缓冲区溢出攻击实验(一)

原创

Venscor 于 2015-11-18 20:43:27 发布 17731 收藏 55
分类专栏： 安全攻防 文章标签： 内存溢出 内存布局 改变程序流程 任意代码执行
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/u010651541/article/details/49849557
版权

安全攻防 专栏收录该内容
3 篇文章 0 订阅
订阅专栏

无聊之余，想弄一下缓冲区溢出实验，之前一直听说这个，也没有亲自动手做一下，发现真正弄起来的时

一、基础知识

这一部分主要是关于程序内存布局相关的知识，也涉及少量at&t汇编、gdb调试、gcc编译的知识。再

1、at&t汇编

决定弄这个缓冲区实验之前，我还是花了10来天把汇编语言学习了一下，主要看的是王爽编写的《汇编语言

AT&T汇编的细节不在这里解释，主要可以参看资料：相关资料及全部源代码这是后面我的源码下载地址，同时把资料上传了上去。但是，这里还是说一下几个特殊的语句，也就是我很费解的部分；

lea src ,add指令：手册是说的是传送地址，简单一句话，我没有理解，一个博文上写的很好：地址是：深入理解计算机系统（3.4）---算数与逻辑运算指令详解，我还是复述一下上面的内容：

lea 4(%edx,%ex,4),%eax和movl 4(%edx,%edx,4),%eax的区别： “假设%edx寄存器的值为x的话，那么lea 4(%edx,%ex,4),%eax这条指令的作用就是将 4 + x + 4x = 5x + 4赋给%eax寄存器。它和mov指令的区别就在于，假设是movl 4(%edx,%edx,4),%eax这个指令，它的作用是将内存地址为5x+4的内存区域的值赋给%eax寄存器，而leal指令只是将5x+4这个地址赋给目的操作数%eax而已，它并不对存储器进行引用的值的计算。”上面的博文中还有详尽的图文描述，为了方便，我一并摘下；假设执行指令之前，寄存器和存储器的状态如图1，则执行了movl 4(%edx,%edx,4),%eax后，寄存器和存储器的状态如图2，而执行了lea指令后，寄存器和存储器的状态如图3；

