



恶意样本分析手册(第一版)

■ 文档编号	■ 密级	内部使用
■ 版本编号 1.0	■ 日期	2017 年 2 月



© 2017 绿盟科技

■ 版权声明

本文中出现的任何文字叙述、文档格式、插图、照片、方法、过程等内容，除另有特别注明，版权均属绿盟科技所有，受到有关产权及版权法保护。任何个人、机构未经绿盟科技的书面授权许可，不得以任何方式复制或引用本文的任何片断。

目录

准备工作	8
1) Process monitor	8
2) OllyDbg	10
3) WinDbg	12
4) IDA	18
5) WireShark	22
6) PCHunter	23
分析方法	26
静态分析	26
动态分析	29
动态调试	30
不同格式文件分析方法	32
1) EXE (C\C++)	32
2) SYS	34
3) JS 代码	37
4) Doc	37
5) Elf 文件	40
6) C#程序	43
7) Python	44
8) dll	45
具体分析调试技巧	47
恶意代码分析通用规则:	49
反调试技术总结	50
1. FindWindow	50
2. 枚举窗口	50
3. 枚举进程	52
4. 查看父进程是不是 Explorer	53
5. 检测系统时钟	54
6. 查看 StartupInfo 结构	55
7. BeingDebugged, NTGlobalFlag	56
8. CheckRemoteDebuggerPresent	56
9. NtQueryInformationProcess	57
10. SetUnhandledExceptionFilter	59
11. SeDebugPrivilege 进程权限	60
12. GuardPages	61
13. 软件断点	63
14. 硬件断点	64
15. 封锁键盘, 鼠标输入	66
16. 禁用窗口	66
17. ThreadHideFromDebugger	67
18. OutputDebugString	67

虚拟机检测	69
1. VMWare	69
1.1 查找注册表	69
1.2 搜索特定程序确定是否运行在虚拟机中.....	71
1.3 通过执行特权指令.....	72
1.4 查找特定的驱动模块.....	74
1.5 CPUID	74
1.6 检测目标对象.....	75
2. Virtual PC	77
2.1 使用非法指令:	77
2.2 检测设备对象.....	78
3. VirtualBox	78
3.1 搜索注册表项.....	78
4. SandBox	80
4.1 使用 WinObj 搜索目标对象.....	80
4.2 查看当前进程是否包含目标动态库.....	80
脱壳	82
0x1 单步跟踪法.....	82
0x2 ESP 定律法.....	82
0x3 二次断点法	83
0x4 末次异常法	83
0x5 模拟跟踪法	83
0x6 SFX 自动脱壳法.....	84
0x7 出口标志法	84
0x8 使用脱壳辅助脱壳	84
0x9 使用脱壳工具脱壳	84
总结:	85
样本分析 API 集合	86
0x01 文件类.....	86
kernel32.CreateFile	86
CreateFileMapping	87
kernel32.OpenFile.....	87
FindFirstFile	88
FindNextFile	88
GetModuleFileName	89
GetModuleHandle	89
GetProcAddress	89
GetStartupInfo	90
GetTempPath	90
GetWindowsDirectory	91
MapViewOfFile	91
NtQueryDirectoryFile	91
SetFileTime	92

Wow64DisableWow64FsRedirection	93
0x02 网络类	93
ws2_32.socket	93
accept	93
bind	94
connect	94
wininet.InternetOpenA	94
CoCreateInstance	95
FtpPutFile	96
GetAdaptersInfo	97
gethostbyname	97
gethostname	97
inet_addr	98
InternetOpen	98
InternetOpenUrl	99
InternetReadFile	99
InternetWriteFile	100
NetShareEnum	100
OleInitialize	101
recv	101
send	102
URLDownloadToFile	102
WSAStartup	102
0x03 注册表与服务类	103
CreateService	103
ControlService	104
OpenSCManager	104
RegisterHotKey	105
RegOpenKey	106
StartServiceCtrlDispatcher	106
0x04 进程类	106
AttachThreadInput	106
CheckRemoteDebuggerPresent	107
ConnectNamedPipe	107
CreateProcess	108
CreateRemoteThread	109
CreateToolhelp32Snapshot	110
EnumProcesses	111
EnumProcessModules	111
IsWow64Process	112
ZwQueryInformationProcess	112
OpenProcess	113
PeekNamedPipe	113

Process32First	114
Process32Next	114
ReadProcessMemory	115
ResumeThread	115
SetThreadContext	115
ShellExecute	116
SuspendThread	116
Thread32First	116
Thread32Next	117
WinExec	117
WriteProcessMemory	118
0x05 注入类	118
GetThreadContext	118
QueueUserAPC	119
VirtualAllocEx	119
VirtualProtectEx	120
0x06 驱动类	120
DeviceIoControl	120
LsaEnumerateLogonSessions	121
MmGetSystemRoutineAddress	121
0x07 其他	122
AdjustTokenPrivileges	122
BitBlt	122
CallNextHookEx	123
CertOpenSystemStore	124
CreateMutex	124
CryptAcquireContext	125
EnbaleExecuteProtectionSupport	125
FindResource	125
FindWindow	126
GetAsyncKeyState	126
GetDC	127
GetForegroundWindow	127
GetKeyState	127
GetSystemDefaultLangId	128
GetTickCount	128
GetVersionEx	128
IsDebuggerPresent	128
LoadLibrary	129
LoadResource	129
MapVirtualKey	129
Module32First	130
Module32Next	130

NetScheduleJobAdd	131
OpenMutex	131
OutputDebugString	132
SetWindowsHookEx	132

样本分析概述

恶意样本分析的目标，通常是为一一起网络入侵事件的相应提供所需要的信息。所以，分析目标就是确定此次事件到底发生了什么，并确保能够定位出所有受感染的主机和文件。在分析可疑的恶意样本时，侧重的是功能分析，在有限的时间内对程序的功能进行分析，并尽可能定位功能代码，确定它能够做什么事，如何在网络上检测到它，以及如何衡量并消除它所带来的危害。更深入的，可以对它所使用的新技术进行逆向学习，不断的学习新的知识来充实自己。分析过程中可能会遇到各种问题，发散思维尽量想办法进行调试。

准备工作

掌握常用分析工具的使用

1) Process monitor

功能：

监控进程的行为

使用场景：

应用程序运行时使用此软件来监控程序的各种操作。

备注：此软件主要监控程序的五种行为：文件系统，注册表，进程，网络，分析。

文件系统：

显示 Windows 文件系统的所有活动信息，包括本地的存储文件以及远程文件系统。此软件能够自动侦测到新加入的文件系统设备并且对其进行监控。

注册表：

能够记录所有的注册表操作并且将注册表根键以常用的缩写形式来表示，以显示注册表路径。在注册表方面，重点要关注的是对注册表的添加和设置功能，恶意样本一般用来实现自启动行为，Operation 为 RegSetValue，常用的路径为 Software\Microsoft\Windows\CurrentVersion\Run。

进程：

能够跟踪所有的进程与线程的创建和退出操作，也可以追踪到动态链接库文件和设备驱动程序的加载操作。大多数恶意软件都会创建子进程或者启动 cmd 进程删除自身文件来达到隐藏自己的目的，通过 ProcMon 的进程树可以清楚地知道恶意样本的行为。

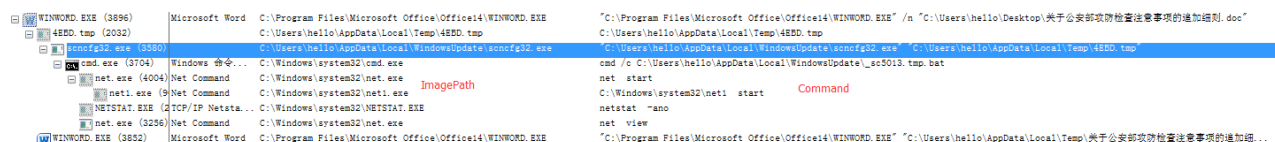


图 1：进程树

如上图：一个 word 的恶意样本，运行时创建了很多的子进程，通过此进程树，我们可以知道样本创建的子进程的路径（此路径也是它释放恶意文件的路径，也可以对它进行单独调试），运行时的命令行参数，通过此图，我们可以尽快的找到样本的关键代码位置。

网络：

使用 Windows 的事件追踪机制（ETW）来追踪并记录 TCP 和 UDP 的活动，每一项网络操作，包括源头与终点的地址，连同大量的发出和接收的数据，但是不包括实际的数据内容。

14:3...	WINWORD.EXE	3852	TCP Connect	WIN-TM03LL4MBTD.localdomain:49161 -> 23.99.112.148:http	SUCCESS	Length: 0, ms...
14:3...	WINWORD.EXE	3852	TCP Send	WIN-TM03LL4MBTD.localdomain:49161 -> 23.99.112.148:http	SUCCESS	Length: 330, ...
14:3...	WINWORD.EXE	3852	TCP Receive	WIN-TM03LL4MBTD.localdomain:49161 -> 23.99.112.148:http	SUCCESS	Length: 1024, ...
14:3...	WINWORD.EXE	3852	TCP Receive	WIN-TM03LL4MBTD.localdomain:49161 -> 23.99.112.148:http	SUCCESS	Length: 436, ...
14:3...	WINWORD.EXE	3852	TCP Receive	WIN-TM03LL4MBTD.localdomain:49161 -> 23.99.112.148:http	SUCCESS	Length: 1277, ...
14:3...	WINWORD.EXE	3852	TCP Connect	WIN-TM03LL4MBTD.localdomain:49162 -> 52.187.48.35:http	SUCCESS	Length: 0, ms...
14:3...	WINWORD.EXE	3852	TCP Send	WIN-TM03LL4MBTD.localdomain:49162 -> 52.187.48.35:http	SUCCESS	Length: 304, ...
14:3...	WINWORD.EXE	3852	TCP Receive	WIN-TM03LL4MBTD.localdomain:49162 -> 52.187.48.35:http	SUCCESS	Length: 1024, ...

图 2：网络信息

比如我们监控到样本会与 23.99.112.148 和 52.187.48.35 进行数据交换，可以结合 WireShark 工具，对 IP 地址进行设置，可以快速的定位到数据包，从而查看数据包的内容

分析：

检测系统中的所有活动线程，并且生成一个详尽的分析报告，包括系统内核和用户 CPU 的事件消耗，以及上下文切换执行的数量。

使用经验：

由于此款软件监控的是系统中所有的进程的行为，数据量往往很大，不利于我们分析数据，所以需要对其设置过滤选项，通过 Filter->Filter 选项可以看到右侧的窗口，在此窗口中增加过滤项。

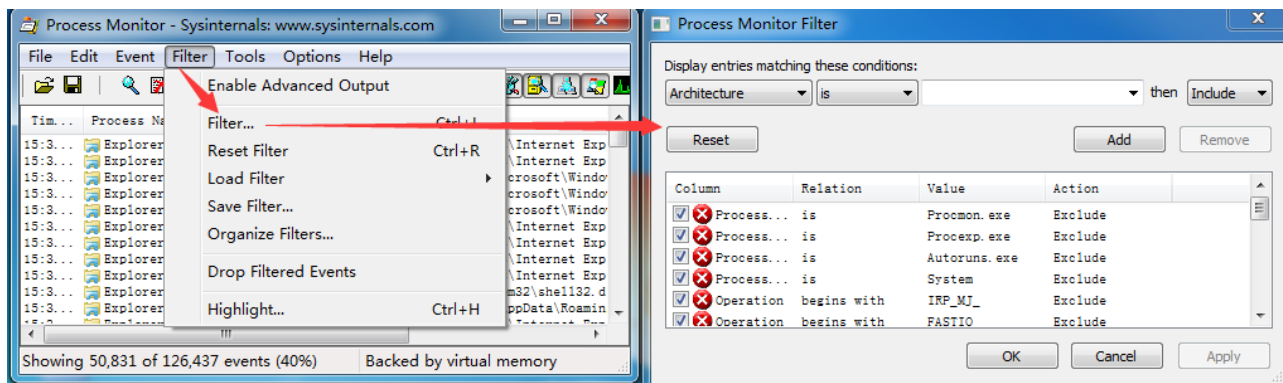


图 3：设置过滤

也可以在数据的显示窗口中进行个别的过滤，例如不想查看 Explorer 进程的任何数据，可以在此进程名上右键，然后选择 'Exclude Explorer.EXE'，就不会看到此进程的任何数据了（注意：一定要在进程名称上右键）。想对哪方面的数据进行过滤，就在对应的数据上右键，然后选择相应的操作。如果选择 Include 'Explorer.EXE'，就会只显示 Explorer.EXE 相关的信息。在搜索数据排除杂项时，使用此方法比较简单。

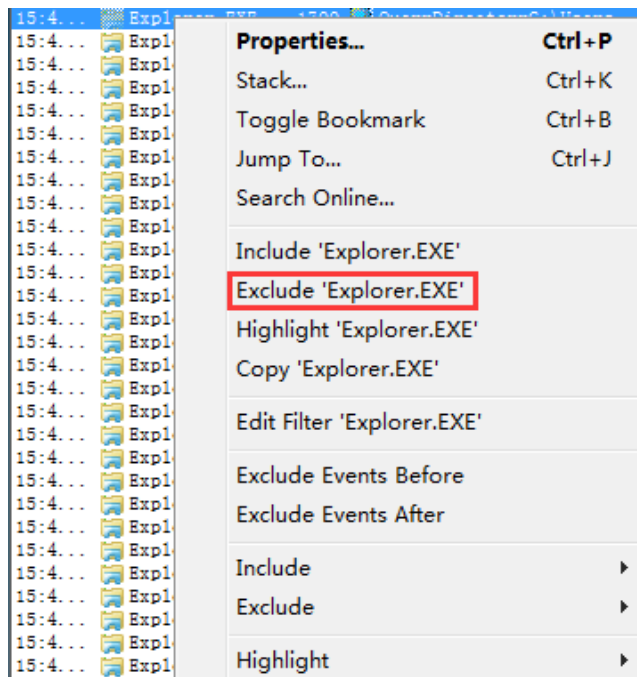


图 4：动态设置过滤选项

2) OllyDbg

功能:

用来对样本进行动态调试

基本调试方法:

F2: 设置断点

F8: 单步步过

F7: 单步步入

F9: 运行

F4: 运行到光标所在处

CTR+F9: 执行到返回, 此命令在执行到一个 ret 指令时暂停, 如果进入到一个函数中, 代码量比较大, 并且没有什么意义, 可以使用此命令, 直接运行到当前函数的结尾处。

ALT+F9: 执行到用户代码。可用于从系统领空快速返回到我们调试的程序领空

查看线程和堆栈:

恶意代码经常使用多线程, 通过 View->Threads 可以调出线程面板窗口, 查看一个程序的当前线程。

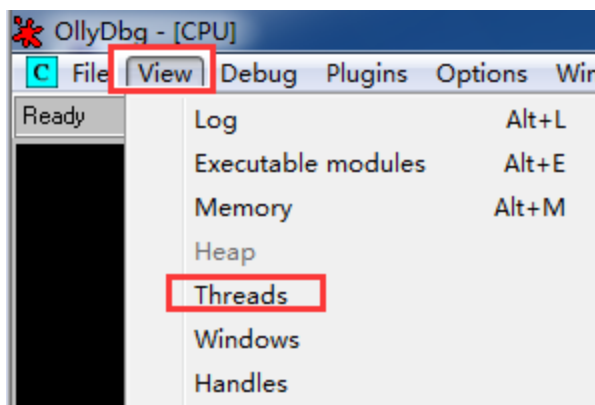


图 5: 查看线程信息

这个窗口可以显示线程的内存位置以及他们的活动状态。由于 OD 是单线程的, 所以需要暂停所有线程, 设置一个断点, 继续运行程序, 这样可以保证在一个特定线程内调试。

给定进程中的每个线程都有自己的栈, 通常情况下, 线程的重要数据都保存在栈中。可以使用 OD 的内存映射来查看内存中的栈内容。

断点:

软件断点:

调试字符串解码函数时, 软件断点比较有用。恶意代码编写者在恶意代码中使用混淆字符串时, 在使用字符串之前, 通常会使用字符串解码函数。混淆数据常被解码成一个有用的字符串。因此, 查看字符串内容的最好办法是, 在字符串解码函数的结束位置设置断点。但是, 这种方法只能在程序使用字符串的时候识别出他们。

条件断点:

对于频繁调用的函数, 仅当特定参数传给他时才中断程序执行, 在这种情况下, 条件软件断点很有用, 可以节省调试时间。

使用方法: 右键目标地址, 选择 BreakPoint->Conditional, 在弹出的对话框中输入条件表达式。

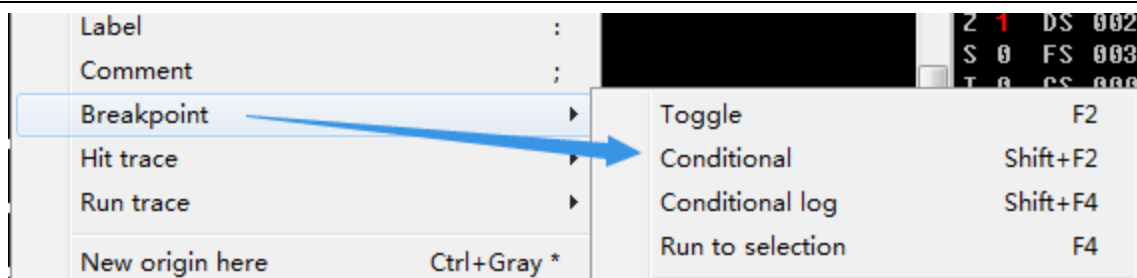


图 6：设置条件断点

硬件断点：

硬件断点可以在不改变代码，堆栈以及任何目标资源的前提下进行调试。另外，在调试时，也不会降低代码的执行速度。

使用方法：右键目标地址，选择 BreakPoint->Hardware, on Execution。

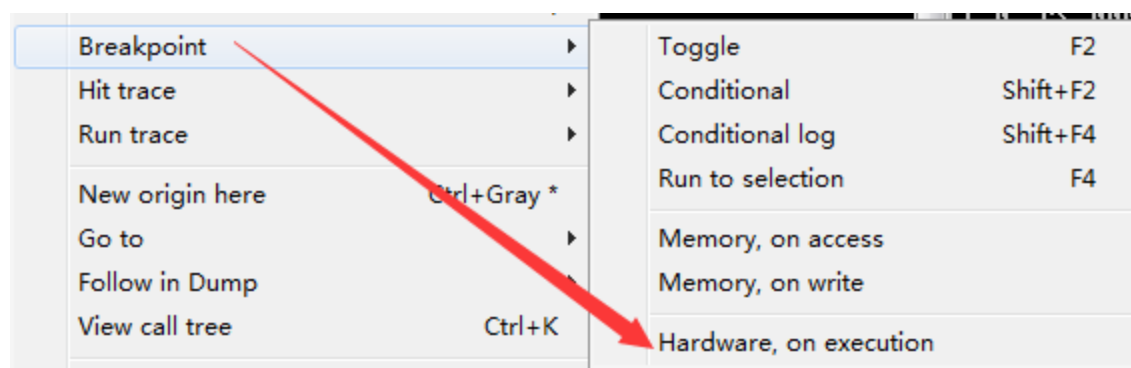


图 7：设置硬件断点

内存断点：

在一块内存上设置断点，可以让被调试程序在访问这段内存时中断执行。OD 支持软件内存断点和硬件内存断点，还支持对内存进行读，写，执行或其他权限访问是否产生中断的设置。

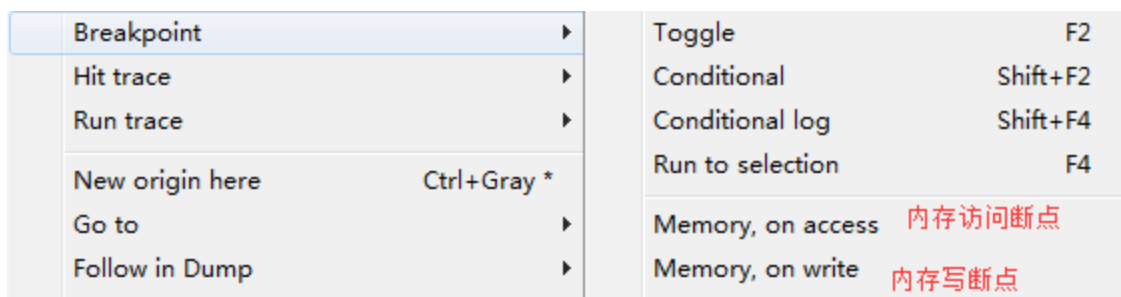


图 8：设置内存断点

回溯跟踪：

可以使用减号键退回到上一步执行的指令，使用加号键，执行下一条指令。如果使用 step into 可以跟踪每一步的执行。如果使用 step over，只能单步跟踪 step over 之前区域，回溯之后再决定是否进入另一个区域。

堆栈调用追踪：

通过此方式可以查看一个给定函数的执行路径，为了查看堆栈调用，在主菜单中选择 view->call stack，会显示当前位置之前的调用序列。如下图所示，函数的调用顺序为

tickbot.00401000->tickbot.004019D0->user32.CreateWindowExW。函数下面的数据就是调用此函数时传入的参数。

Address	Stack	Procedure / arguments	Called from	Frame
0012FE80	00401A07	user32.CreateWindowExW	tickbot.00401A01	0012FEB8
0012FE84	00000000	ExtStyle = 0		
0012FE88	00415CB8	Class = "LineClass"		
0012FE8C	00415CCC	WindowName = "Line"		
0012FE90	00CF0000	Style = WS_OVERLAPPED WS_MINIMIZE		
0012FE94	00000078	X = 78 (120.)		
0012FE98	00000096	Y = 96 (150.)		
0012FE9C	00000320	Width = 320 (800.)		
0012FEA0	000001E0	Height = 1E0 (480.)		
0012FEA4	00000000	hParent = NULL		
0012FEA8	00000000	hMenu = NULL		
0012FEAC	00400000	hInst = 00400000		
0012FEB0	00000000	lParam = NULL		
0012FEC0	0040106F	tickbot.0040106F	tickbot.0040106A	
0012FEC4	00400000	Arg1 = 00400000		
0012FEC8	0000000A	Arg2 = 0000000A		
0012FECC	00400000	Arg3 = 00400000		
0012FF00	00405A92	tickbot.00401000	tickbot.00405A8D	0012FEFC
0012FF04	00400000	Arg1 = 00400000		
0012FF08	00000000	Arg2 = 00000000		
0012FF0C	005D1DC4	Arg3 = 005D1DC4		
0012FF10	0000000A	Arg4 = 0000000A		
0012FF8C	770D3C45	Includes tickbot.00405A92	kernel32.770D3C43	0012FF88

图 9：查看堆栈信息

异常处理：

当异常发生时，OD 会暂停运行，可以使用以下方法来决定是否将异常转到应用程序处理

- Shift+F7 进入异常
- Shift+F8 跳过异常
- Shift+F9：运行异常处理

在恶意代码分析期间最好忽略所有异常，因为调试的目的并不是修复这些异常。

3) WinDbg

功能：

动态调试工具，支持应用层和驱动层的调试

常用窗口说明：

可以通过工具栏来打开不同的窗口



图 10：窗口总图预览

如上图，从左向右依次是 Command, Watch, Locals, Registers, Memory Windows, Call Stack, Disassembly, Scratch pad, Processes and Threads, Command Browser。下面介绍几个常用的。

Command 窗口说明

我们需要在里面输入需要的命令，Windbg 的调试命令和 VC，VS 一样，F9 下断点，F10 步过，F11 步入，F5 运行到断点。

图 11: 命令行窗口

在 **Name** 下输入要查看的变量的名字，在 **Value** 下显示变量的值

图 12: Watch 窗口

显示了所有的寄存器的值

Registers	
Customize...	
Reg	Value
gs	2b
fs	53
es	2b
ds	2b
edi	0
esi	fffffffe
ebx	0
edx	13de08
ecx	8b660000
eax	0
ebp	47f60c
eip	77541003
cs	23
efl	246
esp	47f5e0
ss	2b
dr0	0
dr1	0

图 13: 寄存器窗口

Memory 窗口说明:

用来查看目标地址的内存数据, 并且可以对显示的格式进行选择

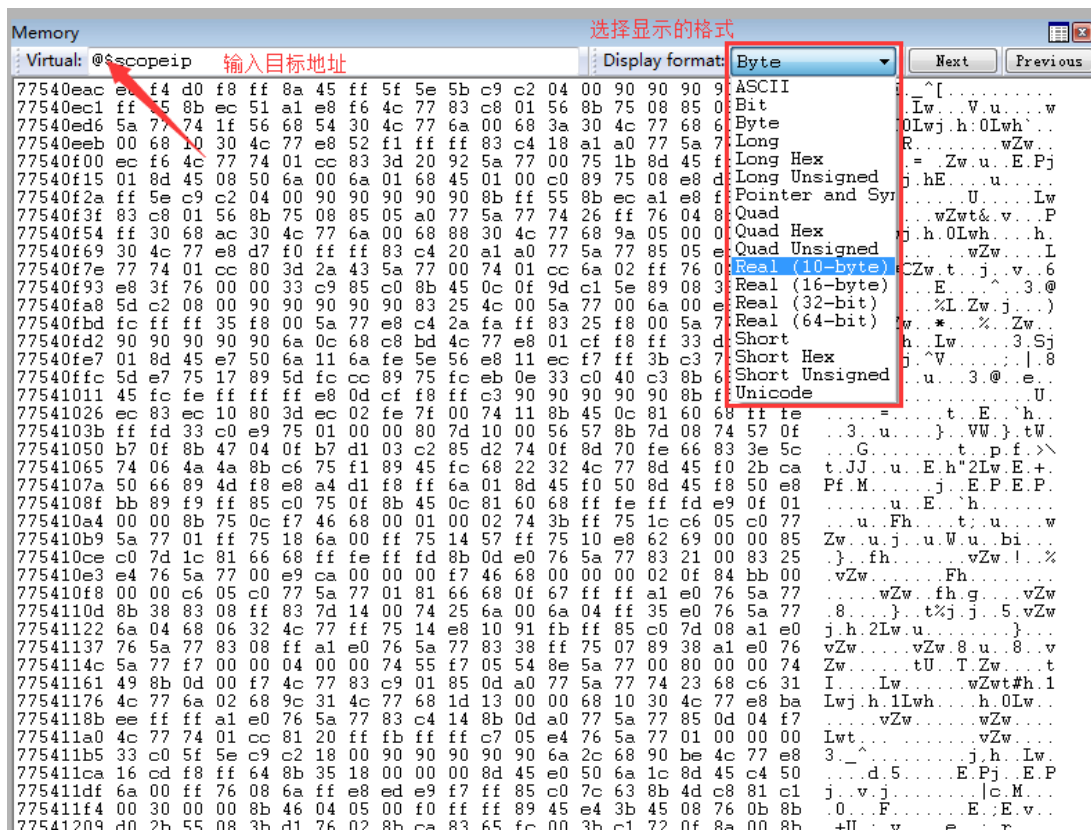


图 14: 内存窗口

Disassembly 窗口说明:

