攻防世界reverse进阶easyre-153 writeup（#gdb调试父子进程、#ida版本差异）

原创

漫小牛 于 2021-05-10 15:31:20 发布 186 收藏 1

分类专栏： CTF Writeup 文章标签： 经验分享

本文链接：https://blog.csdn.net/weixin_43363675/article/details/116596059

文章目录

学习目标：

1）不同ida版本的反编译差异；
2）进程创建和进程间通信；
3）upx壳和脱壳；
4）查看进程id；
5）gdb调试父子进程；
6）静态分析的方法；
7）gdb动态调试的一般方法。

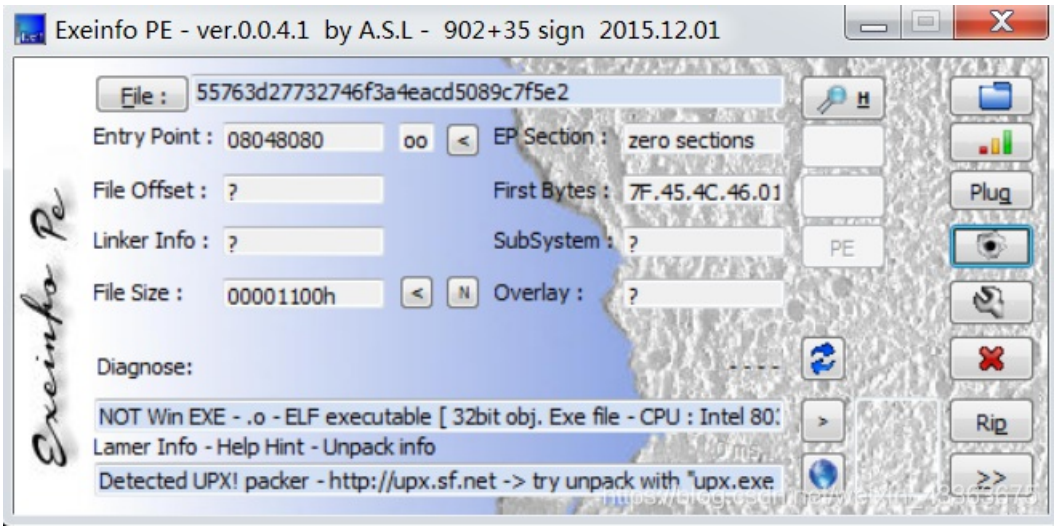
引言

这是攻防世界Reverse中的一道进阶题，从名称上看，是一道简单的逆向题。虽然，这道题中的解密算法比较简单，但涵盖的知识点比较多，有些小坑初学者很难绕过，是一道较为典型的逆向题。
点击附件1下载后，得到二进制文件。