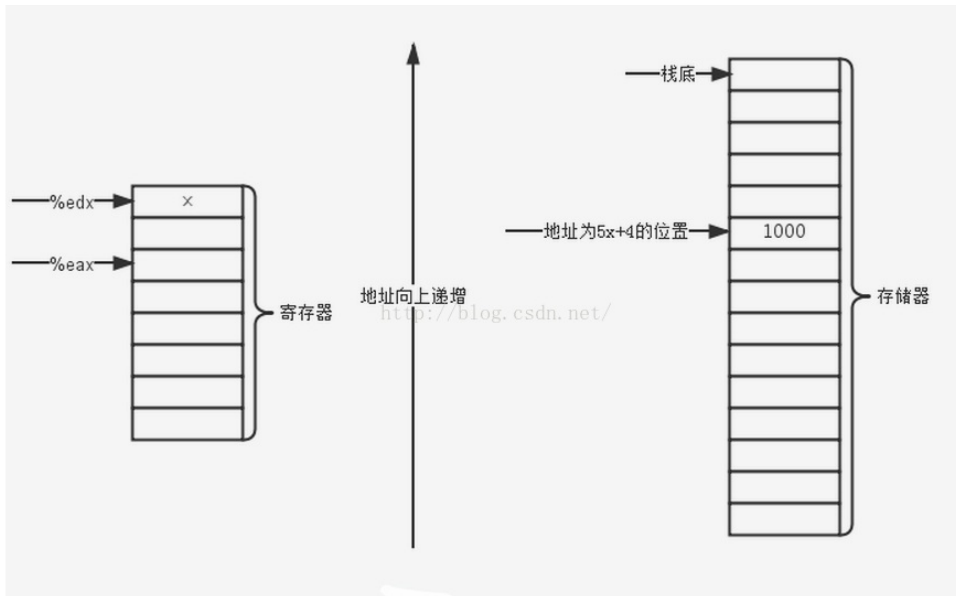


图1：执行前

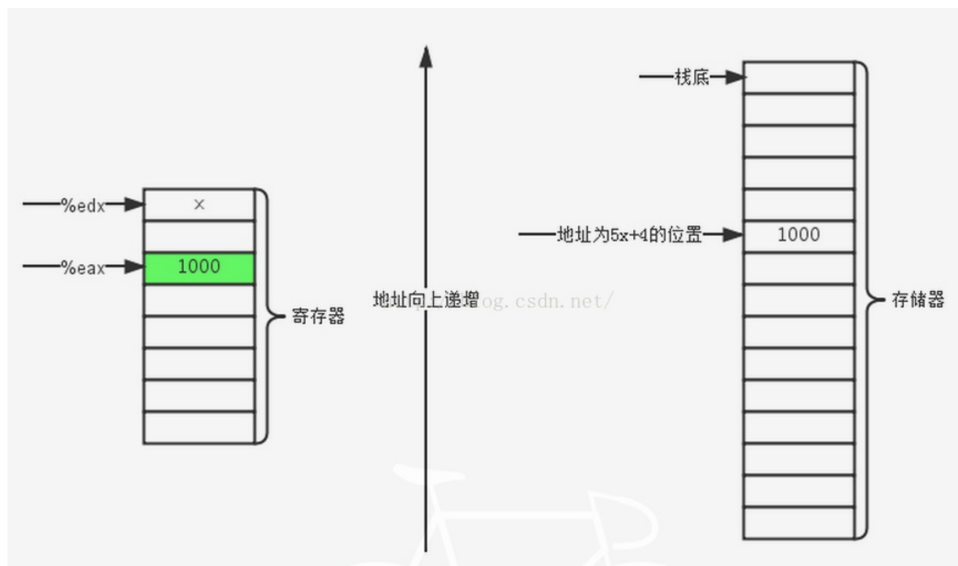


图2：执行了move

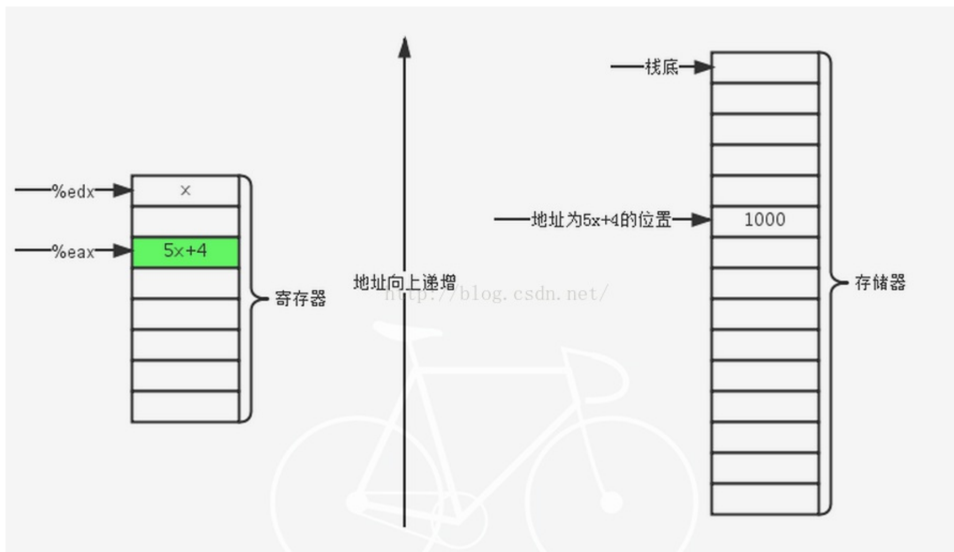


图3：执行了lea

好了，汇编知识只记录这么一点，后面用的汇编只能会给出解释或者注释；

2、gdb调试知识

以前基本没有在Linux开发的经验，写程序都是IDE中进行的，所以没有gdb调试的经历；这里用到的gdb知识

3、gcc编译相关知识：

gcc编译有一堆选项，具体每个选项的作用不在这里意义罗列，其实我也不怎么清楚，知识现用现查，这里
个后面用的编译选项；

-fstack-protector	启用堆栈保护,不过只为局部变量中含有 char 数组的函数插入保护代码
-fstack-protector-all	启用堆栈保护，为所有函数插入保护代码
-fno-stack-protector	禁用堆栈保护
-O[n]	启用优化