用来显示汇编代码

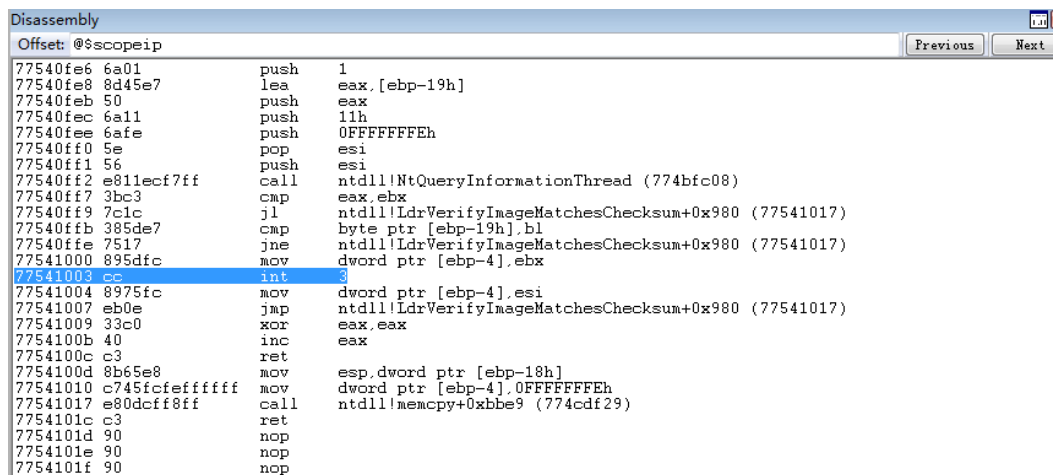


图 15: 汇编代码窗口

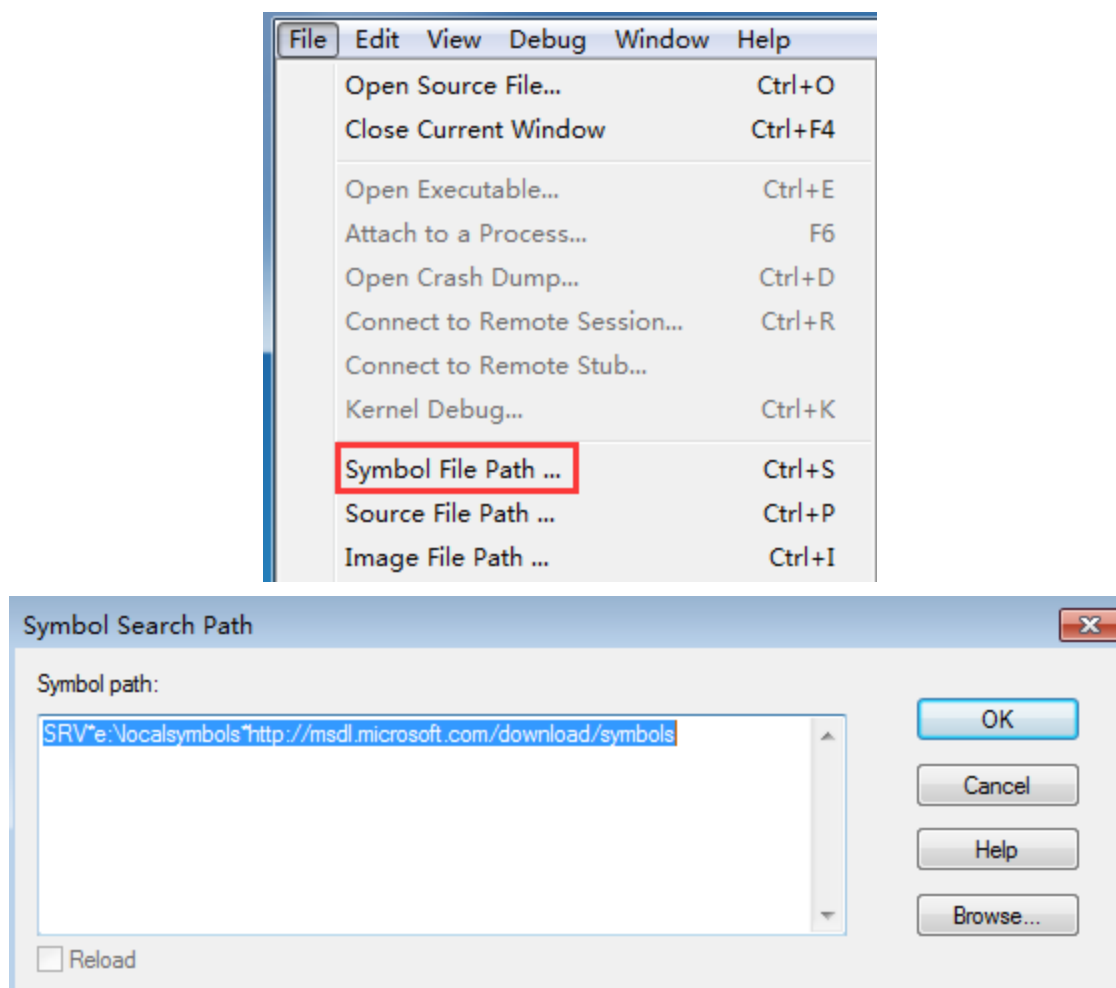
加载微软符号步骤:

图 16: 设置符号表

接着随便加载一个应用程序

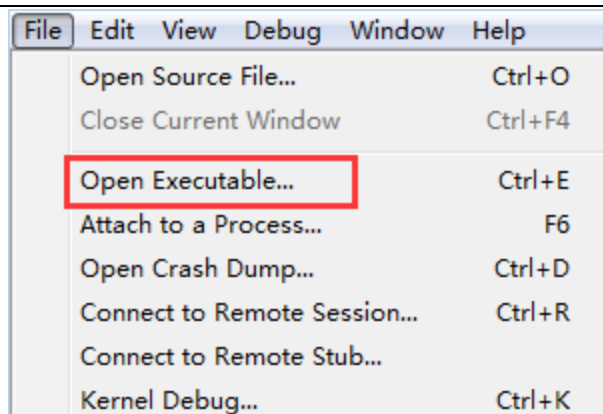


图 17: 加载应用程序

再次打开 Symbol Search Path 窗口，点击 Reload，就会从微软官网下载符号表

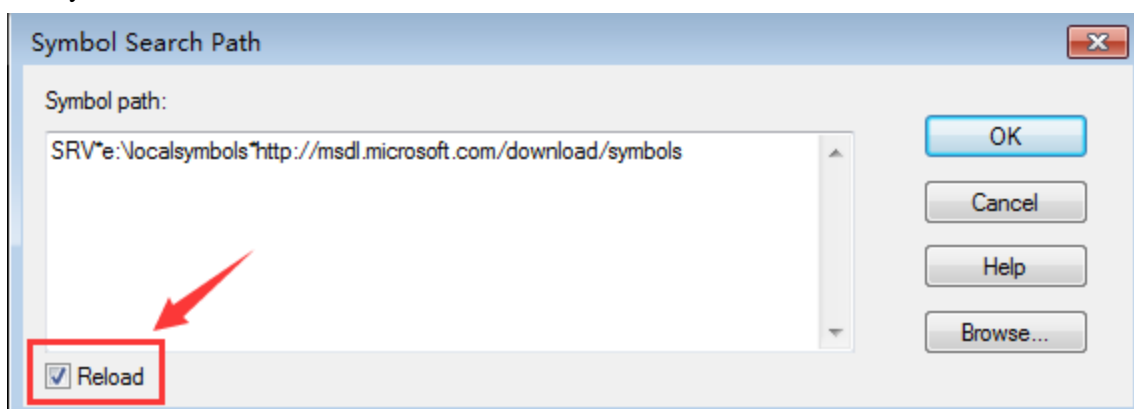


图 18: 重新加载符号表

注意事项:

在 win7 下面 bp_CorExeMain，windbg 一直不认识这个函数，就是读不到符号表。可能的原因是只有 C 盘，符号表放在 C 盘下面，因为权限的问题所以读取不了。所以建议将符号表放在其他盘符下

基本调试命令:

命令 d 用来读取程序数据或堆栈等的内存。

da: 读取内存数据并以 ASCII 文本显示

du: 读取内存数据并以 Unicode 文本显示

dd: 读取内存数据并以 32 位双字显示

db: 读取内存数据并以 byte 的形式显示

```
kd> db edi 1 100
e35bba4b 43 63 43 61 6e 49 57 72-69 74 65 00 43 63 43 6f CcCanIWrite.CcCo
e35bba5b 70 79 52 65 61 64 00 43-63 43 6f 70 79 57 72 69 pyRead.CcCopyWri
e35bba6b 74 65 00 43 63 44 65 66-65 72 57 72 69 74 65 00 te.CcDeferWrite.
e35bba7b 43 63 46 61 73 74 43 6f-70 79 52 65 61 64 00 43 CcFastCopyRead.C
e35bba8b 63 46 61 73 74 43 6f 70-79 57 72 69 74 65 00 43 cFastCopyWrite.C
e35bba9b 63 46 61 73 74 4d 64 6c-52 65 61 64 57 61 69 74 cFastMdlReadWait
e35bbaab 00 43 63 46 61 73 74 52-65 61 64 4e 6f 74 50 6f .CcFastReadNotPo
e35bbabb 73 73 69 62 6c 65 00 43-63 46 61 73 74 52 65 61 ssible.CcFastRea
e35bbacb 64 57 61 69 74 00 43 63-46 6c 75 73 68 43 61 63 dWait.CcFlushCac
e35bbadb 68 65 00 43 63 47 65 74-44 69 72 74 79 50 61 67 he.CcGetDirtyPag
e35bbaeb 65 73 00 43 63 47 65 74-46 69 6c 65 4f 62 6a 65 es.CcGetFileObje
e35bbafb 63 74 46 72 6f 6d 42 63-62 00 43 63 47 65 74 46 ctFromBob.CcGetF
e35bbb0b 69 6c 65 4f 62 6a 65 63-74 46 72 6f 6d 53 65 63 ileObjectFromSec
e35bbb1b 74 69 6f 6e 50 74 72 73-00 43 63 47 65 74 46 6c tionPtrs.CcGetFl
e35bbb2b 75 73 68 65 64 56 61 6c-69 64 44 61 74 61 00 43 ushedValidData.C
e35bbb3b 63 47 65 74 4c 73 6e 46-6f 72 46 69 6c 65 4f 62 cGetLsnForFileOb
```

图 19: 查看内存

上图中 100 前面是小写的 L，不是数字 1。100 表示要显示的数据量的大小，是个 16 进制数，此处表示显示 256byte 的数据。

命令 e 和 d 的使用方法相同，用来改变内存的值

ba: 设置硬件断点

rdmsr: 读 msr 寄存器

.writemem: dump 文件命令

eg: kd> .writemem D:\Temp3 ee8d6000 L10000 表示从地址 ee8d6000 开始，保存的数据的长度为 L10000，将保存的数据放在 D 盘下，名字为 Temp3

s: 搜索命令。

例如: s -a 起始地址 搜索范围 搜索目标

```
eg: s -a ee265000 L10000 "pitopam.com"
kd> s -a ee265000 L10000 "pitopam.com"
ee273be8 70 69 74 6f 70 61 6d 2e-63 6f 6d 00 34 fe 12 00 pitopam.com.4...
kd> !address ee273be8
ee265000 - 00010000
Usage KernelSpaceUsageImage
ImageName 011f.sys
```

图 20: 搜索字符串

lm: 列举出加载到进程空间的所有模块

```
0:000> lm
start      end          module name
00d10000 00d28000 image00d10000 (deferred)
74b50000 74b5c000 CRYPTBASE (deferred)
74b60000 74bc0000 SspiCli (deferred)
74e10000 74e29000 sechost (deferred)
75b10000 75c20000 kernel32 (deferred)
75ed0000 75fc0000 RPCRT4 (deferred)
75fc0000 7606c000 msvcrt (deferred)
76370000 763b7000 KERNELBASE (deferred)
76770000 76810000 ADVAPI32 (deferred)
774a0000 77620000 ntdll (export symbols) C:\Windows\SysWOW64\ntdll.dll
```

图 21: 列举模块

bu: 可以在没有加载的代码中设置一个延迟断点。例如 bu newModule!exporteFunction 命令会指示

WinDbg: 一旦 newModule 模块加载，在 exportedFunction 上设置断点。

x: 使用通配符来搜索函数或符号。

ln: 列出最接近给定内存地址的符号，可以用来确认指针指向的函数。

!devhandles: 可以获得拥有这个设备句柄的所有用户态应用程序列表。

u: 查看函数的汇编代码

```
kd> u obinsertobject
nt!ObInsertObject:
8056ea64 8bff          mov     edi,edi
8056ea66 55            push   ebp
8056ea67 8bec          mov     ebp,esp
8056ea69 81ecc4000000 sub     esp,0C4h
8056ea6f 8b4508          mov     eax,dword ptr [ebp+8]
8056ea72 8b48f0          mov     ecx,dword ptr [eax-10h]
8056ea75 83c0e8          add     eax,0FFFFFFE8h
8056ea78 53            push   ebx
```

图 22: 查看汇编代码

4) IDA

功能:

强大的反汇编工具

基本命令:

- 在反汇编窗口中按键盘上的 G 键，就会出现如下图的一个跳转窗口，在 Jump address 中输入要跳转的虚拟内存地址或命名的位置，可以进行跳转。

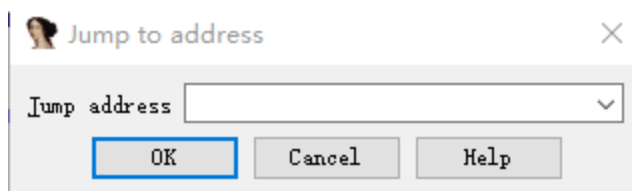


图 23: 跳转指令

- X 命令可以显示目标函数或变量名的交叉引用列表。用鼠标点击目标函数名，然后按下“X”键，就会出现一个窗口，显示所有调用此函数的地址。如下图是 GetCurrentThreadId 函数的交叉引用信息。

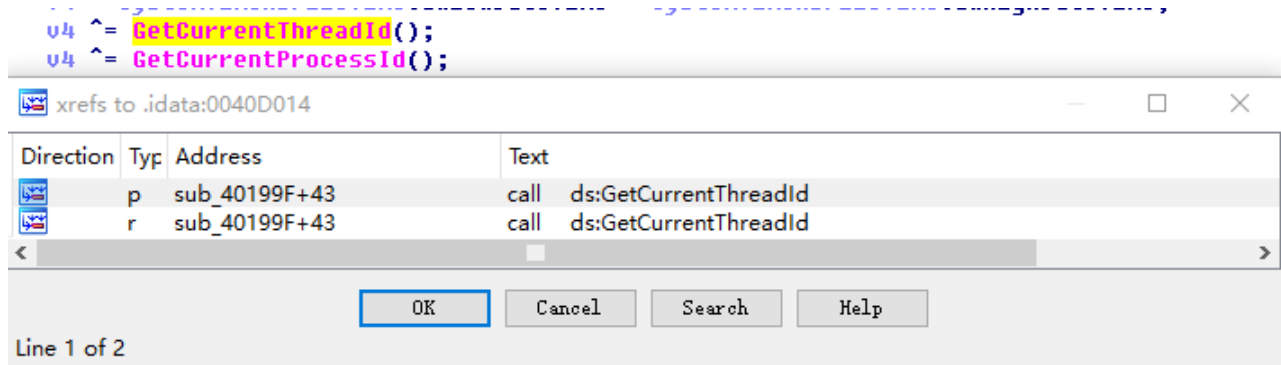


图 24: 交叉引用

- A:以默认的字符串风格对当前选中的位置进行格式化。
- 指定数组: 要创建数组，首先选择数组中的第一个元素，通过 Edit->Array 命令打开“创建数组”对话框，要创建的数组类型由选择的第一个元素的项的类型决定
- D: 可以转换数据类型，也可以使用“Options”菜单的“Setup data types”设置更多的数据类型。U: 可以撤销所有转换。下面两幅图显示了转换前后的区别，将光标放在第一个 db 位置，按下“X”键，将第一个 db 类型转换为了 dw 类型。

```
db 0C7h ;
db 45h ; E
db 0F0h ;
db 60h ; ' OFF32 SEGDEF [_rdata,40D160]
db 0D1h ;
db 40h ; @
db 0
db 6Ah ; j
db 0
```

图 25: 转换前

```

dw 45C7h
db 0F0h ;
db 60h ; \ OFF32 SEGDEF [_rdata,40D160]
db 0D1h ;
db 40h ; @
db 0
db 6Ah ; j
db 0

```

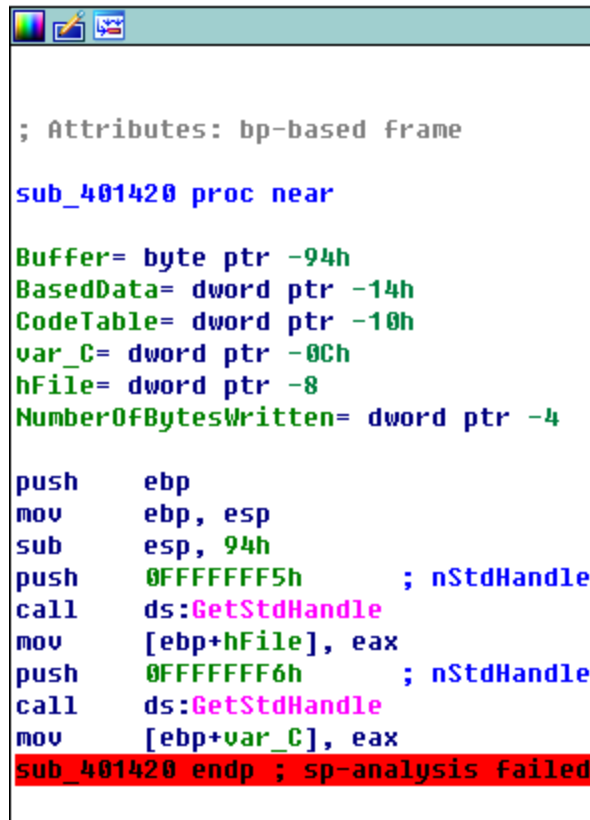
图 26: 转换后

- F5、Tab: 将汇编语言转为高级语言
- 在汇编窗口下, 按下空格键可以在图形模式和汇编模式之间进行转换

```

.text:00401420 sub_401420 proc near ; CODE XREF: start-82↓p
.text:00401420
.text:00401420 Buffer = byte ptr -94h
.text:00401420 BasedData = dword ptr -14h
.text:00401420 CodeTable = dword ptr -10h
.text:00401420 var_C = dword ptr -0Ch
.text:00401420 hFile = dword ptr -8
.text:00401420 NumberOfBytesWritten= dword ptr -4
.text:00401420
.text:00401420 push ebp
.text:00401421 mov ebp, esp
.text:00401423 sub esp, 94h
.text:00401429 push 0FFFFFFF5h ; nStdHandle
.text:0040142B call ds:GetStdHandle
.text:00401431 mov [ebp+hFile], eax
.text:00401434 push 0FFFFFFF6h ; nStdHandle
.text:00401436 call ds:GetStdHandle
.text:0040143C mov [ebp+var_C], eax

```



```

; Attributes: bp-based frame

sub_401420 proc near

Buffer= byte ptr -94h
BasedData= dword ptr -14h
CodeTable= dword ptr -10h
var_C= dword ptr -0Ch
hFile= dword ptr -8
NumberOfBytesWritten= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 94h
push    0FFFFFFF5h      ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+hFile], eax
push    0FFFFFFF6h      ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+var_C], eax
sub_401420 endp ; sp-analysis failed

```

图 27: 汇编窗口

- 在汇编窗口上右键，选择 Xrefs graph from...可以显示函数的调用关系图

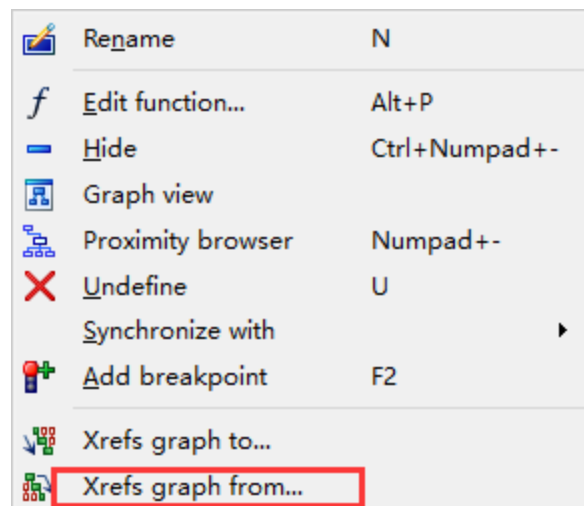


图 28: 查看调用关系图

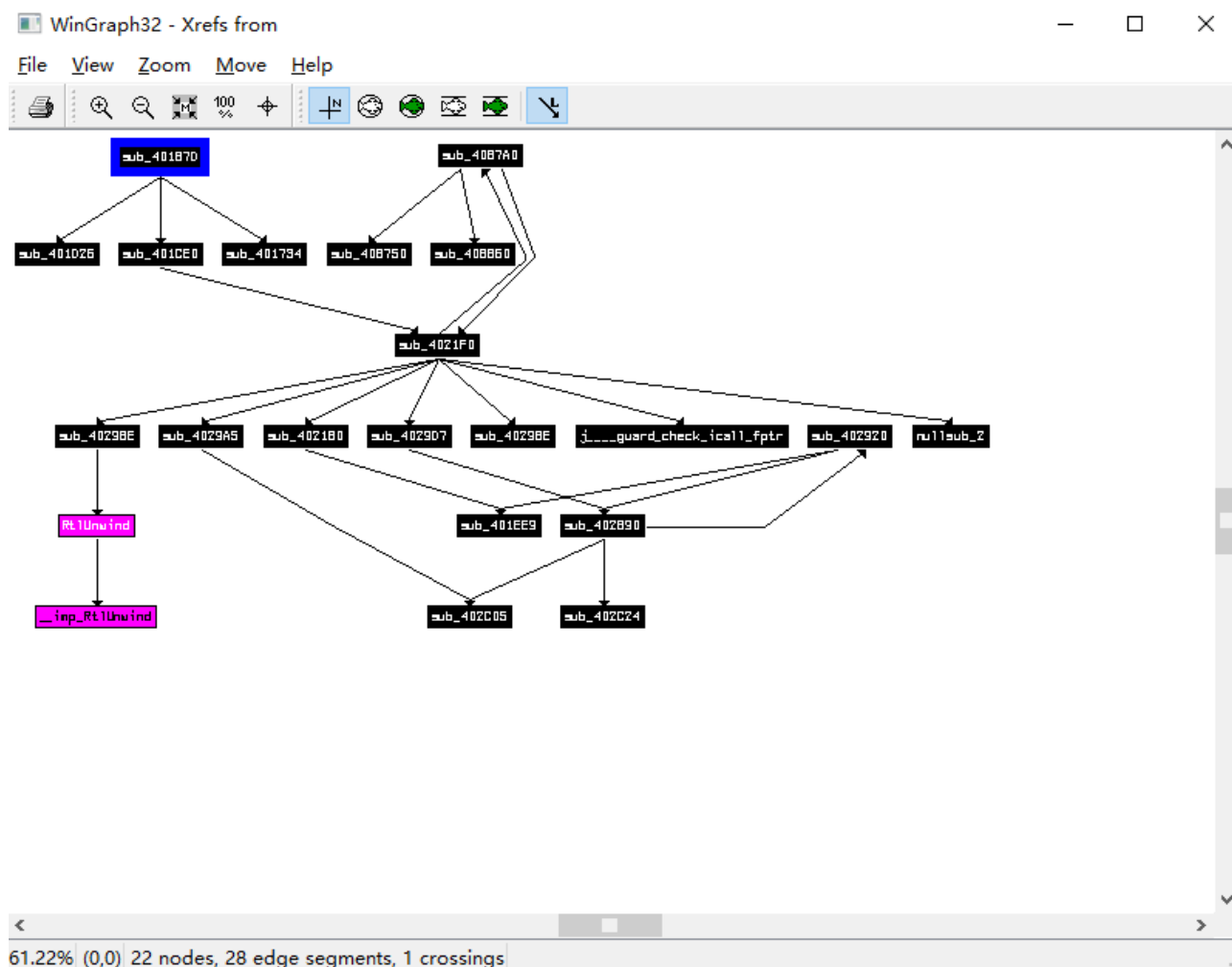


图 29：函数调用关系图

- IDA 经常查看的是字符串和导入表，导出表（适用于动态库），通过查看字符串可以得知样本需要连接的网址或者目标邮箱的名称（没有混淆的情况下），通过查看导入表可以知道样本运行需要用到的函数，如果能够看到一些关键的函数，基本就可以知道它要执行的行为了。