第一步、查脱壳

1.查壳

使用Exeinfo PE查看二进制文件的类型，具体情况见下图：



结果显示，存在简单的UPX壳。

2.脱壳

使用upx脱壳工具进行脱壳，具体脱壳的过程见下图：

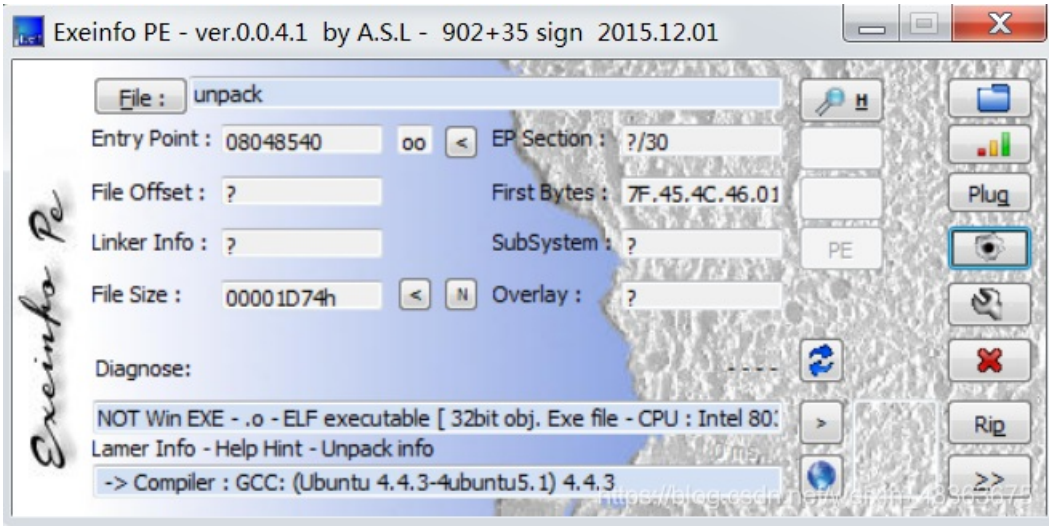
```
┌──(root💀kali)-[~/桌面]
└─# upx -d 55763d27732746f3a4eacd5089c7f5e2
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96        Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
      7540 <-      4352   57.72% linux.exec/i386 55763d27732746f3a4eacd5089c7f5e2

Unpacked 1 file.                                  https://blog.csdn.net/weixin_43363675
```

3.查看文件格式

再次使用Exeinfo PE查看该二进制文件，可看到已经脱掉了upx壳，该文件为32位的elf文件，使用GCC4.4.3进行编译。具体细节见下图：



第二步、IDA静态分析

1.IDA版本的小坑

新手到这里，可能会因为IDA版本的小坑而止步不前，原因是IDA并非每次均能给出准确的反编译伪代码，不同版本的IDA反编译出的伪代码可能会产生极大的差异。
在分析前，先给出核心函数lol反编译伪代码的差异，让大家有一个明显的对比后，再通过正确的版本继续分析。
下面是IDA 7.5版本生成的lol函数的反编译伪代码：

```
int lol()
{
  return printf("flag_is_not_here");
}
```

下面是IDA 6.8版本生成的lol函数的反编译伪代码：

```
int __cdecl lol(int a1)
{
  char v2; // [sp+15h] [bp-13h]@1
  char v3; // [sp+16h] [bp-12h]@1
  char v4; // [sp+17h] [bp-11h]@1
  char v5; // [sp+18h] [bp-10h]@1
  char v6; // [sp+19h] [bp-Fh]@1
  char v7; // [sp+1Ah] [bp-Eh]@1
  char v8; // [sp+1Bh] [bp-Dh]@1

  v2 = 2 * *(_BYTE *)(a1 + 1);
  v3 = *(_BYTE *)(a1 + 4) + *(_BYTE *)(a1 + 5);
  v4 = *(_BYTE *)(a1 + 8) + *(_BYTE *)(a1 + 9);
  v5 = 2 * *(_BYTE *)(a1 + 12);
  v6 = *(_BYTE *)(a1 + 18) + *(_BYTE *)(a1 + 17);
  v7 = *(_BYTE *)(a1 + 10) + *(_BYTE *)(a1 + 21);
  v8 = *(_BYTE *)(a1 + 9) + *(_BYTE *)(a1 + 25);
  return printf("flag_is_not_here");
}
```

通过上述两版本的对比看，6.8版本解析出了关键的解密代码，而7.5版本并未解析出这部分代码。实际上，这部分代码在IDA7.5的汇编窗口是存在的，也可通过与IDA 7.5交互进一步还原出这部分代码。在CTF比赛中，时间非常宝贵，建议安装至少两个版本的IDA，在一个版本一两分钟内无法分析出关键代码时直接换另一个版本反编译。
下面，放弃IDA 7.5的交互式优化，继续沿IDA 6.8版本静态分析。

