反调试技术 VC 版

唐久涛

看雪 ID：tangjiutao

本人空间：http://ucooper.com

写意互联网，关注搜索引擎技术，涉猎搜索引擎优化、软件破解、PHP 网站建设、Wordpress 应用等

声明：这篇文章是本人学习的总结，理论部分参考了《脱壳的艺术》、《加密与解密》以及本人从网络上收集的资料，在此向原作者致敬。本人的贡献在于根据个人理解对各种反调试技术进行了汇总和高度归纳，并提供了本人创作的各种反调试实例及源代码。本人于 09 年 9 月份开始学习软件逆向工程的相关知识，在学习过程中得到大量网友的热心帮助，在此向各位致以诚挚谢意。希望本人的这些工作能够对各位有所帮助，浅陋之处，莫要见笑。各种形式的转载都必须保留作者信息及本声明。

由于本人入门较晚、能力有限，部分方法尚未实现，望高手不吝赐教。实现了的方法大都附有实例程序。很多方法对于修改版的 OD 已经失效，请用原版 OD 进行测试。

发现 OD 的处理

一、如何获取 OD 窗口的句柄

1.已经获取了窗口类名或标题：FindWindow 函数

2.没有获取窗口类名或标题：GetForeGroundWindow 返回前台窗口，这里就是 OD 的窗口句柄了。注意这种方法更为重要，因为大多数情况下不会知道 OD 的窗口类名。

```
        invoke    IsDebuggerPresent
        .if       eax
                    invoke    GetForegroundWindow     ;获得的是 OD 的窗口句柄
                    invoke    SendMessage,eax,WM_CLOSE,NULL,NULL
        .endif
```

二、获取 OD 窗口句柄后的处理

（1）向窗口发送 WM_CLOSE 消息

```
void CDetectODDlg::OnWndcls()
{
    // TODO: Add your control notification handler code here
    HWND hWnd;
    if(hWnd=::FindWindow("OllyDbg",NULL))
    {
        MessageBox("发现 OD");
        ::SendMessage(hWnd,WM_CLOSE,NULL,NULL);
    }else{
        MessageBox("没发现 OD");
    }
}
```

（2）使 OD 窗口不可用

```
HWND hd_od=FindWindow("ollydbg",NULL);
SetWindowLong(hd_od,GWL_STYLE,WS_DISABLED);
```

（3）终止相关进程，根据窗口句柄获取进程 ID，根据进程 ID 获取进程句柄，

```
    HWND hWnd;
    HANDLE hProc;
    DWORD pId;
```

```
    if(hWnd=::FindWindow("OllyDbg",NULL))        //获取窗口句柄
    {
        MessageBox("发现 OD");
        GetWindowThreadProcessId(hWnd,&pId);   //获取进程 ID
        hProc=OpenProcess(PROCESS_TERMINATE,TRUE,pId); //获取进程句柄
        TerminateProcess(hProc,200);   //终止进程
        CloseHandle(hProc);
    }else{
        MessageBox("没发现 OD");
    }
```

（2）程序自身直接退出

1. 窗口类名、窗口名

（1） FindWindow

（2） EnumWindow 函数调用后，系统枚举所有顶级窗口，为每个窗口调用一次回调函数。在回调函数中用 GetWindowText 得到窗口标题，用 strstr 等函数查找有无 Ollydbg 字符串。StrStr(大小写敏感，对应的 StrStrI 大小写不敏感)函数返回 str2 第一次出现在 str1 中的位置，如果没有找到，返回 NULL。

（3） GetForeGroundWindow 返回前台窗口（用户当前工作的窗口）。当程序被调试时，调用这个函数将获得 Ollydbg 的窗口句柄，这样就可以向其发送 WM_CLOSE 消息将其关闭了。

（1）FindWindow

```
void CDetectODDlg::OnWndcls()
{
    // TODO: Add your control notification handler code here
    HWND hWnd;
    if(hWnd=::FindWindow("OllyDbg",NULL))
    {
        MessageBox("发现 OD");
        ::SendMessage(hWnd,WM_CLOSE,NULL,NULL);
    }else{
        MessageBox("没发现 OD");
    }
}
```

（2）EnumWindow

```
包含头文件：#include "Shlwapi.h"
BOOL CALLBACK EnumWindowsProc(
  HWND hwnd,          // handle to parent window
  LPARAM lParam      // application-defined value
  )
{
```

```
    char ch[100];
    CString str="Ollydbg";
    if(IsWindowVisible(hwnd))
    {
        ::GetWindowText(hwnd,ch,100);
        //AfxMessageBox(ch);
        if(::StrStrI(ch,str))
        {
            AfxMessageBox("发现 OD");
            return FALSE;
        }
    }
    return TRUE;
}

void CDetectODDlg::OnEnumwindow()
{
    // TODO: Add your control notification handler code here
    EnumWindows(EnumWindowsProc,NULL);
    AfxMessageBox("枚举窗口结束，未提示发现 OD，则没有 OD");
}
```

2. 检测调试器进程

枚举进程列表，看是否有调试器进程（OLLYDBG.EXE,windbg.exe 等）。
利用 kernel32!ReadProcessMemory()读取进程内存，然后寻找调试器相关的字符串（如”OLLYDBG”）以防止逆向分析人员修改调试器的可执行文件名。

```
需要头文件：#include "tlhelp32.h"
void CDetectODDlg::OnEnumProcess()
{
    // TODO: Add your control notification handler code here

    HANDLE hwnd;
    PROCESSENTRY32 tp32;   //结构体
    CString str="OLLYDBG.EXE";
    BOOL bFindOD=FALSE;
    hwnd=::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,NULL);
    if(INVALID_HANDLE_VALUE!=hwnd)
    {
        Process32First(hwnd,&tp32);
        do{
            if(0==lstrcmp(str,tp32.szExeFile))
            {
                AfxMessageBox("发现 OD");
```

```
                    bFindOD=TRUE;
                    break;
                }
            }while(Process32Next(hwnd,&tp32));
            if(!bFindOD)
                AfxMessageBox("没有 OD");
        }
        CloseHandle(hwnd);
}
```

3. 父进程是否是 Explorer

原理：通常进程的父进程是 explorer.exe（双击执行的情况下），否则可能程序被调试。
下面是实现这种检查的一种方法：
1．通过 TEB(TEB.ClientId)或者使用 GetCurrentProcessId()来检索当前进程的 PID
2．用 Process32First/Next()得到所有进程的列表，注意 explorer.exe 的 PID（通过 PROCESSENTRY32.szExeFile）和通过 PROCESSENTRY32.th32ParentProcessID 获得的当前进程的父进程 PID。Explorer 进程 ID 也可以通过桌面窗口类和名称获得。
3．如果父进程的 PID 不是 explorer.exe，cmd.exe，Services.exe 的 PID，则目标进程很可能被调试
对策：Olly Advanced 提供的方法是让 Process32Next()总是返回 fail，使进程枚举失效，PID 检查将会被跳过。这些是通过补丁 kernel32!Process32NextW()的入口代码（将 EAX 值设为 0 然后直接返回）实现的。
（1）通过桌面类和名称获得 Explorer 的 PID 源码见附件

```
    DWORD ExplorerID;
    ::GetWindowThreadProcessId(::FindWindow("Progman",NULL),&ExplorerID);
```

（2）通过进程列表快照获得 Explorer 的 PID 源码见附件

```
void CDetectODDlg::OnExplorer()
{
    // TODO: Add your control notification handler code here
    HANDLE hwnd;
    PROCESSENTRY32 tp32;   //结构体
    CString str="Explorer.EXE";

    DWORD ExplorerID;
    DWORD SelfID;
    DWORD SelfParentID;
    SelfID=GetCurrentProcessId();
    ::GetWindowThreadProcessId(::FindWindow("Progman",NULL),&ExplorerID);
    hwnd=::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,NULL);
    if(INVALID_HANDLE_VALUE!=hwnd)
    {
        Process32First(hwnd,&tp32);
        do{
```

```
            if(0==lstrcmp(str,tp32.szExeFile))
            {
            //    ExplorerID=tp32.th32ProcessID;
            //    AfxMessageBox("aaa");
            }
            if(SelfID==tp32.th32ProcessID)
            {
                SelfParentID=tp32.th32ParentProcessID;
            }
        }while(Process32Next(hwnd,&tp32));

        str.Format(" 本 进 程 ： %d 父 进 程 ： %d Explorer 进 程 ： %d
",SelfID,SelfParentID,ExplorerID);
        MessageBox(str);
        if(ExplorerID==SelfParentID)
        {
            AfxMessageBox("没有 OD");
        }
        else
        {
            AfxMessageBox("发现 OD");
        }
    }
    CloseHandle(hwnd);
}
```

