前言
项目预览
项目配置
文件命名与作用
ShellCode大小的计算方法
第一部分 ShellCode生成
第二部分 ShellCode部分
ShellCode加载器
如何提取ShellCode
如何扩展ShellCode框架?
参考资料
项目下载

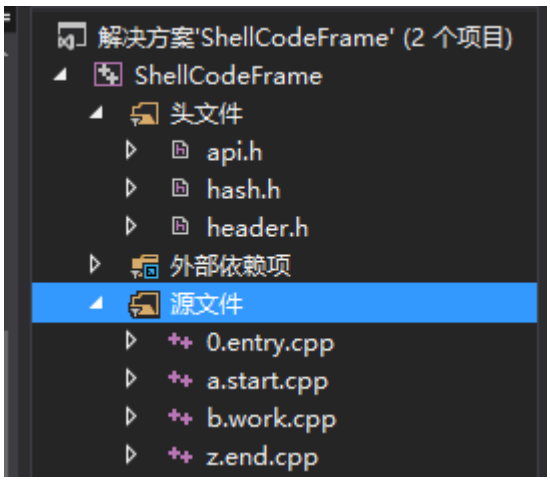
前言

现阶段，shellcode编写门槛高，大多需要有较深的汇编功底，而Metersploit上的Shellcode开源生成框架，功能单一，扩展性差，大多只能在demo中测试使用，难以在实战中发挥作用。

我的这个版本用纯C语言实现了Windows平台下自己的Shellcode生成器，能在实战中根据现实情况，自动生成所需功能的Shellcode。

项目预览

整个项目大致如下：后面会讲解每一个文件的作用



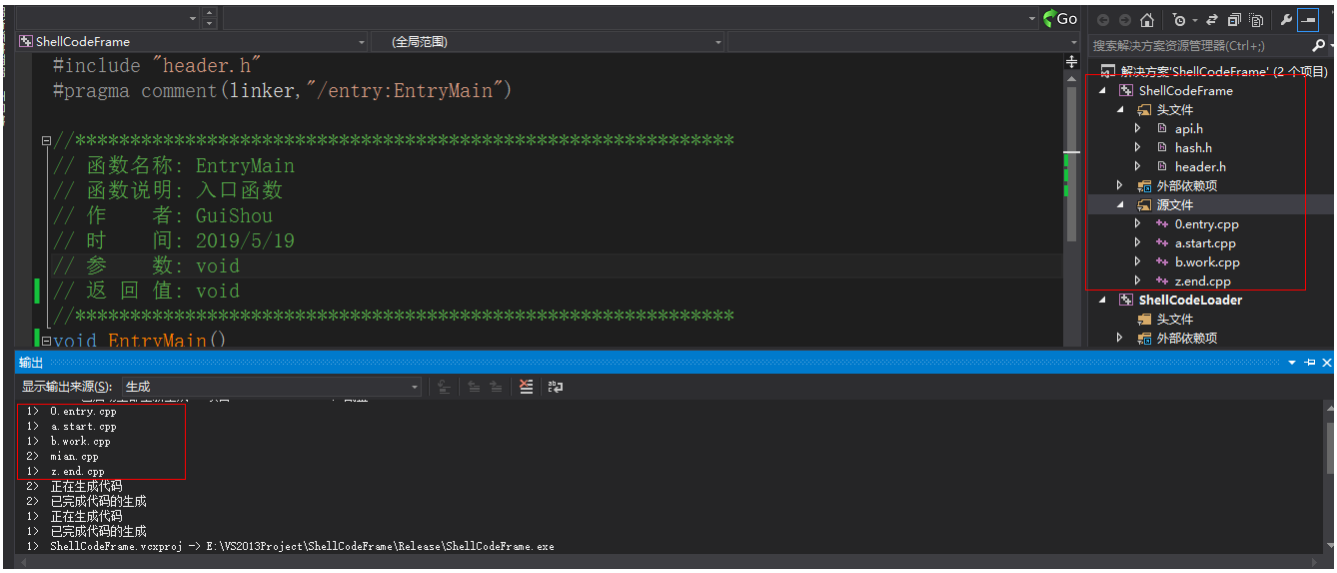
项目配置

首先来说一下自己的这个项目的设置，本项目使用VS2013编译

1. 编译时选择 realse版本 属性->C/C++->代码生成->运行库->多线程 (/MT)
2. 为了防止编译器自动生成的一系列代码造成的干扰 需要修改入口点 在属性->链接器->高级
3. 属性->C/C++->代码生成->禁用安全检查GS
4. 关闭生成清单 属性->链接器->清单文件->生成清单 选择否
5. 关闭调试信息 属性->链接器->生成调试信息->否
6. 取消SDL安全检查
7. 兼容XP 选择属性->常规->平台工具集->Visual Studio 2013 - Windows XP (v120_xp)
8. C/C++优化 优化->使大小最小化 (/O1) 优化大小或速度->代码大小优先 (/Os)