2.分析main函数

下面main函数的反编译代码：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // eax@9
  int v4; // edx@9
  int v5; // [sp+18h] [bp-38h]@1
  int v6; // [sp+1Ch] [bp-34h]@2
  __pid_t v7; // [sp+20h] [bp-30h]@1
  int v8; // [sp+24h] [bp-2Ch]@4
  __int16 v9; // [sp+2Eh] [bp-22h]@4
  int v10; // [sp+4Ch] [bp-4h]@1

  v10 = *MK_FP(__GS__, 20);
  pipe(&v5);
  v7 = fork();
  if ( !v7 )
  {
    puts("\nOMG!!!! I forgot kid's id");
    write(v6, "6980087614356821435692875 3", 0x1Du);
    puts("Ready to exit     ");
    exit(0);
  }
  read(v5, &v9, 0x1Du);
  __isoc99_scanf("%d", &v8);
  if ( v8 == v7 )
  {
    if ( (*(_DWORD *)((char *)lol + 3) & 0xFF) == 204 )
    {
      puts(":D");
      exit(1);
    }
    printf("\nYou got the key\n ");
    lol((int)&v9);
  }
  wait(0);
  result = 0;
  v4 = *MK_FP(__GS__, 20) ^ v10;
  return result;
}
```

- L14 fork了一个进程后，父子进程将同时执行。为了进一步了解父子进程的执行路径，需知道fork函数的返回值的含义，否则分析将无法进行下去。

> fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：
> 1）在父进程中，fork返回新创建子进程的进程ID；
> 2）在子进程中，fork返回0；
> 3）如果出现错误，fork返回一个负值；

- 从上面fork函数的描述看，正常执行时父进程fork子进程，父进程返回子进程的进程ID，V7=子进程的ID；子进程中fork返回0。两种不同的返回值也决定了两个进程不同的执行路径。
- L15-L21为子进程的执行路径，通过进程间通信向管道中写入一串字符串"6980087614356821435692875 3"，之后子进程退出。
- L22之后为父进程执行的代码，L22从管道中读取字符串"6980087614356821435692875 3"，并接受用户的输入，当输入数据与子进程号相等时，会跳到L31 You got the key的位置。lol函数则认为是出现flag的位置。

3.分析lol函数

lol函数的代码为：

```
int __cdecl lol(int a1)
{
  char v2; // [sp+15h] [bp-13h]@1
  char v3; // [sp+16h] [bp-12h]@1
  char v4; // [sp+17h] [bp-11h]@1
  char v5; // [sp+18h] [bp-10h]@1
  char v6; // [sp+19h] [bp-Fh]@1
  char v7; // [sp+1Ah] [bp-Eh]@1
  char v8; // [sp+1Bh] [bp-Dh]@1

  v2 = 2 * *(_BYTE *)(a1 + 1);
  v3 = *(_BYTE *)(a1 + 4) + *(_BYTE *)(a1 + 5);
  v4 = *(_BYTE *)(a1 + 8) + *(_BYTE *)(a1 + 9);
  v5 = 2 * *(_BYTE *)(a1 + 12);
  v6 = *(_BYTE *)(a1 + 18) + *(_BYTE *)(a1 + 17);
  v7 = *(_BYTE *)(a1 + 10) + *(_BYTE *)(a1 + 21);
  v8 = *(_BYTE *)(a1 + 9) + *(_BYTE *)(a1 + 25);
  return printf("flag_is_not_here");
}
```

- 参数a1为子进程经进程间通信传入父进程，进一步传到该函数体的字符串"69800876143568214356928753"。
- L11-L17为7条简单的赋值语句，可认为是计算flag的过程。
- L18的printf语句打印了一行提示flag_is_not_here，这条提示实际是一条干扰项，引诱我们不要分析L11-L17的代码。
- 到这里，如果我们将L11-L17用c或python编写exp程序，已可直接解析出flag，此时这个writeup就完成了，但为了讲解gdb调试父子进程的知识点，绕过将来可能面临同时调试父子进程的小坑，我们再写一节，通过动态调试直接打印出flag。

第三步、gdb动态分析

我们的目的是直接跳到关键函数lol，这就需要在gdb中将控制流逐步引导到lol函数。

1.main函数反汇编

首先来看一下main函数的反汇编代码，在gdb中disas main，可显示如下汇编代码：