4. RDTSC/ GetTickCount 时间敏感程序段

当进程被调试时，调试器事件处理代码、步过指令等将占用 CPU 循环。如果相邻指令之间所花费的时间如果大大超出常规，就意味着进程很可能是在被调试。

（1）RDTSC

将计算机启动以来的 CPU 运行周期数放到 EDX：EAX 里面，EDX 是高位，EAX 是低位。

如果 CR4 的 TSD(time stamp disabled)置位,则 rdtsc 在 ring3 下运行会导致异常(特权指令),所以进入 ring0,把这个标记置上,然后 Hook OD 的 WaitForDebugEvent,拦截异常事件,当异常代码为特权指令时,把异常处的 opcode 读出检查,如果是 rdtsc,把 eip 加 2,SetThreadContext,edx:eax 的返回由你了。

（2）GetTickCount 源码见附件

```
void CDetectODDlg::OnGetTickCount()
{
    // TODO: Add your control notification handler code here
```

```
    DWORD dTime1;
    DWORD dTime2;
    dTime1=GetTickCount();
    GetCurrentProcessId();
    GetCurrentProcessId();
    GetCurrentProcessId();
    GetCurrentProcessId();
    dTime2=GetTickCount();
    if(dTime2-dTime1>100)
    {
        AfxMessageBox("发现 OD");
    }
    else{
        AfxMessageBox("没有 OD");
    }
}
```

5. StartupInfo 结构

原理:Windows 操作系统中的 explorer.exe 创建进程的时候会把 STARTUPINFO 结构中的值设为 0,而非 explorer.exe 创建进程的时候会忽略这个结构中的值，也就是结构中的值不为 0，所以可以利用这个来判断 OD 是否在调试程序.

****

```
结构体
typedef struct _STARTUPINFO
{
    DWORD cb;                0000
    PSTR lpReserved;         0004
    PSTR lpDesktop;          0008
    PSTR lpTitle;            000D
    DWORD dwX;               0010
    DWORD dwY;               0014
    DWORD dwXSize;           0018
    DWORD dwYSize;           001D
    DWORD dwXCountChars;     0020
    DWORD dwYCountChars;     0024
    DWORD dwFillAttribute;   0028
    DWORD dwFlags;           002D
    WORD wShowWindow;        0030
    WORD cbReserved2;        0034
    PBYTE lpReserved2;       0038
    HANDLE hStdInput;        003D
    HANDLE hStdOutput;       0040
    HANDLE hStdError;        0044
```

```
} STARTUPINFO, *LPSTARTUPINFO;
void CDetectODDlg::OnGetStartupInfo()
{
	// TODO: Add your control notification handler code here
	STARTUPINFO info;
	GetStartupInfo(&info);
	if(info.dwX!=0 || info.dwY!=0 || info.dwXCountChars!=0 || info.dwYCountChars!=0
		|| info.dwFillAttribute!=0 || info.dwXSize!=0 || info.dwYSize!=0)
	{
		AfxMessageBox("发现 OD");
	}
	else{
		AfxMessageBox("没有 OD");
	}

}
```

6. BeingDebugged

kernel32!IsDebuggerPresent() API 检测进程环境块(PEB)中的 BeingDebugged 标志检查这个标志以确定进程是否正在被用户模式的调试器调试。
每个进程都有 PEB 结构，一般通过 TEB 间接得到 PEB 地址
Fs:[0]指向当前线程的 TEB 结构，偏移为 0 处是线程信息块结构 TIB
TIB 偏移 18H 处是 self 字段，是 TIB 的反身指针，指向 TIB（也是 PEB）首地址
TEB 偏移 30H 处是指向 PEB 结构的指针
PEB 偏移 2H 处，就是 BeingDebugged 字段，Uchar 类型
（1） 调用 IsDebuggerPresent 函数，间接读 BeingDebugged 字段
（2） 利用地址直接读 BeingDebugged 字段
对策：
（1） 数据窗口中 Ctrl+G fs:[30] 查看 PEB 数据,将 PEB.BeingDebugged 标志置 0
（2） Ollyscript 命令"dbh"可以补丁这个标志

```
void CDetectODDlg::OnIsdebuggerpresent()
{
	// TODO: Add your control notification handler code here
	if(IsDebuggerPresent())
	{
		MessageBox("发现 OD");
	}
	else
	{
		MessageBox("没有 OD");
	}
}
```

7. PEB.NtGlobalFlag , Heap.HeapFlags, Heap.ForceFlags

（1）通常程序没有被调试时，PEB 另一个成员 NtGlobalFlag（偏移 0x68）值为 0，如果进程被调试通常值为 0x70（代表下述标志被设置）：
FLG_HEAP_ENABLE_TAIL_CHECK(0X10)
FLG_HEAP_ENABLE_FREE_CHECK(0X20)
FLG_HEAP_VALIDATE_PARAMETERS(0X40)
这些标志是在 ntdll!LdrpInitializeExecutionOptions()里设置的。请注意 PEB.NtGlobalFlag 的默认值可以通过 gflags.exe 工具或者在注册表以下位置创建条目来修改：
HKLM\Software\Microsoft\Windows Nt\CurrentVersion\Image File Execution Options

```
assume  fs:nothing
mov     eax,fs:[30h]
mov     eax,[eax+68h]
and     eax,70h
```

（2）由于 NtGlobalFlag 标志的设置，堆也会打开几个标志，这个变化可以在 ntdll!RtlCreateHeap()里观测到。正常情况下系统为进程创建第一个堆时会将 Flags 和 ForceFlags 分别设为 2（HEAP_GROWABLE)和 0 。当进程被调试时，这两个标志通常被设为 50000062（取决于 NtGlobalFlag）和 0x40000060（等于 Flags AND 0x6001007D）。

```
assume  fs:nothing
mov     ebx,fs:[30h]        ;ebx 指向 PEB
mov     eax,[ebx+18h]       ;PEB.ProcessHeap
cmp     dword ptr [eax+0ch],2       ;PEB.ProcessHeap.Flags
jne     debugger_found
cmp dword ptr [eax+10h],0           ;PEB.ProcessHeap.ForceFlags
jne   debugger_found
```

这些标志位都是因为 BeingDebugged 引起的。系统创建进程的时候设置 BeingDebugged=TRUE ，后来 NtGlobalFlag 根据这个标记设置 FLG_VALIDATE_PARAMETERS 等标记。在为进程创建堆时，又由于 NtGlobalFlag 的作用，堆的 Flags 被设置了一些标记，这个 Flags 随即被填充到 ProcessHeap 的 Flags 和 ForceFlags 中，同时堆中被填充了很多 BAADF00D 之类的东西（HeapMagic，也可用来检测调试）。
一次性解决这些状态见加密解密 P413

```
//*************************************************
typedef ULONG NTSTATUS;
typedef ULONG PPEB;
typedef ULONG KAFFINITY;
typedef ULONG KPRIORITY;

typedef struct _PROCESS_BASIC_INFORMATION { // Information Class 0
NTSTATUS ExitStatus;
PPEB PebBaseAddress;
KAFFINITY AffinityMask;
KPRIORITY BasePriority;
ULONG UniqueProcessId;
ULONG InheritedFromUniqueProcessId;
```

```
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;

typedef enum _PROCESSINFOCLASS {
ProcessBasicInformation, // 0 Y N
ProcessQuotaLimits, // 1 Y Y
ProcessIoCounters, // 2 Y N
ProcessVmCounters, // 3 Y N
ProcessTimes, // 4 Y N
ProcessBasePriority, // 5 N Y
ProcessRaisePriority, // 6 N Y
ProcessDebugPort, // 7 Y Y
ProcessExceptionPort, // 8 N Y
ProcessAccessToken, // 9 N Y
ProcessLdtInformation, // 10 Y Y
ProcessLdtSize, // 11 N Y
ProcessDefaultHardErrorMode, // 12 Y Y
ProcessIoPortHandlers, // 13 N Y
ProcessPooledUsageAndLimits, // 14 Y N
ProcessWorkingSetWatch, // 15 Y Y
ProcessUserModeIOPL, // 16 N Y
ProcessEnableAlignmentFaultFixup, // 17 N Y
ProcessPriorityClass, // 18 N Y
ProcessWx86Information, // 19 Y N
ProcessHandleCount, // 20 Y N
ProcessAffinityMask, // 21 N Y
ProcessPriorityBoost, // 22 Y Y
ProcessDeviceMap,// 23 Y Y
ProcessSessionInformation, // 24 Y Y
ProcessForegroundInformation, // 25 N Y
ProcessWow64Information // 26 Y N
} PROCESSINFOCLASS;

typedef NTSTATUS (_stdcall *ZwQueryInformationProcess)(
HANDLE ProcessHandle,
PROCESSINFOCLASS ProcessInformationClass,
PVOID ProcessInformation,
ULONG ProcessInformationLength,
PULONG ReturnLength
); //定义函数指针
void CDetectODDlg::OnPebflags()
{
	// TODO: Add your control notification handler code here

	//定义函数指针变量
```

```
        ZwQueryInformationProcess MyZwQueryInformationProcess;

        HANDLE hProcess = NULL;
        PROCESS_BASIC_INFORMATION pbi = {0};
       ULONG peb = 0;
       ULONG cnt = 0;
        ULONG PebBase = 0;
        ULONG AddrBase;
        BOOL bFoundOD=FALSE;
        WORD flag;
        DWORD dwFlag;
        DWORD bytesrw;
        DWORD ProcessId=GetCurrentProcessId();
      hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE,
ProcessId);
      if (hProcess != NULL) {
            //函数指针变量赋值

       MyZwQueryInformationProcess=(ZwQueryInformationProcess)GetProcAddress(LoadLibrary("ntdll.dll"),"Z
wQueryInformationProcess");
           //函数指针变量调用
             if (MyZwQueryInformationProcess(
                    hProcess,
                       ProcessBasicInformation,
                       &pbi,
                       sizeof(PROCESS_BASIC_INFORMATION),
                       &cnt) == 0)
             {
                PebBase = (ULONG)pbi.PebBaseAddress;   //获取 PEB 地址
                  AddrBase=PebBase;
                  if       (ReadProcessMemory(hProcess,(LPCVOID)(PebBase+0x68),&flag,2,&bytesrw)       &&
bytesrw==2)   //读内存地址
                  { //PEB.NtGlobalFlag
                       if(0x70==flag){
                            bFoundOD=TRUE;
                       }
                  }
                  if     (ReadProcessMemory(hProcess,(LPCVOID)(PebBase+0x18),&dwFlag,4,&bytesrw)     &&
bytesrw==4)
                  {
                       AddrBase=dwFlag;
                  }
                  if      (ReadProcessMemory(hProcess,(LPCVOID)(AddrBase+0x0c),&flag,2,&bytesrw)      &&
bytesrw==2)
```

```
                {//PEB.ProcessHeap.Flags
                    if(2!=flag){
                        bFoundOD=TRUE;
                    }
                }
                if (ReadProcessMemory(hProcess,(LPCVOID)(AddrBase+0x10),&flag,2,&bytesrw) &&
bytesrw==2)
                {//PEB.ProcessHeap.ForceFlags
                    if(0!=flag){
                        bFoundOD=TRUE;
                    }
                }
                if(bFoundOD==FALSE)
                {
                    AfxMessageBox("没有 OD");
                }
                else
                {
                    AfxMessageBox("发现 OD");
                }
            }
            CloseHandle(hProcess);
        }
}
```

