《软件安全》实验报告

姓名：蒋枘言　学号：2313546　班级：信息安全班

一、实验名称

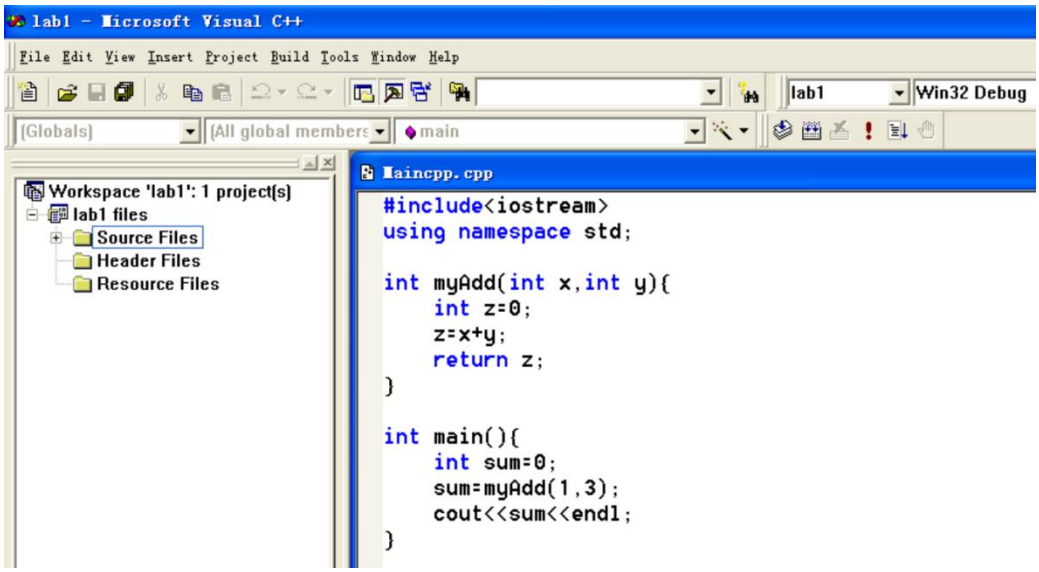
IDE 反汇编实验

二、实验要求

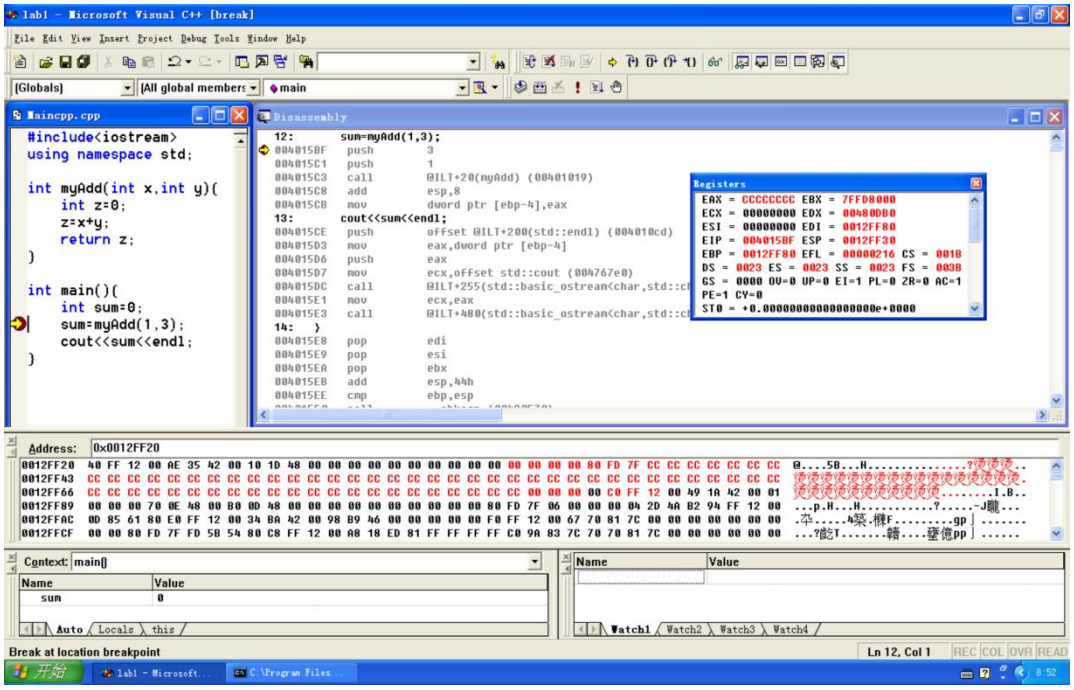
根据第二章示例 2-1，在 Windows XP 环境下进行 Microsoft Visual C++ 6.0(VC6)反汇编调试，熟悉函数调用、栈帧切换、CALL 和 RET 指令等汇编语言实现，将 call 语句执行过程中的寄存器 EIP、ESP、EBP 等的变化状态进行记录，解释变化的主要原因。