文件命名与作用

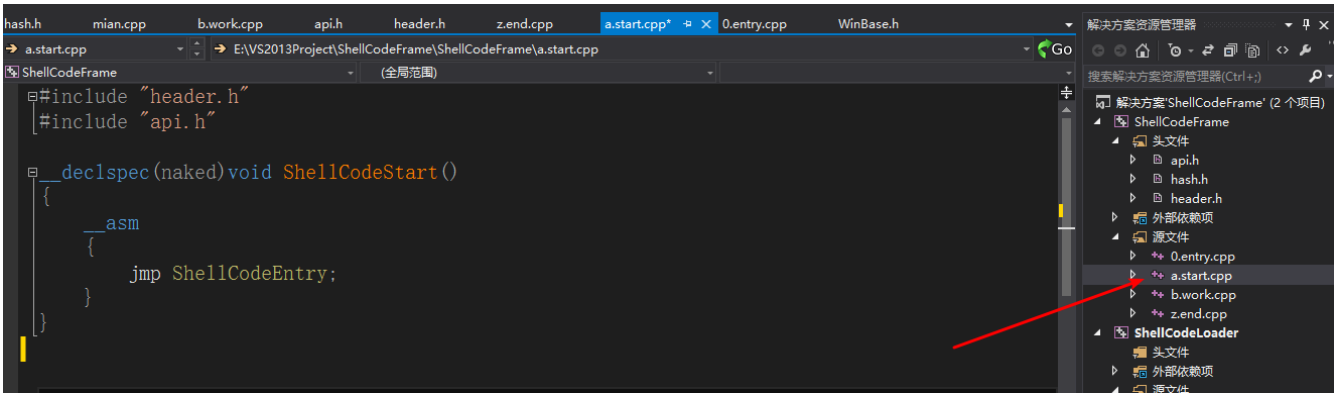
我的这个框架分为两个部分，一个是ShellCode的生成部分，还有一个是ShellCode部分



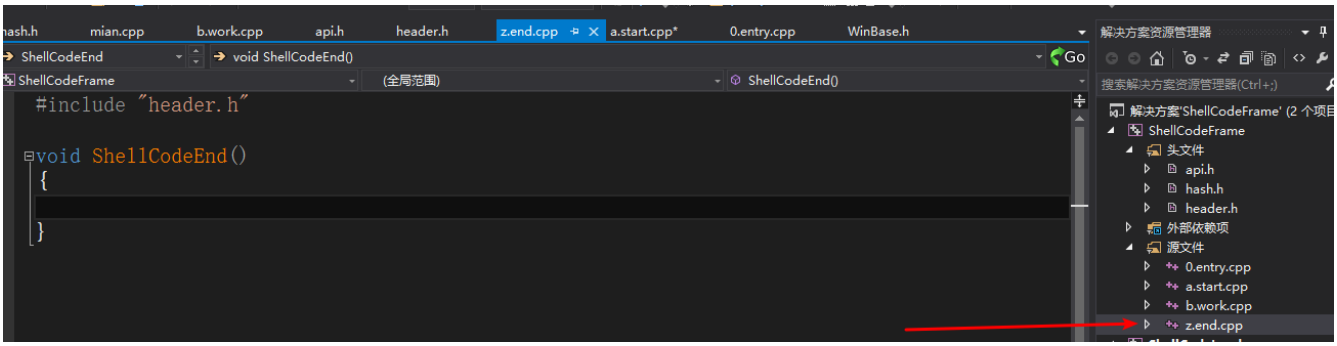
之所以采用这样的文件命名的方式是为了方便计算ShellCoede的大小。文件的编译顺序就是编译后的exe函数的排列顺序。具体来说这个项目的文件编译顺序是0.entry.cpp->a.start.cpp->b.work.cpp->z.end.cpp(main.cpp是另外一个工程)，那么代码段中的函数排列顺序也会和文件的编译顺序一致 下面说一下每个文件的作用