8. DebugPort: CheckRemoteDebuggerPresent()/NtQueryInformationProcess()

Kernel32!CheckRemoteDebuggerPresent()是用于确定是否有调试器被附加到进程。

```
BOOL CheckRemoteDebuggerPresent(
  HANDLE   hProcess,
  PBOOL    pbDebuggerPresent
)
```

Kernel32!CheckRemoteDebuggerPresent()接受 2 个参数，第 1 个参数是进程句柄，第 2 个参数是一个指向 boolean 变量的指针，如果进程被调试，该变量将包含 TRUE 返回值。

这个 API 内部调用了 ntdll!NtQueryInformationProcess()，由它完成检测工作。

```
typedef BOOL (WINAPI *CHECK_REMOTE_DEBUGGER_PRESENT)(HANDLE, PBOOL); //定义函数指针

void CDetectODDlg::OnCheckremotedebuggerpresent()
{
    // TODO: Add your control notification handler code here
    HANDLE          hProcess;
    HINSTANCE       hModule;
    BOOL            bDebuggerPresent = FALSE;
    CHECK_REMOTE_DEBUGGER_PRESENT CheckRemoteDebuggerPresent; //建立函数指针变量
```

```
    hModule = GetModuleHandleA("Kernel32");          //地址要从模块中动态获得
    CheckRemoteDebuggerPresent =
        (CHECK_REMOTE_DEBUGGER_PRESENT)GetProcAddress(hModule,
"CheckRemoteDebuggerPresent");          //获取地址
    hProcess = GetCurrentProcess();
    CheckRemoteDebuggerPresent(hProcess,&bDebuggerPresent);   //调用
     if(bDebuggerPresent==TRUE)
     {
         AfxMessageBox("发现 OD");
     }
     else
     {
         AfxMessageBox("没有 OD");
     }
}
```

ntdll!NtQueryInformationProcess()有 5 个参数。

为了检测调试器的存在，需要将 ProcessInformationclass 参数设为 ProcessDebugPort(7)。NtQueryInformationProcess()检索内核结构 EPROCESS5 的 DebugPort 成员，这个成员是系统用来与调试器通信的端口句柄。非 0 的 DebugPort 成员意味着进程正在被用户模式的调试器调试。如果是这样的话，ProcessInformation 将被置为 0xFFFFFFFF ，否则 ProcessInformation 将被置为 0。

```
ZwQueryInformationProcess(
IN HANDLE ProcessHandle,
IN PROCESSINFOCLASS ProcessInformationClass,
OUT PVOID ProcessInformation,
IN ULONG ProcessInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
//*********************************************************
typedef NTSTATUS (_stdcall *ZW_QUERY_INFORMATION_PROCESS)(
HANDLE ProcessHandle,
PROCESSINFOCLASS ProcessInformationClass, //该参数也需要上面声明的数据结构
PVOID ProcessInformation,
ULONG ProcessInformationLength,
PULONG ReturnLength
); //定义函数指针

void CDetectODDlg::OnZwqueryinfomationprocess()
{
    // TODO: Add your control notification handler code here
    HANDLE          hProcess;
    HINSTANCE       hModule;
    DWORD           dwResult;
    ZW_QUERY_INFORMATION_PROCESS MyFunc;
```

```
    hModule = GetModuleHandle("ntdll.dll");

MyFunc=(ZW_QUERY_INFORMATION_PROCESS)GetProcAddress(hModule,"ZwQueryInfor
mationProcess");
    hProcess = GetCurrentProcess();
    MyFunc(
        hProcess,
        ProcessDebugPort,
        &dwResult,
        4,
        NULL);
    if(dwResult!=0)
    {
        AfxMessageBox("发现 OD");
    }
    else
    {
        AfxMessageBox("没有 OD");
    }
}
```

9. SetUnhandledExceptionFilter/ Debugger Interrupts

调试器中步过 INT3 和 INT1 指令的时候，由于调试器通常会处理这些调试中断，所以设置的异常处理例程默认情况下不会被调用，Debugger Interrupts 就利用了这个事实。这样我们可以在异常处理例程中设置标志，通过 INT 指令后如果这些标志没有被设置则意味着进程正在被调试。另外，kernel32!DebugBreak()内部是调用了 INT3 来实现的，有些壳也会使用这个 API。注意测试时，在异常处理里取消选中 INT3 breaks 和 Singal-step break

```
安全地址的获取是关键
//************************************************************
static DWORD lpOldHandler;
typedef LPTOP_LEVEL_EXCEPTION_FILTER (_stdcall    *pSetUnhandledExceptionFilter)(
                                LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
                                );
pSetUnhandledExceptionFilter lpSetUnhandledExceptionFilter;

LONG WINAPI TopUnhandledExceptionFilter(
    struct _EXCEPTION_POINTERS *ExceptionInfo
)
{
    _asm pushad
    AfxMessageBox("回调函数");
    lpSetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER )lpOldHandler);
```

```
	ExceptionInfo->ContextRecord->Eip=NewEip;//转移到安全位置
	_asm popad
	return EXCEPTION_CONTINUE_EXECUTION;
}

void CDetectODDlg::OnSetUnhandledExceptionFilter()
{
	bool isDebugged=0;
	// TODO: Add your control notification handler code here
	lpSetUnhandledExceptionFilter =
(pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(("kernel32.dll")),
  "SetUnhandledExceptionFilter");
	lpOldHandler=(DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilter);
	_asm{	//获取这个安全地址
		call me		//方式一，需要 NewEip 加上一个偏移值
me:
		pop NewEip	//方式一结束
		mov NewEip,offset safe //方式二，更简单
		int 3	//触发异常
	}
	AfxMessageBox("检测到 OD");
	isDebugged=1;
	_asm{
safe:
	}
	if(1==isDebugged){

	}else{
		AfxMessageBox("没有 OD");
	}
}
//*************************************************************
```

由于调试中断而导致执行停止时，在 OllyDbg 中识别出异常处理例程（通过视图->SEH 链）并下断点，然后 Shift+F9 将调试中断/异常传递给异常处理例程，最终异常处理例程中的断点会断下来，这时就可以跟踪了。

另一个方法是允许调试中断自动地传递给异常处理例程。在 OllyDbg 中可以通过 选项-> 调试选项 -> 异常 -> 忽略下列异常 选项卡中钩选"INT3 中断"和"单步中断"复选框来完成设置。

10.Trap Flag 单步标志异常

TF=1 的时候，会触发单步异常。该方法属于异常处理，不过比较特殊：未修改的 OD 无论是 F9 还是 F8 都不能处理异常，有插件的 OD 在 F9 时能正确处理，F8 时不能正确处理。

```
        void CDetectODDlg::OnTrapFlag()
{
    try{
        _asm{
            pushfd                          //触发单步异常
            or          dword ptr [esp],100h    ;TF=1
            popfd
        }
        AfxMessageBox("检测到 OD");
    }catch(...){
        AfxMessageBox("没有 OD");
    }
}
```