三、实验过程

1.打开 VC6，新建一个项目，编写一个简单的 C++程序，如下图所示。



2.在程序第 12 行（sum = myAdd(1, 3);）设置一个断点（快捷键 F9），随后点击 Build->Start Debug->Go（快捷键 F5）开始调试，程序执行到第 12 行时便会停止。



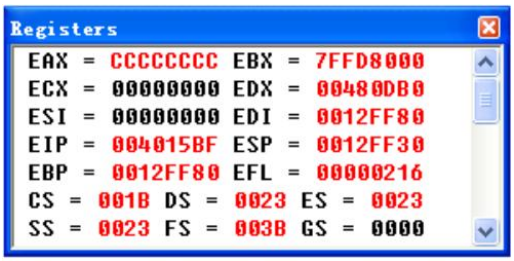
3.右键，选择 Go To Disassembly，转到反汇编代码。

```
1:    #include<iostream>
2:    using namespace std;
3:
4:    int myAdd(int x,int y){
00401560   push        ebp
00401561   mov         ebp,esp
00401563   sub         esp,44h
00401566   push        ebx
00401567   push        esi
00401568   push        edi
00401569   lea         edi,[ebp-44h]
0040156C   mov         ecx,11h
00401571   mov         eax,0CCCCCCCCh
00401576   rep stos    dword ptr [edi]
5:        int z=0;
00401578   mov         dword ptr [ebp-4],0
6:        z=x+y;
0040157F   mov         eax,dword ptr [ebp+8]
00401582   add         eax,dword ptr [ebp+0Ch]
00401585   mov         dword ptr [ebp-4],eax
7:        return z;
00401588   mov         eax,dword ptr [ebp-4]
8:    }
0040158B   pop         edi
0040158C   pop         esi
0040158D   pop         ebx
0040158E   mov         esp,ebp
00401590   pop         ebp
00401591   ret
9:
10:   int main(){
004015A0   push        ebp
004015A1   mov         ebp,esp
004015A3   sub         esp,44h
004015A6   push        ebx
004015A7   push        esi
004015A8   push        edi
004015A9   lea         edi,[ebp-44h]
004015AC   mov         ecx,11h
004015B1   mov         eax,0CCCCCCCCh
004015B6   rep stos    dword ptr [edi]
11:       int sum=0;
004015B8   mov         dword ptr [ebp-4],0
12:       sum=myAdd(1,3);
004015BF   push        3
004015C1   push        1
004015C3   call        @ILT+20(myAdd) (00401019)
004015C8   add         esp,8
004015CB   mov         dword ptr [ebp-4],eax
13:       cout<<sum<<endl;
004015CE   push        offset @ILT+200(std::endl) (004010cd)
004015D3   mov         eax,dword ptr [ebp-4]
004015D6   push        eax
004015D7   mov         ecx,offset std::cout (004767e0)
004015DC   call        @ILT+255(std::basic_ostream<char,std::char_traits<char> >::operator<<) (00401104)
004015E1   mov         ecx,eax
004015E3   call        @ILT+480(std::basic_ostream<char,std::char_traits<char> >::operator<<) (004011e5)
14:   }
004015E8   pop         edi
004015E9   pop         esi
004015EA   pop         ebx
004015EB   add         esp,44h
004015EE   cmp         ebp,esp
004015F0   call        __chkesp (00420570)
004015F5   mov         esp,ebp
004015F7   pop         ebp
004015F8   ret
```