- api.h->存放所有和api函数相关的结构体及函数指针
- hash.h->存放需要用到的API函数的哈希定义宏
- header.h->存放头文件及函数声明
- 0.entry.cpp->存放ShellCode函数的入口
- a.start.cpp->存放ShellCodeStart(标记一个起始位置)和真正的ShellCode代码
- b.work.cpp->存放ShellCode中的起作用的代码
- z.end.cpp->存放ShellCodeEnd函数(标记一个结束位置)

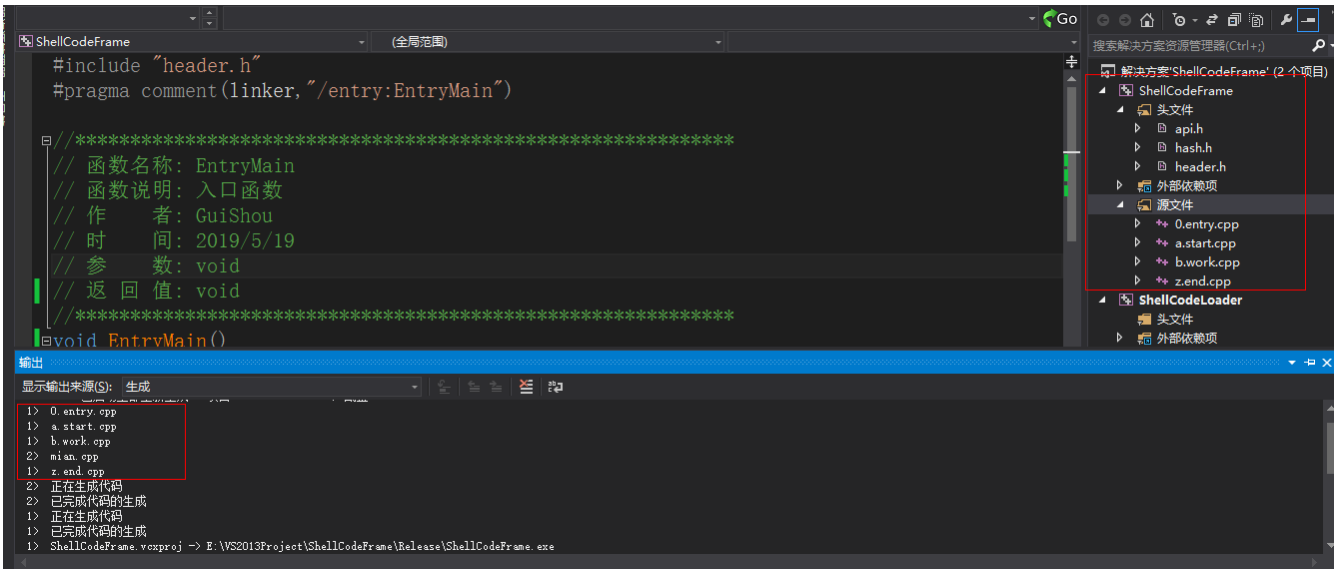
ShellCode大小的计算方法



首先我在a.start.cpp中放了一个ShellCodeStart函数，用于标记ShellCode的开始位置

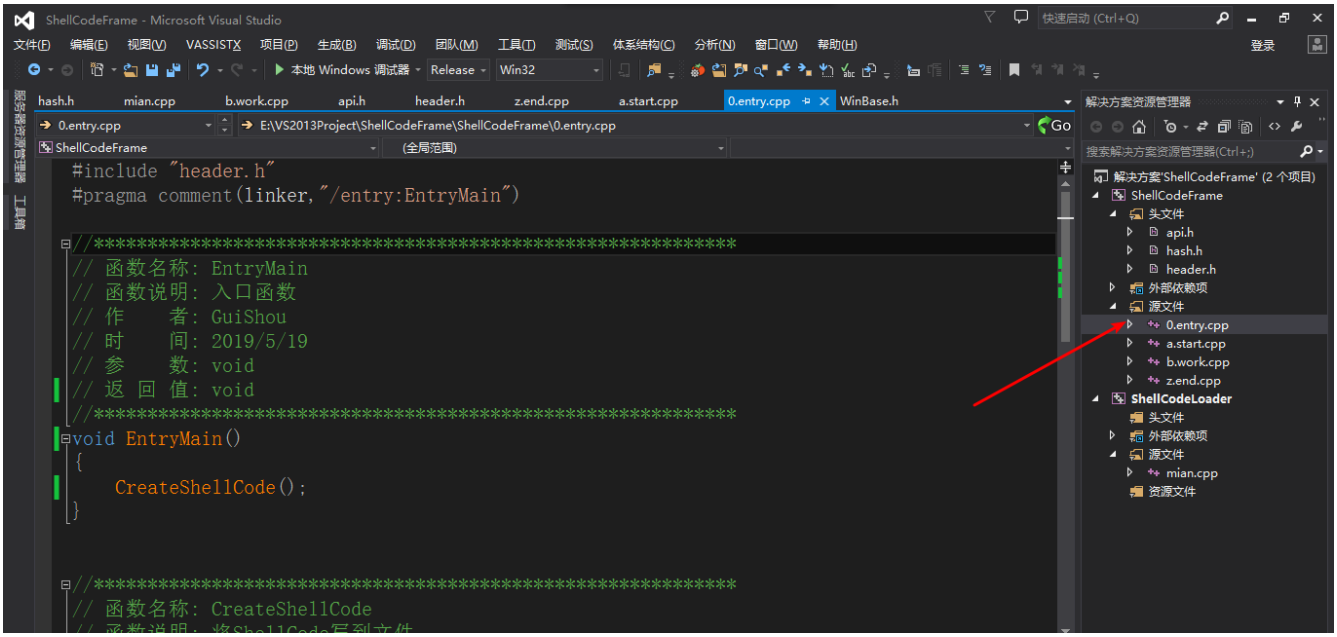


然后在z.end中放了一个ShellCodeEnd函数，用来标记ShellCode的结束位置，然后将真正的ShellCode放在a和z之间

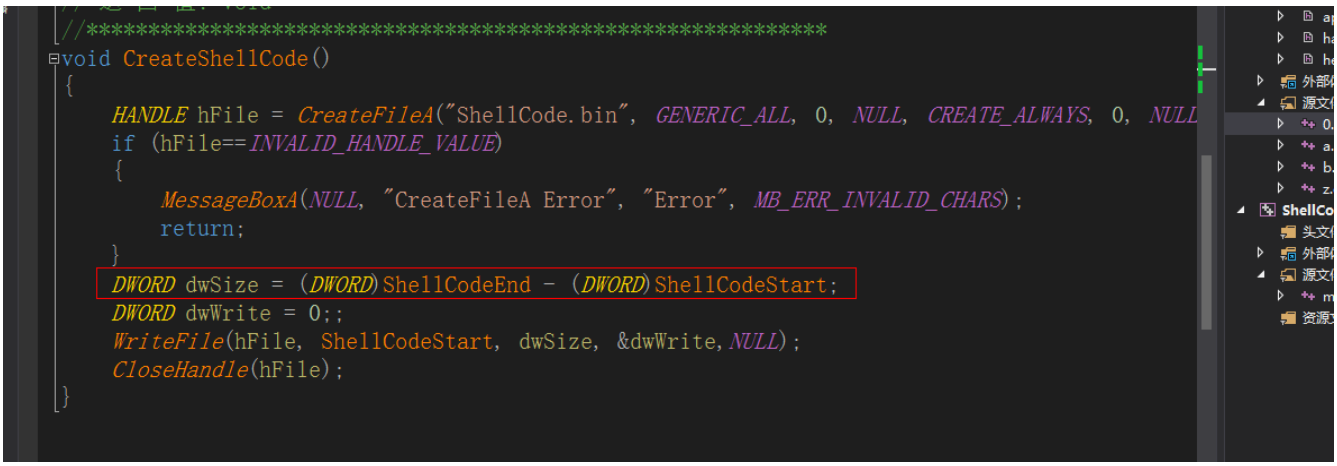


