C++/MFC调试器项目

原创

Mr_Hock 于 2018-11-17 13:59:32 发布 823 收藏 3

分类专栏： 项目分享

本文链接：https://blog.csdn.net/qq_43572067/article/details/84184896

项目分享 专栏收录该内容

10 篇文章 0 订阅

订阅专栏

本次调试器项目已实现的功能

附加，创建调试进程

查看，修改汇编，内存，栈，寄存器

查看任意模块，导入表，导出表。

永久性的断点(调试器重新打开断点依然存在)

无限软件(条件)断点，硬件 (条件)断点，执行，读写，访问

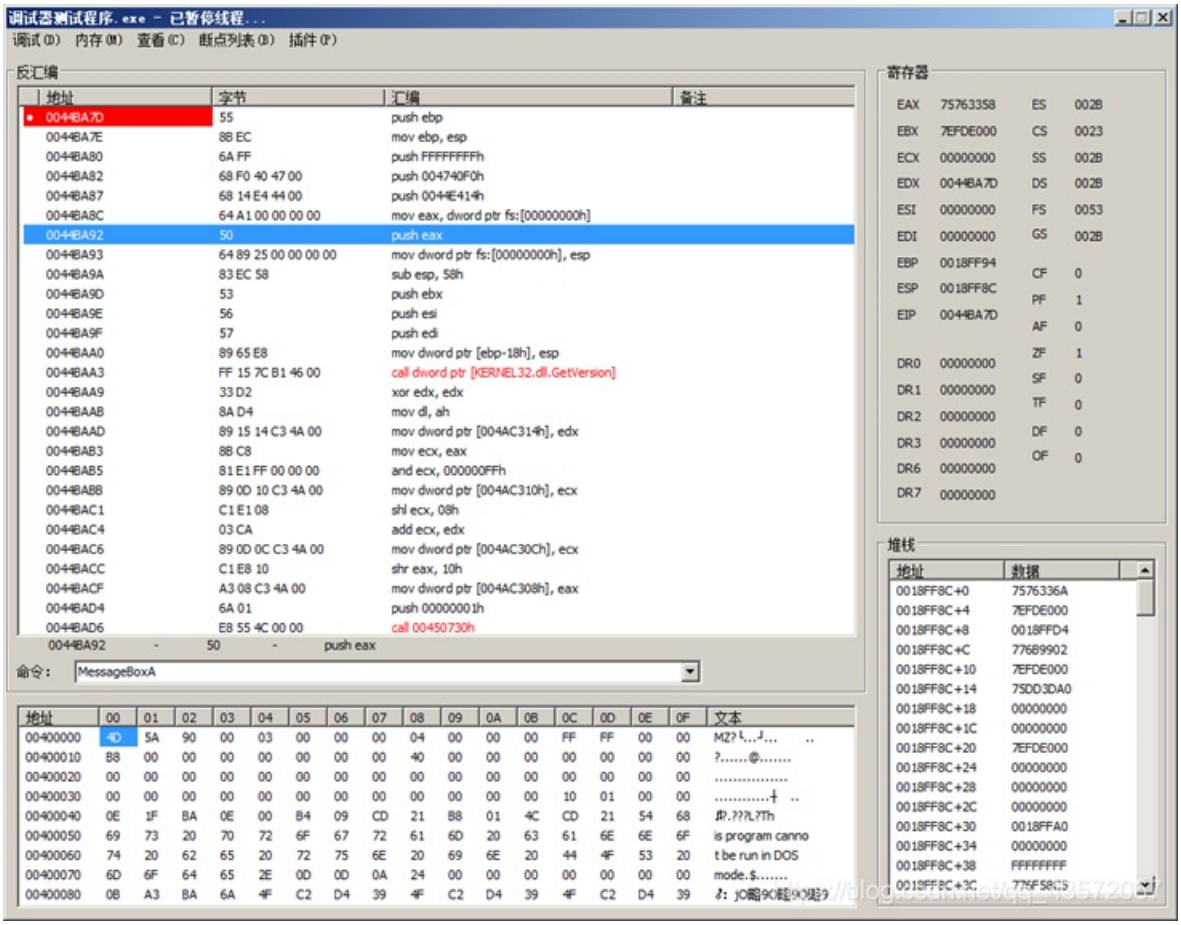
内存断点，执行，读写，访问

反反调试，插件功能

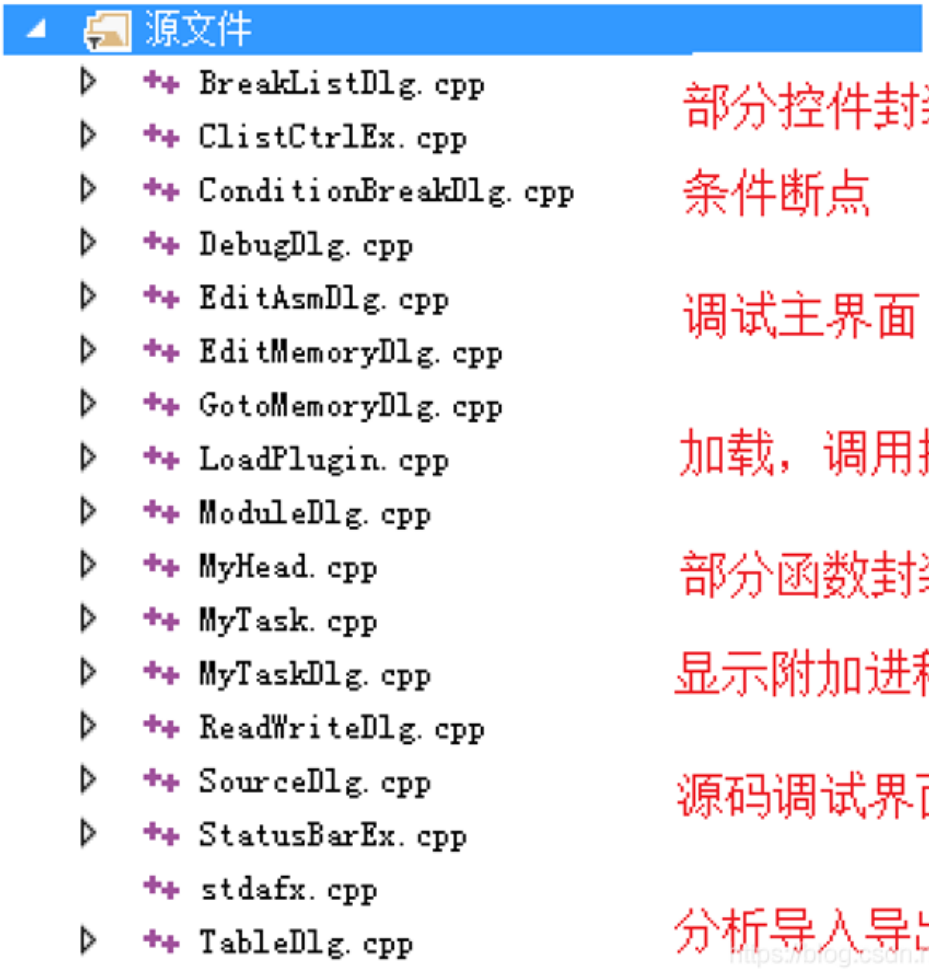
解析符号，源码调试，Dump

关键代码高亮，断点高亮

源码分享：https://github.com/Mr-Hock/MyDebug


调试器测试程序.exe - 已暂停线程...
调试(D) 内存(M) 查看(C) 断点列表(B) 插件(P)
反汇编
地址 字节 汇编 备注
0044BA7D 55 push ebp
0044BA7E 8B EC mov ebp, esp
0044BA80 6A FF push FFFFFFFFh
0044BA82 68 F0 40 47 00 push 004740F0h
0044BA87 68 14 E4 44 00 push 0044E414h
0044BA8C 64 A1 00 00 00 00 mov eax, dword ptr fs:[00000000h]
0044BA92 50 push eax
0044BA93 64 89 25 00 00 00 00 mov dword ptr fs:[00000000h], esp
0044BA9A 83 EC 58 sub esp, 58h
0044BA9D 53 push ebx
0044BA9E 56 push esi
0044BA9F 57 push edi
0044BAA0 89 65 E8 mov dword ptr [ebp-18h], esp
0044BAA3 FF 15 7C B1 46 00 call dword ptr [KERNEL32.dll.GetVersion]
0044BAA9 33 D2 xor edx, edx