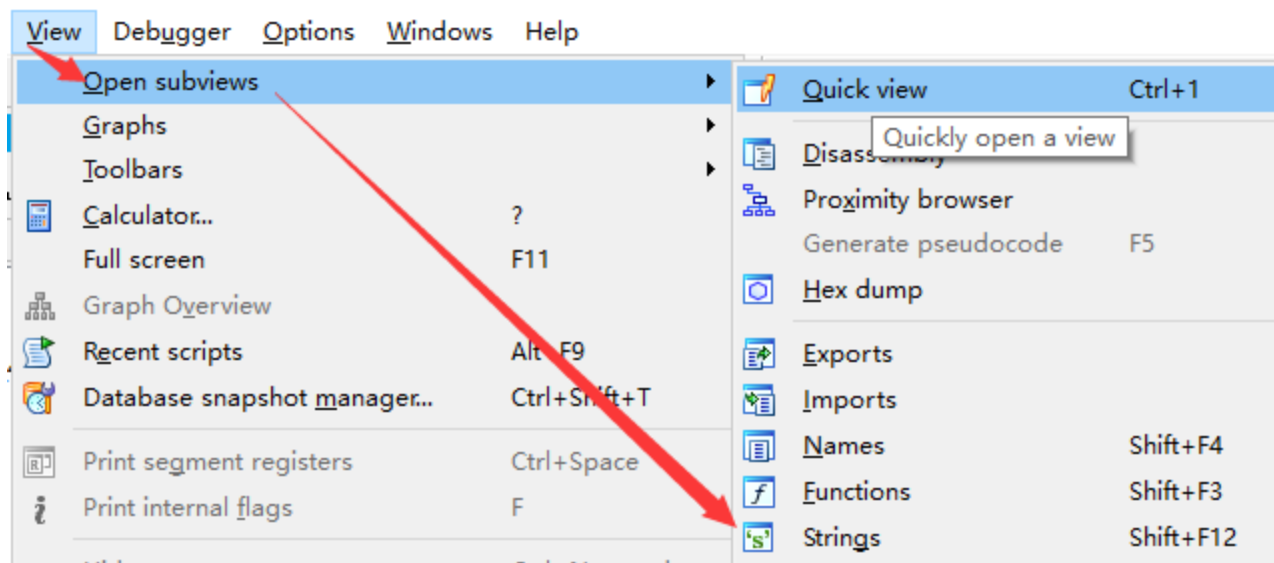


图 30：查看字符串

5) Wireshark

功能：

用于抓取网络传送的数据包，显示网络封包的详细信息。

基本使用方法：

过滤：

过滤器会帮助我们在大量的数据中迅速找到我们需要的信息。过滤器有两种，一种是显示过滤器，就是主界面上那个，用来在捕获的记录中找到所需要的记录。另一种是捕获过滤器，用来过滤捕获的封包，以免捕获太多的记录。例如过滤表达式为 `http`，用来指查看 HTTP 协议的记录。`Ip.src == x.x.x.x or ip.dst == x.x.x.x` 表示只显示源地址或目标地址是 x.x.x.x 的数据包

封包详细信息：

各行信息行为分别为：

Frame：物理层的数据帧概况

Ethernet II：数据链路层以太网帧头部信息

Internet Protocol Version 4：互联网层 IP 包头部信息

Transmission Control Protocol：传输层的数据段头部信息

Hypertext Transfer protocol：应用层信息。

使用方法：

打开 Wireshark 选择网卡后开始抓包，在过滤器中输入过滤条件，如 `“ip.addr == [ip 地址] && tcp.port == 80”` 来对 ip 地址和协议端口进行过滤，分析我们想要分析的包。在封包列表点击一个包，该行会高亮显示，并且在下面的两个窗口显示该包的详细信息。

Wireshark 界面介绍

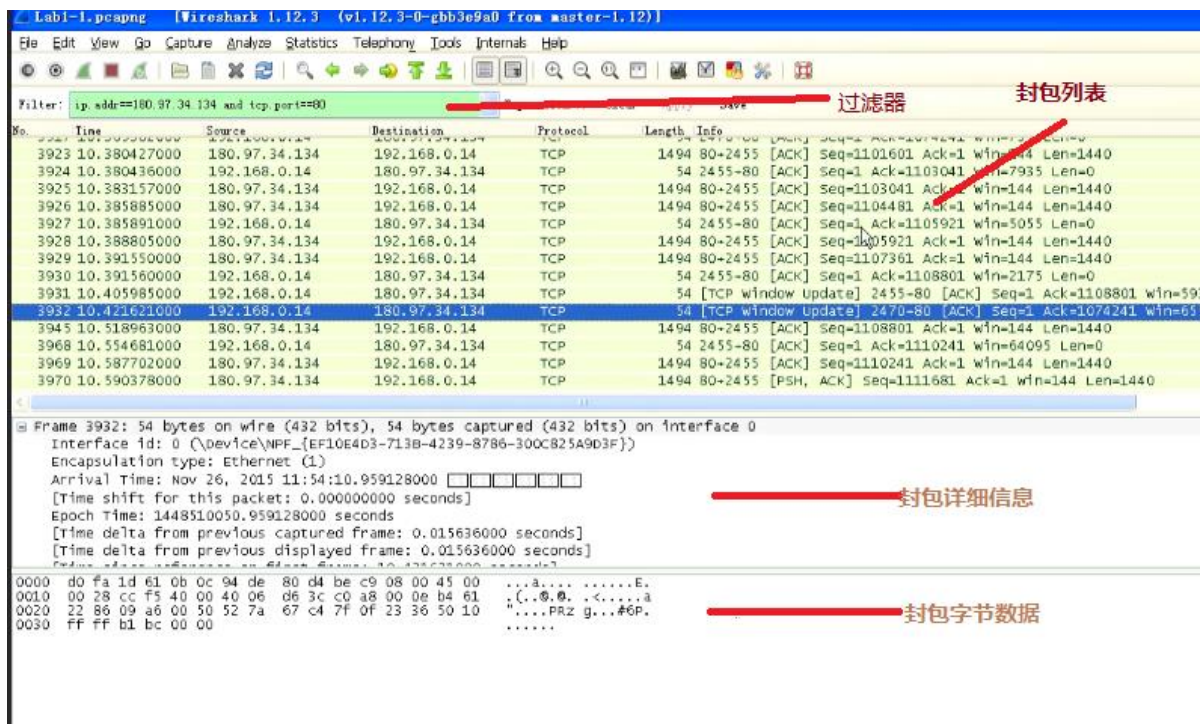


图 31: Wireshark 界面

6) PCHunter

简介: 强大的内核级监控软件。

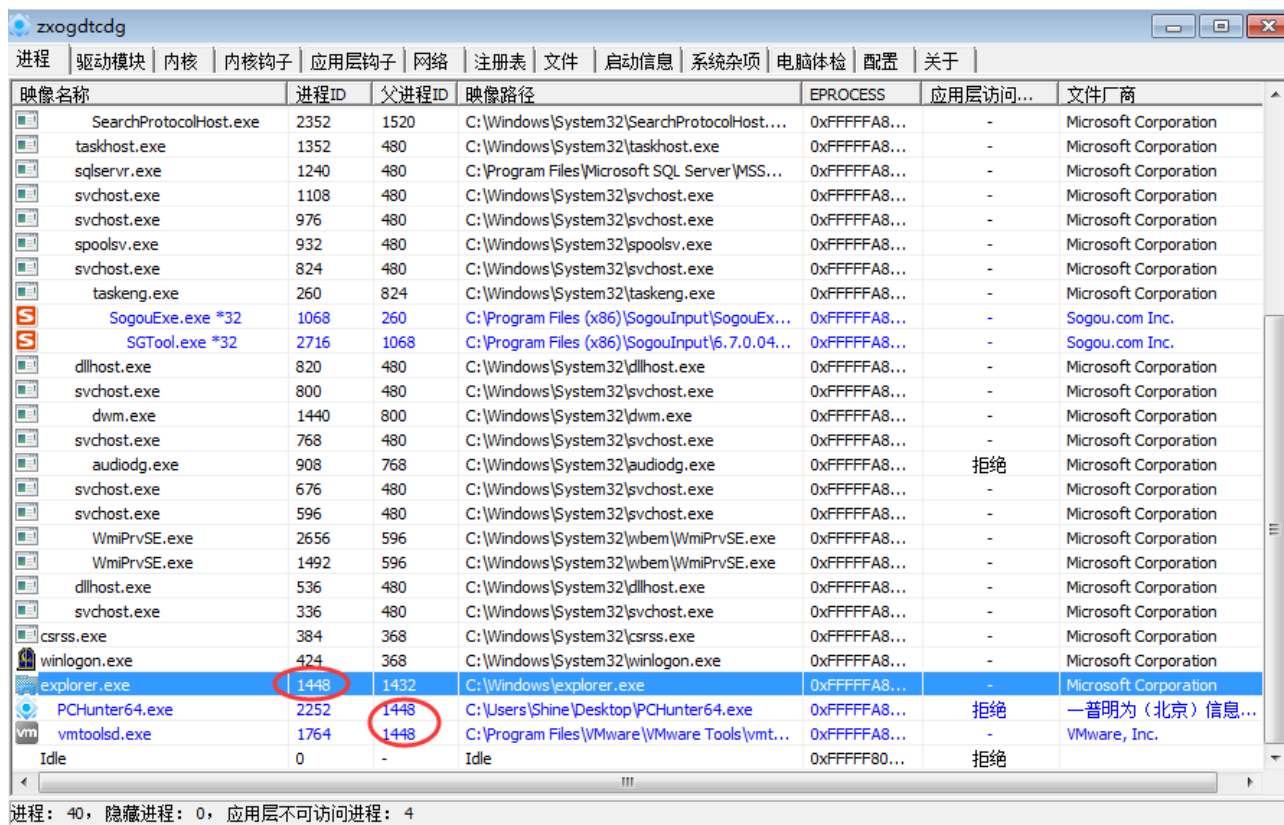
功能:

1. 进程、线程、进程模块、进程窗口、进程内存信息查看, 杀进程、杀线程、卸载模块等功能
2. 内核驱动模块查看, 支持内核驱动模块的内存拷贝
3. SSDT、Shadow SSDT、FSD、KBD、TCPIP、Nsiproxy、Tdx、Classnpn、Atapi、Acpi、SCSI、IDT、GDT 信息查看, 并能检测和恢复 ssdt hook 和 inline hook
4. CreateProcess、CreateThread、LoadImage、CmpCallback、BugCheckCallback、Shutdown、Lego 等近 20 多种 Notify Routine 信息查看, 并支持对这些 Notify Routine 的删除
5. 端口信息查看, 目前不支持 2000 系统
6. 查看消息钩子
7. 内核模块的 iat、eat、inline hook、patches 检测和恢复
8. 磁盘、卷、键盘、网络层等过滤驱动检测, 并支持删除
9. 注册表编辑
10. 进程 iat、eat、inline hook、patches 检测和恢复
11. 文件系统查看, 支持基本的文件操作
12. 查看(编辑) IE 插件、SPI、启动项、服务、Hosts 文件、映像劫持、文件关联、系统防火墙规则、IME
13. ObjectType Hook 检测和恢复
14. DPC 定时器检测和删除
15. MBR Rootkit 检测和修复

16. 内核对象劫持检测
17. WorkerThread 枚举
18. Ndis 中一些回调信息枚举
19. 硬件调试寄存器、调试相关 API 检测
20. 枚举 SFilter/Flgmgr 的回调
21. 系统用户名检测

以上是 PCHunter 的全部功能，下面介绍一些分析样本时经常用到的功能：

PCHunter 中的进程选项，显示了自身的 ID 和父进程的 ID，可以方便的确认进程之间的父子关系



映像名称	进程ID	父进程ID	映像路径	EPROCESS	应用层访问...	文件厂商
SearchProtocolHost.exe	2352	1520	C:\Windows\System32\SearchProtocolHost.exe	0xFFFFFA8...	-	Microsoft Corporation
taskhost.exe	1352	480	C:\Windows\System32\taskhost.exe	0xFFFFFA8...	-	Microsoft Corporation
sqlservr.exe	1240	480	C:\Program Files\Microsoft SQL Server\MSS...	0xFFFFFA8...	-	Microsoft Corporation
svchost.exe	1108	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
svchost.exe	976	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
spoolsv.exe	932	480	C:\Windows\System32\spoolsv.exe	0xFFFFFA8...	-	Microsoft Corporation
svchost.exe	824	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
taskeng.exe	260	824	C:\Windows\System32\taskeng.exe	0xFFFFFA8...	-	Microsoft Corporation
SogouExe.exe *32	1068	260	C:\Program Files (x86)\SogouInput\SogouEx...	0xFFFFFA8...	-	Sogou.com Inc.
SGTool.exe *32	2716	1068	C:\Program Files (x86)\SogouInput\6.7.0.04...	0xFFFFFA8...	-	Sogou.com Inc.
dllhost.exe	820	480	C:\Windows\System32\dllhost.exe	0xFFFFFA8...	-	Microsoft Corporation
svchost.exe	800	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
dwm.exe	1440	800	C:\Windows\System32\dwm.exe	0xFFFFFA8...	-	Microsoft Corporation
svchost.exe	768	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
audiodg.exe	908	768	C:\Windows\System32\audiodg.exe	0xFFFFFA8...	拒绝	Microsoft Corporation
svchost.exe	676	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
svchost.exe	596	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
WmiPrvSE.exe	2656	596	C:\Windows\System32\wbem\WmiPrvSE.exe	0xFFFFFA8...	-	Microsoft Corporation
WmiPrvSE.exe	1492	596	C:\Windows\System32\wbem\WmiPrvSE.exe	0xFFFFFA8...	-	Microsoft Corporation
dllhost.exe	536	480	C:\Windows\System32\dllhost.exe	0xFFFFFA8...	-	Microsoft Corporation
svchost.exe	336	480	C:\Windows\System32\svchost.exe	0xFFFFFA8...	-	Microsoft Corporation
csrss.exe	384	368	C:\Windows\System32\csrss.exe	0xFFFFFA8...	-	Microsoft Corporation
winlogon.exe	424	368	C:\Windows\System32\winlogon.exe	0xFFFFFA8...	-	Microsoft Corporation
explorer.exe	1448	1432	C:\Windows\explorer.exe	0xFFFFFA8...	-	Microsoft Corporation
PCHunter64.exe	2252	1448	C:\Users\Shine\Desktop\PCHunter64.exe	0xFFFFFA8...	拒绝	一普明为(北京)信息...
vmtoolsd.exe	1764	1448	C:\Program Files\VMware\VMware Tools\vmt...	0xFFFFFA8...	-	VMware, Inc.
Idle	0	-	Idle	0xFFFFF80...	拒绝	

进程：40，隐藏进程：0，应用层不可访问进程：4

图 32：进程列表

PCHunter 进程的查看选项也是一项很强的功能，我们可以查看特定进程的模块，线程，句柄，窗口，内存，定时器，热键等，最常用的应该是内存，如下图，通过内存窗口，可以知道进程中的内存地址，大小和属性。在地址栏中输入地址，在大小栏中输入要查看的内存的大小，然后点击 Dump 就可以保存相应大小的地址中的数据。将数据 dump 出来可以结合 IDA 进行静态分析。

[PCHunter64.exe]进程内存(802)

地址	大小	Protect	State	Type	模块名
0x0000000000000000	0x0000000000001000	No Access	Free		
0x0000000000001000	0x0000000000001000	ReadWrite	Commit	Map	
0x0000000000002000	0x0000000000001000	ReadWrite	Commit	Private	
0x00000000000021000	0x000000000000F000	No Access	Free		
0x0000000000003000	0x000000000000E5000		Reserve	Private	
0x000000000000115000	0x0000000000002000	ReadWrite ...	Commit	Private	
0x000000000000117000	0x00000000000019000	ReadWrite	Commit	Private	
0x000000000000130000	0x0000000000004000	Read	Commit	Map	
0x000000000000134000	0x000000000000C000	No Access	Free		
0x000000000000140000	0x0000000000001000	Read	Commit	Map	
0x000000000000141000	0x000000000000F000	No Access	Free		
0x000000000000150000	0x0000000000001000	ReadWrite	Commit	Private	
0x000000000000151000	0x000000000000F000	No Access	Free		
0x000000000000160000	0x00000000000067000	Read	Commit	Map	
0x0000000000001C7000	0x0000000000009000	No Access	Free		

地址: 大小: Dump

图 33: 进程内存信息

内核钩子和应用层钩子选项用来查看恶意样本是否对系统函数进行挂钩，一般调试驱动程序就查看内核钩子，调试应用层的程序就查看应用层钩子。

过滤驱动用来展示样本是否对系统中的设备添加了过滤驱动。下图展示了一款驱动层的恶意样本创建的过滤驱动，包含了文件系统过滤驱动和网络设备过滤驱动。

进程	内核模块	钩子	系统回调	网络	过滤驱动	注册表	文件	启动项	服务	映像劫持	本工具配置
类型	驱动对象名	驱动路径	设备	宿主驱动对象名							
File	\FileSystem\FltMgr	C:\WINDOWS\system32\drivers\fltMgr.sys	0x822D8890	\FileSystem\Ntfs							
File	\FileSystem\sr	C:\WINDOWS\system32\drivers\sr.sys	0x82072BC8	\FileSystem\FltMgr							
File	\Driver\011f	D:\Virus\011f.sys	0x81ECA020	\FileSystem\sr							
Disk	\Driver\PartMgr	C:\WINDOWS\system32\drivers\PartMgr.sys	0x81EC5A20	\Driver\Disk							
Volume	\Driver\VolSnap	C:\WINDOWS\system32\drivers\VolSnap.sys	0x81E8C020	\Driver\Ftdisk							
Tcpip	\Driver\011f	D:\Virus\011f.sys	0x81FC8768	\Driver\Tcpip							

图 34: 过滤驱动

分析方法

样本分析主要是通过静态分析或者动态调试来查看病毒的反汇编代码，通过断点或者单步来观察病毒的内存数据、寄存器数据等相关内容。逆向分析通过查看病毒的各个分支流程可以完整的、全面的查看病毒的各个流程，包括病毒需要在某些条件下才被触发的流程，都可以通过查看反汇编代码进行查看。相对于行为分析来讲，要求的技术含量更高一些。

常用分析环境：Windows7 32


常用工具：PEID、OllyDBG、IDA Pro

静态分析

静态分析技术通常是研究恶意代码的第一步。静态分析指的是分析程序指令与结构来确定目标程序的功能的过程。在这个时候，病毒本身并不在运行状态。我们一般采用以下几种方式进行静态分析

采用反病毒引擎扫描：如果尚不确定目标程序是否为病毒程序，我们可以首先采用多个不同的反病毒软件来扫描一下这个文件，看是否有哪个引擎能够识别它。（www.virscan.org、www.virustotal.com）

注意：只能通过 MD5 值查询，不允许将样本进行上传



SHA256: 9d22ec3d998a9cd4184f02802e1dec457779d06b6ea6042d38ffba56d2cee604

File name: Aizhan.SEO.Service.exe

Detection ratio: 11 / 56

Analysis date: 2015-08-05 01:05:04 UTC (1 month, 1 week ago)

Analysis File detail Additional information Comments 0 Votes

Antivirus	Result	Update
AVG	MSIL.BLHP	20150804
AVware	Trojan.Win32.Generic!BT	20150805
AhnLab-V3	Trojan/Win32.Pakes	20150804
Baidu-International	Adware.MSIL.iBryte.DBH	20150804
ESET-NOD32	a variant of MSIL/Kryptik.DBH	20150805
Fortinet	MSIL/Kryptik.DBH!tr	20150804
Ikarus	Trojan.MSIL.Crypt	20150805
McAfee	Artemis!729760241744	20150805
McAfee-GW-Edition	Artemis	20150805
Rising	PE:Trojan.Win32.Generic.18E9A264!417964644	20150731
VIPRE	Trojan.Win32.Generic!BT	20150805

图 35: VitusTotal 检测结果界面

计算哈希值：哈希是一种用来唯一标识目标程序的常用方法。目标程序通过一个哈希程序，会产生出一段唯一的用于标识这个样本的哈希值，我们可以将这个值理解为目标程序的指纹。常用的哈希算法有 MD5、Sha-1 以及 CRC32 等。由于仅仅采用一种算法，特别是 MD5 算法，有可能使得不同程序产生同样的哈希结果，所以为了谨慎起见，一般会运用多种哈希方式进行计算。

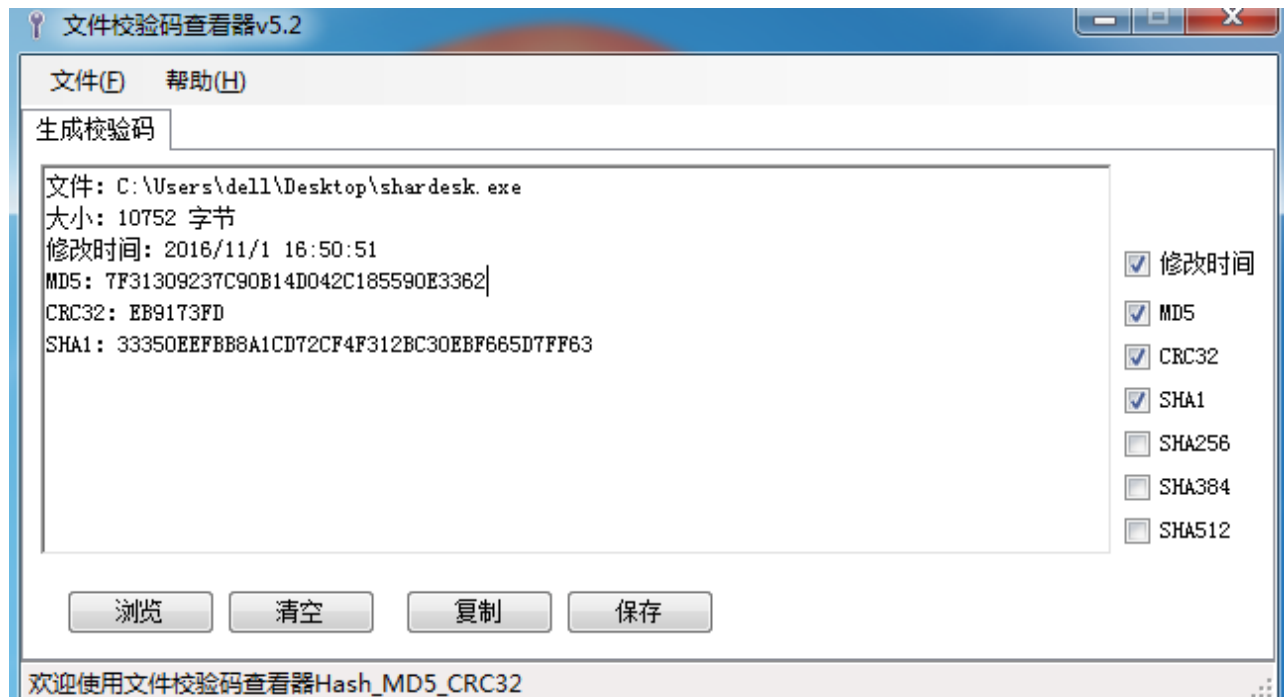


图 36：计算文件校验码

查找字符串：程序中的字符串就是一串可打印的字符序列，一个程序通常都会包含一些字符串，比如打印输出信息、连接的 URL，或者是程序所调用的 API 函数等。从字符串中进行搜索是获取程序功能提示的一种简单方法。（在 IDA 和 OD 中都可以查找字符串）并不是所有的字符串都是有意义的，但是利用这个结果，也能够给我们的静态分析带来很大的便利了。

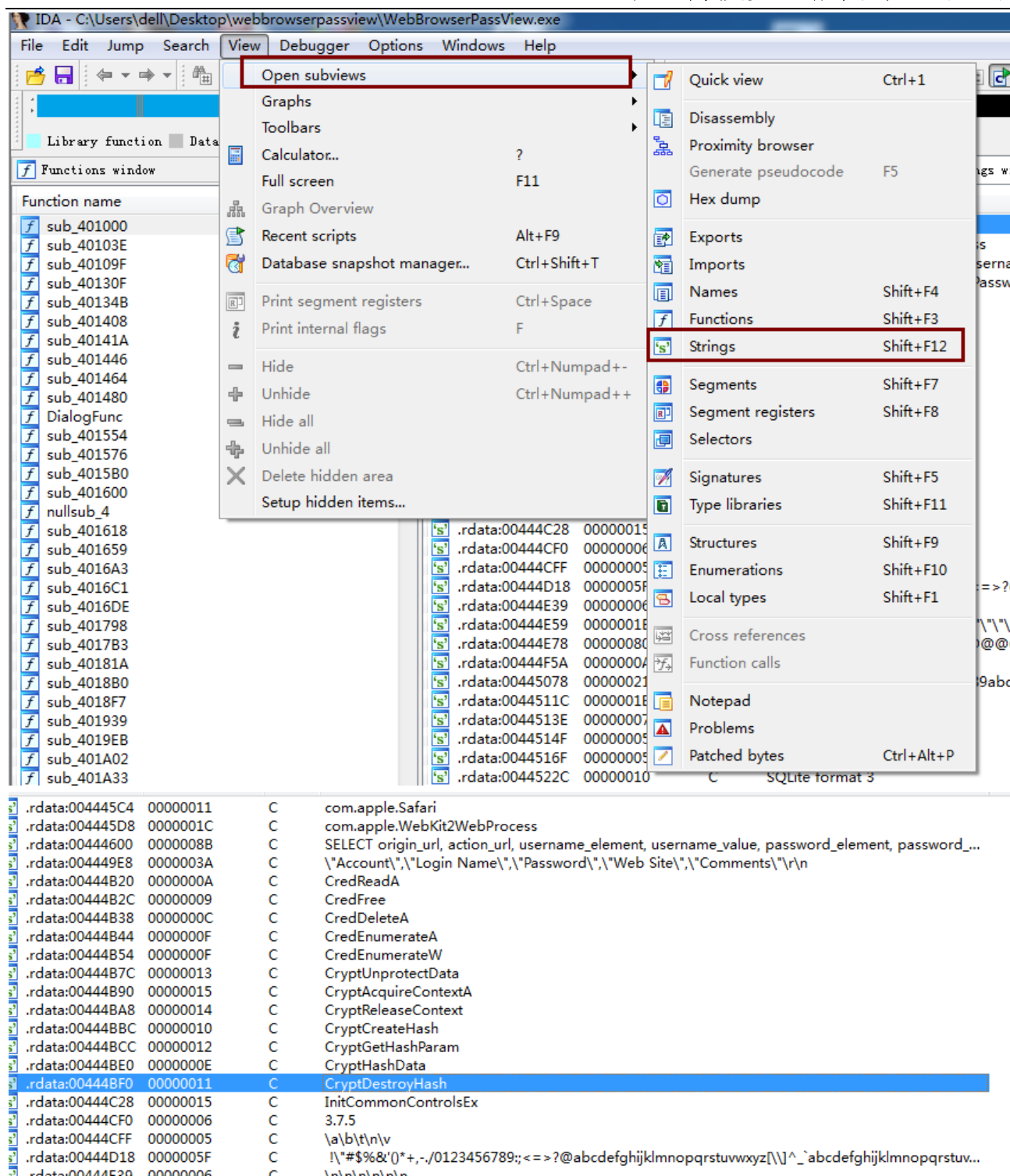


图 37：查看字符串信息

侦壳操作：病毒木马编写者经常会使用加壳技术来让他们的恶意程序难以被检测或分析。正常的程序总是会包含很多字符串。而加了壳的恶意代码通过分析所得到的可打印字符串就会很少。如果查找出的程序的字符串很少时，那么这个程序就很有可能是加了壳的。此时往往就需要使用其它方法来进一步检测它们的行为。(常用 PEiD 进行查壳)

