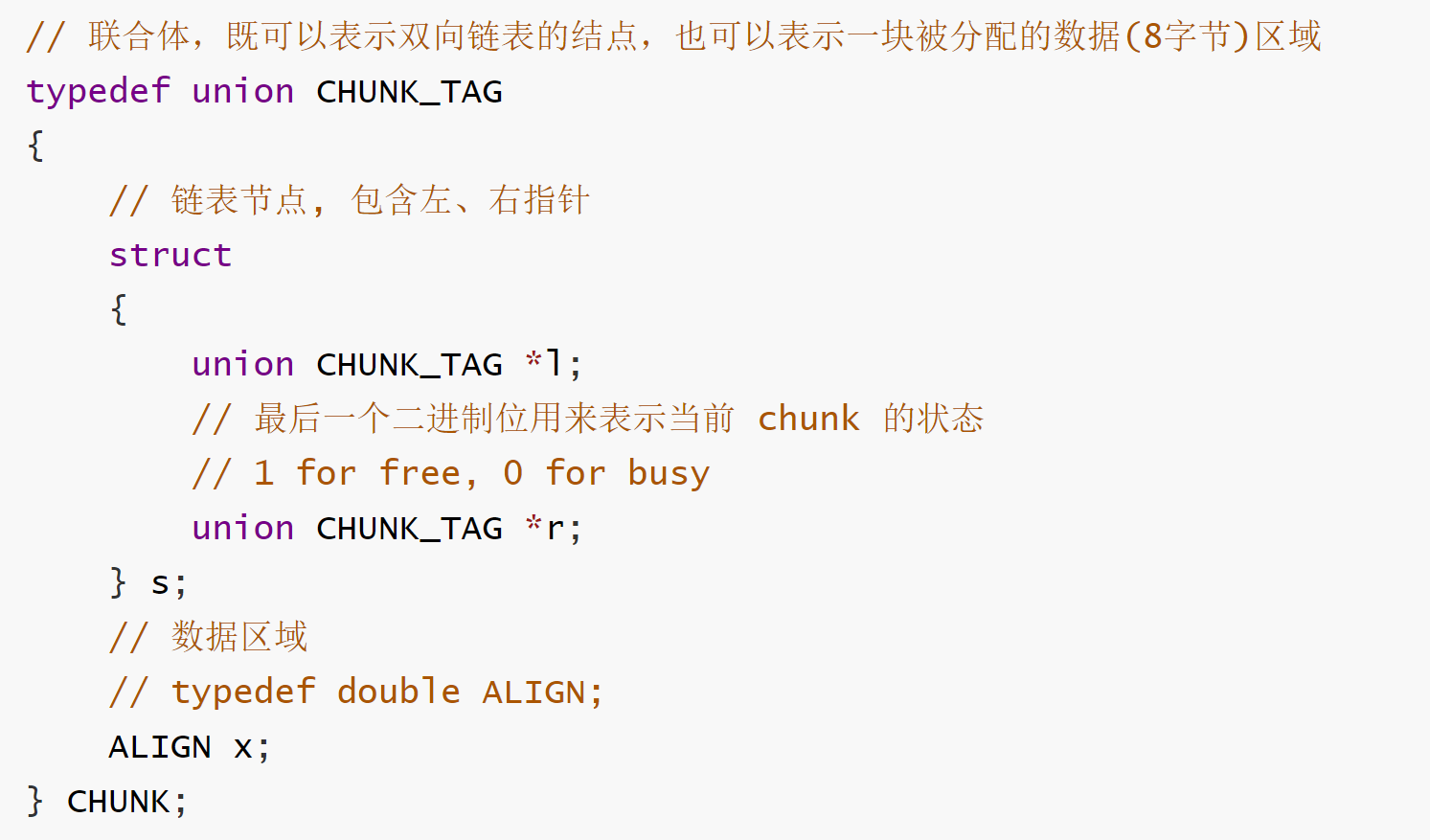
结果

本漏洞的利用点实际上和双向链表删除节点的操作是相同的。在漏洞程序中，每一个堆块由**堆首(8字节)+堆数据部分**组成，每个堆的堆首又形成**双向链表**。***tmalloc***和***tfree***实际上就是在堆首形成的双向链表中插入和删除某个节点。那么漏洞利用方式就容易想到了

