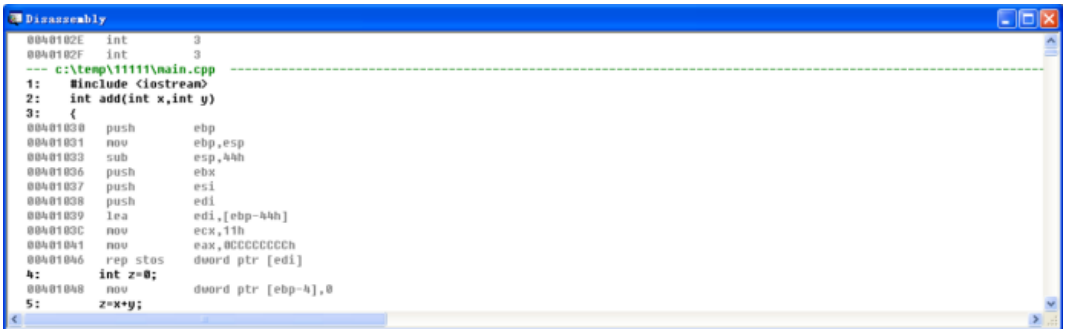
《漏洞利用及渗透测试基础》第二次实验报告

1811463 赵梓杰 信息安全

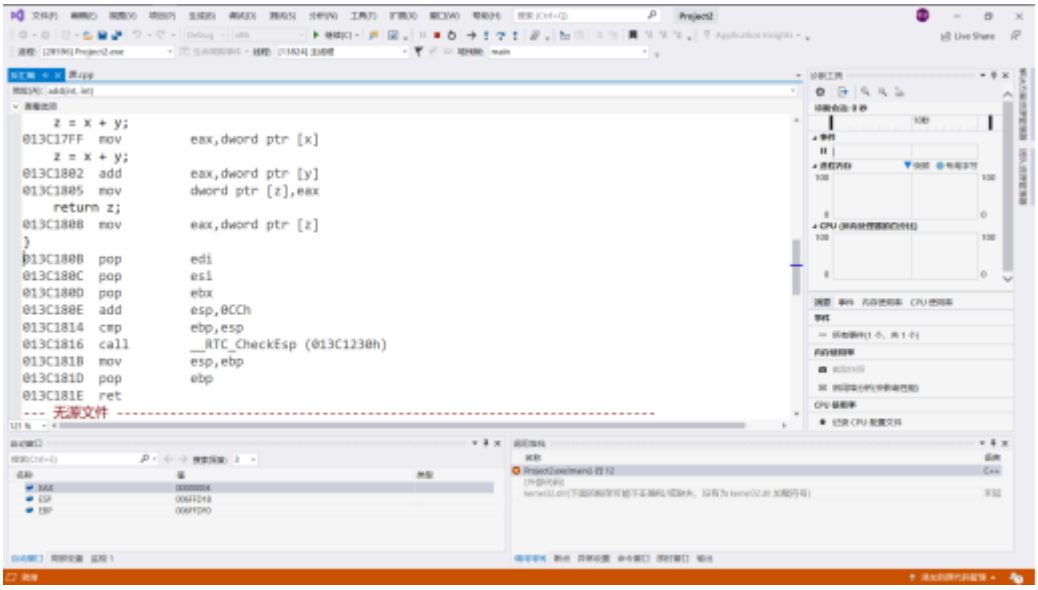
- 实验内容
  - 在XP系统的VC6下调试课本的示例程序并将call语句执行过程中EIP变化、ESP、EBP变化等状态进行记录，解释变化的主要原因。
  - 相同的一个程序，在非XP系统的当前win10操作系统中运行，绘制add函数执行时的栈状态，并与XP下同时刻的状态进行比较。
- 实验步骤
  - 在XP系统的VC6下调试以下程序代码

```
#include <iostream>
int add(int x,int y)
{
    int z=0;
    z=x+y;
    return z;
}
void main()
{
    int n=0;
    n=add(1,3);
    printf("%d\n",n);
}
```