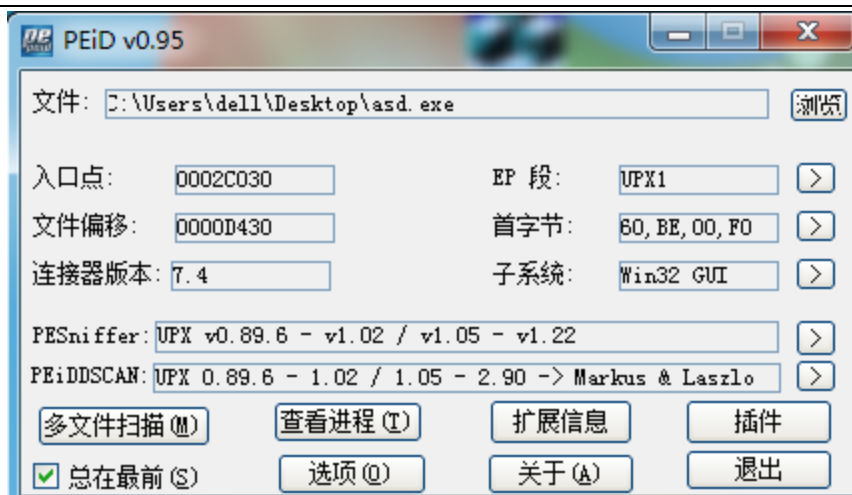


图 38：查壳

PE 分析：对于目标程序 PE 结构的分析，往往可以使我们获取更多的信息。常用的工具有 PView 和 Resource Hacker 等。PE 的分析较为复杂。

使用 IDA Pro 进行高级静态分析：IDA Pro 这款软件可以说是所有反病毒工程师的首选，它除了能够反汇编整个程序以外还能执行查找函数、栈分析、本地变量标识等任务。而且 IDA Pro 生来就是交互式的，其反汇编过程的所有属性都可以被修改、操作、重新安排或重新定义。而我们在实际分析的过程中，也应当及时地将已分析出的内容进行重命名或进行标注。这款软件很好的一点是它能够以图形化的界面来为用户揭示程序的整个执行流程。

动态分析

所谓动态分析就是在运行恶意代码之后进行检查的过程，它能够让你观察到恶意代码的真实功能。虽然动态分析技术非常强大，但是它们还是应该在静态分析之后进行，并且应该在虚拟的环境中进行。通常的选择是在 vmware workstation 中安装分析用的操作系统（如 win7）来进行样本分析。除此之外，还可以通过使用沙盘软件对样本进行动态分析，沙盘是一种在安全环境里运行可疑程序的一种分析方法，无需担心伤害到我们真实的系统。如：沙盘软件 sandboxie 经查资料还有其他一些沙盘软件（defensewall、geswall、bufferzone 等）

sandboxie 的使用

在沙盘中点击右键并选择运行任意程序。

点击浏览，选择需要测试的软件就可以了。



图 39：运行沙箱

行为监控：Process Monitor 是在反病毒领域最为常用的软件了。Process Monitor 一旦开始运行，它就会监控所有能够捕获的系统调用。

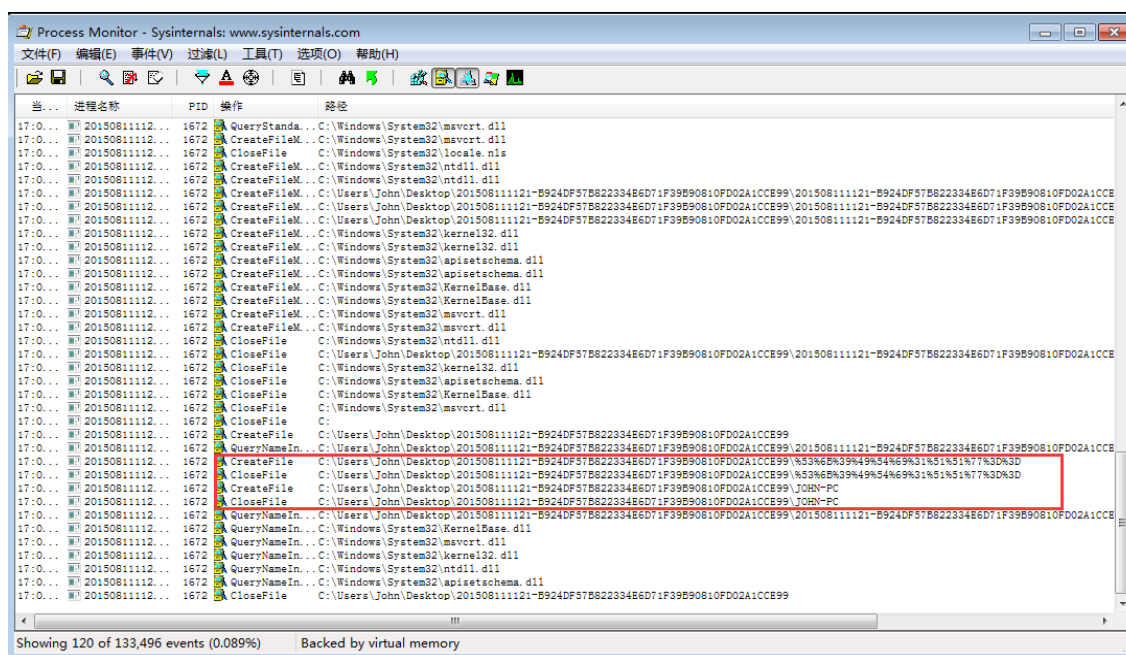


图 40：ProcMon 行为监控

动态调试

使用调试器对病毒进行分析在反病毒工作中扮演着十分重要的角色。调试器允许你查看任意内存地址的内容、寄存器的内容以及每个函数的参数。调试器也允许你在任意时刻改变关于程序执行的任何东西。比如你可以在任意时刻改变一个变量的值——前提是你需要获得关于这个变量足够的信息，包括在内存中的位置。在实际的动态调试过程中，我们最常用的是 OllyDBG 和 WinDbg，前者是病毒分析人员使用最多的调试器，缺点是不支持内核调试，如果想调试内核，WinDbg 基本上就是唯一的选择了。虽然 IDA Pro 也能够进行动态调试，但是它远远不如 OD 方便。因此在实际分析的过程中，往往是将二者结合

The screenshot displays the Immunity Debugger interface with four main panes:

- Assembly Pane (Top Left):** Shows assembly instructions for the function `asd.<ModuleEntryPoint>`. The instruction stream includes:
 - `push edx`
 - `push 0x36`
 - `push 0xFEE39965`
 - `call asd.004010FE`
 - `jmp 到 kernel32.Virtu`
 - `push dword ptr ss:[ebp-0x60]`
 - `lea ecx,dword ptr ss:[ebp-0xBC]`
 - `push ecx`
 - `call asd.00401C8C`
 - `push ebp`
 - `mov ebp,esp`
 - `add esp,-0x11C`
 - `call asd.004011CC`
 - `or eax,eax`
 - `je asd.00401EC9`
 - `push 0x34`
 - `push dword ptr ss:[ebp-0x154]`
 - `user32.75A20BF0`
 - `lea eax,dword ptr ss:[ebp-0x100]`
 - `push eax`
 - `push dword ptr ss:[ebp-0xC]`
 - `push dword ptr ss:[ebp-0x108]`
 - `push 0x98158438`
 - `call asd.00401102`
 - `push 0x8F252E11`
 - `jmp 到 kernel32.Multi`
- Registers Pane (Top Right):** Displays the state of CPU registers. Notable values include:
 - EAX: 0012FF0C
 - ESP: 0012FF8C
 - EIP: 00401CFE
 - LastErr: ERROR_SXS_KEY_NOT_FOUND (000036B7)
- Memory Dump Pane (Bottom Left):** Shows raw hex data and its ASCII representation. The address range shown is from `0012FF6C` to `0012FFFC`.
- Disassembly Pane (Bottom Right):** Provides a detailed view of the instruction at address `0012FF8C`, which is `7587ED6C`. It shows the instruction type as "返回到" (Return) and the target address as `kernel32.7587ED6C`.

图 41: OD 动态调试

不同格式文件分析方法

1) EXE (C\C++)

使用工具:

ProcessMonitor, OD 或者 WinDbg, IDA, WireShark

分析方法:

首先运行一下应用程序, 使用 ProcessMonitor 监控一下样本的行为, 查看一下有没有关键的恶意为, 比如释放了文件, 创建子进程, 修改了注册表, 连接服务器进行数据交换等等。先对样本有一个大概的定位。如果有网络行为, 那么就使用 WireShark 抓取网络封包。

将样本拖进 IDA 中, 查看一下字符串表和导入表, 搜索一下有没有特殊的字符串或者关键的系统函数。

关键的字符串, 例如 ThunderBird, FoxMail 等是邮箱的客户端, 如果存在这些信息, 有很大的可能是窃取用户的账户和密码, 如果有看到网址, IP 之类的, 就会有流量传输, 用来传输窃取到的信息或者想要从服务器上下载其他的恶意软件。

's'	.rdata:00413704	0000001E	C	mm\\Eudora\\CommandLine\\current
's'	.rdata:00413724	00000008	C	Software
's'	.rdata:0041372C	0000001D	C	\\Mozilla\\Mozilla Thunderbird
's'	.rdata:0041374C	00000008	C	%s\\Main
's'	.rdata:00413754	00000012	C	Install Directory
's'	.rdata:00413768	0000000C	C	sqlite3.dll
's'	.rdata:00413774	00000009	C	nss3.dll
's'	.rdata:00413784	0000001F	C	gramfiles%\\Mozilla Thunderbird
's'	.rdata:004137A4	0000000E	C	ShowGridLines
's'	.rdata:004137B4	00000010	C	SaveFilterIndex
's'	.rdata:004137C4	00000014	C	AddExportHeaderLine
's'	.rdata:004137D8	00000010	C	MarkOddEvenRows
's'	.rdata:004137E8	0000000B	C	%s %s %s
's'	.rdata:004137FC	0000000B	C	User Name
's'	.rdata:00413808	0000000F	C	IMAP User Name
's'	.rdata:00413818	00000013	C	HTTPMail User Name
's'	.rdata:00413830	0000000B	C	User Name
's'	.rdata:0041383C	0000000C	C	POP3 Server
's'	.rdata:00413848	0000000C	C	IMAP Server
's'	.rdata:00413854	00000010	C	HTTPMail Server
's'	.rdata:00413864	0000000C	C	SMTP Server
's'	.rdata:00413874	0000000B	C	Password2
's'	.rdata:00413880	00000008	C	IMAP Pas
's'	.rdata:00413888	00000007	C	sword2
's'	.rdata:00413890	00000013	C	HTTPMail Password2
's'	.rdata:004138A4	0000000F	C	SMTP Password2
's'	.rdata:004138B4	0000000A	C	POP3 Port
's'	.rdata:004138C0	0000000A	C	IMAP Port
's'	.rdata:004138CC	0000000E	C	HTTPMail Port
's'	.rdata:004138DC	0000000A	C	SMTP Port
's'	.rdata:004138E8	00000017	C	POP3 Secure Connection
's'	.rdata:00413900	00000017	C	IMAP Secure Connection
's'	.rdata:00413918	0000001B	C	HTTPMail Secure Connection
's'	.rdata:00413934	00000010	C	SMTP Secure Conn
's'	.rdata:00413944	00000007	C	ection
's'	.rdata:0041394C	00000008	C	SMTP Dis
's'	.rdata:00413954	0000000A	C	play Name
's'	.rdata:00413960	00000013	C	SMTP Email Address
's'	.rdata:00413974	0000000E	C	POP3 Password
's'	.rdata:00413988	0000000A	C	Password
's'	.rdata:00413998	0000000A	C	Password
's'	.rdata:004139A4	0000000E	C	SMTP Password
's'	.rdata:004139B4	0000000A	C	POP3 User
's'	.rdata:004139C4	00000006	C	User
's'	.rdata:004139CC	0000000A	C	HTTP User

图 42: 查看字符串

关键的系统函数：CreateProcess 经常被用来创建子进程。RegSetValue 设置注册表键值，常用来自身设置为开机启动。Http 开头的函数，用来和服务器连接进行数据传输














	004030E8	CopyFileW	KERNEL32
	00403100	CreateFileW	KERNEL32
	0040308C	CreateMutexW	KERNEL32
	004030B0	CreateProcessW	KERNEL32
	00403044	CreateSolidBrush	GDI32
	0040305C	CreateThread	KERNEL32
	00403074	CreateToolhelp32Snapshot	KERNEL32
<hr/>			
	00403010	RegCloseKey	ADVAPI32
	00403000	RegDeleteValueW	ADVAPI32
	00403028	RegDisableReflectionKey	ADVAPI32
	00403034	RegOpenKeyExW	ADVAPI32
	0040300C	RegSetValueExW	ADVAPI32
	004031C8	RegisterClassExW	USER32

图 43: 查看导入表

对样本有个大概的定位后，使用 OD 或者使用 Windbg 对样本进行调试。

2) SYS

使用工具:

WinDbg, IDA, WireShark, PCHunter

分析方法:

首先加载驱动程序，使用 PCHunter 查看一下有没有在系统空间创建什么过滤设备，Hook 某个函数，有没有注入到某个进程

将样本拖进 IDA 中，查看一下字符串表和导入表，搜索一下有没有特殊的字符串或者关键的系统函数。可能此时看到的都是些乱码级的字符串，所以可能是样本在运行的时候会自己对用到的字符串进行解密，所以可以在调试时，样本运行了一部分后再 dump 内存，这时候再看他的字符串表，可能就不会有那么多的乱码了。

对样本有一个大概的定位后，使用 Windbg 进行调试。由于是驱动文件，不会像应用层那样，直接断在程序的入口处，所以需要先找到程序的入口。

uf nt!IopLoadDriver

32 位系统下，在 nt!IopLoadDriver+663 处下断点

64 位系统下，在 nt!IopLoadDriver+9FE 处下断点

接着运行代码，在断点处断下来，会看到要给 call 指令，F11 进去就到了 DriverEntry。

例如：（以 Win7 x64 举例）

打开虚拟机和 Windbg，连接上以后，在 Command 窗口的命令行中输入 uf nt!IopLoadDriver，会显示出 IopLoadDriver 函数的汇编代码：

Command

```

kd> uf nt!IopLoadDriver
nt!IopLoadDriver:
fffff800`04264a60 48895c2410      mov     qword ptr [rsp+10h],rbx
fffff800`04264a65 55             push    rbp
fffff800`04264a66 56             push    rsi
fffff800`04264a67 57             push    rdi
fffff800`04264a68 4154           push    r12
fffff800`04264a6a 4155           push    r13
fffff800`04264a6c 4156           push    r14
fffff800`04264a6e 4157           push    r15
fffff800`04264a70 4881ec90020000 sub     rsp,290h
fffff800`04264a77 488b0562dfd8ff mov     rax,qword ptr [nt!_security_cookie (fffff800`03f
fffff800`04264a7e 4833c4         xor     rax,rax
fffff800`04264a81 4889842480020000 mov     qword ptr [rsp+280h],rax
fffff800`04264a89 33ed           xor     ebp,ebp
fffff800`04264a8b 488d842480000000 lea     rax,[rsp+80h]
fffff800`04264a93 4c898c2420010000 mov     qword ptr [rsp+120h],r9
fffff800`04264a9b 418ad8         mov     bl,r8b
fffff800`04264a9e 408af2         mov     sil,dl
fffff800`04264aa1 418929         mov     dword ptr [r9],ebp
fffff800`04264aa4 4533c9         xor     r9d,r9d
fffff800`04264aa7 4533c0         xor     r8d,r8d
fffff800`04264aaa 33d2           xor     edx,edx
fffff800`04264aac 4c8be9         mov     r13,rcx
fffff800`04264aaf 4c8bf5         mov     r14,rbp
fffff800`04264ab2 66896c2468     mov     word ptr [rsp+68h],bp
fffff800`04264ab7 66896c246a     mov     word ptr [rsp+6Ah],bp
fffff800`04264abc 48896c2470     mov     qword ptr [rsp+70h],rbp
fffff800`04264ac1 448bfd         mov     r15d,ebp
fffff800`04264ac4 4889ac24a0000000 mov     qword ptr [rsp+0A0h],rbp
fffff800`04264acc 6689ac2498000000 mov     word ptr [rsp+98h],bp
fffff800`04264ad4 6689ac249a000000 mov     word ptr [rsp+9Ah],bp
fffff800`04264adc 48896c2460     mov     qword ptr [rsp+60h],rbp
fffff800`04264ae1 89ac24b0000000 mov     dword ptr [rsp+0B0h],ebp
fffff800`04264ae8 4889442420     mov     qword ptr [rsp+20h],rax
fffff800`04264aed e8eeb7eeff     call    nt!NtQueryKey (fffff800`041502e0)
fffff800`04264af2 448d6502       lea     r12d,[rbp+2]
fffff800`04264af6 3d050000080    cmp     eax,80000005h
fffff800`04264afb 7411           je      nt!IopLoadDriver+0xae (fffff800`04264b0e)

nt!IopLoadDriver+0x9d:
fffff800`04264afd 3d2300000c0    cmp     eax,0C0000023h
fffff800`04264b02 740a           je      nt!IopLoadDriver+0xae (fffff800`04264b0e)

nt!IopLoadDriver+0xa4:

```

kd>

英

Ln 0, Col 0 Sys 0:KdSrv:S Proc 000:0 Thrd 000:0 ASM OVR CAPS NUN

图 44

使用快捷键 Ctrl+F 搜索 9FE

```

nt!IopLoadDriver+0x9fe:
fffff800`0426545e 488bd6         mov     rdx,rsi
fffff800`04265461 488bcb         mov     rcx,rbx
fffff800`04265464 ff5358         call    qword ptr [rbx+58h]
fffff800`04265467 4c8b15da3bdaff mov     r10,qword ptr [nt!PnpEtwHandle (fffff800`0400904
fffff800`0426546e 8bf8           mov     edi,eax
fffff800`04265470 898424e0000000 mov     dword ptr [rsp+0E0h],eax
fffff800`04265477 4c3bd5         cmp     r10,rbp
fffff800`0426547a 0f848e00000000 je      nt!IopLoadDriver+0xaae (fffff800`0426550e)

```

图 45

在命令行输入 bp fffff800`04265464，在地址 fffff800`04265464 出处下断点，接着输入“g”，让虚拟机运行。

回到虚拟机中，使用软件 KmdManager 加载驱动，打开 KmdManager，界面如下：

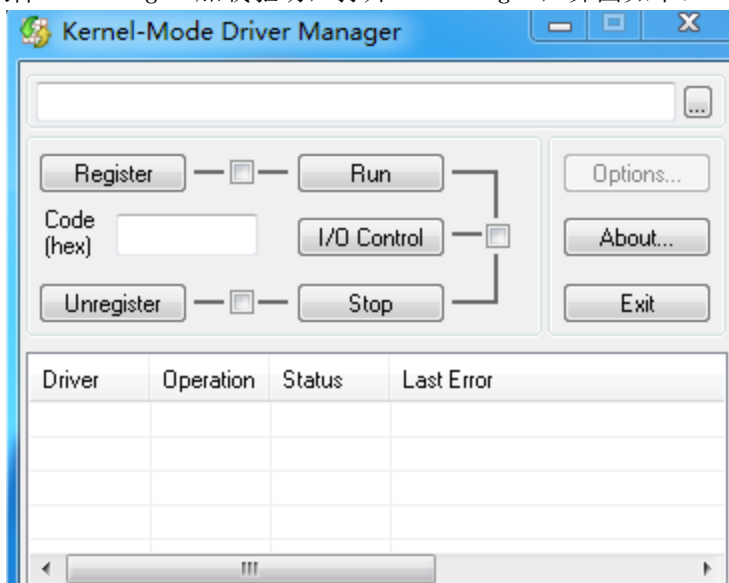


图 46

将 sys 文件拖入上图所示的窗口中，在小方框中打勾（可以在卸载驱动的时候再打下面的勾），如下图所示：

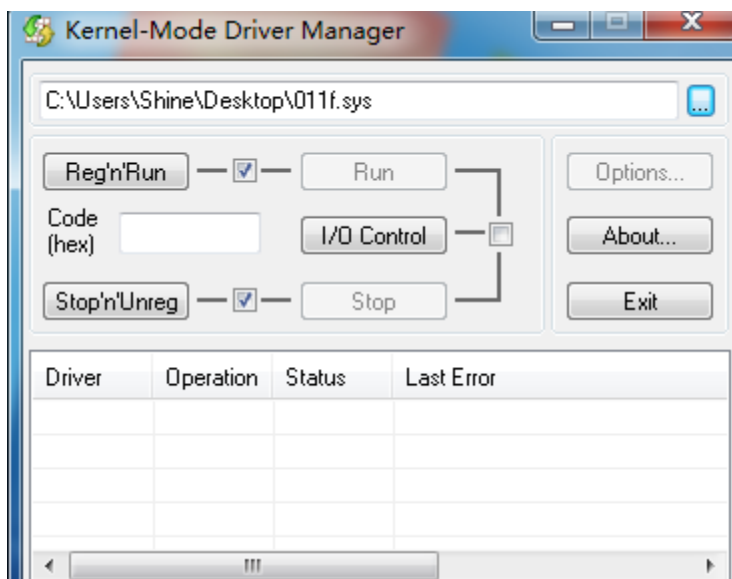


图 47

然后点击 Reg`nRun 加载驱动，会看到断点断在我们设置的地方，然后 F11 进入就达到了 DriverEntry 开始调试。

注意：32 位的驱动程序要在 32 位系统下调试，64 的驱动程序要在 64 位系统下调试

3) JS 代码

js 代码通常嵌入在 html 文件中，一般经过编码或者加密，无法直接看懂源码。可以对其进行解码和解密或者在浏览器中调试看处理后的代码。功能通常是作为下载器从互联网上下载其他恶意代码并在本地执行，或者作为一个启动器安装恶意代码来执行。

可以对其进行解码和解密或者在浏览器中调试看处理后的代码，js 的代码可以通过 process monitor 监控它下载或生成的文件然后对其分析。

4) Doc

使用工具：

OllyDbg 或者 Winddbg 或者 Immunity Debugger, IDA, Wireshark

对于 word 类的恶意样本，基本上都是漏洞利用。这里只讲解公开的漏洞。使用 virustotal 查看一下样本利用的漏洞的 CVE 编号。确定了 CVE 号之后，可以通过网上查找相关的资料（一般都有很多分析案例），先对此漏洞有个认识。然后是搭建相关的环境。因为不同的漏洞，对 office 的版本是对应的，需要搭建存在此漏洞的 office 版本。

Word 类样本通常是通过释放应用程序来执行恶意功能，释放程序的方式有通过运行宏来释放，还有就是在恶意的 word 中嵌入 shellcode，此 shellcode 会连接服务器，从服务器上下载恶意程序。

通过 shellcode：

shellcode 是嵌入在 word 文档中的，所以需要对 word 文档进行漏洞分析，定位到 shellcode，然后对其进行分析。一般情况下样本都是通过 jmp esp 跳转到 shellcode 的。

通过宏：

当我们打开恶意的 doc 的时候，如果他的恶意代码保存在宏中，就会弹出一个框，让我们启动宏。我们可以使用如下的方法来查看宏：点击工具栏中的“视图”->“宏”->“查看宏”。