***如果我们能够控制被删除堆堆首节点的左右指针，那么当执行tfree操作时，就能够使修改返回地址，使其指向shellcode的地址***

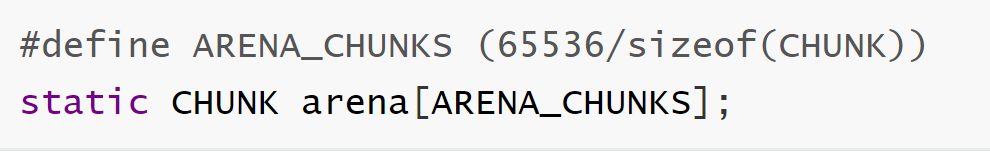
下面是对堆管理中数据结构的一些具体分析。

一个比较重要的数据结构是联合体***CHUNK***，它的定义如下图所示。它即可以用来表示**8字节的数据区域**，也可以用来表示**8字节的堆首信息**。当用来表示堆首信息时，右指针***s.r***的最后一位用来表示当前chunk的状态，为1表示空闲，为0表示已经被分配



联合体CHUNK

另一个比较重要的数据结构是***arena***，它是一个由chunk组成的数组，即整个堆所占据的空间。实际上，***arena***代表着从操作系统申请的一块大的、连续的内存区域，并且手动管理这部分的内存空间。



数组arena

在漏洞程序中被用到的两个函数是***tmalloc***和***tfree***，其中***tmalloc***主要包含以下步骤

1. 计算需要分配的大小
   1. 该大小需要按照***CHUNK***的大小对齐，对于32位程序来说即8字节
   2. 需要留1个chunk存放堆首信息
2. 找到双向链表中第一个足够大的空闲节点
3. 设置该节点的状态为已分配
4. 插入新空闲节点至双向链表



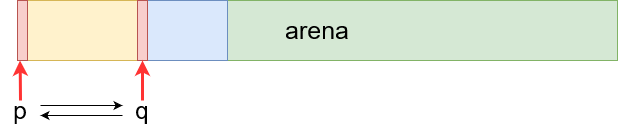
tmalloc函数

函数***tfree***主要包含以下步骤

1. 尝试与左空闲节点q合并
   1. 需要判断节点q是否空闲并且不为空
   2. 从双向链表中删除节点p
2. 类似的与尝试与右空闲节点合并

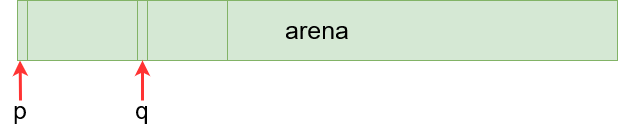
在清楚了堆管理相关的细节之后，再来分析一下漏洞代码。

首先通过***tmalloc***申请了p、q两块内存区域，他们的内存布局如下图(内存布局一)所示，p、q分别指向黄色区域和蓝色区域的开始位置。红色的区域存放8字节的堆首信息



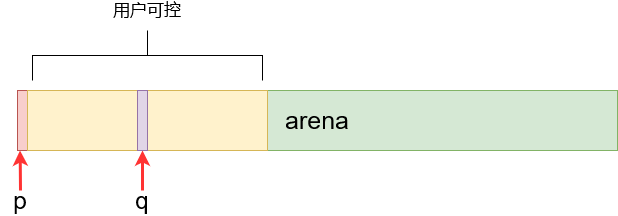
内存布局一

接着通过***tfree***释放了p、q，需要注意的是：**p、q仍然指向原来的位置**

**

内存布局二

之后又申请了内存区域p，由于之前p、q已经被释放，因此这次仍然是从***arena***的开始位置进行内存分配，又由于，因此用户写入的数据可以覆盖堆块q的堆首(504-512字节处)，即能够修改堆首q的左右指针



内存布局三

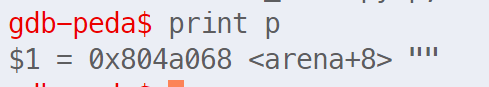
最后程序**重复释放**了堆q，即可能存在从双向链表删除节点的操作，从而造成任意内存写入漏洞