0044BAAB 8A D4 mov dl, ah
0044BAAD 89 15 14 C3 4A 00 mov dword ptr [004AC314h], edx
0044BAB3 8B C8 mov ecx, eax
0044BAB5 81 E1 FF 00 00 00 and ecx, 000000FFh
0044BABB 89 0D 10 C3 4A 00 mov dword ptr [004AC310h], ecx
0044BAC1 C1 E1 08 shl ecx, 08h
0044BAC4 03 CA add ecx, edx
0044BAC6 89 0D 0C C3 4A 00 mov dword ptr [004AC30Ch], ecx
0044BACC C1 E8 10 shr eax, 10h
0044BACF A3 08 C3 4A 00 mov dword ptr [004AC308h], eax
0044BAD4 6A 01 push 00000001h
0044BAD6 E8 55 4C 00 00 call 00450730h
0044BA92 - 50 - push eax
命令: MessageBoxA
寄存器
EAX 75763358 ES 002B
EBX 7EFDE000 CS 0023
ECX 00000000 SS 002B
EDX 0044BA7D DS 002B
ESI 00000000 FS 0053
EDI 00000000 GS 002B
EBP 0018FF94
ESP 0018FF8C
EIP 0044BA7D
CF 0
PF 1
AF 0
ZF 1
SF 0
TF 0
DF 0
OF 0
DR0 00000000
DR1 00000000
DR2 00000000
DR3 00000000
DR6 00000000
DR7 00000000
堆栈
地址 数据
0018FF8C+0 7576336A
0018FF8C+4 7EFDE000
0018FF8C+8 0018FFD4
0018FF8C+C 77689902
0018FF8C+10 7EFDE000
0018FF8C+14 75DD3DA0
0018FF8C+18 00000000
0018FF8C+1C 00000000
0018FF8C+20 7EFDE000
0018FF8C+24 00000000
0018FF8C+28 00000000
0018FF8C+2C 00000000
0018FF8C+30 0018FFA0