那么根据文件的编译顺序，只需要用ShellCodeEnd函数的位置减去ShellCodeStart函数的位置，就能得到ShellCode的大小

第一部分 ShellCode生成

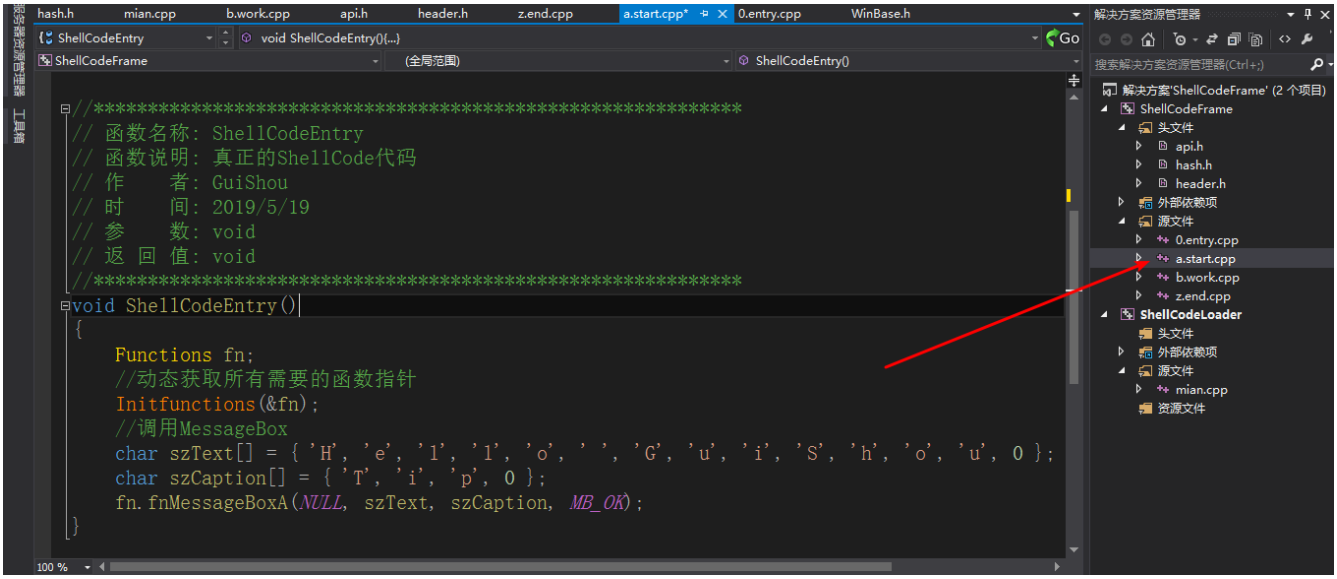


首先来说明ShellCode的生成部分，这个部分在0.entry.cpp中，同时将入口点修改为EntryMain，也就是说这是整个工程的main函数



这个ShellCode生成函数会计算ShellCode的大小，然后将ShellCode写到一个二进制文件，可以省去在OD中提取ShellCode的步骤

第二部分 ShellCode部分



真正的ShellCode代码存放在a.start中的ShellCodeEntry函数里

首先我定义了一个结构体Functions，这个结构体存放所有需要用到的函数指针

```
//函数指针结构体
typedef struct _FUNCTIONS
{
    pfnLoadLibraryA fnLoadLibraryA;
    pfnMessageBoxA fnMessageBoxA;
}Functions,*Pfunctions;
```