#### 4.5.3构造payload

构造payload的步骤如下

1. 确定shellcode地址

获得指针p指向的内存地址**0x804a068**



指针p指向位置

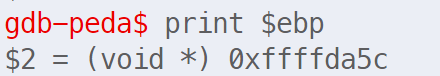
这里我们设置shellcode的偏移为32字节



shellcode偏移

1. 确定返回地址所在的内存地址

获得栈底指针寄存器ebp的值为**0xffffda5c**

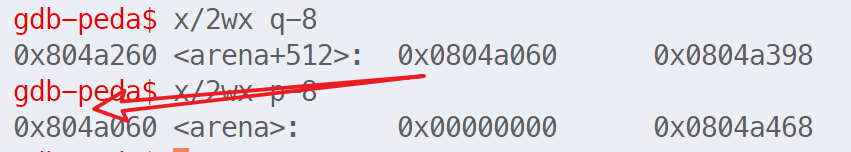


ebp寄存器

返回地址正好在其上方4个字节处，因此其地址为**0xffffda60**

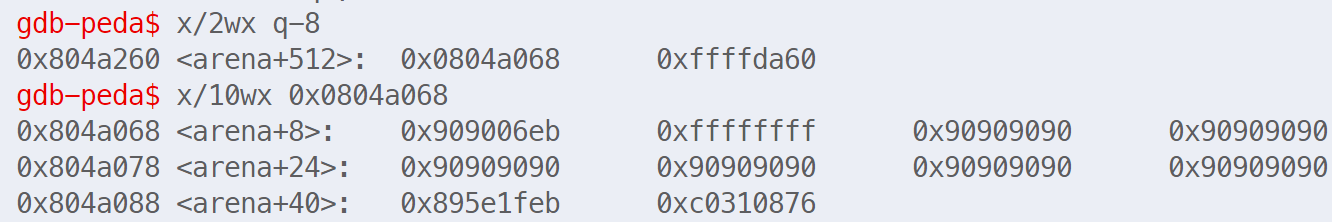
1. 覆盖q堆首的左右指针

查看没有被覆盖前q堆首的值如下，可以发现其左指针指向p的堆首



没有被覆盖时q的堆首

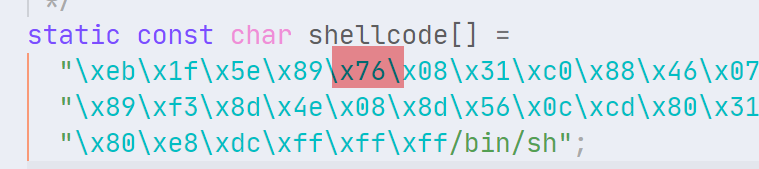
被覆盖后q堆首如下，可以发现其左指针等于p的起始**数据区域**，右指针指向返回值的内存地址



覆盖后q的堆首

1. 跳过非法区域

值得注意的是，这里我们并没有将q的左指针直接等于shellcode的起始位置，这是因为**与左节点合并必须满足左节点空闲**，即左指针的第5位应该是奇数，而我们的shellcode并不满足。



shellcode内容

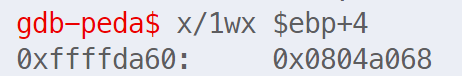
因此我们将其指向p的起始数据区域，同时将其5~8个字节赋值为**0xffffffff**。还需要注意的是，我们需要跳过这些非法的区域，即通过***eb 6***进行6字节的相对跳转



跳过6字节

1. 覆盖返回值

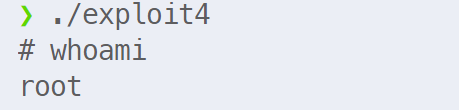
执行完***tfree***后，可以发现返回地址已经被覆盖成**0x0804a068**



覆盖返回地址

1. 执行漏洞利用程序

执行该漏洞利用程序，获取具有root权限的shell



具有root权限的shell

### 4.6漏洞程序五

#### 4.6.1漏洞分析

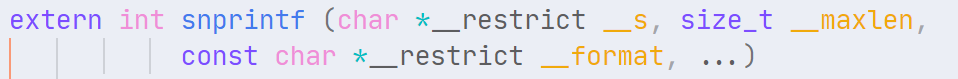


vul5

vul5属于格式化字符串漏洞，造成漏洞的主要原因是**用户可以输入任意格式化字符串**，进而利用特殊格式化占位符***%n***，造成对内存地址进行修改

#### 4.6.2攻击原理

***snprintf***的函数原型如下



snprintf函数原型

其作用为将格式化之后的字符串**最多拷贝*\_\_maxlen***长度至缓冲区***\_\_s***中。这里需要注意的一点是**先格式化再截断**，即当我们格式化之后的字符串超过***\_\_maxlen***时不会提前格式化的过程，而是将超过***\_\_maxlen***的长度进行截断。

