（已弃坑，请看i春秋月刊 Linux pwn入门，地址见文章）鬼哥手把手带你入门栈溢出(0)：了解基本原理

原创

随缘懂点密码学 于 2020-01-13 13:42:55 发布 277 收藏 1
分类专栏： pwn学习 # 栈溢出 文章标签： 栈溢出原理

本文链接：https://blog.csdn.net/qq_42667481/article/details/103952553


pwn学习 同时被 2 个专栏收录
1 篇文章 0 订阅
订阅专栏


栈溢出
1 篇文章 0 订阅
订阅专栏

由于本人水平较有限，并且发现了一个很好的教程，故弃坑。

i春秋月刊的Linux pwn 入门 -》 https://bbs.ichunqiu.com/thread-47090-1-1.html

目录

基础知识准备

汇编基础

首先要对汇编有一定基础，**下面的，尤其加粗部分必须懂（其他的遇到了，现学也行）**

常用指令：**mov lea push pop**

add sub inc dec and or not **xor**

call ret jmp cmp jz ...

寄存器（eax ebx ecx edx **esp ebp eip** eflags cs ds **...)**

C语言

不说了，遇到不懂的函数现查就行

gdb-peda调试

最简单的调试要会

break(b)

run(r)

continue(c)

next(n)

ni ：下一句汇编

step(s) ：单步步入

si:汇编单步步入（可以看到函数中push ebp mov ebp,esp add esp xxh）

stack：看栈用，比如 stack 20

x/<n/f/u> <addr> ：按照格式查看指定的内存地址

----------正文从这里开始----------

了解函数的栈帧布局、调用及返回过程

环境：ubuntu16.04（64位）

调试工具：gdb-peda

编码

写一个测试程序(stack_frame.c),代码如下：

```
#include <stdio.h>

int add(int a,int b){
    int result = 0;
    result = a+b;
    return result;
}

int main(int argc,char *argv[])
{
    int a = 0x41;
    int b = 0x42;
    int tmp = 0;

    tmp = add(a,b);
    printf("result = %d\n",tmp);

    return 0;
}
```