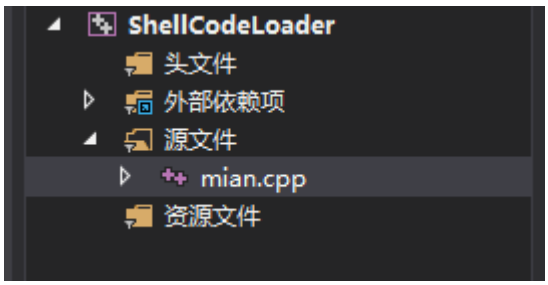
接着通过计算哈希的方式获取到需要的函数地址并将所需要的模块加载进来

```
//************************************************************
void Initfunctions(Pfunctions pfn)
{
    //获取LoadLibraryA函数地址
    pfn->fnLoadLibraryA = (pfnLoadLibraryA)GetProcAddressWithHash(HASH_LoadLibraryA);
    //将user32.dll加载到当前进程中
    char szUser32[] = { 'u', 's', 'e', 'r', '3', '2', '.', 'd', 'l', 'l', 0 };
    pfn->fnLoadLibraryA(szUser32);
    //获取MessageBoxA函数地址
    pfn->fnMessageBoxA = (pfnMessageBoxA)GetProcAddressWithHash(HASH_MessageBoxA);
}
```