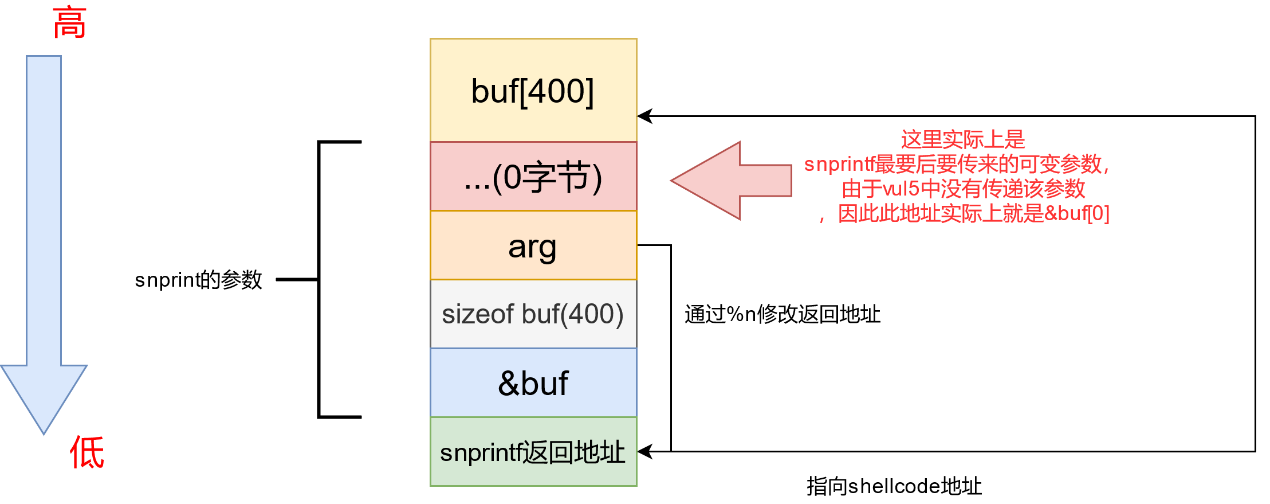
利用格式化字符串漏洞，我们可以读取栈中的数据，也可以修改特定内存地址的值。本实验中，主要是利用了格式化占位符***%n***会将当前字符串的长度保存到某一个变量地址中。例如下图将字符串***”hello, world”***一共12个字节保存到了变量***length***中



%n例子

利用***%n***的特性，我们可以将***foo***函数的返回地址修改为shellcode的地址。

该漏洞的攻击原理可以用下图来表示

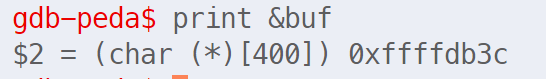


#### 4.6.3构造payload

构造payload的步骤如下

1. 确定返回地址所在的内存地址

首先确定缓冲区buf的起始地址为**0xffffdb3c**



buf地址

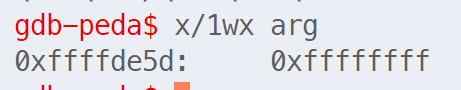
根据攻击原理中的栈布局图我们可以计算得到返回地址所在的内存地址为**0xffffdb3c-16(0x10)=0xffffdb2c**，查看该内存地址的值为**0x080404e8**，对该地址进行反汇编，可以看到它确实对应***foo***函数在调用***snprintf***的后一条指令