0018FF8C+34 00000000
0018FF8C+38 FFFFFFFF
0018FF8C+3C 776F58C5
地址 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 文本
00400000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ?...
00400010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ?.....@.....
00400020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00400030 00 00 00 00 00 00 00 00 00 00 00 00 10 01 00 00 ....
00400040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..??.??L?Th
00400050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00400060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00400070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode.$.....
00400080 0B A3 BA 6A 4F C2 D4 39 4F C2 D4 39 4F C2 D4 39

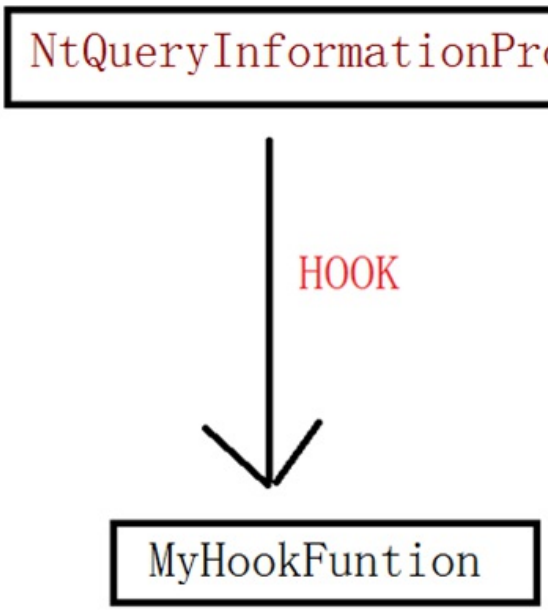
调试器项目代码


源文件
BreakListDlg.cpp
ClistCtrlEx.cpp
ConditionBreakDlg.cpp
DebugDlg.cpp
EditAsmDlg.cpp
EditMemoryDlg.cpp
GotoMemoryDlg.cpp
LoadPlugin.cpp
ModuleDlg.cpp
MyHead.cpp
MyTask.cpp
MyTaskDlg.cpp
ReadWriteDlg.cpp
SourceDlg.cpp
StatusBarEx.cpp
stdafx.cpp
TableDlg.cpp
部分控件封装
条件断点
调试主界面
加载，调用插件类
部分函数封装头文件
显示附加进程列表
源码调试界面
分析导入导出表