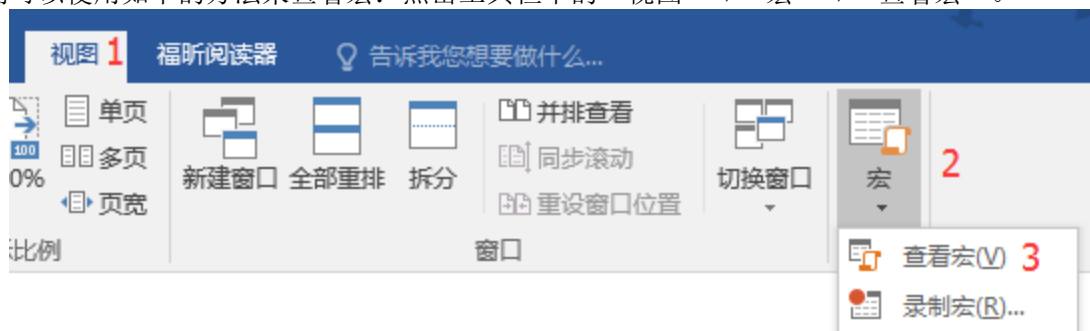


图 48：查看宏

宏的调试方法：

如果样本中存在宏，在打开样本的时候，如果你禁用了宏，会出现类似的对话框

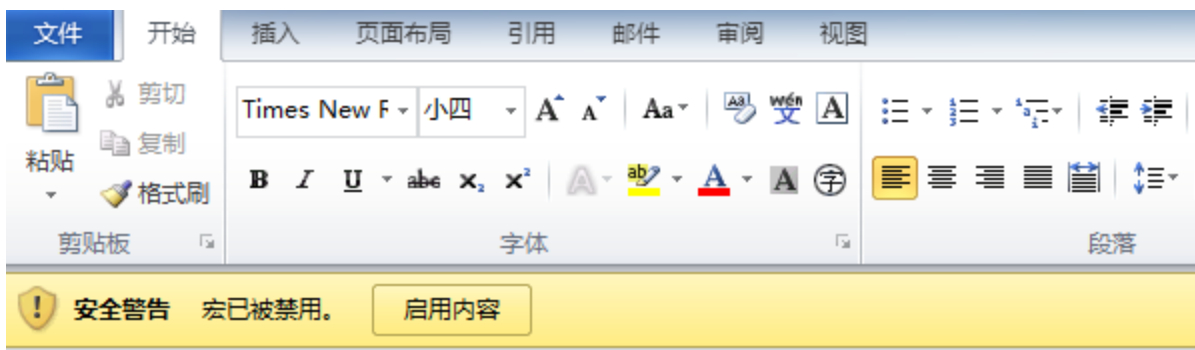


图 49

如果要调试宏，需要点击启用内容，然后依次点击视图->宏->查看宏

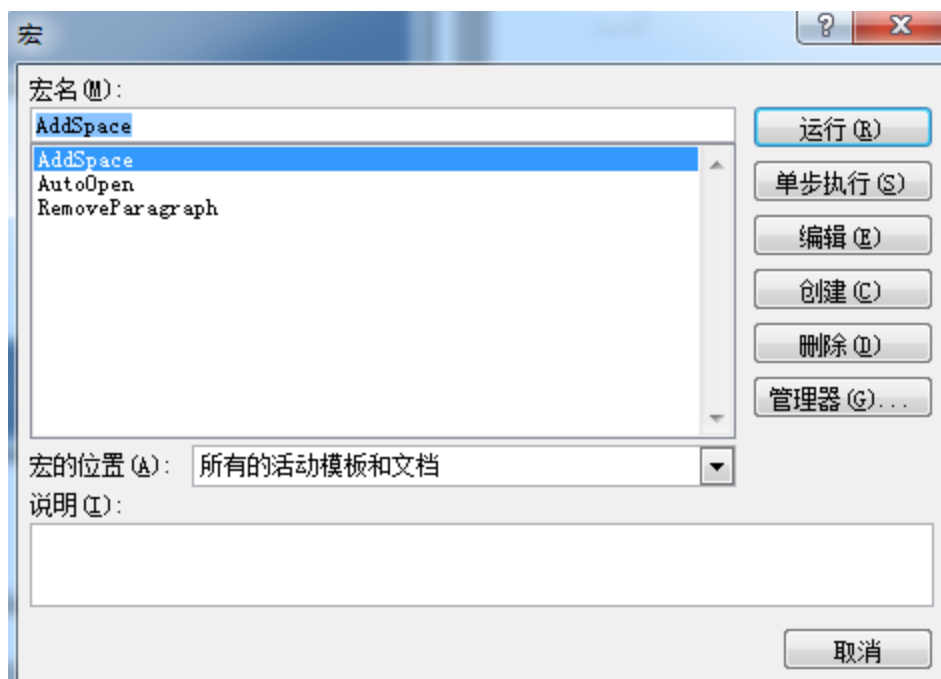


图 50

点击单步执行来调试选中的宏

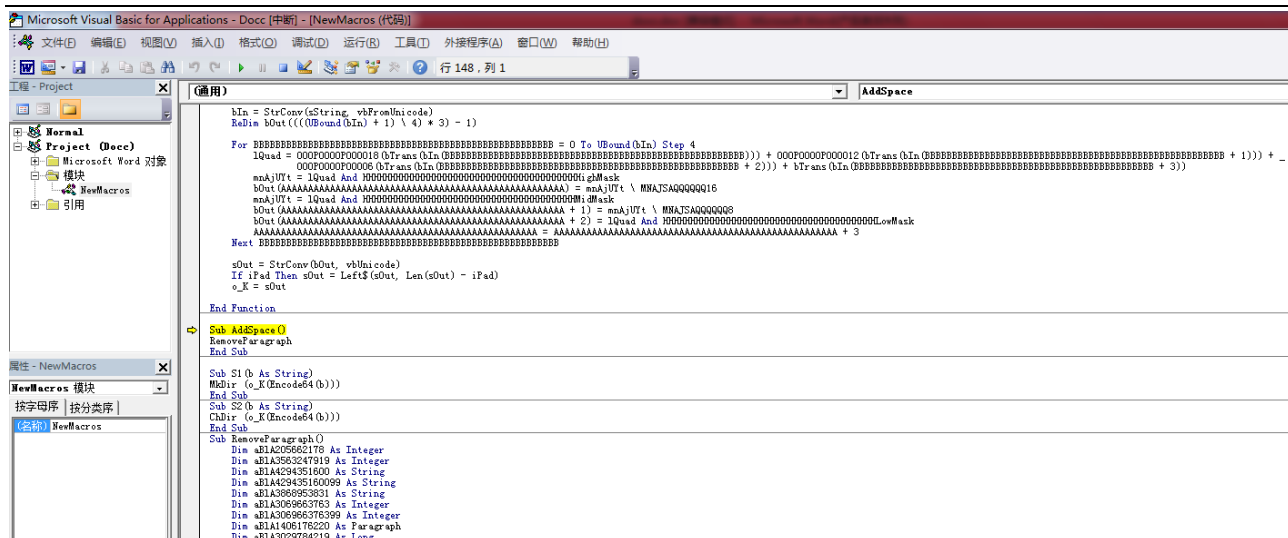


图 51

基本调试命令:

F8: 步入

Shift+F8: 步过

Ctrl+F8: 运行到光标

F9: 下断点

可以通过本地窗口查看变量的值, 打开方法: 视图->本地窗口。可以通过此方法打开其他的窗口

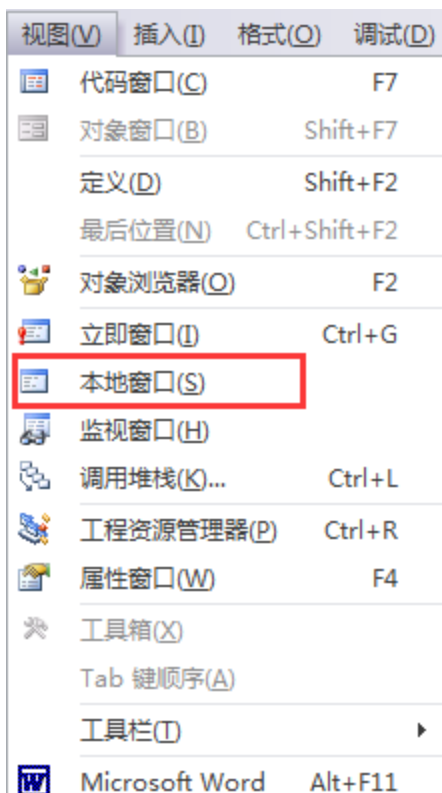


图 52

通过本地窗口查看变量的值:

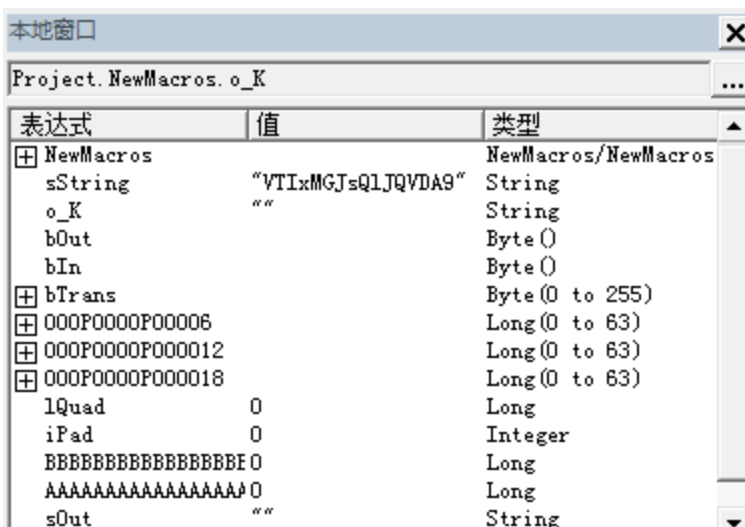


图 53: 查看变量值

5) Elf 文件

Elf 文件属于 linux 下的应用程序，分析此类样本可以使用 linux 下的 GDB 调试，也可以使用 IDA 进行双机调试，这两种方法各有利弊，根据自己的喜好进行选择。

使用 GDB 调试需要的基本命令：

启动 GDB: `gdb exeFileName`

查看寄存器的值: `i r register` 例如: `i r eip` 查看 eip 的值

设置寄存器的值: `set $register = value` 例如: `set $eip=0x12345678` 对 eip 的值进行设置

看汇编代码: `layout asm`

源码调试时: `n`: 步过函数 `s`: 进入函数

非源码调试时: `ni`: 步过函数 `si`: 进入函数

查看内存的命令: `x`。

`x/x` 以十六进制输出

`x/d` 以十进制输出

`x/c` 以单字符输出

`x/i` 反汇编 - 通常，会使用 `x/10i $ip-20` 来查看当前的汇编（\$ip 是指令寄存器）

`x/s` 以字符串输出

查看当前进程的线程: `info threads`

切换线程: `thread <ID>`，切换调试的线程为指定 ID 的线程

GDB 调试是用命令行进行操作，操作不太方便，而且没有友好的用户界面，适合喜欢命令行的人士使用。

IDA 双机调试:

使用 IDA 进行调试，有友好的用户界面，有的样本还可以进行反汇编，可以看到高级语言，大大降低了分析的难度。要使用 IDA 进行双机调试，首先要搭建相关的环境。

将 IDA 安装目录下的 `dbgsvr` 文件下的 `linux_server` 文件拷贝到 linux 中，然后通过命令行定位到文件的存放地方，按如下步骤操作：

1. `chmod +x linux_server` 增加可运行权限

此时通过 `ls -l linux_server` 可以看到其拥有的可执行权限

```
root@ubuntu:/home/wangfeng# chmod +x linux_server
root@ubuntu:/home/wangfeng# ls -l linux_server
-rwxrwxrwx 1 wangfeng 650724 Apr 13 2015 linux_server
```

图 54

2. `./linux_server` 开启执行 server 端，出现如下信息：

```
root@ubuntu:/home/wangfeng# ./linux_server
IDA Linux 32-bit remote debug server(ST) v1.19. Hex-Rays (c) 2004-2015
Listening on port #23946...
```

图 55：开启服务端

1. 回到 IDA 中，选择 Debugger 选项，选择 Seletedebugger

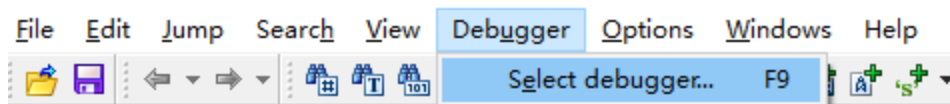


图 56

2. 在弹出的对话框中选择 Remote Linux debugger，点击 OK。

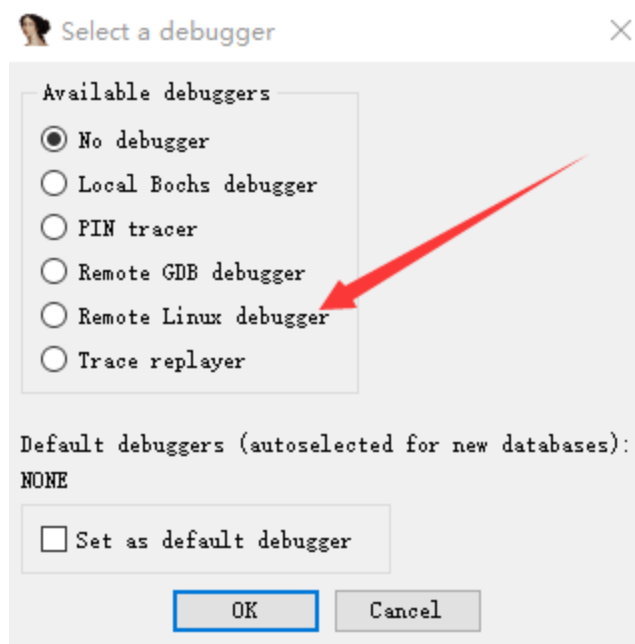


图 57：选择调试

3. 然后再次点击 Debugger，选择 Process Options...

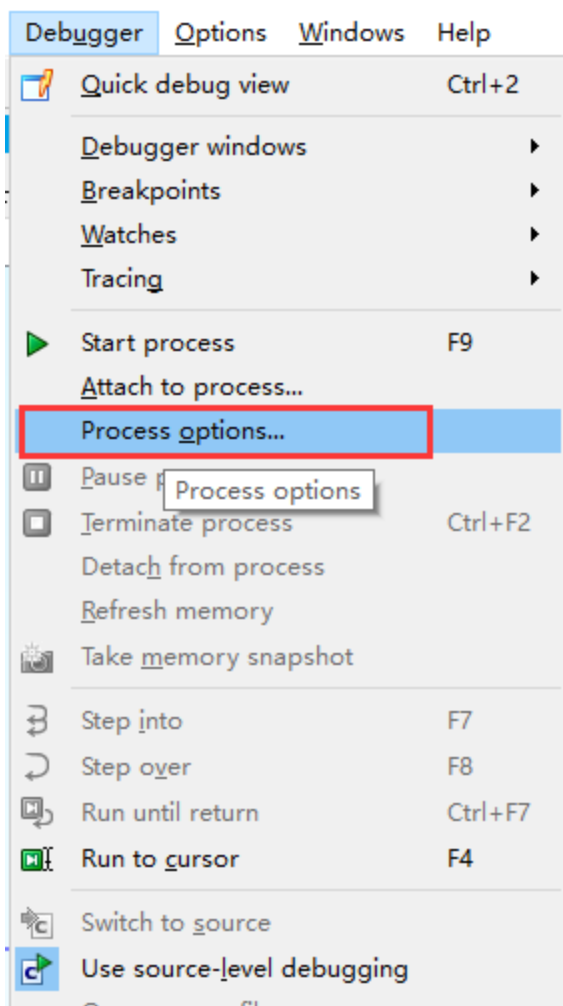


图 58

4. 在 Hostname 框中输入虚拟机的 IP 地址，点击 OK，然后就可以开始调试了

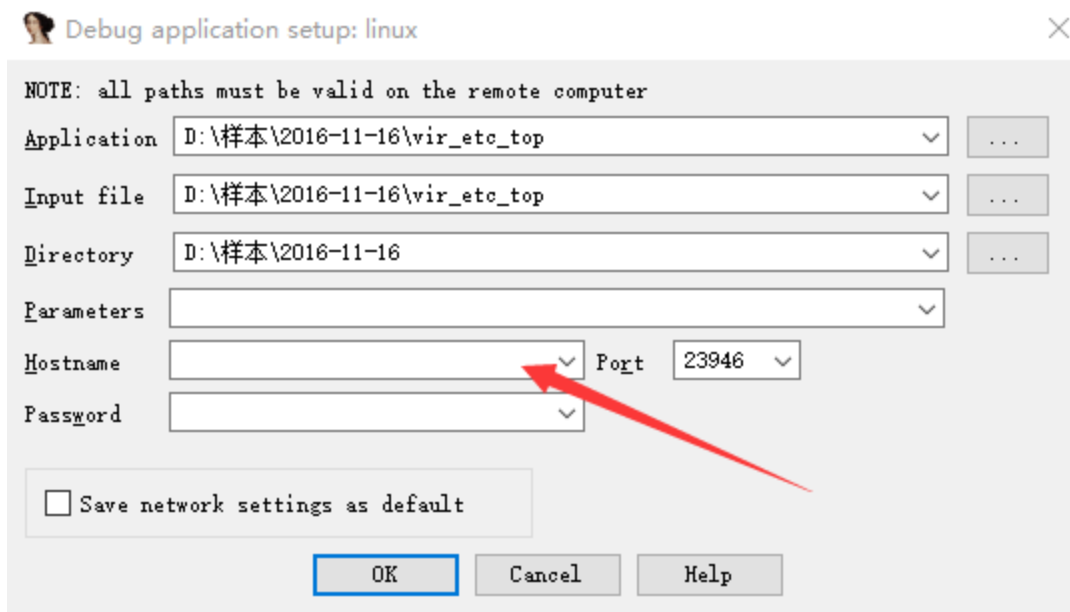


图 59: 配置调试信息

在 linux 下查看 IP 地址：在命令行输入 ifconfig

```
root@ubuntu:/home/...# ifconfig
eth0      Link encap:Ethernet  HWaddr ...
          inet addr:192.168.126.158 Bcast:... Mask:255...
          inet6 addr: fe80::20c:29ff:fe85:ed85/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2865 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1130 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1771728 (1.7 MB)  TX bytes:101216 (101.2 KB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:519 errors:0 dropped:0 overruns:0 frame:0
          TX packets:519 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:46202 (46.2 KB)  TX bytes:46202 (46.2 KB)
```

图 60：查看 IP 地址

6) C#程序

对于 C#样本，使用 OD 和 IDA 调试都比较困难，因此选用针对 C#的逆向工具 dnspy 进行反编译：

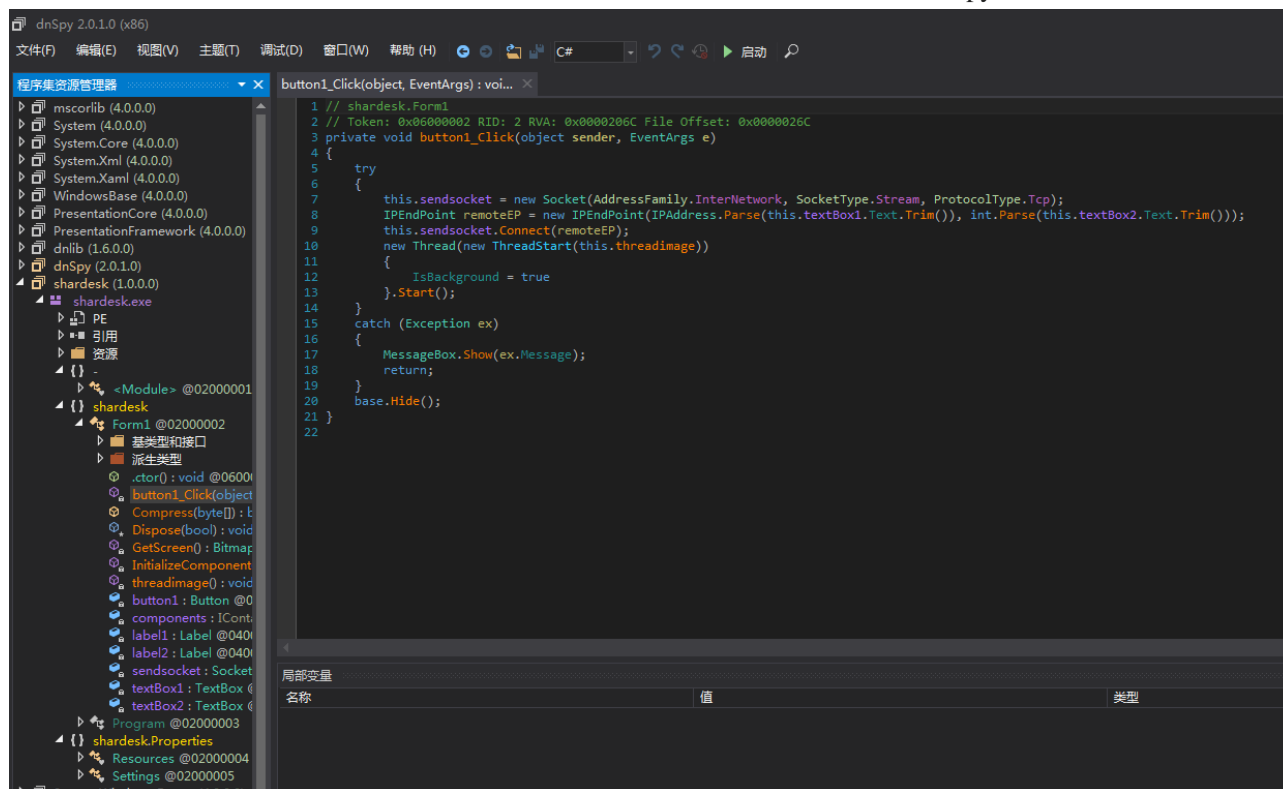


图 61：dnspy 反编译结果

如果程序没有被混淆，则反编译后会看到程序的 C#源码。可以进行静态和动态分析。

如果有混淆, 混淆方法具有多样性, 有些混淆可以通过反混淆工具 de4dot 还原出源码, 从而进行源码分析。

C#程序是由.net framework 提供的库解释执行的, 执行的汇编代码并不在其加载的代码空间中, 而是在临时的代码空间内部 System_Windows_Forms_ni 内。

参考: <http://blog.csdn.net/ATField/article/details/1750890> .net 的调试方法, C#程序是依赖 MSCROEE.dll 中的 _CorExeMain 函数边解释边执行的。

C#程序调试的结论是, 不论上层是什么语言写的程序, 它都会调用底层的 API 函数实现。所以, 可以通过 API 函数的调用来查看行为。

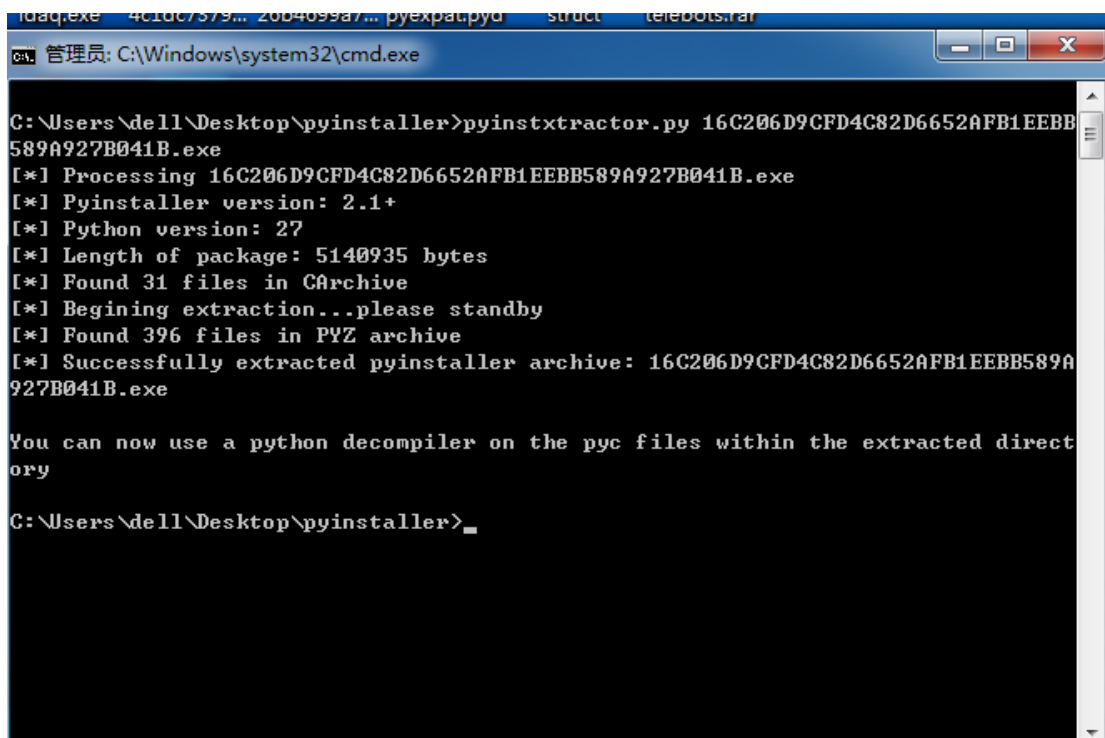
7)Python

PyInstaller 文件打包成的 exe 程序:

Pyinstaller 将 py 文件打包成可以只生成单独的可执行程序, 支持的版本也多: 2.3 到 2.7 都支持。以及 x64 也支持, 也可以自定义图标。

因此遇到单个的可执行程序一般为 pyinstaller 打包的。

分析 pyinstaller 打包的 exe 程序时, 可以使用 pyinstxtractor.py 将 exe 反编译成 py 文件



```

C:\Users\dell\Desktop\pyinstaller>pyinstxtractor.py 16C206D9CFD4C82D6652AFB1EEBB589A927B041B.exe
[*] Processing 16C206D9CFD4C82D6652AFB1EEBB589A927B041B.exe
[*] Pyinstaller version: 2.1+
[*] Python version: 27
[*] Length of package: 5140935 bytes
[*] Found 31 files in CArchive
[*] Beginning extraction...please standby
[*] Found 396 files in PYZ archive
[*] Successfully extracted pyinstaller archive: 16C206D9CFD4C82D6652AFB1EEBB589A927B041B.exe

You can now use a python decompiler on the pyc files within the extracted directory

C:\Users\dell\Desktop\pyinstaller>_
  