  单步调试后右键代码段进入反汇编，即可得到该代码在VC6编译下的汇编代码



  - 在win10系统的vs2019下调试上述代码，在printf那一行添加一个断点，执行后右键进入反汇编

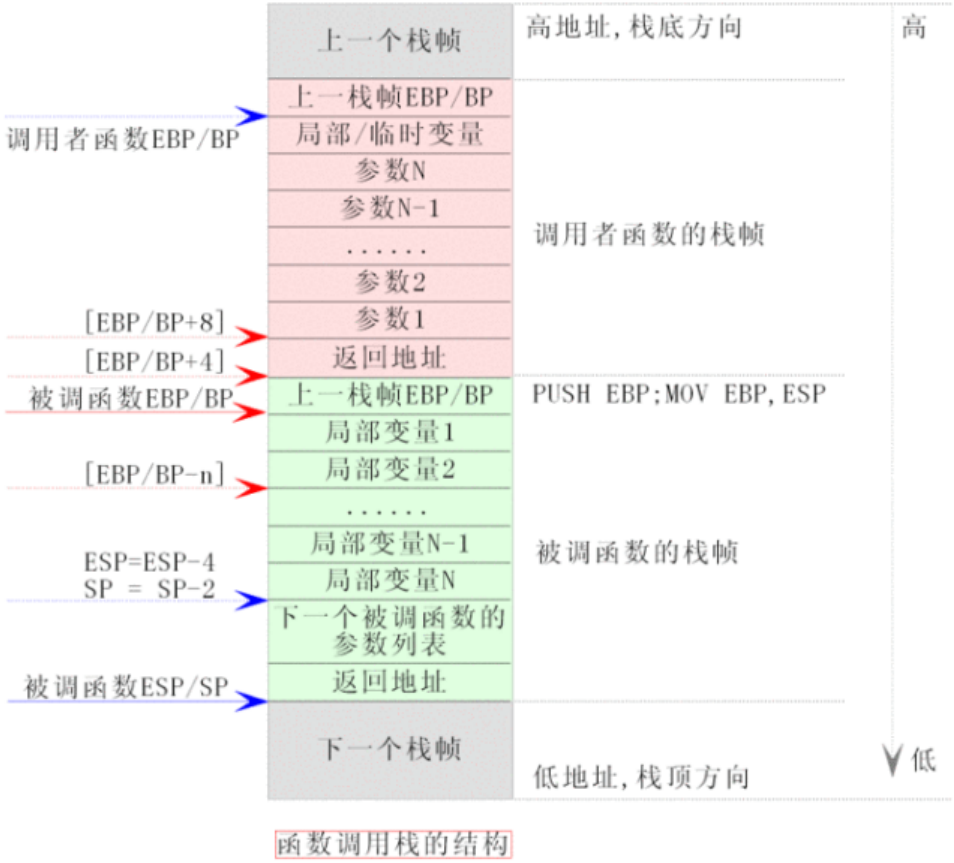


- 实验心得和体会
  - XP的VC6版本的汇编代码及相关分析

```
1:  #include<iostream>
...
2:  int add(int x,int y)
3:  {
00401030   push        ebp
00401031   mov         ebp,esp
00401033   sub         esp,44h
00401036   push        ebx
00401037   push        esi
00401038   push        edi
00401039   lea         edi,[ebp-44h]
0040103C   mov         ecx,11h
00401041   mov         eax,0CCCCCCCCh
00401046   rep stos    dword ptr [edi]
4:          int z=0;
00401048   mov         dword ptr [ebp-4],0
5:          z=x+y;
0040104F   mov         eax,dword ptr [ebp+8]
00401052   add         eax,dword ptr [ebp+0Ch]
00401055   mov         dword ptr [ebp-4],eax
6:          return z;
00401058   mov         eax,dword ptr [ebp-4]
7:      }
0040105B   pop         edi
0040105C   pop         esi
0040105D   pop         ebx
0040105E   mov         esp,ebp
00401060   pop         ebp
00401061   ret
void main()
9:      {
00401070   push        ebp
00401071   mov         ebp,esp
00401073   sub         esp,44h
00401076   push        ebx
00401077   push        esi
00401078   push        edi
```

```
00401079   lea          edi,[ebp-44h]
0040107C   mov          ecx,11h
00401081   mov          eax,0CCCCCCCCh
00401086   rep stos     dword ptr [edi]
10:        int n=0;
00401088   mov          dword ptr [ebp-4],0
11:        n=add(1,3);
0040108F   push         3
00401091   push         1
00401093   call         @ILT+0(add) (00401005)
00401098   add          esp,8
0040109B   mov          dword ptr [ebp-4],eax
12:        printf("%d\n",n);
0040109E   mov          eax,dword ptr [ebp-4]
004010A1   push         eax
004010A2   push         offset string "%d\n" (0043101c)
004010A7   call         printf (004081b0)
004010AC   add          esp,8
13:    }
004010AF   pop          edi
004010B0   pop          esi
004010B1   pop          ebx
004010B2   add          esp,44h
004010B5   cmp          ebp,esp
004010B7   call         __chkesp (00408230)
004010BC   mov          esp,ebp
004010BE   pop          ebp
004010BF   ret
```

本次实验我们主要分析call语句的执行代码，及函数体add和第十一行调用add时相关的寄存器的转换，用一张从网上剽来的很好的图能表示跳转的时候的变换



当函数执行到n = add(1,3)时，首先从右向左的将3和1两个参数依次压入栈中，然后执行call语句，我们可以将call语句进行拆分，拆分成首先跳转到add函数体，然后将当前的eip压入栈中（当前的eip指向的位置为主函数体中下一步应该执行的操作，即返回的地址），最后将函数体的下一步执行操作的地址赋值给eip，可以将一个call语句用汇编语言写成如下

```
call 0x123456 (这里我们只认为其为单纯的跳转操作，可以看作jmp)
push %eip
mov $0x123456,%eip
```