snprintf返回地址

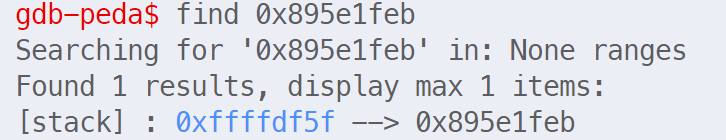
1. 确定shellcode内存地址

获得arg的内存地址为**0xffffde5d**



arg内存地址

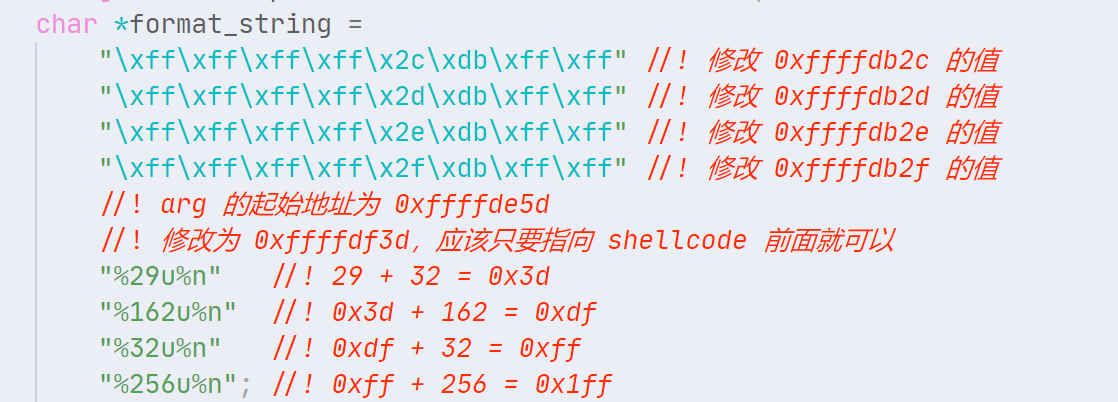
我们将shellcode写入到格式化字符串之后的200个字节开始的位置，即**0xffffde5d+0x3a(58)+0xc8(200)=0xffffdf5f**处，这与***find***搜索出的shellcode的起始地址一致



shellcode起始地址

1. 计算得到格式化字符串

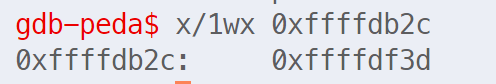
确定***snprintf***的返回地址为**0xffffdb2c**之后，我们只需要通过***%n***修改**0xffffdb2c、0xffffdb2d、0xffffdb2e以及0xffffdb2f**这四个字节的值，具体的计算过程如下图所示。其中格式化占位符后面数字的作用是**不足该长度的部分用空格补齐**。这里我们并没有直接跳转到shellcode的地址，因为跳转到shellcode前面()的nop指令位置也能够滑动到shellcode



构造格式化字符串

1. 修改返回地址

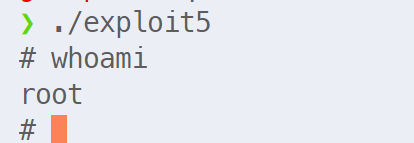
执行***snprintf***的过程中，可以发现其返回地址已经被修改成了shellcode前面nop指令的地址