```

然后直接对其源码进行分析即可。

Py 文件可能有其他的保护措施, 比如动态解密后执行

```

1 cipher = AES.new(key, AES.MODE_CBC, iv)
2 return base64.b64encode(iv + cipher.encrypt(raw))
3
4 def decrypt(self, enc):
5     enc = base64.b64decode(enc)
6     iv = enc[:AES.block_size]
7     cipher = AES.new(self.key, AES.MODE_CBC, iv)
8     return self._unpad(cipher.decrypt(enc[AES.block_size:])).decode('utf-8')
9
10 def _pad(self, s):
11     return s + (self.bs - len(s) % self.bs) * chr(self.bs - len(s) % self.bs)
12
13 @staticmethod
14 def _unpad(s):
15     edit = s[:-ord(s[len(s)-1:])]
16     sleep(randrange(15,26))
17     return edit
18
19 B="buHeNR7GdvFaPSTqYkFOCKFv"
20 u = ""
21 eNoVmcVyrFAURT-IAR5gILs7M9zd-frXrYrDdNHce87eayU3k5TLn5_0ANXtYcROXu6CkqCp9wapp5IozCH2hCDJCcAAG3wgIfTicfB8RyG1z1fZMFHOicTJRp37x8bfhyTO_70zFihG1JUH
22 ROzvPcwa6at3ixRVMRZrPC5Ux7x65m7gd8vntDf-dpRb0QrPxpIVnfUw1TZfueZB5SMdwC1Mn9H8gBWHf72jduXwK08A3164hvn1YNK69ohFr9bHvTfVYr8xovD7rJbZ_5dwJXVyFgF3eHXGQsmf
23 vqMkcyuhFkRVNxdCHDUKzi353PhX-zqyZwawmFHTKub7L1P1Gx89Ze5H7YV5ejHNZIAMrKZog-SuznRjgMbJg487cKYHm7g464huS_z1hNEslwJON5cCaH6ehYLJf5g508oQD11GVXYb5ymMovuJ0oB
24 UpmMpE1VOJg1HYnUtEzeFTZbaTt73Y-s_mDx5Gw7UJIn_ntCvkRE_SnBxPggMiKgjckgsq8RMcIa4-So3Iw7C0yqisZSUSd50WUxeETPaOxtmTKpCzOCpFqW7FZtQ1Ne38nTxRjvOUGszZD57N2mSIY
25 BV7tCphvK9SdrnZzqz3HOTj1a-m6vWuH0aGoxIQohD1HxdtXaD32sIb5K6ue78SAQNj1zKwWBMto7h5aMwY8uE3UA2kkSwsAusF08vniSxtBy0JO-y8B3I25pBg96F3uYI6mqmVkcNOQdtU2q851InJ
26 dyptbnfwCjVZDyxEt80g_BjK8XqIqRugeH4hM53YtT6EtfHFZ2DkXkH-nNRvOu41051CxcplZc4ktaa8PQvoo6Q5HFLdC3ge50Jbc0ZbQ8UWrp2tegtZdAtP_xkw6ZENyUPmPugktGxg84sE0DTJ
27 JyPYmkVL3P86XELDPHb7zBoqSLi4SgyvOkvbZgp5e6Z0Pz8_D7IoC2xV52eL1LBQ8Z58mC1zkh1Tz1cnohUvYrOAMJ3-3VeoY9Gkzwl8LFYbhrQ891b_nYUmv5F6uXEVaLc8agkJNde7UExyHs
28 FYJLHC9AEQvObbWIR196AMID0T6CNI5NhdW-in6KqH5FpH1Yah_tsrJrvwH09a5-YpcvOPI5EZgcaybia-pE_UAab9jIYvZ5ym-k01xlUmvnainzAl-jlAbJFcltceCWN61Zn8wIezjs9SwCA01U
29 Is-4gfj7YIryakt1EK-BEG0s88G0zUQIehgkRde9_rJoCjYfSt-gfmaNk8ACD3bc1qwc7TpacbfckLxI_msUH43m8DrIQZ0Vb1WYbqR-w8_nbmfy7fNcZM2ghVfHm4uhdoEv9T9LDr8_KHw4s6j5u
30 YC-HZLN803NRKc3Tos4j9Qn6j0MTws_baUw1Ng85uAznXf5c3zQAr9o5rybsVFKLB08K19bcldYF-9zEwDH2K6gTWsyI_23hfryANFOPd6hF1Wof8UOM-Oyr1FvWpWC5jdxRwxt0U_mPhuMe30f
31
32 NNN = AESCipher("RVX0Wkdzfw4ynICDqJL9VjUjlyehv7")
33 u = NNN.decrypt(zlib.decompress(base64.urlsafe_b64decode(u))).encode("rot13")
34
35 exec u

```

可以将解密后的数据保存到文件对其进行分析，代码可能对变量名和函数名进行了混淆，这时候需要边分析边修改变量名和函数名

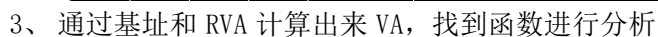
8)dll

使用 OD 调试 dll，需要保证 OD 目录下有 loaddll.exe 文件。

UDD	2017/1/5 17:49	文件夹	
原版	2016/7/27 20:38	文件夹	
DBGHELP.DLL	2008/9/9 6:02	应用程序扩展	1,045
LoadDll.dll	2008/4/21 10:26	应用程序扩展	3
LoadDll.exe	2008/4/21 9:59	应用程序	6
OlllyDBG.EXE	2015/1/12 10:00	应用程序	1,264
OLLYDBG.HLP	2004/12/24 19:55	帮助文件	354
ollydbg.ini	2017/1/5 17:50	配置设置	13
ollydbg_白底黑字配置.ini	2016/1/21 16:13	配置设置	12
Udd Cleaner.exe	2015/1/12 10:02	应用程序	665

调试步骤：

- 1、先借助 LordPE 获取输出表的方法获取 RVA 地址



具体分析调试技巧

将样本拖入 IDA 中，查看导入表和字符串，通过查看这两项，有时可以发现一些敏感的函数或者字符串，一般情况下可以大概知道样本的行为了。比如看到 `CreateThread`，可以知道样本有创建线程操作，在动态调试的时候关注一下这个函数，查看创建的线程并且同样需要对创建的线程进行调试分析。

如果样本创建了许多线程，这样分析起来会有点乱，可以在样本调用 `CreateThread` 的时候，让它直接返回，这样的话，就可以只分析一个线程，使用同样的方法逐个分析。

在关键函数地方下断点，可以快速定位样本的行为。比如看到一个 `exe` 文件的导入表里的函数很少，一是因为样本有壳，二是因为样本是动态获得函数。如果是动态获得函数，可以在 `GetProcAddress` 中下断点，可以快速知道样本都使用了哪些函数。

下断点的方式：

内存访问断点，在关键数据内存中下断点可以知道样本程序在什么地方什么时候用到了跟踪的数据。

内存写入断点：可以用来跟踪一个关键数据是在什么时候生成的，有时样本会开辟一块内存，写入数据，然后从这块内存中继续执行。所以下一个内存写入断点可以知道什么时候写的，写的数据是什么，继而可以将此块内存 `dump` 出来进行进一步的分析。

修改 eip：通过修改 `eip` 可以修改程序的执行流程。

修改函数返回值：有的函数调用中有反调试或者联网失败之类的行为，样本根据此类函数的返回值走到不同的程序中。如果返回值不对，样本可能直接就退出程序。所以可以通过修改返回值，来让程序走到我们想要的地址中。

修改内存：有的样本会通过判断某块内存中的数据来决定走哪个分支语句，但是有时因为网址已经失效导致内存中没有数据就可能直接导致样本直接退出。此时就可以在内存中写入一些数据，让样本继续走正常流程。

常用的自启动方式：

[1] 注册表

`Software\Microsoft\Windows\Current Version\Run`，这个键下的所有程序在系统每次启动的时候都会按顺序自动执行

`Software\Microsoft\Windows\Current Version\RunOnce`，只会被执行一次

`Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run`，这个键一般不被注意，但是也要查看一下

`Software\Microsoft\Windows\CurrentVersion\RunServicesOnce`，此键中的程序会在系统加载时自动启动一次

`Software\Microsoft\Windows\CurrentVersion\RunServices`，`RunServices` 是继 `RunServiceOnce` 之后启动的程序

`SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx`，此键是 WindowsXP/2003 特有的自启动注册表项

`Software\Microsoft\WindowsNT\CurrentVersion\Windows`，其下的 `load` 键值的程序也可以自启动

`SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon`，下面的 `Notify`，`Userinit`，`Shell` 键值也会有自启动的程序，而且其键值可以用逗号分隔，从而实现登录的时候启动多个程序

以上的注册表路径都包含 HKEY_CURRENT_USER 和 HKEY_LOCAL_MACHINE，不过 HKEY_LOCAL_MACHINE 对所有用户有效，而 HKEY_CURRENT_USER 只对当前用户有效。

一些不常用但是也可以实现自启动的注册表值：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\System\Shell
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\ShellServiceObjectDelayLoad
HKEY_CURRENT_USER\Software\Policies\Microsoft\Windows\System\Scripts
HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\System\Scripts
```

[2] 任务

C:\WINDOWS\System32\Tasks

在该目录下创建自启动文件，由 taskeng.exe 启动执行

[3] 系统配置文件

在 Windows 的配置文件（包括 Win.ini、System.ini 和 wininit.ini 文件）也会加载一些自动运行的程序。

[4] 自动批处理文件

位于系统盘根目录下的 autoexec.bat 也可以在电脑启动时自动运行

多线程调试：

如果进程创建了多个线程，可以在 CreateThread 函数下断点，当进程再次调用此函数时，修改此函数代码，让它直接返回，就不会再创建其他的线程，这样就可以只调试一个线程。

调试子进程：

如果父进程以 suspend 的方式创建子进程，那么在 ResumeThread 前附加到子进程，通过 OD 激活附加的线程进行调试。

恶意代码分析通用规则：

在病毒分析的过程中，不要过于陷入细节。大多数病毒木马程序是庞大而复杂的，我们不可能也没有必要去了解每一个细节。真正需要关注的是恶意程序最关键最主要的功能。在实际分析过程中，当遇到了一些困难和复杂的代码段后，应当在进入到细节之前有一个概要性的了解。

另外，在面对不同的工作任务中，应当善于使用不同的工具和方法。不存在通吃的做法。如果在一个点上被卡住了，不要花太长的时间在这个问题上，尝试转移到其他问题，尝试从不同的角度进行分析。

最后，恶意代码分析技术与恶意代码编写技术是对立统一的，二者都在不断地相互促进并发展着。所以我们应当能够认识、理解和战胜这些不断涌现的新技术，并能够快速适应这个领域中的新的变化。

反调试技术总结

1. FindWindow

使用此函数查找目标窗口，如果找到，可以禁用窗口，也可以直接退出程序

函数声明：

```
HWND FindWindow  
(  
LPCSTR lpClassName,  
LPCSTR lpWindowName  
);
```

参数表：

lpClassName：指向一个以 NULL 字符结尾的、用来指定类名的字符串或一个可以确定类名字符串的原子。如果这个参数是一个原子，那么它必须是一个在调用此函数前已经通过 GlobalAddAtom 函数创建好的全局原子。这个原子（一个 16bit 的值），必须被放置在 lpClassName 的低位字节中，lpClassName 的高位字节置零。

如果该参数为 null 时，将会寻找任何与 lpWindowName 参数匹配的窗口

lpWindowName：指向一个以 NULL 字符结尾的、用来指定窗口名（即窗口标题）的字符串。如果此参数为 NULL，则匹配所有窗口名。

```
void CAntiDebugDlg::OnBnClickedBtnFindwindow()  
{  
    HWND Hwnd = NULL;  
    Hwnd = ::FindWindow(L"OllyDbg", NULL);  
    if (Hwnd == NULL) {  
        MessageBoxW(L"Not Being Debugged!");  
    }  
    else {  
        MessageBoxW(L"Being Debugged!");  
    }  
}
```

2. 枚举窗口

使用 EnumWindow 函数枚举窗口，并且为每一窗口调用一次回调函数，在回调函数中可以调用 GetWindowText 获取窗口的标题。与目标窗口名进行比对，如果比对成功，则说明发现调试器。

函数声明：

```
WINUSERAPI  
BOOL
```

WINAPI

EnumWindows (

 In WNDENUMPROC lpEnumFunc,

 In LPARAM lParam

);

参数表:

lpEnumFunc: 回调函数指针。

lParam: 指定一个传递给回调函数的应用程序定义值。

回调函数原型:

BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);

参数表:

Hwnd: 顶层窗口的句柄。

Lparam: 应用程序定义的一个值 (即 EnumWindows 中的 lParam)。

Int GetWindowText (HWND hWnd, LPTSTR lpString, Int nMaxCount);

参数表:

hWnd: 带文本的窗口或控件的句柄。

lpString: 指向接收文本的缓冲区的指针。

nMaxCount: 指定要保存在缓冲区内的字符的最大个数, 其中包含 NULL 字符。如果文本超过界限, 它就被截断。

BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam)

```
{
    WCHAR wzChar[100] = { 0 };
    CStringW strData = L"OllyDbg";
    if (IsWindowVisible(hwnd)) {
        GetWindowText(hwnd, wzChar, 100);
        if (wcsstr(wzChar, strData)) {
            MessageBoxW(NULL, L"Being Debugged!", NULL, 0);
            g_bDebugged = TRUE;
            return FALSE;
        }
    }
    return TRUE;
}
```

void CAntiDebugDlg::OnBnClickedBtnEnumwindow ()

```
{
    EnumWindows(EnumWindowsProc, NULL);
    if (g_bDebugged == FALSE) {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

3. 枚举进程

枚举进程列表，查看是否有调试器进程

函数声明：

```
HANDLE WINAPI CreateToolhelp32Snapshot(
    DWORD dwFlags,
    DWORD th32ProcessID
);
```

通过获取进程信息为指定的进程、进程使用的堆[HEAP]、模块[MODULE]、线程建立一个快照。

参数表：

dwFlags：用来指定“快照”中需要返回的对象，可以是 TH32CS_SNAPPROCESS 等

th32ProcessID：一个进程 ID 号，用来指定要获取哪一个进程的快照，当获取系统进程列表或获取当前进程快照时可以设为 0

BOOL

WINAPI

```
Process32FirstW(
    HANDLE hSnapshot,
    LPPROCESSENTRY32W lppe
);
```

process32First 是一个进程获取函数，当我们利用函数 CreateToolhelp32Snapshot() 获得当前运行进程的快照后，我们可以利用 process32First 函数来获得第一个进程的句柄。

BOOL

WINAPI

```
Process32NextW(
    HANDLE hSnapshot,
    LPPROCESSENTRY32W lppe
);
```

Process32Next 是一个进程获取函数，当我们利用函数 CreateToolhelp32Snapshot() 获得当前运行进程的快照后，我们可以利用 Process32Next 函数来获得下一个进程的句柄。

```
void CAntiDebugDlg::OnBnClickedBtnEnumprocess()
```

```
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE hwnd = NULL;
    PROCESSENTRY32W pe32 = { 0 };
    pe32.dwSize = sizeof(pe32); //如果没有这句，得出的路径不对
    WCHAR str[] = L"OLLYDBG";
    CStringW strTemp;
    BOOL bOK = FALSE;
    hwnd = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
    if (hwnd != INVALID_HANDLE_VALUE) {
        bool bMore = Process32FirstW(hwnd, &pe32);
        do {
```

```

    strTemp = pe32.szExeFile;
    //统一转换为大写进行比较
    strTemp.MakeUpper();
    if (wcsstr(strTemp, str)) {
        MessageBoxW(L"Being Debugged!");
        bOK = TRUE;
        break;
    }
    else if (wcsstr(pe32.szExeFile, L"WINDBG")) {
        MessageBoxW(L"Being Debugged!");
        bOK = TRUE;
        break;
    }
} while (Process32NextW(hwnd, &pe32));
}
if (bOK == FALSE) {
    MessageBoxW(L"Not Being Debugged!");
}
CloseHandle(hwnd);
}

```

4. 查看父进程是不是 Explorer

当我们双击运行应用程序的时候，父进程都是 Explorer，如果是通过调试器启动的，父进程就不是 Explorer。

通过 GetCurrentProcessId() 获得当前进程的 ID

通过桌面窗口类和名称获得 Explorer 进程的 ID

使用 Process32First/Next() 函数枚举进程列表，通过 PROCESSENTRY32.th32ParentProcessID 获得的当前进程的父进程 ID 与 Explorer 的 ID 进程比对。如果不一样的很可能被调试器附加

函数声明：

```

DWORD GetWindowThreadProcessId(
    HWND hwnd,
    LPDWORD lpdwProcessId
);

```

找出某个窗口的创建者（线程或进程），返回创建者的标志符。

参数说明：

hwnd：（向函数提供的）被查找窗口的句柄。

lpdwProcessId：进程号的存放地址（变量地址）

源码展示：

```

void CAntiDebugDlg::OnBnClickedBtnExplorer()
{
    // TODO: 在此添加控件通知处理程序代码

```

```

HANDLE hwnd = NULL;
HANDLE hexplorer = NULL;
PROCESSENTRY32 pe32 = { 0 };
pe32.dwSize = sizeof(pe32);
CStringW str = L"explorer";
DWORD ExplorerId = 0;
DWORD SelfId = 0;
DWORD SelfParentId = 0;
SelfId = GetCurrentProcessId();
hexplorer = ::FindWindowW(L"Progman", NULL);
GetWindowThreadProcessId((HWND)hexplorer, &ExplorerId);
hwnd = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
if (hwnd != INVALID_HANDLE_VALUE)
{
    Process32FirstW(hwnd, &pe32);
    do
    {
        if (SelfId == pe32.th32ProcessID)
        {
            SelfParentId = pe32.th32ParentProcessID;
        }
    } while (Process32NextW(hwnd, &pe32));
}
if (ExplorerId == SelfParentId) {
    MessageBoxW(L"Not Being Debugged!");
}
else
{
    MessageBoxW(L"Being Debugged!");
}
CloseHandle(hwnd);
}

```

5. 检测系统时钟

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。

GetTickCount 返回从操作系统启动所经过的毫秒数。调用两次此函数，如果返回值的差值相差反常，就说明在调试状态。

```

void CAntiDebugDlg::OnBnClickedBtnGettickcount()
{
    // TODO: 在此添加控件通知处理程序代码

```

```
DWORD dwTime1 = 0;
DWORD dwTime2 = 0;
dwTime1 = GetTickCount();
GetCurrentProcessId();
GetCurrentProcessId();
GetCurrentProcessId();
dwTime2 = GetTickCount();
if (dwTime2 - dwTime1 > 100)
{
    MessageBoxW(L"Being Debugged!");
}
else {
    MessageBoxW(L"Not Being Debugged!");
}
}
```

6. 查看 StartupInfo 结构

在 windows 操作系统中, Explorer 创建进程的时候会把 STARTUPINFO 结构中的某些值设为 0, 非 Explorer 创建进程的时候会忽略这个结构中的值, 所以可以通过查看这个结构中的值是不是为 0 来判断是否在调试状态。

函数说明:

WINBASEAPI

VOID

WINAPI

GetStartupInfoW(

 Out LPSTARTUPINFOW lpStartupInfo
);

取得进程在启动时被指定的 STARTUPINFO 结构

typedef struct _STARTUPINFOW {

 DWORD cb;

 LPWSTR lpReserved;

 LPWSTR lpDesktop;

 LPWSTR lpTitle;

 DWORD dwX;

 DWORD dwY;

 DWORD dwXSize;

 DWORD dwYSize;

 DWORD dwXCountChars;

 DWORD dwYCountChars;

 DWORD dwFillAttribute;

 DWORD dwFlags;

```
WORD    wShowWindow;
WORD    cbReserved2;
LPBYTE  lpReserved2;
HANDLE  hStdInput;
HANDLE  hStdOutput;
HANDLE  hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

源码展示:

```
void CAntiDebugDlg::OnBnClickedBtnStartupinfor()
{
    // TODO: 在此添加控件通知处理程序代码
    STARTUPINFO info = { 0 };
    GetStartupInfo(&info);
    if (info.dwX != 0 || info.dwY != 0 || info.dwXCountChars != 0 || info.dwYCountChars !=
0
        || info.dwFillAttribute != 0 || info.dwXSize != 0 || info.dwYSize != 0)
    {
        MessageBoxW(L"Being Debugged!");
    }
    else {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

7 . BeingDebugged, NTGlobalFlag

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。通过 fs:[30]可以获得 PEB 地址。然后通过不同的偏移访问不同的值

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)
```

8. CheckRemoteDebuggerPresent

函数声明:

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


此函数用来确定是否有调试器附加到进程。

参数表:

hProcess: 进程句柄

pbDebuggerPresent: 指向一个 BOOL 的变量, 如果进程被调试, 此变量被赋值为 TRUE。

源码展示:

```
typedef BOOL(WINAPI *CHECK_REMOTE_DEBUGGER_PRESENT)(HANDLE, PBOOL);
void CAntiDebugDlg::OnBnClickedBtnCheckremote()
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE hProcess;
    HINSTANCE hModule;
    BOOL bDebuggerPresent = FALSE;
    CHECK_REMOTE_DEBUGGER_PRESENT CheckRemoteDebuggerPresent;
    hModule = GetModuleHandleA("Kernel32");
    CheckRemoteDebuggerPresent =
        (CHECK_REMOTE_DEBUGGER_PRESENT) GetProcAddress(hModule,
"CheckRemoteDebuggerPresent");
    hProcess = GetCurrentProcess();
    CheckRemoteDebuggerPresent(hProcess, &bDebuggerPresent);
    if (bDebuggerPresent == TRUE)
    {
        MessageBoxW(L"Being Debugged!");
    }
    else
    {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

9. NtQueryInformationProcess

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 EPROCESS 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。

函数声明:

```
NTSTATUS WINAPI NtQueryInformationProcess(
    _In_ HANDLE ProcessHandle,
    _In_ PROCESSINFOCLASS ProcessInformationClass,
    _Out_ PVOID ProcessInformation,
    _In_ ULONG ProcessInformationLength,
```

```

_Out_opt_ PULONG ReturnLength
);

```

参数表:

ProcessHandle: 进程句柄

ProcessInformationClass: 信息类型

ProcessInformation: 缓冲指针

ProcessInformationLength: 以字节为单位的缓冲大小

ReturnLength: 写入缓冲的字节数

源码展示:

```

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 CAntiDebugDlg::OnBnClickedBtnQueryinforpro()
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE hProcess;
    HINSTANCE hModule;
    DWORD dwResult;
    ZW_QUERY_INFORMATION_PROCESS ZwQueryInformationProcess;
    hModule = GetModuleHandleW(L"ntdll.dll");
    ZwQueryInformationProcess =
(ZW_QUERY_INFORMATION_PROCESS) GetProcAddress(hModule, "ZwQueryInformationProcess");
    hProcess = GetCurrentProcess();
    ZwQueryInformationProcess(
        hProcess,
        ProcessDebugPort,
        &dwResult,
        4,
        NULL);
    if (dwResult != 0)
    {
        MessageBoxW(L"Being Debugged!");
    }
    else
    {
        MessageBoxW(L"Not Being Debugged!");
    }
}

```

10. SetUnhandledExceptionFilter

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。

函数说明：

```
LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter
( _In_ LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
) ;
```

设置异常捕获函数.，当异常没有处理的时候, 系统就会调用 SetUnhandledExceptionFilter 所设置异常处理函数.