11.SeDebugPrivilege 进程权限

默认情况下进程没有 SeDebugPrivilege 权限，调试时，会从调试器继承这个权限，可以通过打开 CSRSS.EXE 进程间接地使用 SeDebugPrivilege 确定进程是否被调试。注意默认情况下这一权限仅仅授予了 Administrators 组的成员。可以使用 ntdll!CsrGetProcessId() API 获取 CSRSS.EXE 的 PID，也可以通过枚举进程来得到 CSRSS.EXE 的 PID。
实例测试中，OD 载入后，第一次不能正确检测，第二次可以，不知为何。

```
        void CDetectODDlg::OnSeDebugPrivilege()
{
    // TODO: Add your control notification handler code here
    HANDLE hProcessSnap;
    HANDLE hProcess;
    PROCESSENTRY32 tp32;   //结构体
    CString str="csrss.exe";
    hProcessSnap=::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,NULL);
    if(INVALID_HANDLE_VALUE!=hProcessSnap)
    {
        Process32First(hProcessSnap,&tp32);
        do{
            if(0==lstrcmpi(str,tp32.szExeFile))
            {

    hProcess=OpenProcess(PROCESS_QUERY_INFORMATION,NULL,tp32.th32ProcessID);
                if(NULL!=hProcess)
                {
```

```
                    AfxMessageBox("发现 OD");
                }
                else
                {
                    AfxMessageBox("没有 OD");
                }
                CloseHandle(hProcess);
            }
        }while(Process32Next(hProcessSnap,&tp32));
    }
    CloseHandle(hProcessSnap);
}
```

12.DebugObject: NtQueryObject()

除了识别进程是否被调试之外，其他的调试器检测技术牵涉到检查系统当中是否有调试器正在运行。逆向论坛中讨论的一个有趣的方法就是检查 DebugObject 类型内核对象的数量。这种方法之所以有效是因为每当一个应用程序被调试的时候，将会为调试对话在内核中创建一个 DebugObject 类型的对象。

DebugObject 的数量可以通过 ntdll!NtQueryObject()检索所有对象类型的信息而获得。NtQueryObject 接受 5 个参数，为了查询所有的对象类型，ObjectHandle 参数被设为 NULL，ObjectInformationClass 参数设为 ObjectAllTypeInformation(3)：

```
NTSTATUS NTAPI NtQueryObject(
  IN      HANDLE                    ObjectHandle,
  IN      OBJECT_INFORMATION_CLASS  ObjectInformationClass,
  OUT     PVOID                     ObjectInformation,
  IN      ULONG                     Length,
  OUT     PULONG                    ResultLength
)
```

这个 API 返回一个 OBJECT_ALL_INFORMATION 结构，其中 NumberOfObjectsTypes 成员为所有的对象类型在 ObjectTypeInformation 数组中的计数：

```
typedef struct _OBJECT_ALL_INFORMATION{
    ULONG                       NumberOfObjectsTypes;
    OBJECT_TYPE_INFORMATION     ObjectTypeInformation[1];
}
```

检测例程将遍历拥有如下结构的 ObjectTypeInformation 数组：

```
typedef struct _OBJECT_TYPE_INFORMATION{
  [00] UNICODE_STRING       TypeName;
  [08] ULONG                TotalNumberofHandles;
  [0C] ULONG              TotalNumberofObjects;
  ...more fields...
}
```

TypeName 成员与 UNICODE 字符串"DebugObject"比较，然后检查 TotalNumberofObjects 或 TotalNumberofHandles 是否为非 0 值。

```
#ifndef STATUS_INFO_LENGTH_MISMATCH
#define STATUS_INFO_LENGTH_MISMATCH    ((UINT32)0xC0000004L)
#endif

typedef enum _POOL_TYPE {
    NonPagedPool,
    PagedPool,
    NonPagedPoolMustSucceed,
    DontUseThisType,
    NonPagedPoolCacheAligned,
    PagedPoolCacheAligned,
    NonPagedPoolCacheAlignedMustS
} POOL_TYPE;

typedef struct _UNICODE_STRING {
        USHORT Length;
        USHORT MaximumLength;
        PWSTR Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;
typedef const UNICODE_STRING *PCUNICODE_STRING;

typedef enum _OBJECT_INFORMATION_CLASS
{
        ObjectBasicInformation,                 // Result is OBJECT_BASIC_INFORMATION structure
        ObjectNameInformation,                  // Result is OBJECT_NAME_INFORMATION structure
        ObjectTypeInformation,                  // Result is OBJECT_TYPE_INFORMATION structure
        ObjectAllTypesInformation,              // Result is OBJECT_ALL_INFORMATION structure
        ObjectDataInformation               // Result is OBJECT_DATA_INFORMATION structure

} OBJECT_INFORMATION_CLASS, *POBJECT_INFORMATION_CLASS;

typedef struct _OBJECT_TYPE_INFORMATION {
        UNICODE_STRING TypeName;
        ULONG TotalNumberOfHandles;
        ULONG TotalNumberOfObjects;
        WCHAR Unused1[8];
        ULONG HighWaterNumberOfHandles;
        ULONG HighWaterNumberOfObjects;
        WCHAR Unused2[8];
        ACCESS_MASK InvalidAttributes;
        GENERIC_MAPPING GenericMapping;
        ACCESS_MASK ValidAttributes;
        BOOLEAN SecurityRequired;
```

```
        BOOLEAN MaintainHandleCount;
        USHORT MaintainTypeList;
        POOL_TYPE PoolType;
        ULONG DefaultPagedPoolCharge;
        ULONG DefaultNonPagedPoolCharge;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

typedef struct _OBJECT_ALL_INFORMATION {
        ULONG NumberOfObjectsTypes;
        OBJECT_TYPE_INFORMATION ObjectTypeInformation[1];
} OBJECT_ALL_INFORMATION, *POBJECT_ALL_INFORMATION;

typedef struct _OBJECT_ALL_TYPES_INFORMATION {
       ULONG NumberOfTypes;
       OBJECT_TYPE_INFORMATION TypeInformation[1];
} OBJECT_ALL_TYPES_INFORMATION, *POBJECT_ALL_TYPES_INFORMATION;

typedef UINT32 (__stdcall    *ZwQueryObject_t) (
        IN HANDLE ObjectHandle,
        IN OBJECT_INFORMATION_CLASS ObjectInformationClass,
        OUT PVOID ObjectInformation,
        IN ULONG Length,
        OUT PULONG ResultLength );

void CDetectODDlg::OnNTQueryObject()
{
        // TODO: Add your control notification handler code here
        //  调试器必须正在调试才能检测到，仅打开 OD 是检测不到的
        HMODULE hNtDLL;
        DWORD dwSize;
        UINT i;
        UCHAR    KeyType=0;
        OBJECT_ALL_TYPES_INFORMATION *Types;
        OBJECT_TYPE_INFORMATION        *t;
        ZwQueryObject_t ZwQueryObject;

        hNtDLL = GetModuleHandle("ntdll.dll");
        if(hNtDLL){
                ZwQueryObject = (ZwQueryObject_t)GetProcAddress(hNtDLL, "ZwQueryObject");
                UINT32 iResult = ZwQueryObject(NULL, ObjectAllTypesInformation, NULL, NULL, &dwSize);
                if(iResult==STATUS_INFO_LENGTH_MISMATCH)
                {
                        Types                                                                                =
(OBJECT_ALL_TYPES_INFORMATION*)VirtualAlloc(NULL,dwSize,MEM_COMMIT,PAGE_READWRITE)
```

```
;
                if (Types == NULL)    return;
               if (iResult=ZwQueryObject(NULL,ObjectAllTypesInformation, Types, dwSize, &dwSize)) return;

                for (t=Types->TypeInformation,i=0;i<Types->NumberOfTypes;i++)
                {
                    if ( !_wcsicmp(t->TypeName.Buffer,L"DebugObject")) //比较两个是否相等，这个 L 很特
殊，本地的意思
                    {
                        if(t->TotalNumberOfHandles > 0 || t->TotalNumberOfObjects > 0)
                        {
                            AfxMessageBox("发现 OD");
                            VirtualFree (Types,0,MEM_RELEASE);
                            return;
                        }
                        break; // Found Anyways
                    }
                    t=(OBJECT_TYPE_INFORMATION                                    *)((char
*)t->TypeName.Buffer+((t->TypeName.MaximumLength+3)&~3));
                }
            }
            AfxMessageBox("没有 OD!");
            VirtualFree (Types,0,MEM_RELEASE);
        }
}
```