插件代码

```
//初始化插件接口，必选!
CHAR* WINAPI InitPlugin()
{
    char * nRet = "MyDebug 反反调试插件";
    return nRet;
}

//创建进程时调用的接口(两个参数，调试器会返回进程id和线程id)
VOID WINAPI CreateProcessEvent(DWORD m_Pid, DWORD m_Tid)
{
    //打开进程句柄
    m_ProcessHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, m_Pid);
    if (m_ProcessHandle == nullptr)MessageBox(NULL, TEXT("C++Plugin打开

    //打开线程句柄
    m_Tid = GetProcessThreadId(m_Pid);
    m_ThreadHandle = OpenThread(THREAD_ALL_ACCESS, FALSE, m_Tid);
    if (m_ProcessHandle == nullptr)MessageBox(NULL, TEXT("C++Plugin打开

    AntiPeb();                          //隐藏PEB
    AntiNtQueryInformationProcess();    //HOOK函数
    AntiZwSetInformationThread();       //HOOK函数
}

//创建线程时调用的接口
VOID WINAPI CreateThreadEvent(HANDLE nHandle, DWORD nThreadLocalBase)
{

}
```


反反调试




```
MyHookFuntion:

mov eax,[esp+08]        取出第2个参数
cmp eax,07              判断是否要Hook的功能
jne @@
mov eax,[esp+0C]        取出第3个参数地址
mov [eax], 0            写0
retn 0014               返回
@@:
mov eax,16              还原代码，跳回
jmp NtQueryInformationProcess + 5
```


解析符号准备

```
//一个模块的导入导出函数表记录结构
typedef struct _TABLE
{
    CString ModuleName;                     //模块的名字
    DWORD ModuleAddress;                    //模块的加载地址
    DWORD ModuleSize;                       //模块的尺寸

    CString ExportModuleName;               //导出表模块的名字
    vector<DWORD>ExportFunctionAddress;     //导出函数地址_十进制
    vector<CString>ExportFunctionAddress_str;   //导出函数地址_十六进制
    vector<CString>ExportFunctionName;      //导出的函数名字

    vector<CString>ImportModuleName;        //导入的模块名字
    vector<CString>ImportFunctionName;      //导入的函数名字
    vector<DWORD>ImportFunctionAddress;     //导入的函数地址_十进制
    vector<CString>ImportFunctionAddress_str;   //导入的函数地址_十六进制
    vector<DWORD>IAT;                       //导入函数所IAT地址_十进制
    vector<CString>IAT_str;                 //导入函数所IAT地址_十六进制

}TABLE, *PTABLE;
```


解析导出符号

```
//循环遍历所有模块的
for (DWORD i = 0; i < m_Table.size(); i++)
{
    //计算出模块的起始地址和结束
    nMinAddress = m_Table[i].ModuleAddress;
    nMaxAddress = nMinAddress + m_Table[i].ModuleSize;

    //判断传入的地址是否属于该模块
    if (nAddress >= nMinAddress && nAddress <= nMaxAddress)
    {
        //确定模块后，遍历此模块的所有导出函数地址
        for (DWORD x = 0; x < m_Table[i].ExportFunctionAddress.size(); x++)
        {
            //如果导出函数的地址与传入的地址相等
            if (nAddress == m_Table[i].ExportFunctionAddress[x])
            {
                //返回该导出函数的地址所对应的模块名和导出函数名字
                return CString(m_Table[i].ExportModuleName + CString(".") +
```

反汇编

地址	字节
ntdll.dll.NtQueryInformationProcess	E9 13 05 B1 88
7769FAED	33 C9
7769FAEF	8D 54 24 04
7769FAF3	64 FF 15 C0 00 00 00
7769FAFA	83 C4 04
7769FAFD	C2 14 00
ntdll.dll.NtWaitForMultipleObjects32	B8 17 00 00 00
7769FB05	B9 1E 00 00 00
7769FB0A	8D 54 24 04
7769FB0E	64 FF 15 C0 00 00 00
7769FB15	83 C4 04
7769FB18	C2 14 00
7769FB1B	90
ntdll.dll.NtWriteFileGather	B8 18 00 00 00
7769FB21	B9 1A 00 00 00
7769FB26	8D 54 24 04