4.使用 Step Over（快捷键 F10）和 Step Into（快捷键 F11）逐行执行汇编指令，观察各个寄存器以及内存中值的变化。

以下是对函数调用过程的详细分析：

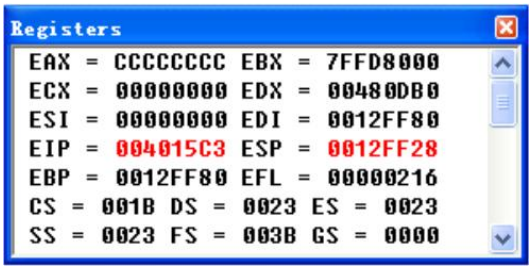
（1）还未调用 myAdd9 函数并且未开始执行第 12 行语句时，当前 EBP 保存主函数栈帧的基准位置，ESP 指向栈顶。EIP 的值为 004015BF，即下一条即将执行的指令 push 3 的地址。



（2）未执行 call 指令（函数调用）之前，参数从右向左入栈。

```
004015BF push 3
004015C1 push 1
```

观察到由于两个 dword 大小的参数入栈，ESP 寄存器的值从 0012FF30 变为了 0012FF28，相当于减去了 8（十六进制）。说明栈向低地址增长。EIP 寄存器每执行一条语句都会变化，永远保存下一条即将执行的指令的地址。



（3）执行 call 指令。

```
004015C3 call @ILT+20(myAdd) (00401019)
```

@ILT+20(myAdd)是修饰符，表示通过 ILT（Incremental Link Table，增量链接表）间接调用函数 myAdd。00401019 是 ILT 中跳转指令的实际地址（十六进制）。

再执行一条指令才跳转到函数 myAdd（00401560 处）。

（4）接着进行**栈帧切换**。

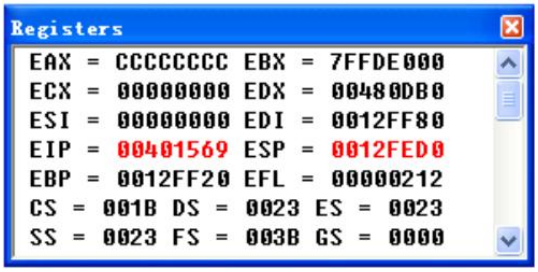
```
00401560 push    ebp
00401561 mov     ebp,esp
00401563 sub     esp,44h
```

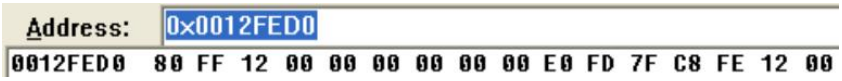
将寄存器 EBP 的值压入栈中，暂时保存主函数的栈帧；随后将 ESP 的值赋给 EBP，将 ESP 的值减去 44h（十进制 68，相当于栈的空间增长了 44h），为 myAdd 函数分配了栈帧空间。

（5）随后的几行指令进行函数状态保存。

```
00401566 push    ebx
00401567 push    esi
00401568 push    edi
```

以上 3 行用于保存现场，ebx 作为内存偏移指针，esi 作为源地址指针，edi 作为目的地址指针。执行后寄存器和内存的值如下图所示。






```
00401569 lea          edi,[ebp-44h]
0040156C mov          ecx,11h                 ;ecx 作为计数器使用
                                              ;相当于设置循环次数 11h 次（十进制 17）
00401571 mov          eax,0CCCCCCCCh          ;将十六进制值 CCCCCCCC 赋给 eax
00401576 rep stos     dword ptr [edi]         ;rep 代表重复执行，每执行一次 ecx 减 1
                                              ;stos 是储存字符串的指令,储存的目标地址是[edi]
```

未初始化的局部变量或栈空间会被编译器填充为 0xCC，故先将 0CCCCCCCCh 赋值给 eax；

程序中出现大量“烫烫烫”乱码的典型原因是未初始化的栈内存被当作字符串访问，因为在 GBK 编码中，0xCCCC 对应中文“烫”。

stos 是存储字符串指令（Store String），操作模式有

①stosb：按字节操作（使用 AL）；

②stosw：按字操作（使用 AX）；

③stosd：按双字操作（使用 EAX，32 位模式）；

④stosq：按四字操作（使用 RAX，64 位模式）。

此处 dword ptr 显式指定操作大小为双字（4 字节），对应 stosd。

目标地址由 EDI（或 RDI，64 位）寄存器指向，并根据方向标志 DF 更新 EDI：

若 DF=0（默认，CLD 指令设置），EDI 递增，此处就是递增；

若 DF=1（STD 指令设置），EDI 递减。

（6）执行函数体

```
0040157F mov    eax,dword ptr [ebp+8]
00401582 add    eax,dword ptr [ebp+0Ch]
00401585 mov    dword ptr [ebp-4],eax
00401588 mov    eax,dword ptr [ebp-4]
```