被修改后的返回地址

1. 执行漏洞利用程序

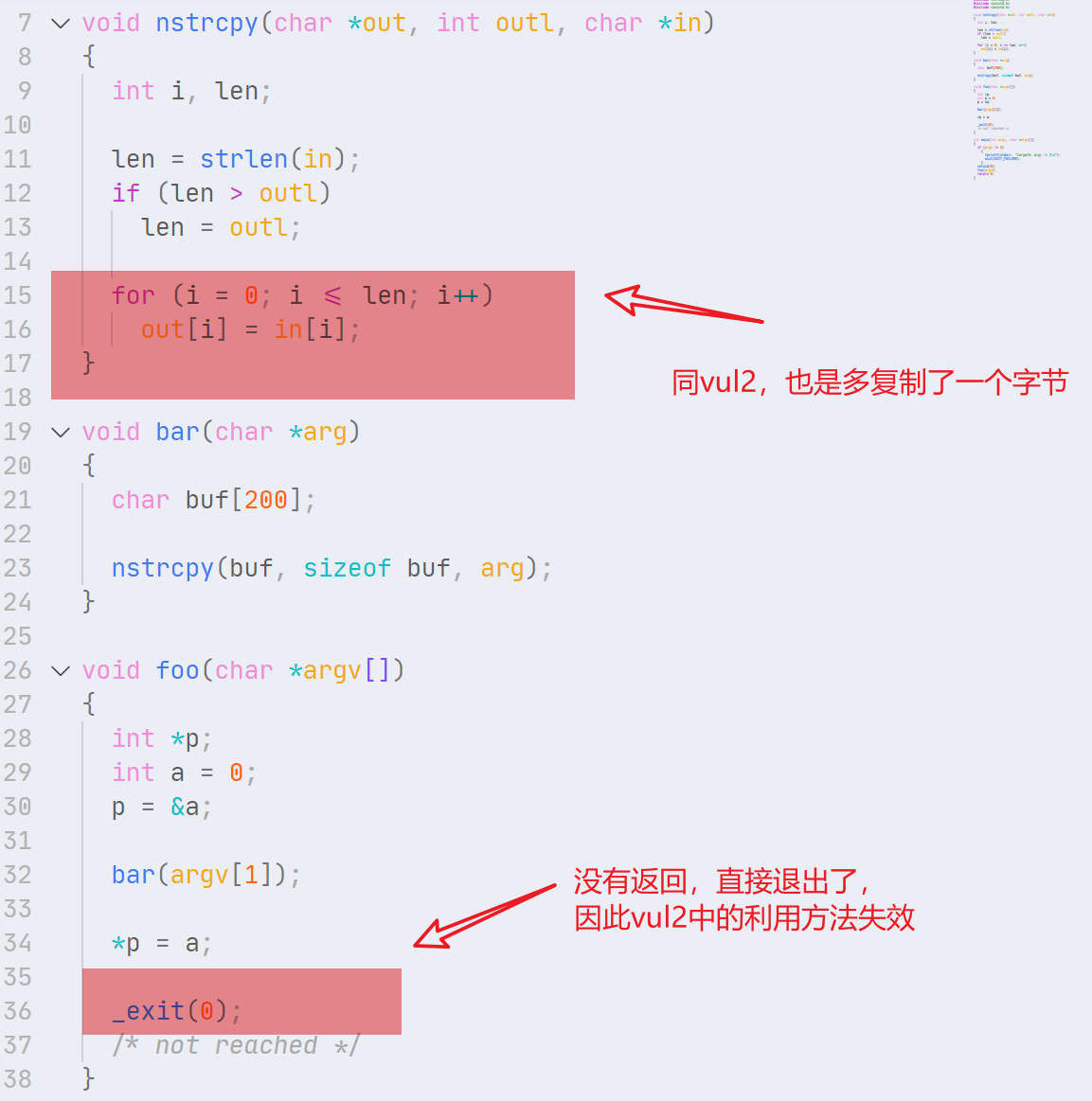
执行该漏洞利用程序，成功获取具有root权限的shell



获取具有root权限的shell

### 4.7漏洞程序六

#### 4.7.1漏洞分析

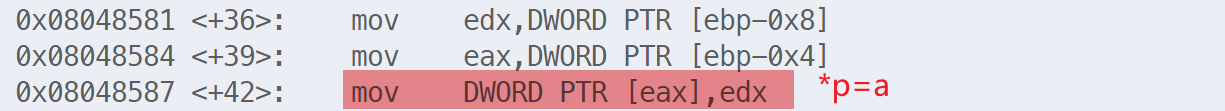


vul6

乍一看，vul6和vul2长的几乎一样。但在vul6中，***foo***函数直接通过***\_exit(0)***退出了，这导致我们在vul2中的利用方法失效(因为根本不会有***leave***和***ret***这两条汇编指令的执行)。注意到***foo***函数中多了一条对指针赋值的指令，那么，如果我们**能控制该指针以及变量*a*的值，就有可能通过修改某条跳转指令(*jmp*)要跳转的地址**来进行攻击。