```
   0x080486e2 <+0>:     push   ebp
   0x080486e3 <+1>:     mov    ebp,esp
   0x080486e5 <+3>:     and    esp,0xfffffff0
   0x080486e8 <+6>:     sub    esp,0x50
   0x080486eb <+9>:     mov    eax,gs:0x14
   0x080486f1 <+15>:    mov    DWORD PTR [esp+0x4c],eax
   0x080486f5 <+19>:    xor    eax,eax
   0x080486f7 <+21>:    lea    eax,[esp+0x18]
   0x080486fb <+25>:    mov    DWORD PTR [esp],eax
   0x080486fe <+28>:    call   0x8048520 <pipe@plt>
   0x08048703 <+33>:    call   0x8048510 <fork@plt>
   0x08048708 <+38>:    mov    DWORD PTR [esp+0x20],eax
   0x0804870c <+42>:    cmp    DWORD PTR [esp+0x20],0x0
   0x08048711 <+47>:    jne    0x8048753 <main+113>
   0x08048713 <+49>:    mov    DWORD PTR [esp],0x80488d4
   0x0804871a <+56>:    call   0x8048500 <puts@plt>
   0x0804871f <+61>:    mov    eax,DWORD PTR [esp+0x1c]
   0x08048723 <+65>:    mov    DWORD PTR [esp+0x8],0x1d
   0x0804872b <+73>:    mov    DWORD PTR [esp+0x4],0x80488ef
```