参数表：

lpTopLevelExceptionFilter : 函数指针。当异常发生时，且程序不处于调试模式（在 vs 或者别的调试器里运行）则首先调用该函数。

源码展示：

```
static DWORD lpOldHandler;
static DWORD NewEip;
typedef LPTOP_LEVEL_EXCEPTION_FILTER(_stdcall *pSetUnhandledExceptionFilter)(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
pSetUnhandledExceptionFilter lpSetUnhandledExceptionFilter;

LONG WINAPI TopUnhandledExceptionFilter(
    struct _EXCEPTION_POINTERS *ExceptionInfo
)
{
    _asm pushad

    lpSetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)lpOldHandler);
    ExceptionInfo->ContextRecord->Eip = NewEip; //转移到安全位置
    _asm popad
    return EXCEPTION_CONTINUE_EXECUTION;
}

void CAntiDebugDlg::OnBnClickedBtnSetunhandleexcfiler()
{
    // TODO: 在此添加控件通知处理程序代码
    bool isDebugged = 0;
    // TODO: Add your control notification handler code here
    lpSetUnhandledExceptionFilter =
    (pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(L"kernel32.dll"),
        "SetUnhandledExceptionFilter");
```

```
//当异常没有处理的时候，系统就会调用此函数所设置的异常处理函数，此函数返回以前设置的回调函数
lpOldHandler = (DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilter);
_asm { //获取这个安全地址
    call me;
me :
    pop NewEip;
    mov NewEip, offset safe;
    int 3; //触发异常,如果被调试，不会走自己定义的异常处理函数
}
// MessageBoxW(L"Being Debugged!");
isDebugged = 1;
_asm {
safe:
}
if (1 == isDebugged) {
    MessageBoxW(L"Being Debugged!");
}
else {
    MessageBoxW(L"Not Being Debugged!");
}
}
```

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 来判断进程是否被调试。

源码展示：

```
void CAntiDebugDlg::OnBnClickedBtnSedebugpre ()
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE hProcessSnap;
    HANDLE hProcess;
    PROCESSENTRY32 tp32 = { 0 }; //结构体
    tp32.dwSize = sizeof(tp32);
    CString str = L"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)
{
    MessageBoxW(L"Being Debugged!");
}
else
{
    MessageBoxW(L"Not Being Debugged!");
}
CloseHandle(hProcess);
}
} while (Process32Next(hProcessSnap, &tp32));
}
CloseHandle(hProcessSnap);
}

```

12.GuardPages

这个检查是针对 01lyDbg 的，因为它和 01lyDbg 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，01lyDbg 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 01lyDbg 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

源码展示：

```

static bool isDebugged = 1;
LONG WINAPI TopUnhandledExceptionFilter2(
    struct _EXCEPTION_POINTERS *ExceptionInfo
)
{
    _asm pushad

    lpSetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)lpOldHandler);
    ExceptionInfo->ContextRecord->Eip = NewEip;
    isDebugged = 0;
    _asm popad
    return EXCEPTION_CONTINUE_EXECUTION;
}

void CAntiDebugDlg::OnBnClickedBtnGuidpages()
{
    // TODO: 在此添加控件通知处理程序代码
}

```

```
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(L"kernel32.dll"),
        "SetUnhandledExceptionFilter");
lpOldHandler = (DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilter2);

// 分配内存
lpvBase = VirtualAlloc(NULL, dwPageSize, MEM_COMMIT, PAGE_READWRITE);
if (lpvBase == NULL)
    MessageBoxW(L"VirtualAlloc Error");
_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)) {
    MessageBoxW(L"VirtualProtect Error");
}
_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) {
    MessageBoxW(L"Being Debugged!");
}
else {
    MessageBoxW(L"Not Being Debugged!");
}
VirtualFree(lpvBase, dwPageSize, MEM_DECOMMIT);
}
```

13 . 软件断点

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
可以搜索在一般地方下的断点, 或者是在函数中下的断点

源码展示:

普通断点:

```

BOOL DetectBreakPoints()
{
    BOOL bDebugged = FALSE;
    __asm
    {
        jmp CodeEnd;
    CodeStart:
        mov eax, ecx;
        nop;
        push eax;
        push ecx;
        pop ecx;
        pop eax;
    CodeEnd:
        cld;
        mov edi, offset CodeStart;
        mov edx, offset CodeStart;
        mov ecx, offset CodeEnd;
        sub ecx, edx;
        mov al, 0CCH;
        repne scasb;
        jnz NotDebugged;
        mov bDebugged, 1;
    NotDebugged:
    }
    return bDebugged;
}

void CAntiDebugDlg::OnBnClickedBtnBreakpoint ()
{
    // TODO: 在此添加控件通知处理程序代码
    if (DetectBreakPoints())
    {
        MessageBoxW(L"Being Debugged!");
    }
    else

```

```

{
    MessageBoxW(L"Not Being Debugged");
}
}

函数断点:
BOOL DetectFuncBreakpoints()
{
    BOOL bFoundOD;
    bFoundOD = FALSE;
    DWORD dwAddr;
    dwAddr = (DWORD)::GetProcAddress(LoadLibrary(L"user32.dll"), "MessageBoxA");
    __asm
    {
        cld; 检测代码开始
        mov     edi, dwAddr
        mov     ecx, 100; 100bytes
        mov     al, 0CCH
        repne   scasb
        jnz     ODNotFound
        mov     bFoundOD, 1
    ODNotFound:
    }
    return bFoundOD;
}

void CAntiDebugDlg::OnBnClickedBtnFuncbreakpoint()
{
    // TODO: 在此添加控件通知处理程序代码
    if (DetectFuncBreakpoints())
    {
        MessageBoxW(L"Being Debugged!");
    }
    else
    {
        MessageBoxW(L"Not Being Debugged!");
    }
}
}

```

14. 硬件断点

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。

源码展示：

```
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 CAntiDebugDlg::OnBnClickedBtnHdbreakpoint()
{
    // TODO: 在此添加控件通知处理程序代码
    lpSetUnhandledExceptionFilter =
    (pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(L"kernel32.dll"),
        "SetUnhandledExceptionFilter");
    lpOldHandler = (DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilterHBP);
    _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) {
        MessageBoxW(L"Being Debugged!");
    }
    else {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

}

15 . 封锁键盘，鼠标输入

WINUSERAPI

BOOL

WINAPI

BlockInput(
 BOOL fBlockIt);

BlockInput 函数阻塞键盘及鼠标事件到达应用程序。该参数指明函数的目的。如果参数为 TRUE，则鼠标和键盘事件将被阻塞。如果参数为 FALSE，则鼠标和键盘事件不被阻塞。

可以在代码中的关键位置调用此函数。

源码展示：

```
void CAntiDebugDlg::OnBnClickedBtnBlockinput()
{
    // TODO: 在此添加控件通知处理程序代码
    DWORD dwNoUse;
    DWORD dwNoUse2;
    ::BlockInput(TRUE);
    dwNoUse = 2;
    dwNoUse2 = 3;
    dwNoUse = dwNoUse2;
    ::BlockInput(FALSE);
}
```

16.禁用窗口

与 BlockInput 函数的功能类似，用来禁用窗口

函数说明：

BOOL EnableWindow (HWND hWnd, BOOL bEnable)

hWnd：被允许/禁止的窗口句柄

bEnable：定义窗口是被允许，还是被禁止。若该参数为 TRUE，则窗口被允许。若该参数为 FALSE，则窗口被禁止。

Windows API 函数。该函数允许/禁止指定的窗口或控件接受鼠标和键盘的输入，当输入被禁止时，窗口不响应鼠标和按键的输入，输入允许时，窗口接受所有的输入。

源码展示：

```
void CAntiDebugDlg::OnBnClickedBtnEnbalewindow()
{
    // TODO: 在此添加控件通知处理程序代码
    CWnd *wnd;
    wnd = GetForegroundWindow();
    wnd->EnableWindow(FALSE);
}
```

```

DWORD dwNoUse;
DWORD dwNoUse2;
dwNoUse = 2;
dwNoUse2 = 3;
dwNoUse = dwNoUse2;
wnd->EnableWindow(TRUE);
}

```

17.ThreadHideFromDebugger

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。

函数声明:

```

NTSTATUS NTAPI NtSetInformationThread(
    IN HANDLE ThreadHandle,
    IN THREAD_INFORMATION_CLASS ThreadInformationClass,
    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

源码展示:

```

void CAntiDebugDlg::OnBnClickedBtnSetinforthread()
{
    HANDLE hwnd;
    HMODULE hModule;
    hwnd = GetCurrentThread();
    hModule = LoadLibrary(L"ntdll.dll");
    pfnZwSetInformationThread ZwSetInformationThread;
    ZwSetInformationThread = (pfnZwSetInformationThread)GetProcAddress(hModule,
        "ZwSetInformationThread");
    ZwSetInformationThread(hwnd, ThreadHideFromDebugger, NULL, NULL);
}

```

18. OutputDebugString

OutputDebugString 函数用于向调试器发送一个格式化的字符串, 01lydbg 会在底端显示相应的信息。01lydbg 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 01lydbg 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。

源码展示:

```

//能够让OD崩溃

```

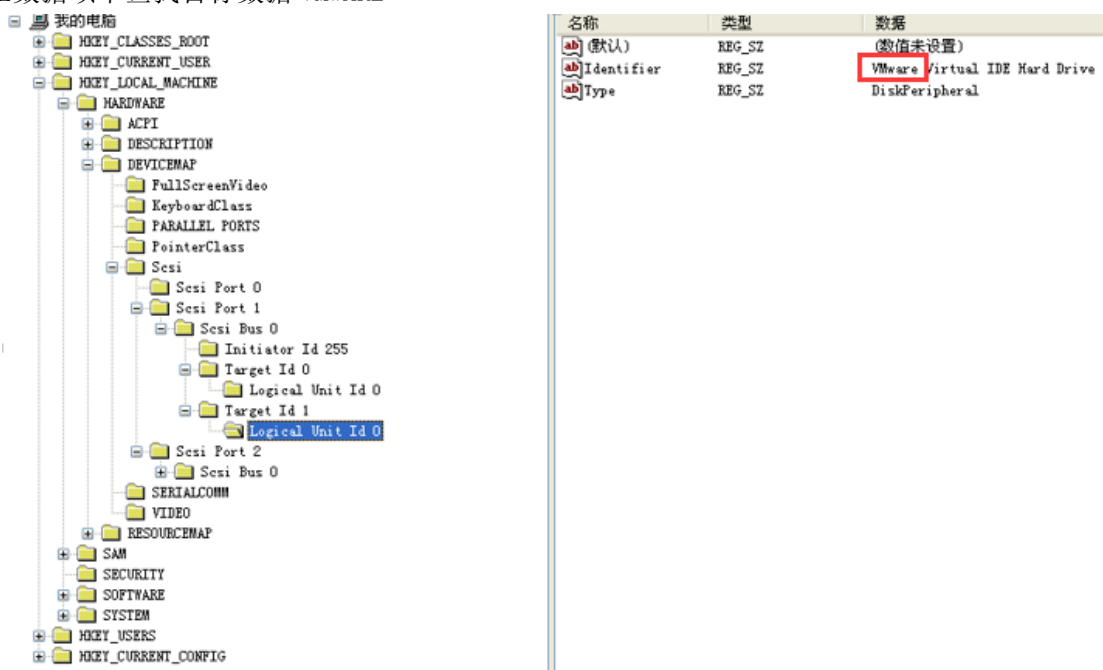
```
void CAntiDebugDlg::OnBnClickedBtnOutputdebugstring()
{
    // TODO: 在此添加控件通知处理程序代码
    ::OutputDebugString(L"%s%s");
}
```

虚拟机检测

1 . VMWare

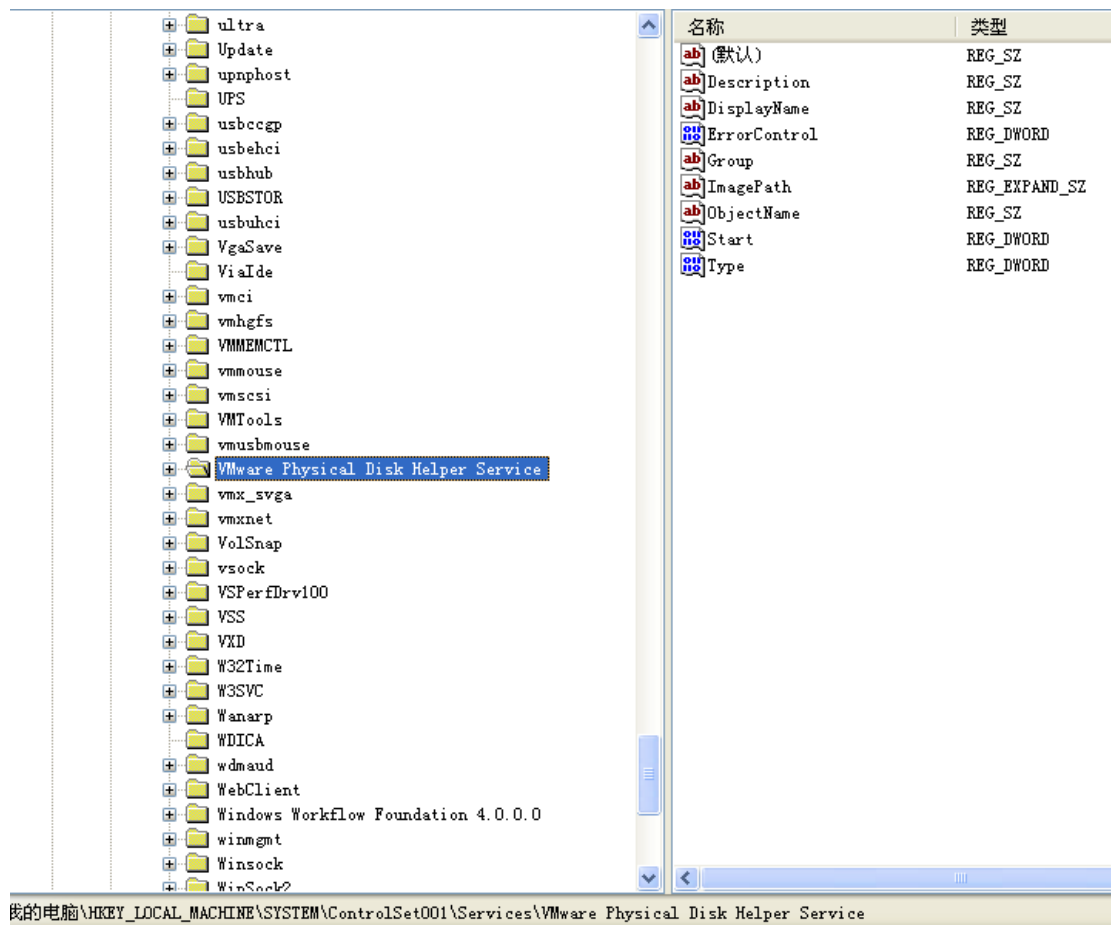
1.1 查找注册表

通过检测注册表“HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VMware Pyhsical Disk Helper Service”，在数据项中查找目标数据 VMWARE



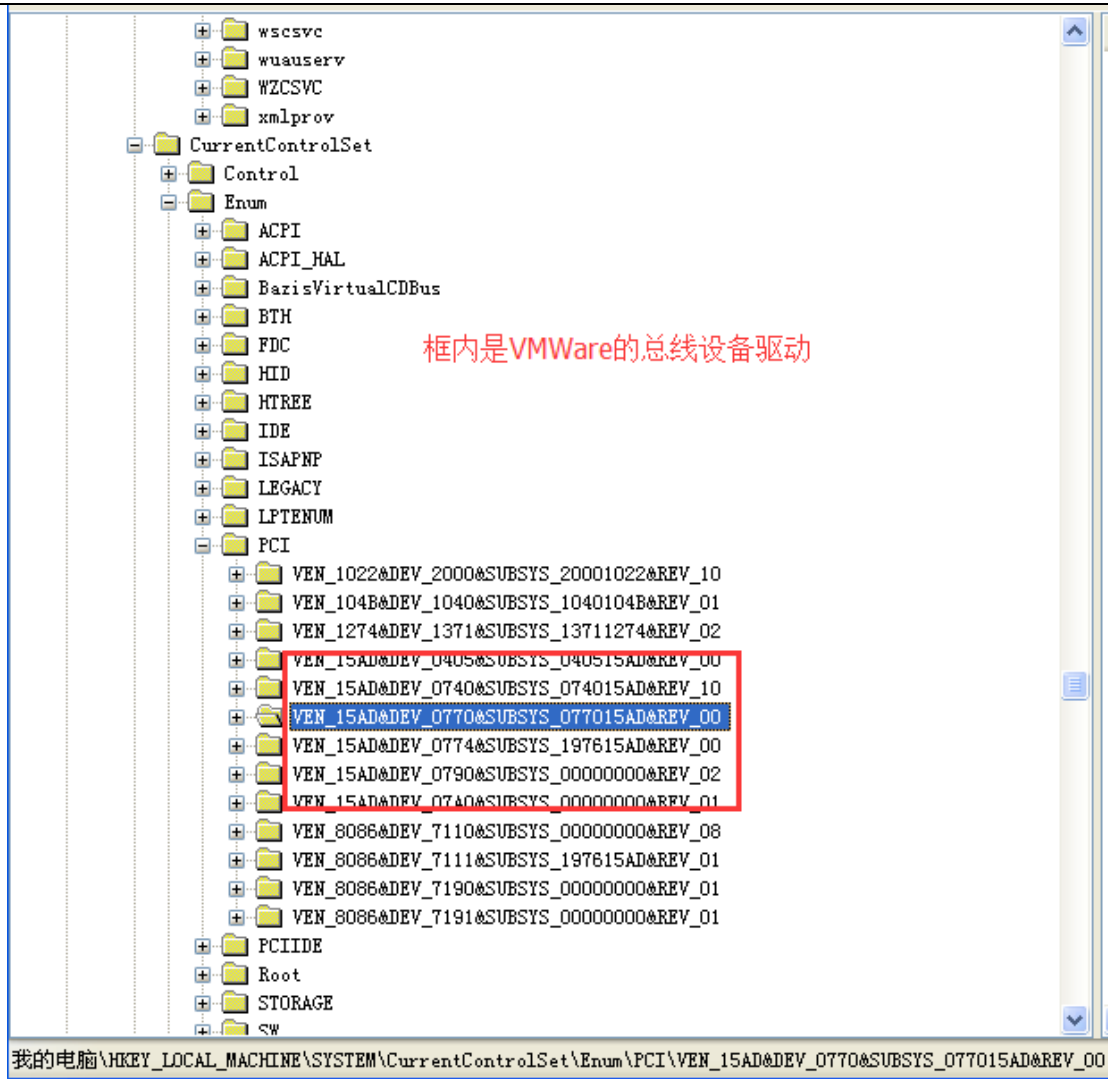
名称	类型	数据
(默认)	REG_SZ	(数值未设置)
Identifier	REG_SZ	Vmware Virtual IDE Hard Drive
Type	REG_SZ	DiskPeripheral

HKEY_LOCAL_MACHINE_SYSTEM_ControlSet001\Services\VMware Pyhsical Disk Helper Service



名称	类型
(默认)	REG_SZ
Description	REG_SZ
DisplayName	REG_SZ
ErrorControl	REG_DWORD
Group	REG_SZ
ImagePath	REG_EXPAND_SZ
ObjectName	REG_SZ
Start	REG_DWORD
Type	REG_DWORD

我的电脑\HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VMware Physical Disk Helper Service



1.2 搜索特定程序确定是否运行在虚拟机中

Vmtoolsd.exe

vmacthlp.exe

VMwareUser.exe

VMwareTray.exe

VMUpgradeHelper.exe

```
bool SearchTargetPro(WCHAR* strProName)
```

```
{
```

```
    PROCESSENTRY32 pe32;
```

```
    pe32.dwSize = sizeof(pe32);
```

```
    HANDLE hProcessSnap = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
```

```
    if (hProcessSnap == INVALID_HANDLE_VALUE)
```

```
    {
```

```
        exit(1);
```

```

}
bool bMore = ::Process32First(hProcessSnap, &pe32);
while (bMore)
{
    if (wcsstr(pe32.szExeFile, strProName))
    {
        return true;
        bMore = false;
    }
    else
    {
        bMore = ::Process32Next(hProcessSnap, &pe32);
    }
}
CloseHandle(hProcessSnap);
return bMore;
}

```

1.3 通过执行特权指令

Vmware 为真主机与虚拟机之间提供了相互沟通的通讯机制，它使用“IN”指令来读取特定端口的数据以进行两机通讯，但由于 IN 指令属于特权指令，在处于保护模式下的真机上执行此指令时，除非权限允许，否则将会触发类型为“EXCEPTION_PRIV_INSTRUCTION”的异常，而在虚拟机中并不会发生异常，在指定功能号 0A（获取 VMware 版本）的情况下，它会在 EBX 中返回其版本号“VMXH”；而当功能号为 0x14 时，可用于获取 VMware 内存大小，当大于 0 时则说明处于虚拟机中。VMDetect 正是利用前一种方法来检测 VMware 的存在，其检测代码分析如下：

```

void IsInsideVMWare()
{
    bool rc = true;
    __try
    {
        __asm
        {
            push    edx
            push    ecx
            push    ebx
            mov     eax, 'VMXH'
            mov     ebx, 0 // 将ebx设置为非幻数'VMXH'的其它值
            mov     ecx, 10 // 指定功能号，用于获取VMWare版本，当它为0x14时用于获取VMWare内存大小
            mov     edx, 'VX' // 端口号
            in      eax, dx // 从端口dx读取VMWare版本到eax
            //若上面指定功能号为0x14时，可通过判断eax中的值是否大于0，若是则说明处于虚拟机中
        }
    }
}

```

```
        cmp     ebx, 'VMXh' // 判断ebx中是否包含VMware版本'VMXh'，若是则在虚拟机中
        setz    [rc] // 设置返回值
        pop     ebx
        pop     ecx
        pop     edx
    }
}

__except (EXCEPTION_EXECUTE_HANDLER) //如果未处于VMware中，则触发此异常
{
    rc = false;
}
if (rc == true)
{
    cout << "In VMWare!" << endl;
}
else
{
    cout << "Not In VMWare!" << endl;
}
}
```

```
void test6(void)
{
    unsigned int a = 0;
    __try {
        __asm {
            // save register values on the stack
            push eax
            push ebx
            push ecx
            push edx
            // perform fingerprint
            mov eax, 'VMXh'
            mov ecx, 14h
            mov dx, 'VX'
            in eax, dx
            mov a, eax
            pop edx
            pop ecx
            pop ebx
            pop eax
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {}
}
```

```
printf("\n[+] Test 6: VMware \"get memory size\" command\n");
if (a > 0)
    printf("Result : VMware detected\n\n");
else
    printf("Result : Native OS\n\n");
}
```

1.4 查找特定的驱动模块

hgfs.sys
vmhgfs.sys
prlETH.sys
prlfs.sys
prlmouse.sys
prlvideo.sys
prl_pv32.sys
vpc-s3.sys
vmsrvc.sys
vmx86.sys
vmnet.sys

1.5 CPUID

原理介绍：当 `eax=1` 时，运行 `CPUID` 之后，`ecx` 的高 31 位可以判断出是否在虚拟机中，如果 `ecx` 的高 31 位为 0 表示在虚拟机下，否则在主机下

参考代码：

```
void CheckCPUID()
{
    DWORD dwECX = 0;
    bool isVM = true;
    __asm {
        pushad;
        pushfd;
        mov eax, 1;
        cpuid;
        mov dwECX, ecx;
        and ecx, 0x80000000;
        test ecx, ecx;
        setz[isVM];
        popfd;
        popad;
    }
}
```

```
if (isVM)//主机下
{
    cout << "在主机下" << endl;
}
else//虚拟机下
{
    cout << "在虚拟机下" << endl;
}
}
```

当 `eax=0x40000000` 时，运行 `CPUID` 之后，`ebx+ecx+edx="VMWareVMWare"`；

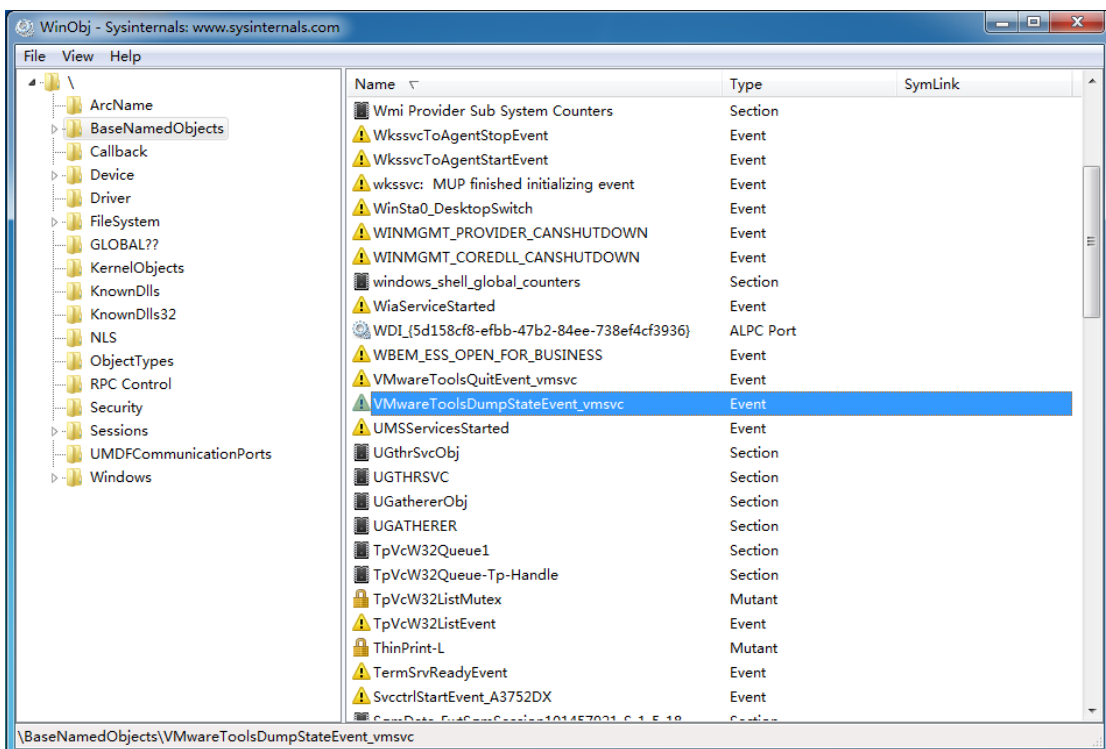
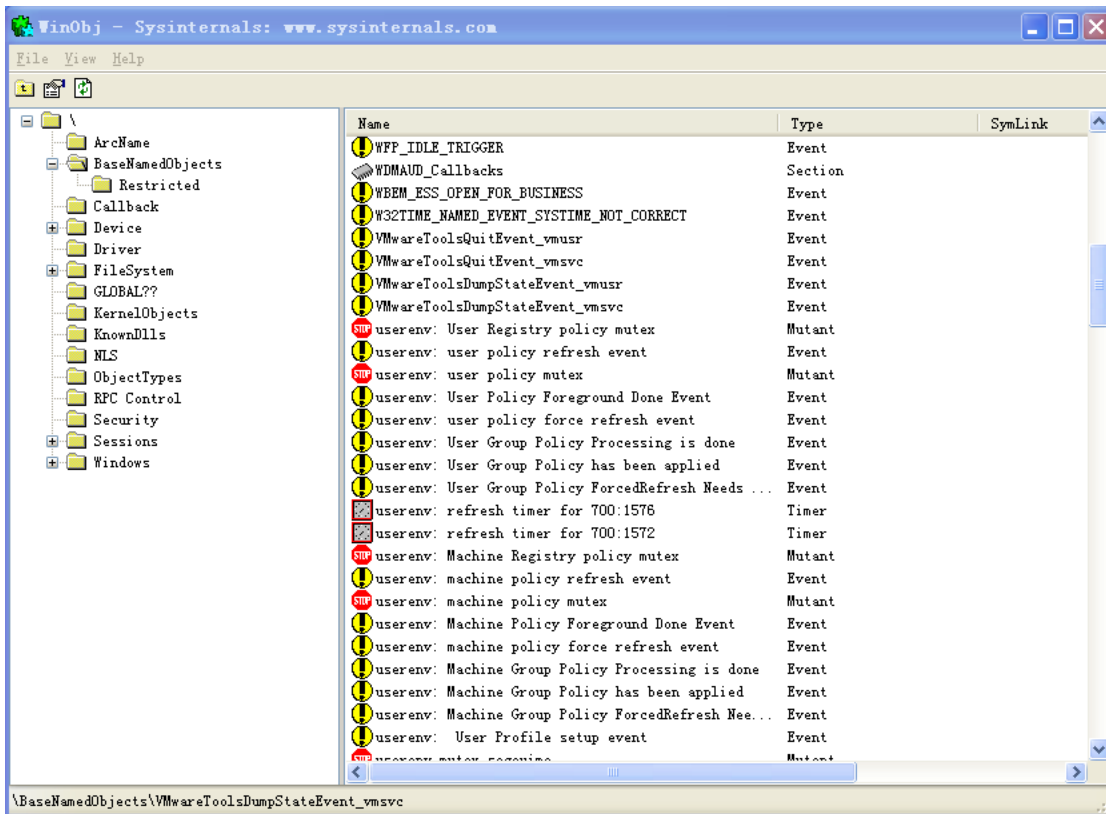
参考代码：

```
void CheckCPUID()
{
    DWORD dwECX = 0;
    bool isVM = true;
    DWORD dwReg[3] = { 0 };
    _asm {
        pushad;
        pushfd;
        mov eax, 0x40000000;
        cpuid;
        mov dword ptr[dwReg], ebx;
        mov dword ptr[dwReg + 4], ecx;
        mov dword ptr[dwReg + 8], edx;

        popfd;
        popad;
    }
    printf("%s\r\n", dwReg);
}
```