编译：

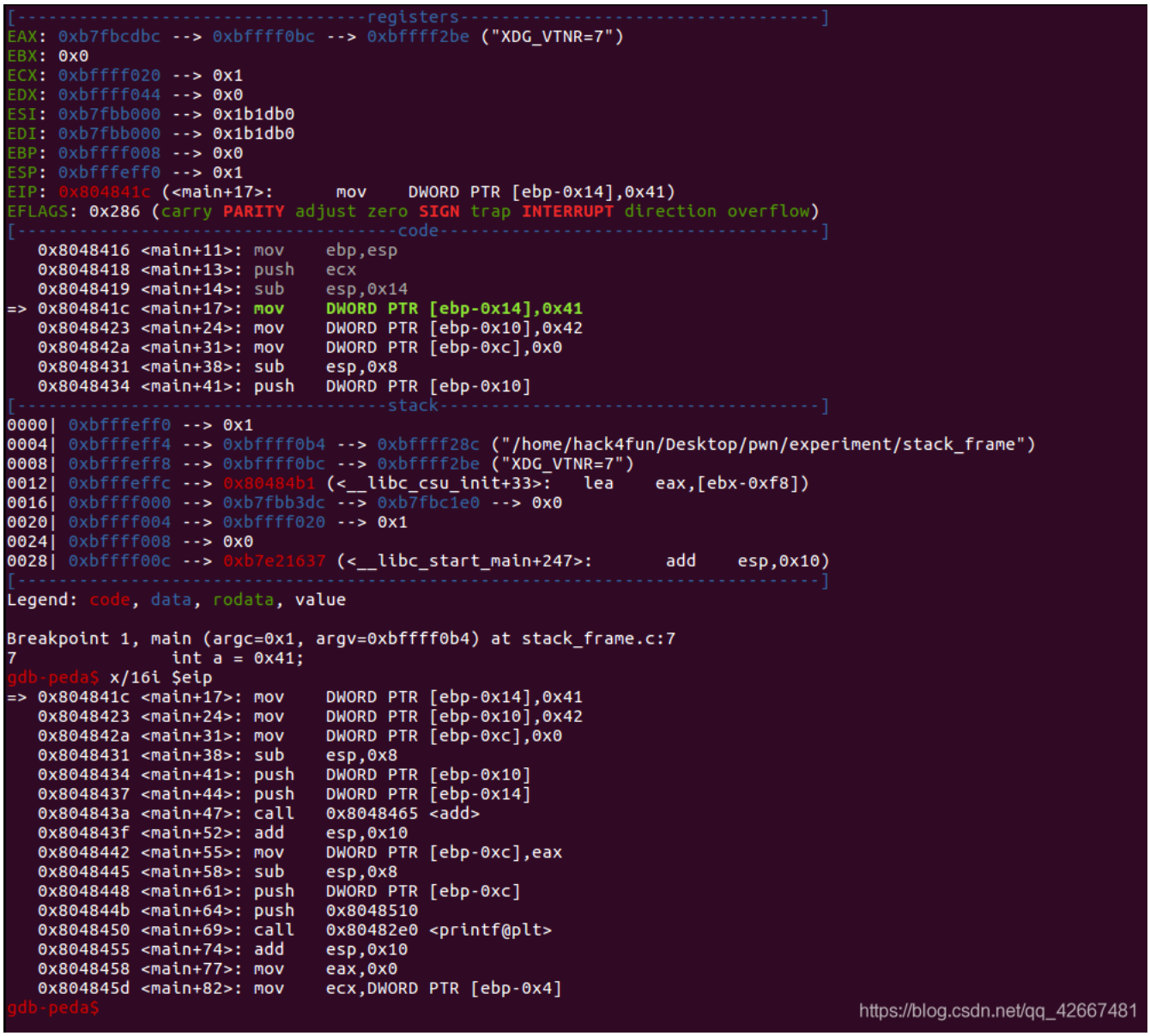
```
gcc -g -o stack_frame stack_frame.c #带有调试信息
```

调试（可跳过看总结，建议自行调试加深理解）

```
gdb stack_frame

b main

r
```



main函数局部变量初始化

三条指令很明显是在给main函数局部变量初始化，**注意观察变量在栈中的位置（与ebp和esp相对位置）**

查看更多指令:

```
x/16i $eip
```

继续调试，观察参数压入栈中的顺序和位置

步入call add，并查看保存原ebp，调整ebp和esp的过程：

```
si
ni
```

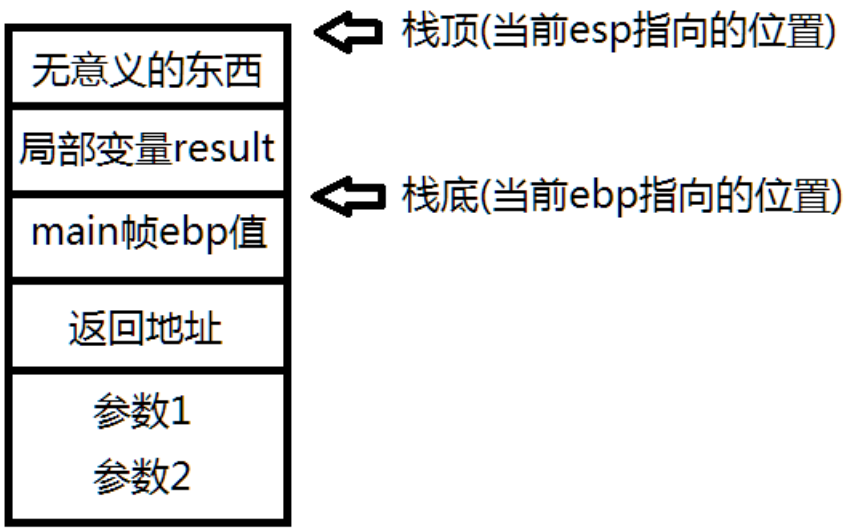
继续执行直到leave，可以看到

```
[----------------------------------registers-----------------------------------]
EAX: 0x83
EBX: 0x0
ECX: 0xbffff020 --> 0x1
EDX: 0x41 ('A')
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbfffefd8 --> 0xbffff008 --> 0x0
ESP: 0xbfffefc8 --> 0xb7e15dc8 --> 0x2b76 ('v+')
EIP: 0x8048480 (<add+27>:       leave)
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048478 <add+19>:  add    eax,edx
   0x804847a <add+21>:  mov    DWORD PTR [ebp-0x4],eax
   0x804847d <add+24>:  mov    eax,DWORD PTR [ebp-0x4]
=> 0x8048480 <add+27>:  leave
   0x8048481 <add+28>:  ret
   0x8048482:   xchg   ax,ax
   0x8048484:   xchg   ax,ax
   0x8048486:   xchg   ax,ax
[-------------------------------------stack------------------------------------]
0000| 0xbfffefc8 --> 0xb7e15dc8 --> 0x2b76 ('v+')
0004| 0xbfffefcc --> 0xb7fd51b0 --> 0xb7e09000 --> 0x464c457f
0008| 0xbfffefd0 --> 0x8000
0012| 0xbfffefd4 --> 0x83
0016| 0xbfffefd8 --> 0xbffff008 --> 0x0
0020| 0xbfffefdc --> 0x804843f (<main+52>:      add    esp,0x10)
0024| 0xbfffefe0 --> 0x41 ('A')
0028| 0xbfffefe4 --> 0x42 ('B')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
22      }
gdb-peda$
```