13.OllyDbg：Guard Pages

这个检查是针对 OllyDbg 的，因为它和 OllyDbg 的内存访问/写入断点特性相关。
除了硬件断点和软件断点外，OllyDbg 允许设置一个内存访问/写入断点，这种类型的断点是通过页面保护来实现的。简单地说，页面保护提供了当应用程序的某块内存被访问时获得通知这样一个途径。
页面保护是通过 PAGE_GUARD 页面保护修改符来设置的，如果访问的内存地址是受保护页面的一部分，将会产生一个 STATUS_GUARD_PAGE_VIOLATION(0x80000001)异常。如果进程被 OllyDbg 调试并且受保护的页面被访问，将不会抛出异常，访问将会被当作内存断点来处理，而壳正好利用了这一点。
示例
下面的示例代码中，将会分配一段内存，并将待执行的代码保存在分配的内存中，然后启用页面的 PAGE_GUARD 属性。接着初始化标设符 EAX 为 0，然后通过执行内存中的代码来引发 STATUS_GUARD_PAGE_VIOLATION 异常。如果代码在 OllyDbg 中被调试，因为异常处理例程不会被调用所以标设符将不会改变。
对策
由于页面保护引发一个异常，逆向分析人员可以故意引发一个异常，这样异常处理例程将会被调用。在示例中，逆向分析人员可以用 INT3 指令替换掉 RETN 指令，一旦 INT3 指令被

执行，Shift+F9 强制调试器执行异常处理代码。这样当异常处理例程调用后，EAX 将被设为正确的值，然后 RETN 指令将会被执行。
如果异常处理例程里检查异常是否真地是 STATUS_GUARD_PAGE_VIOLATION，逆向分析人员可以在异常处理例程中下断点然后修改传入的 ExceptionRecord 参数，具体来说就是 ExceptionCode， 手工将 ExceptionCode 设为 STATUS_GUARD_PAGE_VIOLATION 即可。
实例：

```
//需要用到在 UnhandledExceptionHandler 里定义的一些结构
//*************************************************************
static bool isDebugged=1;
LONG WINAPI TopUnhandledExceptionFilter2(
	struct _EXCEPTION_POINTERS *ExceptionInfo
)
{
	_asm pushad
	AfxMessageBox("回调函数");
	lpSetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER )lpOldHandler);
	ExceptionInfo->ContextRecord->Eip=NewEip;
	isDebugged=0;
	_asm popad
	return EXCEPTION_CONTINUE_EXECUTION;
}

void CDetectODDlg::OnGuardPages()
{
	// TODO: Add your control notification handler code here

	ULONG dwOldType;
	DWORD dwPageSize;
	LPVOID lpvBase;			// 获取内存的基地址
	SYSTEM_INFO sSysInfo;		// 系统信息
	GetSystemInfo(&sSysInfo);	// 获取系统信息
	dwPageSize=sSysInfo.dwPageSize;		//系统内存页大小

	lpSetUnhandledExceptionFilter =
(pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(("kernel32.dll")),
  "SetUnhandledExceptionFilter");
	lpOldHandler=(DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilter2);

  // 分配内存
	lpvBase = VirtualAlloc(NULL,dwPageSize,MEM_COMMIT,PAGE_READWRITE);
	if (lpvBase==NULL)  AfxMessageBox("内存分配失败");
	_asm{
		mov	NewEip,offset safe //方式二，更简单
		mov	eax,lpvBase
```

```
            push    eax
            mov     byte ptr [eax],0C3H //写一个 RETN 到保留内存，以便下面的调用
        }
        if(0==::VirtualProtect(lpvBase,dwPageSize,PAGE_EXECUTE_READ | PAGE_GUARD,&dwOldType)){
            AfxMessageBox("执行失败");
        }
        _asm{
            pop     ecx
            call    ecx     //调用时压栈
safe:
            pop     ecx     //堆栈平衡，弹出调用时的压栈
        }
        if(1==isDebugged){
            AfxMessageBox("发现 OD");
        }else{
            AfxMessageBox("没有 OD");
        }
        VirtualFree(lpvBase,dwPageSize,MEM_DECOMMIT);
}
```

14. Software Breakpoint Detection

软件断点是通过修改目标地址代码为 0xCC（INT3/Breakpoint Interrupt）来设置的断点。通过在受保护的代码段和（或）API 函数中扫描字节 0xCC 来识别软件断点。这里以普通断点和函数断点分别举例。

（1） 实例一 普通断点

注意：在被保护的代码区域下 INT3 断点进行测试

```
BOOL DetectBreakpoints()
{
    BOOL bFoundOD;
    bFoundOD=FALSE;
    __asm
    {
                jmp     CodeEnd
    CodeStart:  mov     eax,ecx    ;被保护的程序段
                nop
                push    eax
                push    ecx
                pop     ecx
                pop     eax
    CodeEnd:
                cld                 ;检测代码开始
                mov     edi,offset CodeStart
                mov     edx,offset CodeStart
```

```
                mov         ecx,offset CodeEnd
                sub         ecx,edx
                mov         al,0CCH
                repne       scasb
                jnz         ODNotFound
                mov bFoundOD,1
        ODNotFound:
        }
        return bFoundOD;
}
void CDetectODDlg::OnDectectBreakpoints()
{
        // TODO: Add your control notification handler code here
        HANDLE hProcess;
        hProcess=::GetCurrentProcess();
        CString str="利用我定位";
        if(DetectBreakpoints())
        {
                AfxMessageBox("发现 OD");
        }
        else
        {
                AfxMessageBox("没有 OD");
        }
}
```

（2） 实例二 函数断点 bp

利用 GetProcAddress 函数获取 API 的地址

注意：检测时，BP MessageBoxA

```
BOOL DetectFuncBreakpoints()
{
        BOOL bFoundOD;
        bFoundOD=FALSE;
        DWORD dwAddr;
        dwAddr=(DWORD)::GetProcAddress(LoadLibrary("user32.dll"),"MessageBoxA");
        __asm
        {
                cld                     ;检测代码开始
                mov         edi,dwAddr  ;起始地址
                mov         ecx,100     ;100bytes   ;检测 100 个字节
                mov         al,0CCH
                repne       scasb
                jnz         ODNotFound
                mov bFoundOD,1
        ODNotFound:
```

```
    }
    return bFoundOD;
}
void CDetectODDlg::OnDectectFuncBreakpoints()
{
    // TODO: Add your control notification handler code here
    CString str="利用我定位";
    if(DetectFuncBreakpoints())
    {
        AfxMessageBox("发现 OD");
    }
    else
    {
        AfxMessageBox("没有 OD");
    }
}
```

15. Hardware Breakpoints Detection

硬件断点是通过设置名为 Dr0 到 Dr7 的调试寄存器来实现的。Dr0-Dr3 包含至多 4 个断点的地址，Dr6 是个标志，它指示哪个断点被触发了，Dr7 包含了控制 4 个硬件断点诸如启用/禁用或者中断于读/写的标志。

由于调试寄存器无法在 Ring3 下访问，硬件断点的检测需要执行一小段代码。可以利用含有调试寄存器值的 CONTEXT 结构，该结构可以通过传递给异常处理例程的 ContextRecord 参数来访问。

```
//*********************************************************
static bool isDebuggedHBP=0;
LONG WINAPI TopUnhandledExceptionFilterHBP(
    struct _EXCEPTION_POINTERS *ExceptionInfo
)
{
    _asm pushad
    AfxMessageBox("回调函数被调用");
    ExceptionInfo->ContextRecord->Eip=NewEip;
    if(0!=ExceptionInfo->ContextRecord->Dr0||0!=ExceptionInfo->ContextRecord->Dr1||
        0!=ExceptionInfo->ContextRecord->Dr2||0!=ExceptionInfo->ContextRecord->Dr3)
        isDebuggedHBP=1;   //检测有无硬件断点
    ExceptionInfo->ContextRecord->Dr0=0; //禁用硬件断点，置 0
    ExceptionInfo->ContextRecord->Dr1=0;
    ExceptionInfo->ContextRecord->Dr2=0;
    ExceptionInfo->ContextRecord->Dr3=0;
    ExceptionInfo->ContextRecord->Dr6=0;
    ExceptionInfo->ContextRecord->Dr7=0;
    ExceptionInfo->ContextRecord->Eip=NewEip; //转移到安全位置
```

```
    _asm popad
    return EXCEPTION_CONTINUE_EXECUTION;
}

void CDetectODDlg::OnHardwarebreakpoint()
{
    // TODO: Add your control notification handler code here

    lpSetUnhandledExceptionFilter                                    =
(pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(("kernel32.dll")),
  "SetUnhandledExceptionFilter");
    lpOldHandler=(DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilterHB
P);

    _asm{
        mov     NewEip,offset safe //方式二，更简单
        int     3
        mov     isDebuggedHBP,1 //调试时可能也不会触发异常去检测硬件断点
safe:
    }
    if(1==isDebuggedHBP){
        AfxMessageBox("发现 OD");
    }else{
        AfxMessageBox("没有 OD");
    }
}
//*********************************************************
```