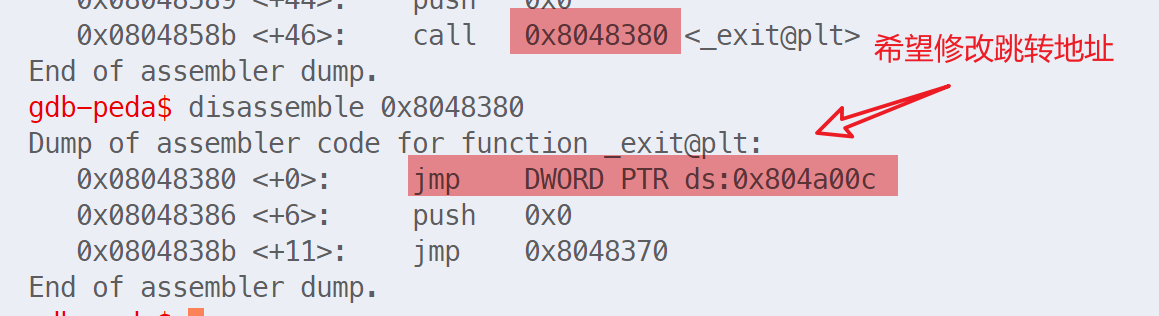
#### 4.7.2攻击原理

***\*p=a***实际上是被翻译成了如下三条汇编指令



\*p=a对应汇编指令

***\_exit(0)***函数对应的汇编代码如下



\_exit(0)对应反汇编

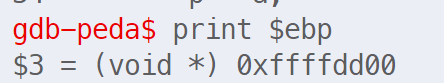
根据vul2中的经验，***nstrcpy***多拷贝一个字节可以导致**foo的栈底指针ebp被修改**。**由于与指针*p*和变量*a*均是根据*ebp*来进行寻址**，因此可以通过修改ebp，使得***p***指向跳转地址所在的内存地址**0x804a00c**，变量***a***的值等于shellcode的起始地址。

#### 4.7.3构造payload

构造payload的步骤如下

1. 修改***foo***栈底指针ebp

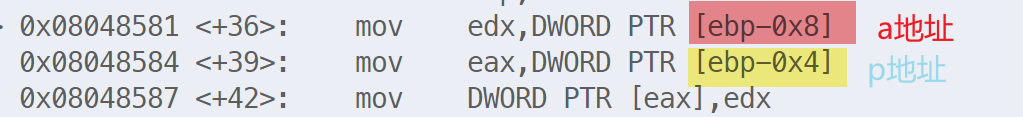
同vul2，我们将ebp的最后一位修改成**0x00**



修改***foo***函数ebp

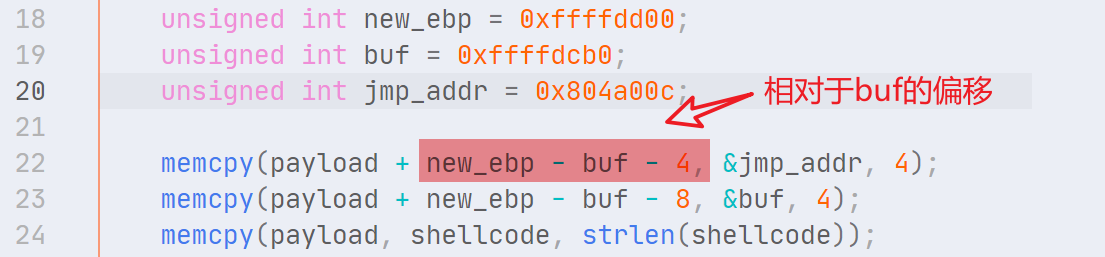
1. 控制指针***p***和变量***a***

反汇编***foo***函数可知指针***p***和变量***a***的地址分别为***ebp-0x4***以及***ebp-0x8***，因此通过对地址**0xffffdd00-0x4**和**0xffffdd00-8**的修改可以使得***p***和***a***变成我们想要的值



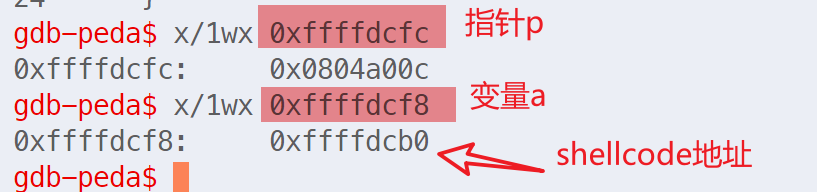
***p***和***a***的地址

由于这两个地址正好位于buf所在的地址空间，因此我们只需要对输入的参数进行简单的赋值即可



控制***p***和***a***

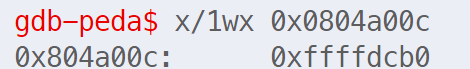
执行完***nstrcpy***函数之后，可以发现***p***和***a***的值已经被成功修改



被修改的***p***和***a***

1. 修改***\_exit(0)***函数的跳转地址

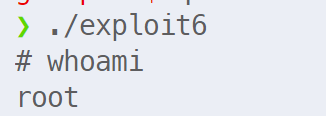
执行完***\*p=a***之后，可以发现跳转已经已经被修改成为shellcode地址



被修改的跳转地址

1. 执行漏洞利用程序

执行该漏洞利用程序，成功获取具有root权限的shell



获取具有root权限的shell