接着调用MessageBox函数

```
//调用MessageBox
char szText[] = { 'H', 'e', 'l', 'l', 'o', ' ', 'G', 'u', 'i', 'S', 'h', 'o', 'u', 0 };
char szCaption[] = { 'T', 'i', 'p', 0 };
fn.fnMessageBoxA(NULL, szText, szCaption, MB_OK);
```

ShellCode加载器



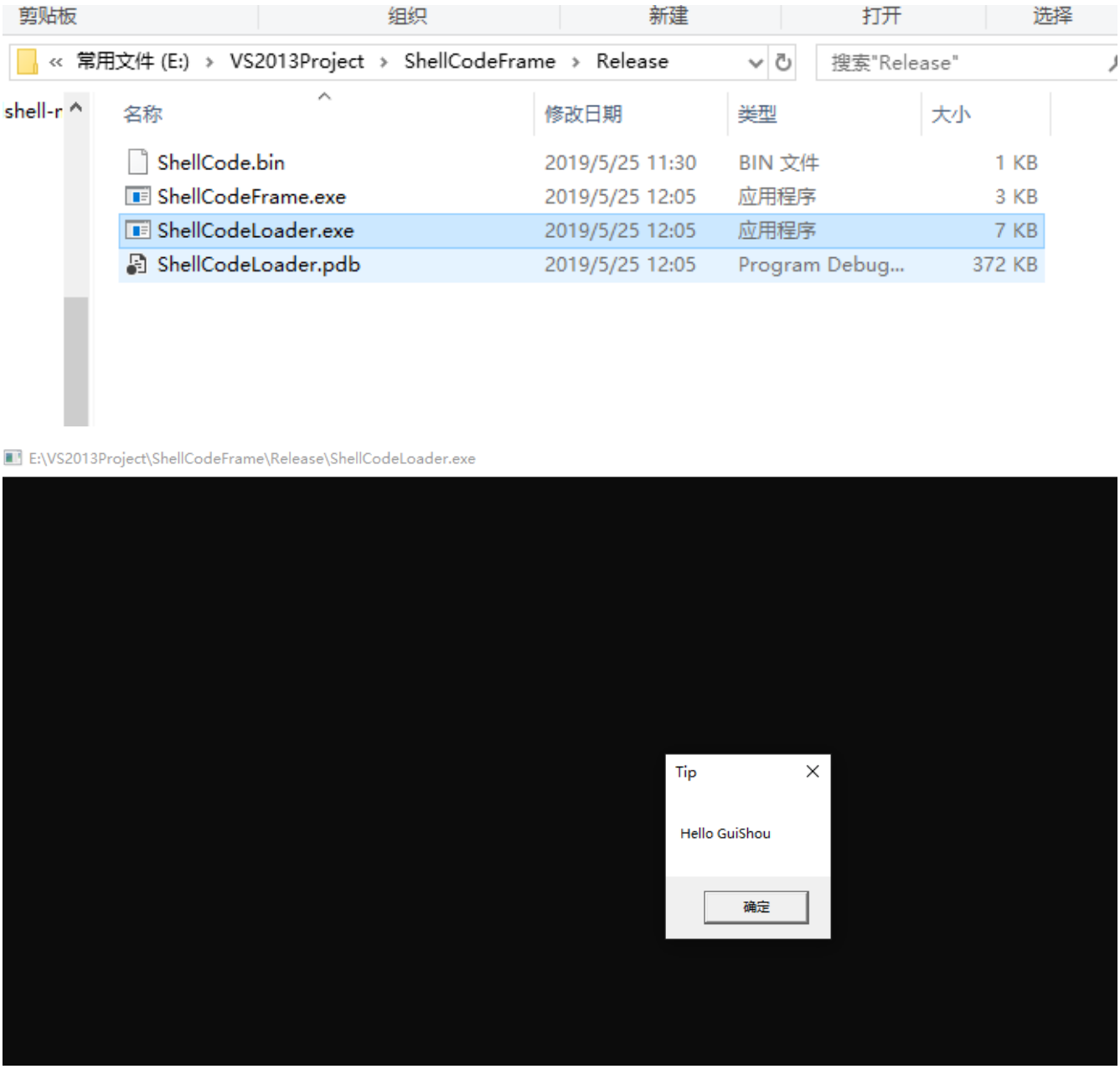
另外我还写了一个ShellCodeLoader用于测试写好的ShellCode，代码相对来说比较简单

```
ShellCodeLoader    (全局范围)    main(int argc, char * argv[])
    HANDLE hFile = CreateFileA("ShellCode.bin", GENERIC_READ, 0, NULL, OPEN_ALWAYS, 0, NULL);
    if (hFile==INVALID_HANDLE_VALUE)
    {
        printf("CreateFile Error");
        return -1;
    }

    DWORD dwSize = 0;
    //获取ShellCode的总大小
    dwSize = GetFileSize(hFile, NULL);

    //申请一块可读可写可执行的内存
    LPVOID lpAddress = VirtualAlloc(NULL, dwSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (lpAddress == NULL)
    {
        printf("VirtualAlloc Error");
        CloseHandle(hFile);
        return -1;
    }
```

```
    //将文件读取到申请的内存中
    DWORD dwRead = 0;
    ReadFile(hFile, lpAddress, dwSize, &dwRead, 0);

    //执行ShellCode
    __asm
    {
        call lpAddress;
    }
    return 0;
}
```