16.PatchingDetection CodeChecksumCalculation 补丁检测，代码检验和

补丁检测技术能识别壳的代码是否被修改，也能识别是否设置了软件断点。补丁检测是通过代码校验来实现的，校验计算包括从简单到复杂的校验和/哈希算法。
实例：改动被保护代码的话，CHECKSUM 需要修改，通过 OD 等找出该值
注意：在被保护代码段下 F2 断点或修改字节来测试

```
/*********************************************************/
BOOL CheckSum()
{
    BOOL bFoundOD;
    bFoundOD=FALSE;
    DWORD CHECK_SUM=5555; //正确校验值
    DWORD dwAddr;
    dwAddr=(DWORD)CheckSum;
    __asm
    {   ;检测代码开始
```

```
                    mov         esi,dwAddr
                    mov         ecx,100
                    xor         eax,eax
 checksum_loop:
                    movzx     ebx,byte ptr [esi]
                    add  eax,ebx
                    rol   eax,1
                    inc   esi
                    loop        checksum_loop

                    cmp         eax,CHECK_SUM
                    jz          ODNotFound
                    mov         bFoundOD,1
      ODNotFound:
      }
      return bFoundOD;
}
void CDetectODDlg::OnChecksum()
{
      // TODO: Add your control notification handler code here
      if(CheckSum())
      {
            AfxMessageBox("发现 OD");
      }
      else
      {
            AfxMessageBox("没有 OD");
      }
}
```

17.block input 封锁键盘、鼠标输入

user32!BlockInput() API 阻断键盘和鼠标的输入。
典型的场景可能是逆向分析人员在 GetProcAddress()内下断，然后运行脱壳代码直到被断下。但是跳过一段垃圾代码之后壳调用 BlockInput()。当 GetProcAddress()断点断下来后，逆向分析人员会突然困惑地发现无法控制调试器了，不知究竟发生了什么。
示例：源码看附件
BlockInput()参数 fBlockIt，true，键盘和鼠标事件被阻断；false，键盘和鼠标事件解除阻断：

```
; Block input
push                TRUE
call            [BlockInput]

;...Unpacking code...
```

```
;Unblock input
push                    FALSE
call                [BlockInput]

void CDetectODDlg::OnBlockInput()
{     // #include "Winable.h"
      // TODO: Add your control notification handler code here
      CString str="利用我定位";
      DWORD dwNoUse;
      DWORD dwNoUse2;
      ::BlockInput(TRUE);
      dwNoUse=2;
      dwNoUse2=3;
      dwNoUse=dwNoUse2;
      ::BlockInput(FALSE);
}
```

对策

（1）最简单的方法就是补丁 BlockInput()使它直接返回。

（2）同时按 CTRL+ALT+DELETE 键手工解除阻断。

18.EnableWindow 禁用窗口

与 BlockInput 异曲同工，也是禁用窗口然后再解禁

在资源管理器里直接双击运行的话，会使当前的资源管理器窗口被禁用。

在 OD 里面的话，就会使 OD 窗口被禁用。　　MFC 里对 OD 貌似无效

```
 void CDetectODDlg::OnEnableWindow()
{
      // TODO: Add your control notification handler code here
      CString str="利用我定位";
      CWnd *wnd;
      wnd=GetForegroundWindow();
      wnd->EnableWindow(FALSE);
      DWORD dwNoUse;
      DWORD dwNoUse2;
      dwNoUse=2;
      dwNoUse2=3;
      dwNoUse=dwNoUse2;
      wnd->EnableWindow(TRUE);
}t
```

19.ThreadHideFromDebugger

ntdll!NtSetInformationThread()用来设置一个线程的相关信息。把 ThreadInformationClass

参数设为 ThreadHideFromDebugger(11H)可以禁止线程产生调试事件。

ntdll!NtSetInformationThread 的参数列表如下。ThreadHandle 通常设为当前线程的句柄(0xFFFFFFFE)：

```
NTSTATUS NTAPI NtSetInformationThread(
    IN   HANDLE                    ThreadHandle,
    IN   THREAD_INFORMATION_CLASS  ThreadInformaitonClass,
    IN   PVOID                     ThreadInformation,
    IN   ULONG                     ThreadInformationLength
);
```

ThreadHideFromDebugger 内部设置内核结构 ETHREAD 的 HideThreadFromDebugger 成员。一旦这个成员设置以后，主要用来向调试器发送事件的内核函数_DbgkpSendApiMessage()将不再被调用。

```
            invoke   GetCurrentThread
            invoke   NtSetInformationThread,eax,11H,NULL,NULL
```

对策：

（1）在 ntdll!NtSetInformationThread()里下断，断下来后，操纵 EIP 防止 API 调用到达内核

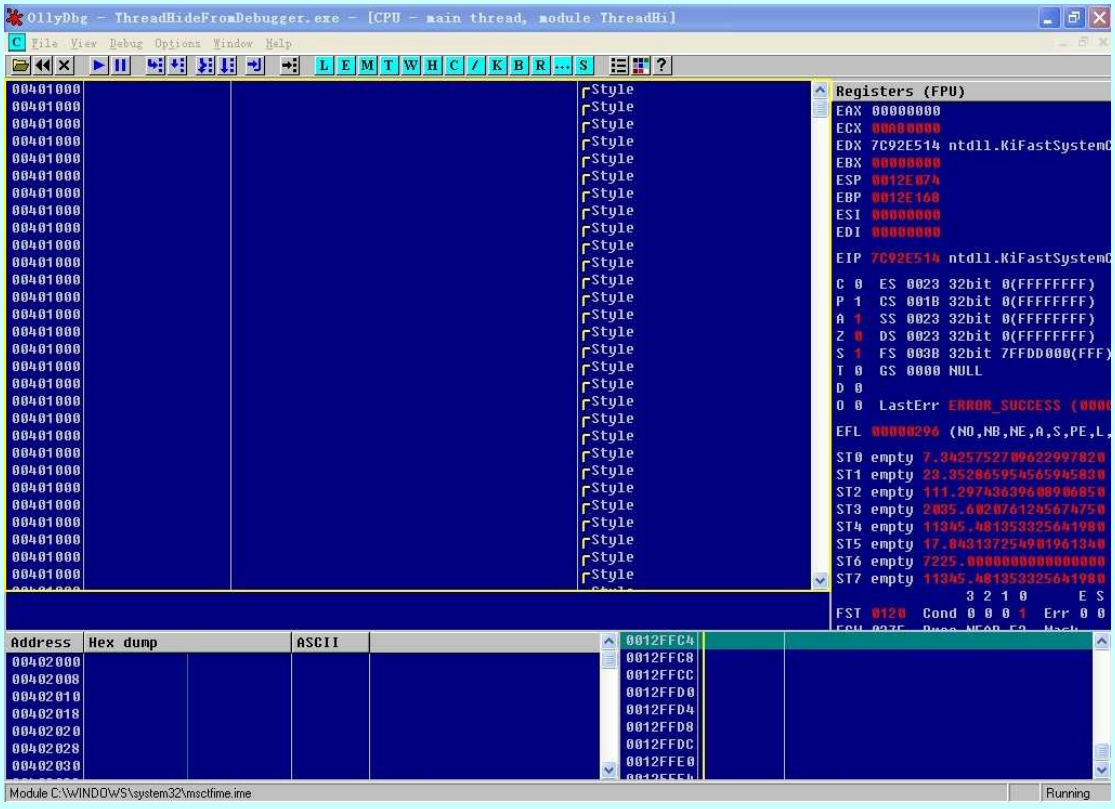
（2）Olly Advanced 插件也有补这个 API 的选项。补过之后一旦 ThreadInformaitonClass 参数为 HideThreadFromDebugger，API 将不再深入内核仅仅执行一个简单的返回。

```
/*********************************************************/
typedef enum _THREADINFOCLASS {
ThreadBasicInformation, // 0 Y N
ThreadTimes, // 1 Y N
ThreadPriority, // 2 N Y
ThreadBasePriority, // 3 N Y
ThreadAffinityMask, // 4 N Y
ThreadImpersonationToken, // 5 N Y
ThreadDescriptorTableEntry, // 6 Y N
ThreadEnableAlignmentFaultFixup, // 7 N Y
ThreadEventPair, // 8 N Y
ThreadQuerySetWin32StartAddress, // 9 Y Y
ThreadZeroTlsCell, // 10 N Y
ThreadPerformanceCount, // 11 Y N
ThreadAmILastThread, // 12 Y N
ThreadIdealProcessor, // 13 N Y
ThreadPriorityBoost, // 14 Y Y
ThreadSetTlsArrayAddress, // 15 N Y
ThreadIsIoPending, // 16 Y N
ThreadHideFromDebugger // 17 N Y
} THREAD_INFO_CLASS;

typedef NTSTATUS (NTAPI *ZwSetInformationThread)(
IN   HANDLE                    ThreadHandle,
IN   THREAD_INFO_CLASS         ThreadInformaitonClass,
IN   PVOID                     ThreadInformation,
```

```
IN    ULONG                                 ThreadInformationLength
);

void CDetectODDlg::OnZwSetInformationThread()
{
      // TODO: Add your control notification handler code here
            CString str="利用我定位";
      HANDLE hwnd;
      HMODULE hModule;
      hwnd=GetCurrentThread();
      hModule=LoadLibrary("ntdll.dll");
      ZwSetInformationThread myFunc;
      myFunc=(ZwSetInformationThread)GetProcAddress(hModule,"ZwSetInformationThread");
      myFunc(hwnd,ThreadHideFromDebugger,NULL,NULL);
}
/************************************************************/
```