4、还涉及一些gcc中的一些栈保护话题，后面再进行介绍；

5、程序内存布局

程序加载到cup执行过程中，会对程序的不同部分放入不同的地方，内存布局如图4；集体用法不做介绍，可
相关资料，值得提出的是，本文所说的缓冲区溢出攻击针对的是栈区；栈：存放程序形参和局部变量。

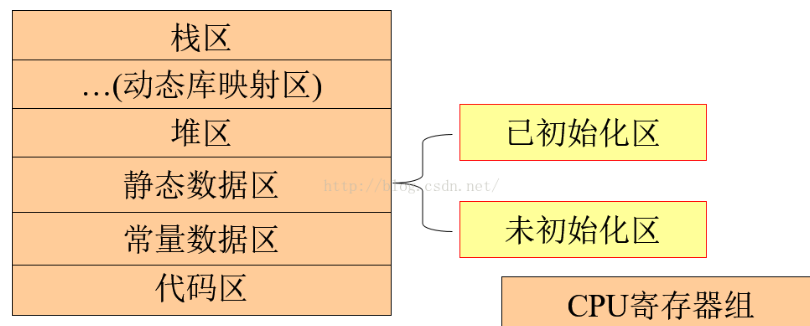


图4：程序内存布局

下图给出了一个函数调用过程的例子，函数调用过程的参数传递、返回地址压栈、以及返回时的现场恢复都
从图5和图6得到一些启示；这里不做具体说明：

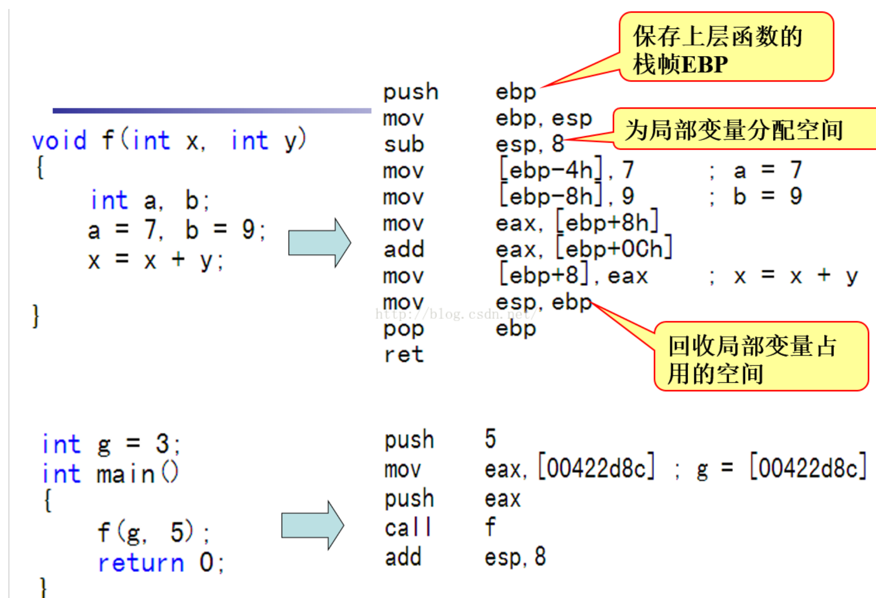


图5:函数调用图

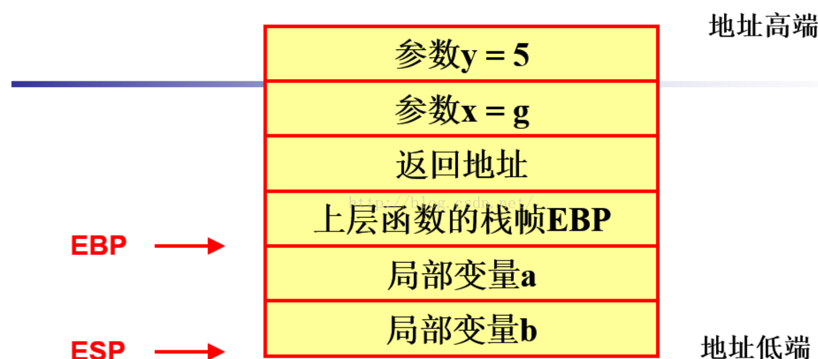


图6：函数调用栈变化

二、攻击过程详解

1、如何获取函数返回地址

缓冲区溢出攻击：由于c/c++对数据没有越界检查，对于函数中的局部变量是存储在栈中的，如图6所示；如函数中的局部变量中存在数据；那么攻击者可以尝试向该数据中写入超过数组长度的数据，这样就会造成数据越界，而没有做数据越界检查的程序并不能发觉这一点。于是溢出的数据会继续“入栈”，只要设计好写入的数据，就可以覆盖原先程序中的其他局部变量和ebp寄存器(图6)以及调用前已经压栈的返回地址。这样，如果设计写入的数据，使得“被覆盖的返回地址”栈处的内容为“指向恶意程序的地址(或shell)”，函数返回时将不再返回到原先调用它的语句的下一条语句处，而是返回到了恶意程序地址处，这样，恶意程序就得到了执行或者是获得了shell。那么，重要的就是获得栈中存储“返回地址”的单元的地址，然后修改这个返回地址，就可以导致程序执行流程发生改变。那么怎么才能获得存储“返回地址”栈单元的地址？下面以一个有漏洞的demo进行研究；

程序1：example1.c

```
#include<stdio.h>
void function(int a,int b,int c){
 char buffer1[5];
 char buffer2[10];
 int *ret;
 ret=(int*)(buffer1+13);
 (*ret)+=8;
}

void main(){
 int x;
 x=0;
 function(1,2,3);
 x=1;
 printf("%d\n",x);
}
```

编译：gcc -g -O0 -fno-stack-protector -o example1 example1.c，对程序进行反汇编，得带汇编代码如下：

```
0x0804843a <+0>:     push   %ebp
0x0804843b <+1>:     mov    %esp,%ebp
0x0804843d <+3>:     and    $0xfffffff0,%esp
0x08048440 <+6>:     sub    $0x20,%esp
0x08048443 <+9>:     movl   $0x0,0x1c(%esp)