```
0x08048733 <+81>:    mov    DWORD PTR [esp],eax
0x08048736 <+84>:    call   0x8048490 <write@plt>
0x0804873b <+89>:    mov    DWORD PTR [esp],0x804890b
0x08048742 <+96>:    call   0x8048500 <puts@plt>
0x08048747 <+101>:   mov    DWORD PTR [esp],0x0
0x0804874e <+108>:   call   0x8048530 <exit@plt>
0x08048753 <+113>:   mov    eax,DWORD PTR [esp+0x18]
0x08048757 <+117>:   mov    DWORD PTR [esp+0x8],0x1d
0x0804875f <+125>:   lea    edx,[esp+0x2e]
0x08048763 <+129>:   mov    DWORD PTR [esp+0x4],edx
0x08048767 <+133>:   mov    DWORD PTR [esp],eax
0x0804876a <+136>:   call   0x80484c0 <read@plt>
0x0804876f <+141>:   mov    eax,0x804891e
0x08048774 <+146>:   lea    edx,[esp+0x24]
0x08048778 <+150>:   mov    DWORD PTR [esp+0x4],edx
0x0804877c <+154>:   mov    DWORD PTR [esp],eax
0x0804877f <+157>:   call   0x80484f0 <__isoc99_scanf@plt>
0x08048784 <+162>:   mov    eax,DWORD PTR [esp+0x24]
0x08048788 <+166>:   cmp    eax,DWORD PTR [esp+0x20]
0x0804878c <+170>:   jne    0x80487d5 <main+243>
0x0804878e <+172>:   mov    eax,0x80485f4
0x08048793 <+177>:   add    eax,0x3
0x08048796 <+180>:   mov    eax,DWORD PTR [eax]
0x08048798 <+182>:   and    eax,0xff
0x0804879d <+187>:   cmp    eax,0xcc
0x080487a2 <+192>:   jne    0x80487bc <main+218>
0x080487a4 <+194>:   mov    DWORD PTR [esp],0x8048921
0x080487ab <+201>:   call   0x8048500 <puts@plt>
0x080487b0 <+206>:   mov    DWORD PTR [esp],0x1
0x080487b7 <+213>:   call   0x8048530 <exit@plt>
0x080487bc <+218>:   mov    eax,0x8048924
0x080487c1 <+223>:   mov    DWORD PTR [esp],eax
0x080487c4 <+226>:   call   0x80484d0 <printf@plt>
0x080487c9 <+231>:   lea    eax,[esp+0x2e]
0x080487cd <+235>:   mov    DWORD PTR [esp],eax
0x080487d0 <+238>:   call   0x80485f4 <lol>
0x080487d5 <+243>:   mov    DWORD PTR [esp],0x0
0x080487dc <+250>:   call   0x80484b0 <wait@plt>
0x080487e1 <+255>:   mov    eax,0x0
0x080487e6 <+260>:   mov    edx,DWORD PTR [esp+0x4c]
0x080487ea <+264>:   xor    edx,DWORD PTR gs:0x14
0x080487f1 <+271>:   je     0x80487f8 <main+278>
0x080487f3 <+273>:   call   0x80484e0 <__stack_chk_fail@plt>
0x080487f8 <+278>:   leave
0x080487f9 <+279>:   ret
```

关键位置有：

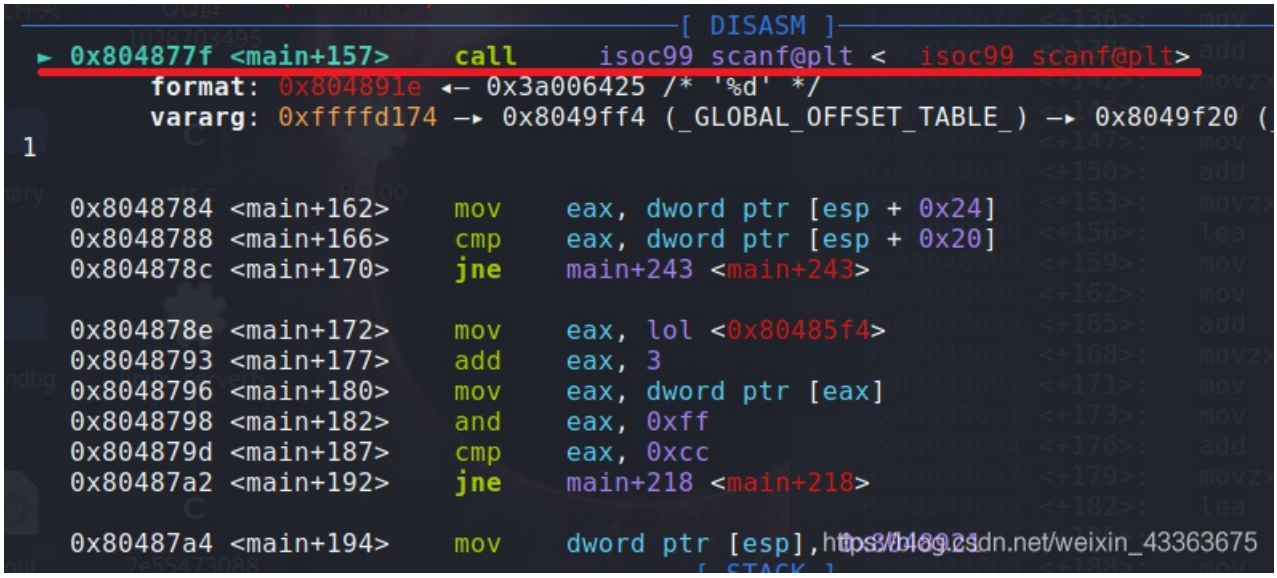
- 0x08048703 <+33>的fork子进程；
- 0x0804877f <+157>scanf函数需输入子进程进程id；
- 0x0804878c <+170>将输入的数字与子进程的id号比较；
- 0x080487d0 <+238>关键解码函数lol。

2.动态调试父子进程

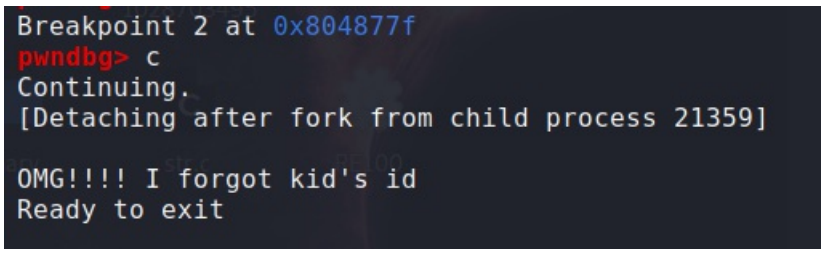
- gdb动态调试时，在没有fork子进程时，是对当前进程，也即父进程的调试。这里有一个小坑，就是fork子进程后，会出现两个同时运行的进程，这时，在一个gdb命令行中，如何进入到指定的进程进行调试呢？