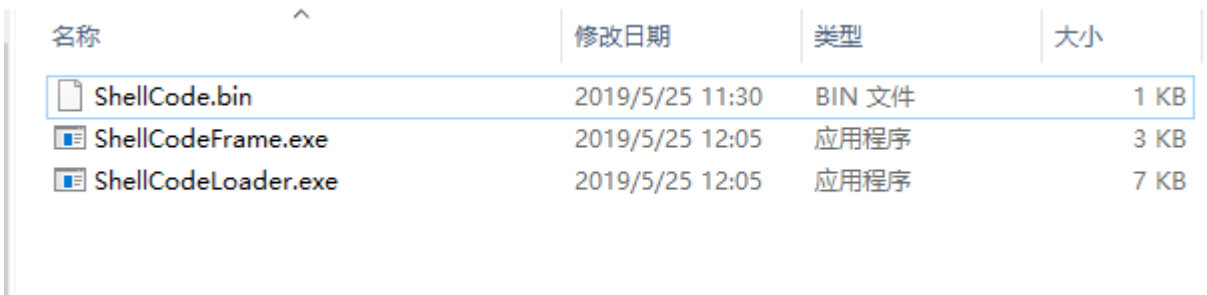
就是将ShellCode读取到内存然后执行

如果你所编写的ShellCode没有文件，当双击ShellCodeLoader时，就会执行生成的ShellCode.bin文件

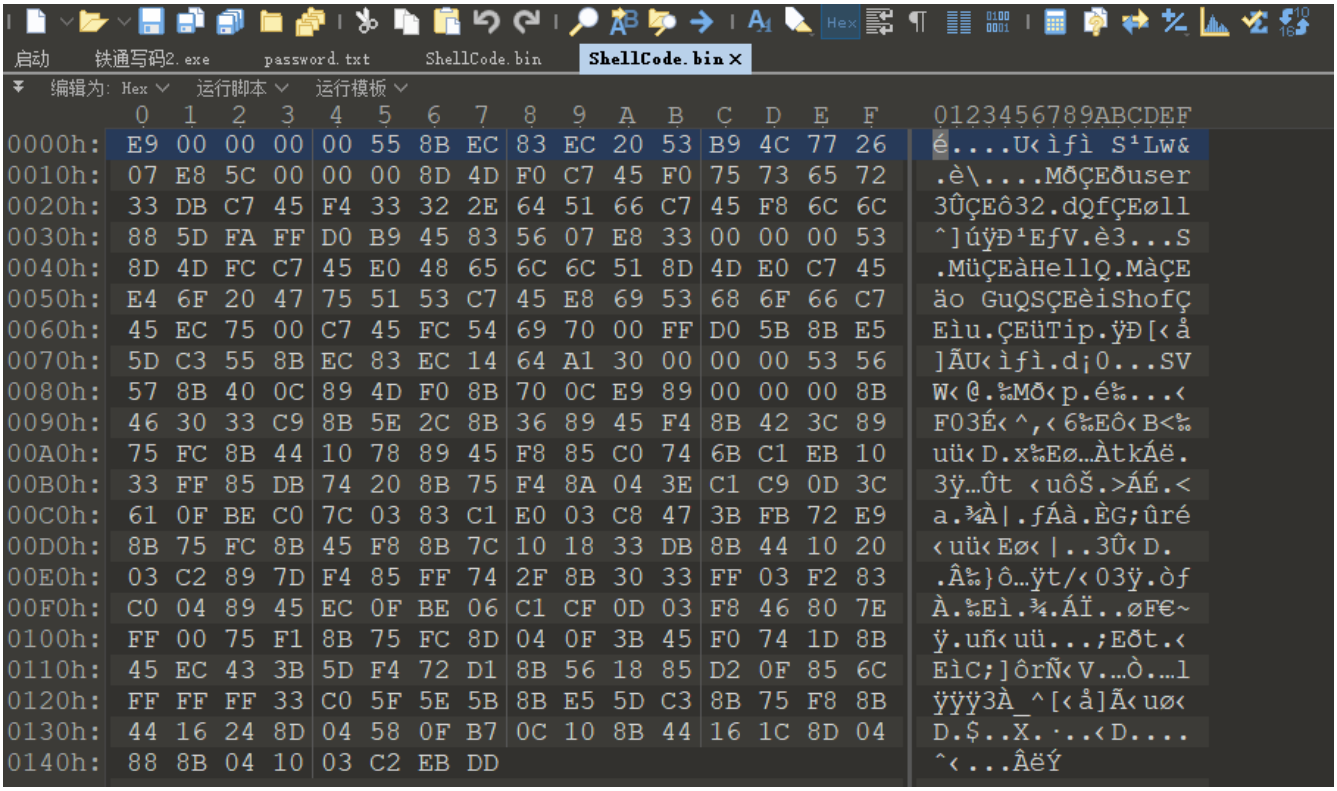


如果执行成功，说明ShellCode没有问题

如何提取ShellCode



在编写好ShellCode之后点击ShellCodeFrame.exe会生成ShellCode.bin，然后用二进制文件打开ShellCode.bin，复制所有代码即可



如何扩展ShellCode框架?

我的这个框架并只写了一个示例的MessageBox函数，具体扩展的步骤如下：

1. 在api.h中定义所需要的函数指针，并将函数指针存放到结构体

```
//定义函数指针
typedef HMODULE(WINAPI *pfnLoadLibraryA)(LPCSTR lpLibFileName);
typedef int (WINAPI *pfnMessageBoxA)(HWND hWnd,LPCSTR lpText,LPCSTR lpCaption,UINT uType);

//函数指针结构体
typedef struct _FUNCTIONS
{
    pfnLoadLibraryA fnLoadLibraryA;
    pfnMessageBoxA fnMessageBoxA;
}Functions,*Pfunctions;
```

2. 在hash.h中定义需要用的到函数的哈希值

3. 在b.work的Initfunctions函数中获取函数指针和加载需要的模块

```
//*************************************************************
void Initfunctions(Pfunctions pfn)
{
    //获取LoadLibraryA函数地址
    pfn->fnLoadLibraryA = (pfnLoadLibraryA)GetProcAddressWithHash(HASH_LoadLibraryA);