当前栈顶

当前栈底

此时，add函数栈的结构如下：



执行leave指令后（相当于mov esp,ebp　pop ebp 这两条)，栈顶为返回地址，ebp恢复为main栈帧的ebp

再执行ret指令（相当于pop eip），返回到main的下一条执行。

总结

int add(int a,int b)的函数调用和返回流程（对于采用cdecl调用方式的函数而言）

1.在main函数中

(1)从右往左压入参数：

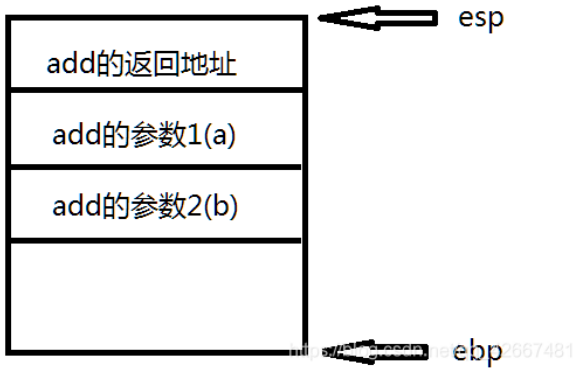
push b

push a

(2)将返回地址入栈并转向add：

call add

此时栈帧如下



2.在add函数中

(1)保存main栈帧的ebp：

push ebp

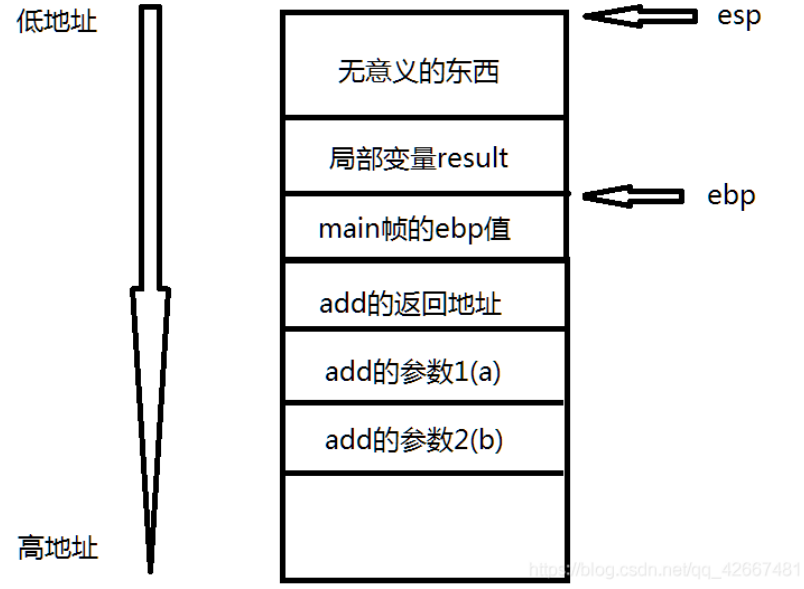
(2)在main栈帧的上面建立add的栈帧：

mov ebp,esp

sub esp,0x10 ; 开辟一段空间，大于等于局部变量占用的内存，具体大小由编译器决定

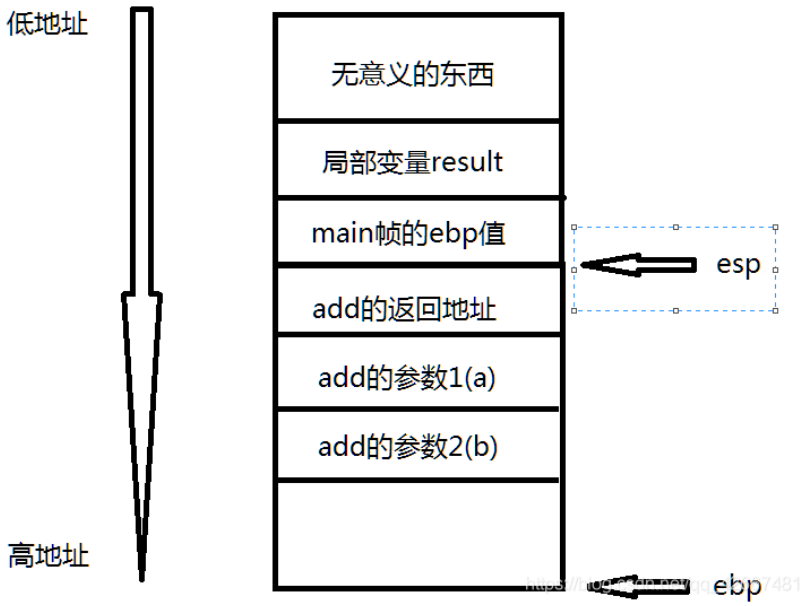
(3)执行函数语句

此时栈帧如下



(4)销毁栈帧：

leave (mov esp,ebp pop ebp)



(5)返回：

ret (相当于pop eip，这一步会将返回地址从栈里取出)

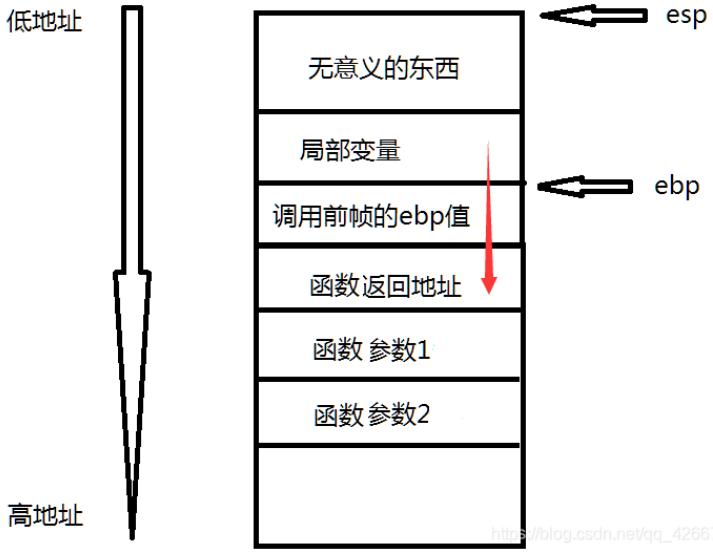
3.回到main函数中

(1)进行堆栈平衡（恢复main栈帧中 原esp值）：

add esp,0x10

(2)继续执行

栈溢出原理



如果局部变量中有字符数组，输入时不检查越界，那么写入足够数量的字符便可修改函数返回地址

在当前函数执行ret指令的时候，便能控制eip的指向，控制程序执行的流程