```
#命令设置gdb在fork之后跟踪子进程
set follow-fork-mode child
#命令设置gdb在fork之后跟踪父进程
set follow-fork-mode parent
```

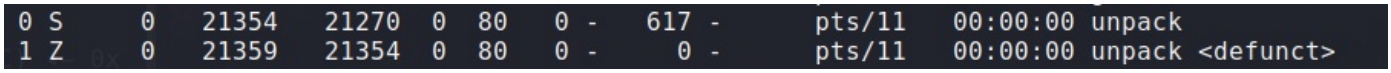
- gdb动态调试时，创建子进程后，会默认进入子进程，为了在父进程调试需要在gdb中，fork之前，输入set follow-fork-mode parent跟踪父进程。
- b main设置断点，输入r运行后，输入set follow-fork-mode parent命令。
- 在scanf的位置设置断点b *80x0804877f，按c继续执行并跳转到scanf断点处，见下图：



- 到scanf断点表示当前已进入父进程进行调试，那么在父进程在执行到scanf之前，子进程的行为又如何呢？下图所示为子进程打印的字符串，可知子进程执行过了将字符串写入管道的语句。



- 为调试lol函数，我们先在main函数中lol函数的调用点设置断点b *0x080487d0，按c继续执行，这是需要输入子进程的id，通过在linux终端输入ps -al命令可查找子进程的id号，也可在gdb中向上翻，找到子进程创建时打印的id号。



- 上图中的两个进程即为父进程和子进程，第二条记录中的pid 21359是当前进程的id号，后面的ppid 21354为21359的父进程，因此，子进程的进程号为21359。输入该id号后，停在lol的调用点位置。

3.动态调试lol函数

输入disas lol查看反汇编代码：

```
0x080485f4 <+0>:     push   ebp
0x080485f5 <+1>:     mov    ebp,esp
0x080485f7 <+3>:     sub    esp,0x28
0x080485fa <+6>:     mov    eax,DWORD PTR [ebp+0x8]
0x080485fd <+9>:     add    eax,0x1
0x08048600 <+12>:    movzx  eax,BYTE PTR [eax]
0x08048603 <+15>:    mov    edx,eax
0x08048605 <+17>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048608 <+20>:    add    eax,0x1
0x0804860b <+23>:    movzx  eax,BYTE PTR [eax]
0x0804860e <+26>:    lea    eax,[edx+eax*1]
0x08048611 <+29>:    mov    BYTE PTR [ebp-0x13],al
0x08048614 <+32>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048617 <+35>:    add    eax,0x4
0x0804861a <+38>:    movzx  eax,BYTE PTR [eax]
0x0804861d <+41>:    mov    edx,eax
0x0804861f <+43>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048622 <+46>:    add    eax,0x5
0x08048625 <+49>:    movzx  eax,BYTE PTR [eax]
0x08048628 <+52>:    lea    eax,[edx+eax*1]
0x0804862b <+55>:    mov    BYTE PTR [ebp-0x12],al
0x0804862e <+58>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048631 <+61>:    add    eax,0x8
0x08048634 <+64>:    movzx  eax,BYTE PTR [eax]
0x08048637 <+67>:    mov    edx,eax
0x08048639 <+69>:    mov    eax,DWORD PTR [ebp+0x8]
0x0804863c <+72>:    add    eax,0x9
0x0804863f <+75>:    movzx  eax,BYTE PTR [eax]
0x08048642 <+78>:    lea    eax,[edx+eax*1]
0x08048645 <+81>:    mov    BYTE PTR [ebp-0x11],al
0x08048648 <+84>:    mov    eax,DWORD PTR [ebp+0x8]
0x0804864b <+87>:    add    eax,0xc
0x0804864e <+90>:    movzx  eax,BYTE PTR [eax]
0x08048651 <+93>:    mov    edx,eax
0x08048653 <+95>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048656 <+98>:    add    eax,0xc
0x08048659 <+101>:   movzx  eax,BYTE PTR [eax]
```