    //将user32.dll加载到当前进程中
    char szUser32[] = { 'u', 's', 'e', 'r', '3', '2', '.', 'd', 'l', 'l', 0 };
    pfn->fnLoadLibraryA(szUser32);
    //获取MessageBoxA函数地址
    pfn->fnMessageBoxA = (pfnMessageBoxA)GetProcAddressWithHash(HASH_MessageBoxA);
}
```

4. 在ShellCodeEntry中调用函数

```
ShellCodeFrame                (全局范围)                ShellCodeEntry()
// 函数说明: 真正的ShellCode代码
// 作    者: GuiShou
// 时    间: 2019/5/19
// 参    数: void
// 返 回 值: void
//*************************************************************
void ShellCodeEntry()
{
    Functions fn;
    //动态获取所有需要的函数指针
    Initfunctions(&fn);
    //调用MessageBox
    char szText[] = { 'H', 'e', 'l', 'l', 'o', ' ', 'G', 'u', 'i', 'S', 'h', 'o', 'u', 0 };
    char szCaption[] = { 'T', 'i', 'p', 0 };
    fn.fnMessageBoxA(NULL, szText, szCaption, MB_OK);
}
```

参考资料

《Windows平台高效Shellcode编程技术实战》

PIC_BINDSHELL(Github)： https://github.com/mattifestation/PIC_Bindshell

项目下载

https://github.com/TonyChen56/ShellCodeFrame