解析导入符号

```
//遍历导入表所有信息
for (DWORD x = 0; x < m_Table[i].ImportFunctionAddress.size(); x++)
{
    //在汇编代码里查找是否有导入函数的地址
    INT nIndex = szAsm.Find(m_Table[i].ImportFunctionAddress_str[x]);
    //查找到了导入函数的地址
    if (nIndex != -1)
    {
        //直接将汇编代码内的地址替换为导入模块名与导入函数名
        szAsm.Replace(m_Table[i].ImportFunctionAddress_str[x] + TEXT("h"), m_Ta
        return szAsm;
    }
```


反汇编

地址	字节	汇编
KERNELBASE.dll.GetFileSize	8B FF	mov edi, edi
7632E12F	55	push ebp
7632E130	8B EC	mov ebp, esp
7632E132	51	push ecx
7632E133	51	push ecx
7632E134	8D 45 F8	lea eax, dword ptr [ebp-08h]
7632E137	50	push eax
7632E138	FF 75 08	push dword ptr [ebp+08h]
7632E13B	E8 35 EE FF FF	call GetFileSizeEx
7632E140	85 C0	test eax, eax
7632E142	74 1C	je 7632E160h
7632E144	8B 45 0C	mov eax, dword ptr [ebp+0Ch]
7632E147	85 C0	test eax, eax
7632E149	74 05	je 7632E150h
7632E14B	8B 4D FC	mov ecx, dword ptr [ebp-04h]
7632E14E	89 08	mov dword ptr [eax], ecx
7632E150	83 7D F8 FF	cmp dword ptr [ebp-08h], FFFFFFFFh
7632E154	75 0E	jne 7632E164h
7632E156	6A 00	push 00000000h
7632E158	FF 15 44 10 32 76	call dword ptr [ntdll.dll.RtlSetLastWin32Error]

源码调试

```
//打开进程获得进程句柄
m_SymHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, m_Pid);

//初始化符号处理器
SymInitialize(m_SymHandle,NULL,FALSE);

//载入符号文件
SymLoadModule64(m_SymHandle,pInfo->hFile,NULL,NULL,(DWORD64)pInfo->lpBaseOfImage,0)
```



```
DWORD nDisplacement = 0;
IMAGEHLP_LINE64 nSourceInfo{};                          //源码信息结构体
nSourceInfo.SizeOfStruct = sizeof(IMAGEHLP_LINE64); //初始化结构体
//取出源码信息(进程句柄，发生异常的地址，传0，输出源码相关信息结构体)
SymGetLineFromAddr64(m_SymHandle, nAddress,&nDisplacement,&nSourceInfo);
```

```
typedef struct _IMAGEHLP_LINE64 {
    DWORD    SizeOfStruct;           // set to sizeof
    PVOID    Key;                    // internal
    DWORD    LineNumber;             //源码在第几行
    PCHAR    FileName;               //源码文件的路径
    DWORD64  Address;                // first instruct
} IMAGEHLP_LINE64, *PIMAGEHLP_LINE64;
```


Dump文件

```
//读取文件头信息
ReadMemoryBytes(nImageBassAddress, nPeHeadData, 4096);

//获取PE信息
PIMAGE_DOS_HEADER nDosHead = (PIMAGE_DOS_HEADER)nPeHeadData;
PIMAGE_NT_HEADERS nNtHead = (PIMAGE_NT_HEADERS)(nPeHeadData + nDosHead->e_lfanew);
PIMAGE_SECTION_HEADER nSecetionHead = IMAGE_FIRST_SECTION(nNtHead);

//PE头大小
nPeSize = nNtHead->OptionalHeader.SizeOfHeaders;
//文件的尺寸
nImageSize = nNtHead->OptionalHeader.SizeOfImage;
//区段数量
nSectionNum = nNtHead->FileHeader.NumberOfSections;

//申请exe所需的堆空间
nImageBuf = new BYTE[nImageSize]{};

//读取PE数据
ReadMemoryBytes(nImageBassAddress, nImageBuf, nPeSize);

nFileSize += nPeSize;
//读取每个区段的数据
for (DWORD i = 0; i < nSectionNum; i++)
{
    ReadMemoryBytes(nImageBassAddress + nSecetionHead[i].VirtualAddress, nImageBuf
    nFileSize += nSecetionHead[i].SizeOfRawData;
}
```