```
0x0804865c <+104>:   lea    eax,[edx+eax*1]
0x0804865f <+107>:   mov    BYTE PTR [ebp-0x10],al
0x08048662 <+110>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048665 <+113>:   add    eax,0x12
0x08048668 <+116>:   movzx  eax,BYTE PTR [eax]
0x0804866b <+119>:   mov    edx,eax
0x0804866d <+121>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048670 <+124>:   add    eax,0x11
0x08048673 <+127>:   movzx  eax,BYTE PTR [eax]
0x08048676 <+130>:   lea    eax,[edx+eax*1]
0x08048679 <+133>:   mov    BYTE PTR [ebp-0xf],al
0x0804867c <+136>:   mov    eax,DWORD PTR [ebp+0x8]
0x0804867f <+139>:   add    eax,0xa
0x08048682 <+142>:   movzx  eax,BYTE PTR [eax]
0x08048685 <+145>:   mov    edx,eax
0x08048687 <+147>:   mov    eax,DWORD PTR [ebp+0x8]
0x0804868a <+150>:   add    eax,0x15
0x0804868d <+153>:   movzx  eax,BYTE PTR [eax]
0x08048690 <+156>:   lea    eax,[edx+eax*1]
0x08048693 <+159>:   mov    BYTE PTR [ebp-0xe],al
0x08048696 <+162>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048699 <+165>:   add    eax,0x9
0x0804869c <+168>:   movzx  eax,BYTE PTR [eax]
0x0804869f <+171>:   mov    edx,eax
0x080486a1 <+173>:   mov    eax,DWORD PTR [ebp+0x8]
0x080486a4 <+176>:   add    eax,0x19
0x080486a7 <+179>:   movzx  eax,BYTE PTR [eax]
0x080486aa <+182>:   lea    eax,[edx+eax*1]
0x080486ad <+185>:   mov    BYTE PTR [ebp-0xd],al
0x080486b0 <+188>:   mov    DWORD PTR [ebp-0xc],0x0
0x080486b7 <+195>:   cmp    DWORD PTR [ebp-0xc],0x1
0x080486bb <+199>:   jne    0x80486d3 <lol+223>
0x080486bd <+201>:   mov    eax,0x80488c0
0x080486c2 <+206>:   lea    edx,[ebp-0x13]
0x080486c5 <+209>:   mov    DWORD PTR [esp+0x4],edx
0x080486c9 <+213>:   mov    DWORD PTR [esp],eax
0x080486cc <+216>:   call   0x80484d0 <printf@plt>
0x080486d1 <+221>:   jmp    0x80486e0 <lol+236>
0x080486d3 <+223>:   mov    eax,0x80488c3
0x080486d8 <+228>:   mov    DWORD PTR [esp],eax
0x080486db <+231>:   call   0x80484d0 <printf@plt>
0x080486e0 <+236>:   leave
0x080486e1 <+237>:   ret
```

分析及调试的大致情况为：

- 0x080485fa <+6>到0x080486ad <+185>为ida中7条c伪代码对应的汇编语句，字符数组按字节相加的操作采用了更简洁的lea指令，可见gcc编译器在接近机器码的中间表示时，进行了指令优化。
- 从0x080485fa <+6>到0x080486ad <+185>汇编指令中写入的地址看，为ebp-0x13到ebp-0xd共7个字节。
- 在0x080486b0设置断点b *0x080486b0，按c继续执行到该断点，在gdb中输入x/s $ebp-0x13即可打印出该flag。