20. Disabling Breakpoints 禁用硬件断点

;执行过后，OD 查看硬件断点还存在，但实际已经不起作用了
;利用 CONTEXT 结构，该结构利用异常处理获得，异常处理完后会自动写回

见 Hardware Breakpoints Detection

21.OllyDbg:OutputDebugString() Format String Bug

OutputDebugString 函数用于向调试器发送一个格式化的串，Ollydbg 会在底端显示相应的信息。OllyDbg 存在格式化字符串溢出漏洞，非常严重，轻则崩溃，重则执行任意代码。这个漏洞是由于 Ollydbg 对传递给 kernel32!OutputDebugString()的字符串参数过滤不严导致的，它只对参数进行那个长度检查，只接受 255 个字节，但没对参数进行检查，所以导致缓冲区溢出。

例如：printf 函数：%d，当所有参数压栈完毕后调用 printf 函数的时候，printf 并不能检测参数的正确性，只是机械地从栈中取值作为参数，这样堆栈就被破坏了，栈中信息泄漏。。

示例:下面这个简单的示例将导致 OllyDbg 抛出违规访问异常或不可预期的终止。

```
szFormatStr         db          '%s%s',0
      push        offset szFormatStr
      call        OutputDebugString
```

对策:补丁 kernel32!OutputDebugStringA()入口使之直接返回

```
void CDetectODDlg::OnOutputDebugString()
{
      // TODO: Add your control notification handler code here
      ::OutputDebugString("%s%s%s");
}
```

22.TLS Callbacks

使用 Thread Local Storage (TLS)回调函数可以实现在实际的入口点之前执行反调试的代码，这也是 OD 载入程序就退出的原因所在。（Anti-OD）

线程本地存储器可以将数据与执行的特定线程联系起来，一个进程中的每个线程在访问同一个线程局部存储时，访问到的都是独立的绑定于该线程的数据块。动态绑定（运行时）线程特定数据是通过 TLS API（TlsAlloc、TlsGetValue、TlsSetValue 和 TlsFree）的方式支持的。除了现有的 API 实现，Win32 和 Visual C++ 编译器现在还支持静态绑定（加载时间）基于线程的数据。当使用_declspec(thread)声明的 TLS 变量时，编译器把它们放入一个叫.tls 的区块里。当应用程序加载到内存时，系统寻找可执行文件中的.tls 区块，并动态的分配一个足够大的内存块，以便存放 TLS 变量。系统也将一个指向已分配内存的指针放到 TLS 数组里，这个数组由 FS:[2CH]指向。

数据目录表中第 9 索引的 IMAGE_DIRECTORY_ENTRY_TLS 条目的 VirtualAddress 指向 TLS 数据，如果非零，这里是一个 IMAGE_TLS_DIRECTORY 结构，如下：

```
IMAGE_TLS_DIRECTORY32       STRUC
    StartAddressOfRawData   DWORD   ?    ; 内存起始地址，用于初始化新线程的 TLS
    EndAddressOfRawData     DWORD   ?    ; 内存终止地址
    AddressOfIndex          DWORD   ?    ; 运行库使用该索引来定位线程局部数据
    AddressOfCallBacks      DWORD   ?    ; PIMAGE_TLS_CALLBACK 函数指针数组的地址
    SizeOfZeroFill          DWORD   ?    ; 用 0 填充 TLS 变量区域的大小
    Characteristics          DWORD   ?    ; 保留，目前为 0
IMAGE_TLS_DIRECTORY32       ENDS
```

AddressOfCallBacks 是线程建立和退出时的回调函数，包括主线程和其它线程。当一个线程创建或销毁时，在列表中的每一个函数被调用。一般程序没有回调函数，这个列表是空的。TLS 数据初始化和 TLS

回调函数调用都在入口点之前执行，也就是说 TLS 是程序最开始运行的地方。程序退出时，TLS 回调函数再被执行一次。回调函数：

TLS_CALLBACK proto Dllhandle : LPVOID, Reason : DWORD, Reserved : LPVOID

参数如下:

Dllhandle： 为模块的句柄

Reason 可取以下值:

DLL_PROCESS_ATTACH 1： 启动一个新进程被加载

DLL_THREAD_ATTACH 2： 启动一个新线程被加载

DLL_THREAD_DETACH 3： 终止一个新线程被加载

DLL_PROCESS_DETACH 0： 终止一个新进程被加载

Reserverd:用于保留，设置为 0

IMAGE_TLS_DIRECTORY 结构中的地址是虚拟地址，而不是 RVA。这样，如果可执行文件不是从基地址装入，则这些地址会通过基址重定位修正。而且 IMAGE_TLS_DIRECTORY 本身不在.TLS 区块中，而在.rdata 里。

TLS 回调可以使用诸如 pedump 之类的 PE 文件分析工具来识别。如果可执行文件中存在 TLS 条目，数据条目将会显示出来。

Data directory

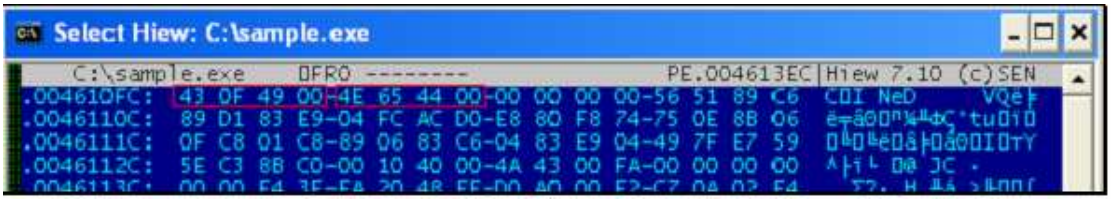
EXPORT	rva:00000000	size:00000000
IMPORT	rva:00061000	size:000000E0
:::		
TLS	rva:000610E0	size:00000018
:::		
IAT	rva:00000000	size:00000000
DELAY_IMPORT	rva:00000000	size:00000000
COM_DESCRPTR	rva:00000000	size:00000000
unused	rva:00000000	size:00000000

接着显示 TLS 条目的实际内容。AddressOfCallBacks 成员指向一个以 null 结尾的回调函数数组。

TLS directory：

StartAddressOfRawData:	00000000
EndAddressOfRawData:	00000000
AddressOfIndex:	004610F8
AddressOfCallBacks:	004610FC
SizeOfZeroFill:	00000000
Characteristics:	00000000

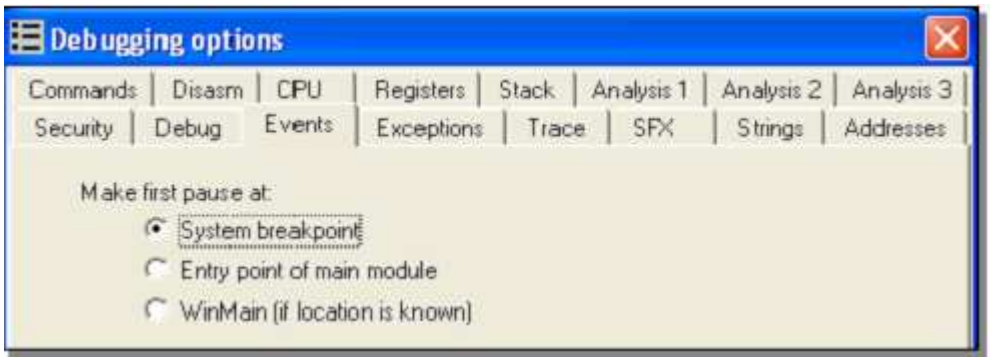
在这个例子中，RVA 0x4610fc 指向回调函数指针（0x490f43 和 0x44654e）：