当执行call语句进入add函数体后，首先执行的是push ebp,mov ebp,esp，即将之前的栈底指针压入栈中，保存上一个栈帧，然后进行一些参数的赋值（此处将esp下移了44h的空间目的为了给函数提前留出足够的空间），而当add函数体结束后，除了对一些使用的寄存器进行pop之外，还有以下汇编代码：

```
0040105E   mov         esp,ebp
00401060   pop         ebp
00401061   ret
```

mov esp,ebp 将当前的栈顶esp移动至ebp位置（即对中间的数据进行了清楚处理）

pop ebp 将之前存下来的上一个栈帧的地址弹出并赋值给ebp

简单总结：在XP系统的VC6环境下的call相关机制，首先将参数入栈，入栈后将当前eip压栈然后跳转至函数体并更新eip至当前函数体，然后将ebp压栈并抬高，实现前一栈帧的保存操作，随后执行函数体结束后，将esp移动至ebp位置，此时esp高一节的地址位置为前一栈帧的保存地址，因此最后pop一下ebp即可

- win10的VS2019版本的汇编代码及相关分析

```
#include <iostream>
int add(int x, int y)
{
00E817D0  push        ebp
00E817D1  mov         ebp,esp
00E817D3  sub         esp,0CCh
00E817D9  push        ebx
00E817DA  push        esi
00E817DB  push        edi
00E817DC  lea         edi,[ebp-0CCh]
00E817E2  mov         ecx,33h
00E817E7  mov         eax,0CCCCCCCCh
00E817EC  rep stos    dword ptr es:[edi]
00E817EE  mov         ecx,offset _A05806CF_源@cpp (0E8C026h)
00E817F3  call        @__CheckForDebuggerJustMyCode@4 (0E81226h)
    int z = 0;
00E817F8  mov         dword ptr [z],0
    z = x + y;
00E817FF  mov         eax,dword ptr [x]
00E81802  add         eax,dword ptr [y]
00E81805  mov         dword ptr [z],eax
    return z;
00E81808  mov         eax,dword ptr [z]
}
00E8180B  pop         edi
00E8180C  pop         esi
00E8180D  pop         ebx
```

```
00E8180E  add         esp,0CCh
00E81814  cmp         ebp,esp
00E81816  call        __RTC_CheckEsp (0E81230h)
00E8181B  mov         esp,ebp
00E8181D  pop         ebp
00E8181E  ret

void main()
{
00E81940  push        ebp
00E81941  mov         ebp,esp
00E81943  sub         esp,0CCh
00E81949  push        ebx
00E8194A  push        esi
00E8194B  push        edi
00E8194C  lea         edi,[ebp-0CCh]
00E81952  mov         ecx,33h
00E81957  mov         eax,0CCCCCCCCh
00E8195C  rep stos    dword ptr es:[edi]
00E8195E  mov         ecx,offset _A05806CF_源@cpp (0E8C026h)
00E81963  call        @__CheckForDebuggerJustMyCode@4 (0E81226h)
    int n = 0;
00E81968  mov         dword ptr [n],0
    n = add(1, 3);
00E8196F  push        3
00E81971  push        1
00E81973  call        add (0E81181h)
00E81978  add         esp,8
00E8197B  mov         dword ptr [n],eax
    printf("%d\n", n);
00E8197E  mov         eax,dword ptr [n]
00E81981  push        eax
00E81982  push        offset string "%d\n" (0E87B30h)
00E81987  call        _printf (0E81046h)
00E8198C  add         esp,8
}
00E8198F  xor         eax,eax
00E81991  pop         edi
00E81992  pop         esi
00E81993  pop         ebx
00E81994  add         esp,0CCh
00E8199A  cmp         ebp,esp
00E8199C  call        __RTC_CheckEsp (0E81230h)
00E819A1  mov         esp,ebp
00E819A3  pop         ebp
00E819A4  ret
```

从win10的VS2019版本上编译的汇编代码和VC6上基本类似，首先将参数入栈，入栈后将当前eip压栈然后跳转至函数体并更新eip至当前函数体，然后将ebp压栈并抬高，实现前一栈帧的保存操作，随后执行函数体结束后，将esp移动至ebp位置，此时esp高一节的地址位置为前一栈帧的保存地址，因此最后pop一下ebp。

而其中存在的一些不同操作为

1. 首先在执行add函数体的时候，给函数体开了更大的空间

```
sub         esp,0CCh
//上面为win10
sub         esp,44h
//上面为xp
```

个人理解将空间扩大是一种栈溢出的防护措施

2. 在函数体结尾阶段，出现了一个判断

```
00E8180E  add         esp,0CCh
00E81814  cmp         ebp,esp
00E81816  call        __RTC_CheckEsp (0E81230h)
00E8181B  mov         esp,ebp
00E8181D  pop         ebp
00E8181E  ret
```

因为我们给esp下移了0CCh，因此再次加上（栈顶esp位置地址低），如果函数体正常运行那么我们上移的esp应该与esp重合，因此在此处做出了一个判断，进行ebp和esp的比较，然后再进行移动和pop操作实现，此项操作可以有效的减少一定的栈溢出或者数据篡改操作，我觉得是一种安全的防护措施，在一定程度上避免了一些安全问题的发生，如果发生会跳转到__RTC_CheckEsp (0E81230h)这一段check代码中进行复查