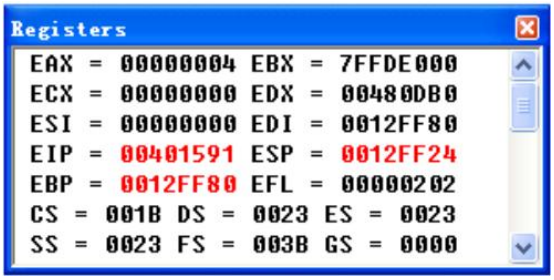
通过 EBP 加上某个数来访问传入的参数。

最终函数的返回值会储存在 EAX 寄存器中。

（7）恢复状态

```
0040158B pop    edi
0040158C pop    esi
0040158D pop    ebx
0040158E mov    esp,ebp
00401590 pop    ebp
```

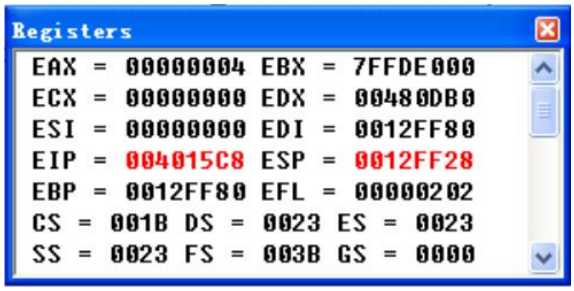
这几行恢复了寄存器的值，即恢复了主函数 main 的栈帧。




```
00401591 ret
```

ret 是函数返回的指令，与 call 相对应。

可以观察到 ret 指令执行前 EIP 的值为 00401591，执行后为 004015C8，回到了主函数中。



004015C8 add esp,8

将栈顶恢复（对应于最开始 push 的两个参数），函数调用结束，回到主函数中继续执行。

四、心得体会

由于上学期上过**王志**老师《**汇编语言与逆向技术**》课程，所以第一次的《软件安全》实验对于我来说比较熟悉，很多理论知识都有所了解，但做完还是有不小收获。

1.理论与实践的结合

反汇编调试让我直观地看到 C++代码如何转化为机器指令。例如，call 和 ret 指令的配合实现了函数的调用与返回，而 EBP 和 ESP 的协同工作管理了栈帧的创建与销毁。这种从高级语言到底层指令的映射，加深了我对程序执行流程的理解。

2.栈帧与内存管理的核心机制

实验中观察到参数按“从右向左”顺序压栈，以及函数内通过 sub esp, 44h 分配局部变量空间的操作，揭示了栈“向低地址增长”的特性。此外，EBP 作为栈帧基准指针的作用（如访问参数[ebp+8]和局部变量[ebp-4]）让我认识到栈帧结构化管理的必要性。

3.调试模式的内存保护意义（感兴趣，自行查询资料）

通过 0xCCCCCCCC 填充未初始化内存的设计，我理解了编译器在调试模式下的主动防御机制。这不仅帮助检测未初始化变量的使用，还能在程序意外执行到填充区域时触发断点，为发现内存错误（如栈溢出、野指针）提供了有效手段。还明白了为什么有些时候写的 C++程序执行结果会有很多个“*烫烫烫*”。

4.工具使用与逆向分析能力的提升

熟练使用 VC6 的调试器（如断点设置、反汇编视图、寄存器监控）和指令单步执行（F10/F11），增强了我的动态分析能力。这对未来分析二进制程序、排查隐蔽漏洞具有重要意义。

但是也遇到了不少麻烦，首先是由于 Windows XP 系统过于古老，操作起来不习惯，虚拟机都安装了好久。Microsoft Visual C++ 6.0 更是古董级别的东西，用着非常不习惯，为什么非要用这么古老的软件呢(ℸ _ℸ)……

一些小想法：对于上学期没有修过《汇编语言与逆向技术》课程的同学来说（甚至密码专业的同学不能选，信安必修，计科和物联网选修，密码没有），《软件安全》这门课刚开始学起来可能有点陌生，毕竟涉及到汇编语言的基础语法（比如字母开头的十六进制数前面要加 0，伪指令 ptr 的用法等等），还有一些知识比如 Intel x86 的 CPU 架构使用小端序存储数据等，这些如果不了解的话学习起来还是有一定障碍。