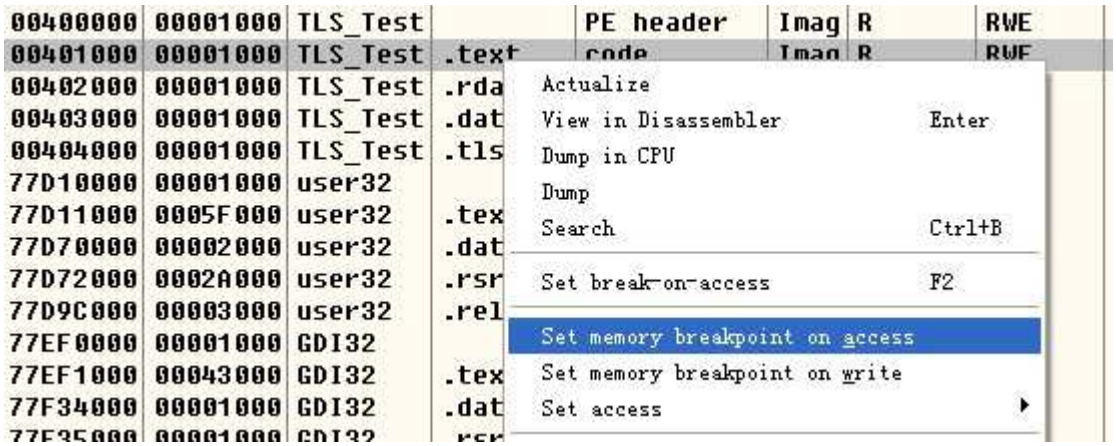


默认情况下 OllyDbg 载入程序将会暂停在入口点，应该配置一下 OllyDbg 使其在 TLS 回调被调用之前中断在实际的 loader。

通过“选项->调试选项->事件->第一次中断于->系统断点”来设置中断于 ntdll.dll 内的实际 loader 代码。这样设置以后，OllyDbg 将会中断在位于执行 TLS 回调的 ntdll!LdrpRunInitializeRoutines()之前的

ntdll!_LdrpInitializeProcess()，这时就可以在回调例程中下断并跟踪了。例如，在内存映像的.text 代码段上设置内存访问断点，可以断在 TLS 回调函数。






```
.386
.model      flat,stdcall
option      casemap:none
include windows.inc
include user32.inc
include kernel32.inc
includelib user32.lib
includelib kernel32.lib

.data?
dwTLS_Index dd    ?

OPTION       DOTNAME
;; 定义一个 TLS 节
.tls    SEGMENT
TLS_Start LABEL    DWORD
  dd       0100h       dup ("slt.")
TLS_End     LABEL    DWORD
.tls      ENDS
OPTION       NODOTNAME

.data
```

```
    TLS_CallBackStart    dd    TlsCallBack0
    TLS_CallBackEnd      dd    0
    szTitle              db    "Hello TLS",0
    szInTls              db    "我在 TLS 里",0
    szInNormal           db    "我在正常代码内",0
    szClassName          db    "ollydbg"          ; OD  类名
    ;这里需要注意的是，必须要将此结构声明为 PUBLIC,用于让连接器连接到指定的位置，
    ;其次结构名必须为_tls_uesd 这是微软的一个规定。编译器引入的位置名称也如此。
    PUBLIC _tls_used
    _tls_used    IMAGE_TLS_DIRECTORY    <TLS_Start,    TLS_End,    dwTLS_Index,
TLS_CallBackStart, 0, ?>

    .code
    ;*****************************************************************
    ;; TLS 的回调函数
    TlsCallBack0 proc Dllhandle:LPVOID,dwReason:DWORD,lpvReserved:LPVOID
        mov      eax,dwReason ;判断 dwReason 发生的条件
        cmp      eax,DLL_PROCESS_ATTACH   ; 在进行加载时被调用
        jnz      ExitTlsCallBack0
        invoke   FindWindow,addr szClassName,NULL   ;通过类名进行检测
        .if      eax      ;找到
                 invoke      SendMessage,eax,WM_CLOSE,NULL,NULL
        .endif
        invoke   MessageBox,NULL,addr szInTls,addr szTitle,MB_OK
        mov      dword ptr[TLS_Start],0
        xor      eax,eax
        inc      eax
    ExitTlsCallBack0:
        ret
    TlsCallBack0    ENDP
    ;*****************************************************************
    Start:
        invoke    MessageBox,NULL,addr szInNormal,addr szTitle,MB_OK
        invoke    ExitProcess, 1
        end   Start
```

VC++ 6.0

VC 里的 TLS 回调，总是有一些问题，基本如下：

1、VC6 不支持。

2、VS2005 的 Debug 版正常，Release 版不正常。

3、VS2005 的 Release 版正常，Debug 版不正常。

VC6 不支持的原因是 VC6 带的 TLSSUP.OBJ 有问题，它已定义了回调表的第一项，并且为 0，0 意味着回调表的结束，因此我们加的函数都不会被调用。[INDENT]对于第 2 个问题，我没遇到，倒是遇到了第 3 个问题。对这个问题进行了一下研究，发现问题所在：在 Link 过程中节.CRT$XLA 和.CRT$XLB 合并时，应该是按字母顺序无间隙合并，但在 DEBUG 版

的输出中实事并非如此，顺序没错，但却产生了很大的间隙，间隙填 0，相当于在我们的回调表前加 0 若干个 0，又是回调表提前结束，这也许是 BUG。针对第二种情况，我没有遇到，不知道是否是这个原因，如果是，则我想应是 LINK 的 BUG。

针对上述问题，本来我想可以使用 VS2008 的 tlssup.obj，但是它与 VC6 的不兼容，改起来比较麻烦，后来我突然想到，也许我们可以自己创建一个 tlssup.obj，基于这个思路，写了自己的 tlssup,目前测试结果显示，它可以兼容 VC6，VS2005，VS2008。

（1）建立一个控制台工程
（2）创建 tlssup.c 文件，代码如下
（3）将该文件加入工程
（4）英文版：右键点击该 tlssup.c 文件，选择 Setting->C/C++->Gategory->Precomliled Headers->Not using precompiled headers。中文版：右键点击该 tlssup.c 文件->设置->C/C++->预编译的头文件->不使用预补偿页眉->确定

```
// tlssup.c 文件代码：
#include <windows.h>
#include <winnt.h>
int _tls_index=0;
#pragma data_seg(".tls")
int _tls_start=0;
#pragma data_seg(".tls$ZZZ")
int _tls_end=0;
#pragma data_seg(".CRT$XLA")
int __xl_a=0;
#pragma data_seg(".CRT$XLZ")
int __xl_z=0;
#pragma data_seg(".rdata$T")
extern PIMAGE_TLS_CALLBACK my_tls_callbacktbl[];
IMAGE_TLS_DIRECTORY32
_tls_used={(DWORD)&_tls_start,(DWORD)&_tls_end,(DWORD)&_tls_index,(DWORD)my_tls_callbacktbl,0,0};
```

然后，我们在其它 CPP 文件中定义 my_tls_callbacktbl 如下即可：

```
extern "C" PIMAGE_TLS_CALLBACK my_tls_callbacktbl[] = {my_tls_callback1,0}; //可以有多个回调，但一定要在最后加一个空项，否则很可能出错。
```

当然下面一行也不能少：

```
#pragma comment(linker, "/INCLUDE:__tls_used")

//  工程 cpp 文件代码：
// TLS_CallBack_test.cpp : Defines the entry point for the console application.
//

#include <windows.h>
#include <winnt.h>
//下面这行告诉链接器在 PE 文件中要创建 TLS 目录
#pragma comment(linker, "/INCLUDE:__tls_used")
```

```
void NTAPI my_tls_callback1(PVOID h, DWORD reason, PVOID pv)
{
//仅在进程初始化创建主线程时执行的代码
if( reason == DLL_PROCESS_ATTACH ){
    MessageBox(NULL,"hi,this is tls callback","title",MB_OK);
}
return;
}
#pragma data_seg(".CRT$XLB")
extern "C" PIMAGE_TLS_CALLBACK my_tls_callbacktbl[] = {my_tls_callback1,0};
#pragma data_seg()
int main(void)
{
MessageBox(NULL,"hi,this is main()","title",MB_OK);
return 0;
}
```

MFC 里

（1）tlssup.c 文件 同样设置

（2）代码

```
#pragma comment(linker, "/INCLUDE:__tls_used")
/*这是 PIMAGE_TLS_CALLBACK()函数的原型，其中第一个和第三个参数保留，第二个参数决定函数在那种情况下*/
void NTAPI my_tls_callback1(PVOID h, DWORD reason, PVOID pv)
{
if( reason == DLL_PROCESS_ATTACH ){
    MessageBox(NULL,"hi,this is tls callback","title",MB_OK);
}
return;
}
#pragma data_seg(".CRT$XLB")
extern "C" PIMAGE_TLS_CALLBACK my_tls_callbacktbl[] = {my_tls_callback1,0};
#pragma data_seg()
```

反反调试技术

本人脱壳逆向的水平不高，这里仅说一下本人的一点体会：

对于初学者来说主要是利用 StrongOD 等各种插件，这些插件能够躲过上面所说的很多检测。有了一定基础以后就可以根据各种反调试方法的弱点寻求反反调试的途径了。

曾经写过一篇关于 ANTI-OD 的原理和应对方法的文章，也可以用于增强自己的 OD，各位可以看一下：

OD 被 Anti 的原因分析及应对之道：

http://www.ucooper.com/od-anti-reasons.html

各种反调试技术原理与实例 汇编版

http://www.ucooper.com/anti-debug-methods-asm.html

欢迎莅临本人空间：http://ucooper.com

写意互联网，关注搜索引擎技术，涉猎搜索引擎优化、软件破解、PHP 网站建设、Wordpress 应用等。

失误之处，敬请指教。

参考文献：《脱壳的艺术》、《加密与解密》、看雪论坛、其它资料