0x0804844b <+17>:    movl   $0x3,0x8(%esp)
0x08048453 <+25>:    movl   $0x2,0x4(%esp)
0x0804845b <+33>:    movl   $0x1,(%esp)
0x08048462 <+40>:    call   0x804841c <function>
0x08048467 <+45>:    movl   $0x1,0x1c(%esp)
0x0804846f <+53>:    mov    0x1c(%esp),%eax
0x08048473 <+57>:    mov    %eax,0x4(%esp)
0x08048477 <+61>:    movl   $0x8048520,(%esp)
0x0804847e <+68>:    call   0x80482f0 <printf@plt>
0x08048483 <+73>:    leave
0x08048484 <+74>:    ret
```

http://blog.csdn.net/

图7：main

```
0x0804841c <+0>:     push   %ebp
0x0804841d <+1>:     mov    %esp,%ebp
0x0804841f <+3>:     sub    $0x20,%esp
0x08048422 <+6>:     lea    -0x9(%ebp),%eax
0x08048425 <+9>:     add    $0xd,%eax
0x08048428 <+12>:    mov    %eax,-0x4(%ebp)
0x0804842b <+15>:    mov    -0x4(%ebp),%eax
0x0804842e <+18>:    mov    (%eax),%eax
0x08048430 <+20>:    lea    0x8(%eax),%edx
0x08048433 <+23>:    mov    -0x4(%ebp),%eax
0x08048436 <+26>:    mov    %edx,(%eax)
0x08048438 <+28>:    leave
0x08048439 <+29>:    ret
```

http://blog.csdn.net/

图8：function

从example1.c源文件中可以看到，function()函数申请了三个局部变量buffer1[5]、buffer2[10]、ret，根据前面的程序内存布局方面的知识，这三个变量是存储在程序的栈空间的,并且存在数组，那么就是存在被攻击的风险的，这里先不讨论工具细节，我们映入了ret变量，为指向int形的指针；后面我们通过分析把ret指针指向function()函数的返回地址；通过改变ret指向的内存单元(也就是栈中存储的function的返回地址)的内容，就可以改变程序的执行流程。

我们先看看程序正常执行情况下，main()函数中的printf()打印的x值为多少，很明显：1；然而我么需要通过ret变量来改变程序的执行流程，使得function()函数执行之后，x=1这一条语句跳过来说明我们修改了function的返回地址；具体来说，就是通过下面的语句改变了程序的执行流程；

```
ret=(int*)(buffer1+13);
```

我们先暂时忽略程序为什么上面这条语句改变了程序执行流程；假定我们已经会得了程序的返回地址之后，怎么使得x=1这一条语句执行的时候被跳过；

根据程序内存布局方面的知识，在function()函数返回的时候，是需要"恢复现场"的，即从栈里面取出function()h函数下一条指令的地址，然后程序执行流程跳转到此地址处，对用的汇编解释就是：pop eax；mov eax,eip(注意只是解释，并没有这样的指令)；那么要让程序跳过eax，只需知道x=1；下一条语句的地址，然后把它写入ret指向的内存单元即可。

从main()函数的汇编可以看到，执行call指令后，压入栈中的地址应该是：0x08048067(图7)，而x=1对应的关键汇编指令为：movl $0x1,0x1c(esp);那么我们把栈中返回地址(即ret中)的内容改为movl $0x1,0x1c(esp)指令的一条语句对应的地址即可，当然程序可以这么写：(*ret)=0x0804806f,为了避免硬编码我们采用：

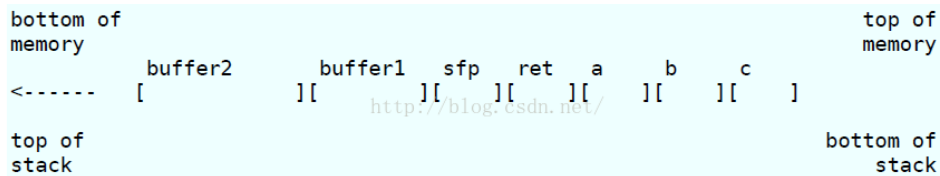
```
(*ret)+=8;
```

达到和上面语句一样的效果；
接下来解释为什么ret指向了程序的返回地址，也就是上面example1.c中最关键的一条代码；

按照刚学的西电李老师《Linux内核分析》课上的内容(图6);程序返回地址相对buffer1的地址偏移量量应该是：buffer1的大小+放ebp栈单元的大小；根据4字节对齐规则，就是8+4=12；所以返回地址ret应该是这样的：

```
ret=buffer1+12;
```

但是经过试验发现，并不是这样的。



没办法，只能打开看function()的汇编来寻找到底ret相对buffer的偏移是多少，奈何汇编新手，看半天试半天还是分析错了，好吧，上必杀技gdb，最终得到ret相对buffer的地址偏移是：13。干！居然不是4的倍数，应该gcc做了什么优化，那就是接受它吧。这样，有了这条语句来获得function()函数的返回地址：

```
ret=(int*)(buffer1+13);
```

默默的试验下，结果如下：

这下结果对了，返回地址顺利get，并且还修改了返回地址。虽然结果对了，怎么止于此，还得看看汇编function的汇编：

```
lea -0x9(%ebp),%eax
add $0xd,%eax
```

由于function()源代码中有一行：

```
ret=(int*)(buffer1+13);
```

再看上面汇编第二行，瞬间对应，再看第一行：瞬间知道eax相对ebp的偏移量-9，所以return相对buffer1偏移量：9+4=13；为毛当时没看出来！！！

好了，这部分先写到这，下面就是要研究return地址应该指向什么样的代码，从实验角度，我们恶意代码就是获得shell。啰嗦了这么多，只怪自己太弱，只能写详尽点了。。。