1.6 检测目标对象

在 WinXP 和 Win7 中都有 `VMwareToolsQuitEvent_wmsvc` 和 `VMwareToolsDumpStateEvent_vmsvc`，可以通过检测这两项事件对象是否存在来判断是不是运行在虚拟机里



2 . Virtual PC

2.1 使用非法指令:

使用非法指令 0x0F, 0x3F, 0xXX, 0xXX。这些指令在 VirtualPC 上不会产生异常，但是在主机和其他虚拟机中会产生异常。并且需要注册异常处理函数来处理

参考代码:

```
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

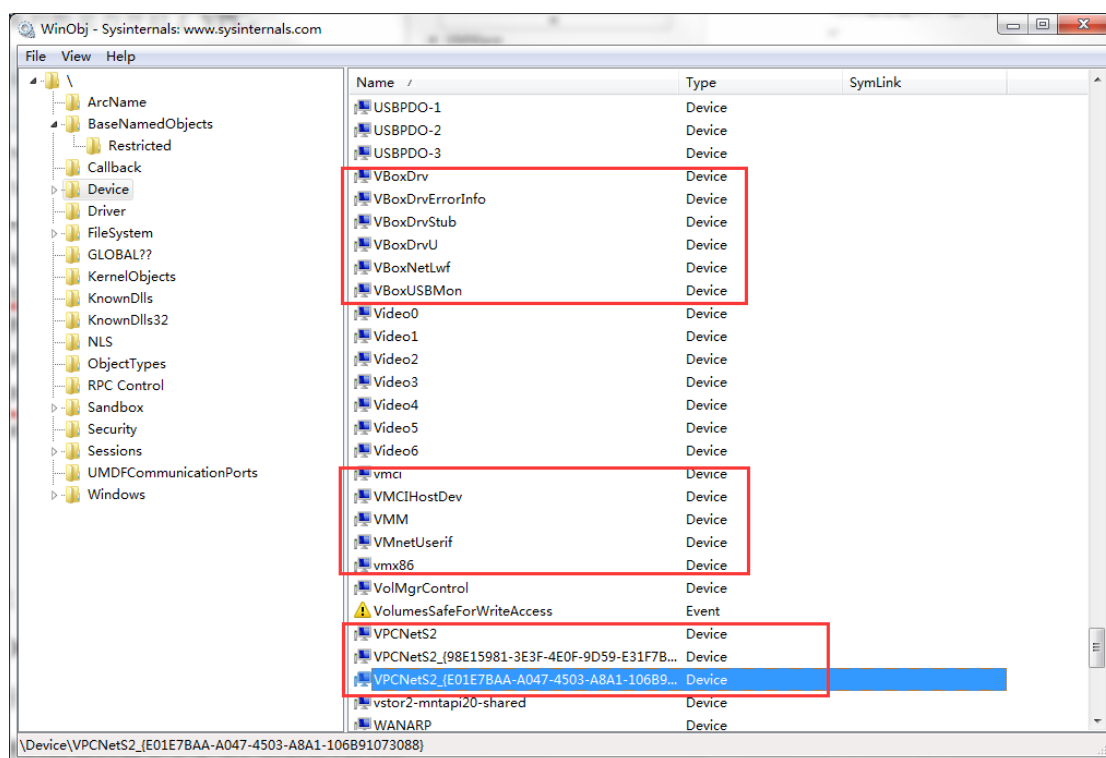
    // lpSetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)lpOldHandler);
    // ExceptionInfo->ContextRecord->Eip = NewEip; //转移到安全位置
    MessageBox(NULL, L"HelloWorld", L"hh", 0);
    _asm popad
    return EXCEPTION_CONTINUE_EXECUTION;
}

void CheckOrder()
{
    // TODO: Add your control notification handler code here
    lpSetUnhandledExceptionFilter =
(pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(L"kernel32.dll"),
        "SetUnhandledExceptionFilter");
    //当异常没有处理的时候，系统就会调用此函数所设置的异常处理函数，此函数返回以前设置的回调函数
    lpOldHandler = (DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilter);
    _asm { //获取这个安全地址
    }
    _asm {
        __emit 0Fh
        __emit 3Fh
        __emit 07h
        __emit 0Bh
    }
}
```

}

2.2 检测设备对象

可以通过查询下图中的对象来判断是不是安装了 VirtualPC（在虚拟机外部检测）。可以检测多种虚拟设备



3 . VirtualBox

3.1 搜索注册表项

搜索特定的注册表项来判断是不是运行在虚拟系统中。

文件(F) 编辑(E) 查看(V) 收藏夹(A) 帮助(H)

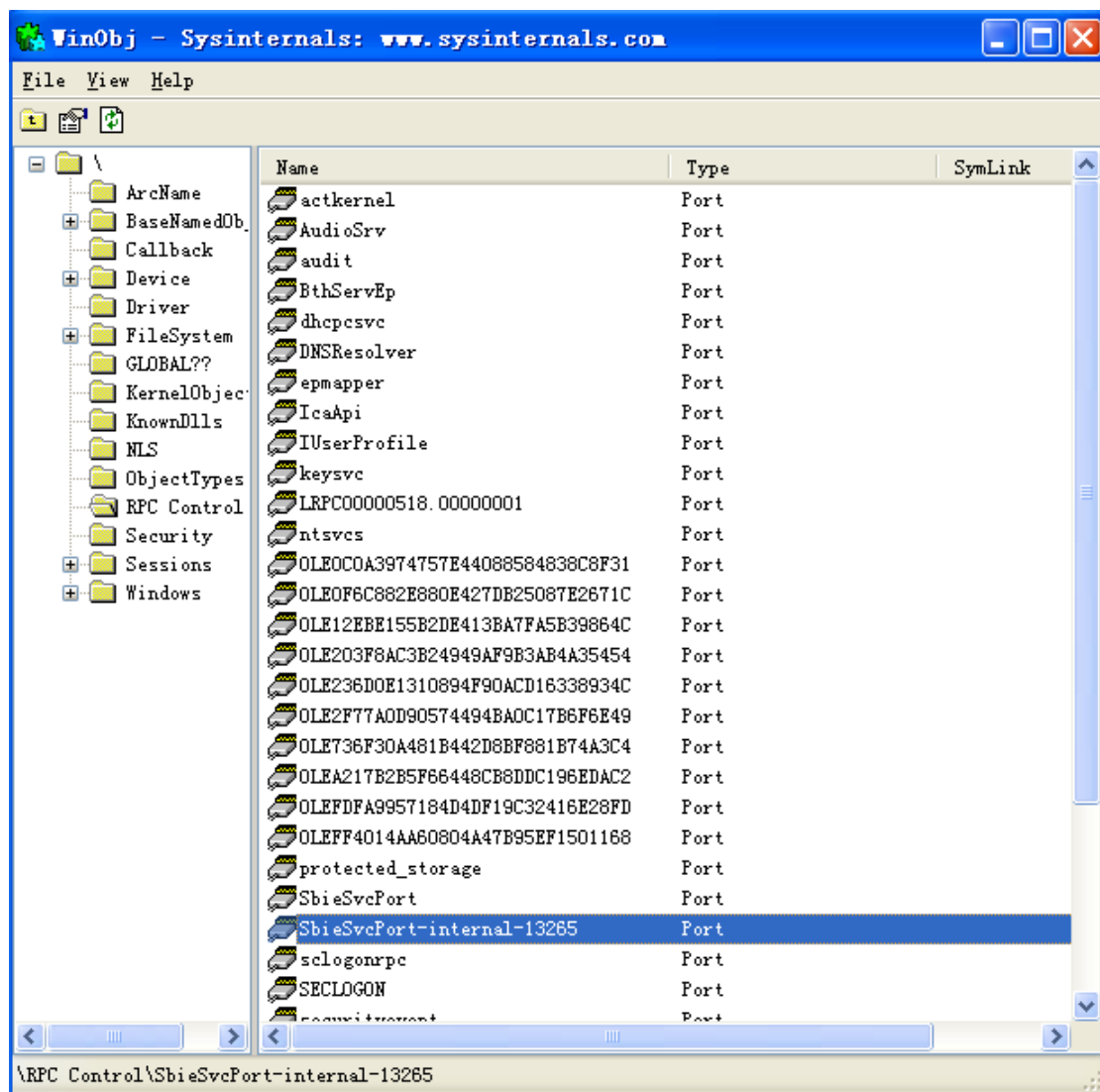
- XML
- xmlfile
- xslfile
- zapfile
- Zone.ClientM
- Zone.ClientM.1
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
 - HARDWARE
 - ACPI
 - DESCRIPTION
 - DEVICEMAP
 - FullScreenVideo
 - KeyboardClass
 - PARALLEL PORTS
 - PointerClass
 - Scsi
 - Scsi Port 0
 - Scsi Bus 0
 - Initiator Id 255
 - Target Id 0
 - Logical Unit Id 0
 - Scsi Port 1
 - VIDEO
 - RESOURCEMAP
 - SAM
 - SECURITY

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
Identifier	REG_SZ	VBOX HARDDISK
Type	REG_SZ	DiskPeripheral

我的电脑\HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0

4 . SandBox

4.1 使用 WinObj 搜索目标对象



但是这种方法只能确定有没有安装 SandBox

4.2 查看当前进程是否包含目标动态库

在当前的进程空间中搜索目标动态库来判断是否运行在沙箱中

Comodo sandbox-----cmdvrt32.dll

Qihoo360 sandbox----SxIn.dll

Sandboxie sandbox-----SbieDll.dll

参考代码:

```
void CheckSbieDll ()
```



```
{
    TCHAR ModuleToFind[MAX_PATH] = L"SbieDll.dll";
    HMODULE moduleHandle[1024];
    DWORD cbNeeded;
    unsigned int i;
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, GetCurrentProcessId());
    if (hProcess == NULL)
    {
        cout << "OpenProcess Error" << endl;
    }
    if (EnumProcessModules(hProcess, moduleHandle, sizeof(moduleHandle), &cbNeeded))
    {
        for (i = 0; i < (cbNeeded / sizeof(HMODULE)); i++)
        {
            TCHAR moduleName[MAX_PATH];

            if (GetModuleFileNameEx(hProcess, moduleHandle[i], moduleName, sizeof(moduleName) /
sizeof(TCHAR))) {
                if (wcsstr(moduleName, ModuleToFind) != nullptr)
                {
                    cout << "In SandBox" << endl;
                    return;
                }
            }
        }
    }
    else
    {
        cout << "EnumProcessModules Error" << endl;
        return;
    }
    cout << "Not In SandBox" << endl;
    return;
}
```

脱壳

保护程序的一种措施是加壳，通过压缩或者加密来组织我们对样本进行分析。那么作为样本分析者，我们首先需要摆脱壳的干扰（可以脱壳，也可以带壳调试，重点在于过壳），找到样本真正的入口点（OEP）来分析样本的功能。

下面介绍一些脱壳的方法：

0x1 单步跟踪法

脱壳的方法有很多，先来讲脱壳方法中最基础的单步跟踪法。单步跟踪法就是利用 OD 的单条指令执行功能，从壳的入口一直执行到 OEP，最终通过这个 OEP 将原程序 dump 出来。然当，在单步跟踪的时候需要跳过一些不能执行到的指令。

使用单步跟踪法追踪 OEP 的常见步骤：

- 1、用 OD 载入待脱壳文件，如果出现压缩提示，选择“不分析代码”；
- 2、向下单步跟踪，实现向下的跳转；
- 3、遇到程序往上跳转的时候（包括循环），在回跳的下一句代码上单击并按键盘上的“F4”键跳过回跳指令；
- 4、OD 中的绿色线条表示跳转没有实现，不必理会，红色线条表示跳转已经实现；
- 5、如果刚载入程序的时候，在附近有一个 CALL 指令，那么就要按键盘上的“F7”键跟进这个 CALL 内，不然程序很容易运行起来；
- 6、在跟踪的时候，如果执行某个 CALL 指令后就运行，一定要按键盘上的“F7”键进入这个 CALL 之内再单步跟踪；
- 7、遇到在 popad 指令下的远转移指令时，要格外注意，因为这个远转移指令的目的地很可能就是 OEP

0x2 ESP 定律法

ESP 定律法是脱壳的利器，是国外友人发现的。有了 ESP 定律，可以方便我们脱掉大多数的压缩壳。可谓是本世纪破解界中最伟大的发现之一。这里只简单的看一下狭义 ESP 定律的原理。

使用 ESP 定律追踪 OEP 的常见步骤：

- 1、将待脱壳程序载入到 OD 中，开始就按键盘上的“F8”键单步跟踪一步，这时如果看到 OD 右边的寄存器窗口中的 ESP 寄存器的值有没有变为红色，如果发现 ESP 寄存器的值变为红色，执行第 2 步；
- 2、在 OD 的命令行窗口中执行命令 `hrXXXXXXXX, xxxxxxxx` 就是变为红色的 ESP 寄存器的值，在输入命令之后，一定不要忘记按键盘上的回车键；
- 3、按键盘上的“F9”键让程序运行起来；
- 4、使用单步跟踪的方法跟踪到 OEP 即可。

0x3 二次断点法

二次断点是有技巧的下两个断点，在两个断点之后就可以很轻松的找到 OEP。

使用二次断点法追踪 OEP 的常见步骤：

- 1、将待脱壳程序载入到 OD 中，单击 OD 的“选项”菜单下的“调试设置”命令，在弹出的“调试选项”对话框中切换到“异常”选项卡，勾选该选项卡下的所有复选框，也就是忽略所有异常；
- 2、按键盘上的“ALT+M”组合键打开 OD 的内存窗口；
- 3、在 OD 的内存窗口中找到“.rsrc" 区段，单击该区段后按键盘上的“F2”键在该区段上下一断点；
- 4、按“Shift+F9”让程序运行到断点处，而后再次打开 OD 的内存窗口，这次在“.rsrc" 区段上面的“.code" 区段（有的时候就是“.text"）上下一个断点；
- 5、按“shift+F9”让程序运行到第二次下的断点处，然后单步跟踪既可以来到 OEP。

0x4 末次异常法

在脱壳方法中，末次异常法又被称为最后一次异常法，这是最基础的脱壳技术之一。末次异常法脱壳很是简单，但就是这简单的脱壳方法可以挑掉曾经风靡一时的强壳。

使用末次异常法追踪 OEP 的常见步骤：

- 1、将待脱壳程序载入到 OD 中，单击 OD 的“选项”菜单，在弹出的菜单中单击“调试设置”命令，在随后弹出的“调试选项”对话框中切换到“异常”选项卡，清除该选项卡下所有复选框，也就是不忽略任何异常；
- 2、连续按键盘上的“Shift+F9”组合键让程序运行起来，记录按键的次数 X；
- 3、回到 OD 中，按键盘上的“Ctrl+F2”组合键重新载入程序，按 X-1 次“Shift+F9”组合键；
- 4、在 OD 右下角窗口中找到“SE 句柄”或是“SE 处理程序”，记录此处的内存地址；
- 5、在 OD 的反汇编窗口中跟随到上一步记录下的内存地址，并在此内存地址处下一个断点；
- 6、按键盘上的“Shift+F9”组合键让程序运行到上一步下的断点处，按键盘上的“F2”键取消此处的断点；
- 7、使用单步跟踪法追踪到 OEP。

0x5 模拟跟踪法

在这章中讲到的众多脱壳方法中，我们首先讲了单步跟踪法脱壳，因为单步跟踪脱壳法是脱壳技术中最基础的方法，在后面其它的一些脱壳方法中总会或多或少的配合单步跟踪法才能顺利完成脱壳工作。便是始终是一次次的按“F8”键来单步跟踪程序，偶尔遇到回跳就跳过执行，这样机械性的操作很是烦人，那么能不能让机器来代替人力，让工具帮我们单步跟踪呢？答案是肯定的，这也就是这节讲的内容——模拟跟踪法。模拟脱壳法就是模拟单步跟踪来进行查找 OEP。

模拟跟踪法的常见步骤：

- 1、将待脱壳程序载入 OD 中，先简单的跟踪一下程序，看看有没有 SEH 暗桩；
- 2、按键盘上的“ALT+F9”打开 OD 的内存窗口，找到“SFX，输入表，资源”的行，并记录此行的内存地址；

3、在 OD 的命令行窗口执行命令“tc eip<上一步中记录下的地址”，命令执行后就会跟踪到 OEP。

0x6 SFX 自动脱壳法

在上一节，我们使用模拟跟踪法代替手动单步跟踪法进行脱壳。在 OD 中，不但可以利用模拟跟踪来代替单步跟踪进行脱壳，从而节省劳动力，还有一种 SFX 自动脱壳的方法也可以节省劳动力，并能快速有效的将程序的壳脱掉。

使用 SFX 自动脱壳法脱壳的常见步骤：

- 1、将 OD 设置为忽略所有异常；
- 2、在 OD 的“调试选项”对话框的“SFX”选项卡中选择“字节模式跟踪实际入口”选项并确定；
- 3、将待脱壳程序载入 OD，待程序载入完成后，会直接停在 OEP 处。

0x7 出口标志法

前面几个脱壳方法中有一个共同点，就是在单步跟踪到 popad 指令后面不远处的 jmp 指令的时候，就可以大胆的判断这个 jmp 指令的目的地址就是 OEP。原因很简单，popad 指令用于将壳运行之前保存的环境恢复，使原程序能正常运行。有些壳的 popad 指令很少，我们就可以查看被这种壳加壳的程序的所有 popad 指令，找到后面存在 jmp 指令的 popad 指令，然后来到其后的 jmp 指令的目的地址，这很可能就是 OEP，然后就可以进行 dump 脱壳了。

使用出口标志法脱壳的常见步骤：

- 1、将待脱壳程序载入 OD 中，在 OD 的反汇编窗口中单击鼠标右键，在弹出的右键菜单中单击“查找”→“所有命令”，在弹出的输入框中输入“popad”并按“查找”按钮；
- 2、逐一尝试跟踪查找到的所有“popad”指令，最终达到程序的 OEP。

0x8 使用脱壳辅助脱壳

在脱壳的时候，使用模拟跟踪法可以让 OD 代替我们单步跟踪程序直到 OEP，这样大大提高了脱壳的效率。但是模拟跟踪法并不能跟踪到一些较强的壳的 OEP，这时我们可以使用高手们写的脱壳脚本来帮助我们完成脱壳工作，使用脱壳脚本来脱壳要比手动跟踪方便得多。脱壳脚本就是高手们为了方便自己或他人脱壳，把自己手动脱壳的步骤记录下来，保存的一个文本文件。虽然脱壳脚本是一个文本文件，可以使用记事本将其打开，但是轻易不要用这种方式修改脱壳脚本，因为直接修改脱壳脚本，很可能造成脱壳脚本不能正确完成对应的脱壳工作。

0x9 使用脱壳工具脱壳

脱壳工具很多，这里只介绍最为实用的全自动脱壳机——超级巡警脱壳工具。

超级巡警脱壳工具的工作方法：

超级巡警脱壳工具会自动侦测待脱壳程序所加的壳头，从而判断出带脱壳程序是用哪种壳程序加壳的。如果超级巡警脱壳工具支持对该壳的脱壳，就可以很方便的将程序的壳脱掉；如果不支持对该壳的脱壳，则会给我们一个简单明了的提示。

大所说新手尽量使用脱壳机脱壳。

总结：

对于加壳程序，我们不一定必须要进行脱壳，主要是找到程序真正的功能代码进行执行，

样本分析 API 集合

0x01 文件类

kernel32.CreateFile

功能：这是一个多功能的函数，可打开或创建以下对象，并返回可访问的句柄：控制台，通信资源，目录（只读打开），磁盘驱动器，文件，邮槽，管道

函数原型：

```
HANDLE WINAPI CreateFile(  
_In_ LPCTSTR lpFileName,  
_In_ DWORD dwDesiredAccess,  
_In_ DWORD dwShareMode,  
_In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
_In_ DWORD dwCreationDisposition,  
_In_ DWORD dwFlagsAndAttributes,  
_In_opt_ HANDLE hTemplateFile  
);
```

参数介绍：

lpFileName 要打开的文件的名或设备名。这个字符串的最大长度在 ANSI 版本中为 MAX_PATH，在 unicode 版本中为 32767。

dwDesiredAccess 指定类型的访问对象。如果为 GENERIC_READ 表示允许对设备进行读访问；如果为 GENERIC_WRITE 表示允许对设备进行写访问（可组合使用）；如果为零，表示只允许获取与一个设备有关的信息

dwShareMode，如果是零表示不共享；如果是 FILE_SHARE_DELETE 表示随后打开操作对象会成功只要删除访问请求；如果是 FILE_SHARE_READ 随后打开操作对象会成功只有请求读访问；如果是 FILE_SHARE_WRITE 随后打开操作对象会成功只有请求写访问。

lpSecurityAttributes，指向一个 SECURITY_ATTRIBUTES 结构的指针，定义了文件的安全特性（如果操作系统支持的话）

dwCreationDisposition，创建配置

dwFlagsAndAttributes，扩展属性

hTemplateFile，hTemplateFile 为一个文件或设备句柄，表示按这个参数给出的句柄为模板创建文件（就是将该句柄文件拷贝到 **lpFileName** 指定的路径，然后再打开）。它将指定该文件的属性扩展到新创建的文件上面，这个参数可用于将某个新文件的属性设置成与现有文件一样，并且这样会忽略 dwAttrsAndFlags。通常这个参数设置为 NULL，为空表示不使用模板，一般为空。

备注:

CreateFileMapping

功能: 创建一个新的文件映射内核对象。

函数原型:

```
HANDLE WINAPI CreateFileMapping(  
    _In_ HANDLE hFile,  
    _In_opt_ LPSECURITY_ATTRIBUTES lpAttributes,  
    _In_ DWORD flProtect,  
    _In_ DWORD dwMaximumSizeHigh,  
    _In_ DWORD dwMaximumSizeLow,  
    _In_opt_ LPCTSTR lpName);
```

参数介绍:

hFile:Long, 指定欲在其中创建映射的一个文件句柄。0xFFFFFFFF (-1, 即 INVALID_HANDLE_VALUE) 表示在页面文件中创建一个可共享的文件映射。

lpFileMappingAttributes:SECURITY_ATTRIBUTES, 它指明返回的句柄是否可以被子进程所继承, 指定一个安全对象, 在创建文件映射时使用。如果为 NULL (用 ByVal As Long 传递零), 表示使用默认安全对象。

flProtect: 保护属性设置

dwMaximumSizeHigh:Long, 文件映射的最大长度的高 32 位。

dwMaximumSizeLow:Long, 文件映射的最大长度的低 32 位。如这个参数和 dwMaximumSizeHigh 都是零, 就用磁盘文件的实际长度。

lpName:String, 指定文件映射对象的名字。如存在这个名字的一个映射, 函数就会打开它。用

vbNullString 可以创建一个无名的文件映射。

返回值:

Long, 新建文件映射对象的句柄; 零意味着出错。会设置 GetLastError。即使函数成功, 但倘若返回的句柄属于一个现成的文件映射对象, 那么 GetLastError 也会设置成 ERROR_ALREADY_EXISTS。在这种情况下, 文件映射的长度就是现有对象的长度, 而不是这个函数指定的尺寸。

备注:

创建一个映射到文件的句柄, 将文件装载到内存, 并使得他可以通过内存地址进行访问。启动器, 装载器和注入器会使用这个函数来读取和修改 PE 文件。

kernel32.OpenFile

功能: 用于打开文件的

函数原型:

```
HFILE WINAPI OpenFile(  
    _In_ LPCSTR lpFileName,
```

```

_Out_ LPOFSTRUCT lpReOpenBuff,
_In_   UINT          uStyle
);

```

参数介绍：（参数与返回值）

lpFileName：文件名

lpReOpenBuff：变量指针，用于存储文件被首次打开时接收信息

uStyle：打开文件的常量类型

成功返回打开的文件句柄，失败返回 HFILE_ERROR

FindFirstFile

功能：根据文件名查找文件。该函数到一个文件夹(包括子文件夹)去搜索指定文件

函数原型：

```

HANDLE FindFirstFile(
    LPCTSTR lpFileName, //filename
    LPWIN32_FIND_DATA lpFindFileData //databuffer
);

```

参数介绍：

LPCTSTR lpFileName 文件名（包括路径）

LPWIN32_FIND_DATA lpFindFileData 指向一个用于保存文件信息的结构体

返回值：

如果调用成功返回一个句柄，可用来做为 FindNextFile 或 FindClose 参数

调用失败 返回为 INVALID_HANDLE_VALUE(即-1)，可调用 GetLastError 来获取错误信息

备注：用来搜索文件目录和枚举文件系统的函数

FindNextFile

功能：可以用来遍历目录或文件时，判断当前目录下是否有下一个目录或文件。

函数原型：

```

BOOL FindNextFile(
    HANDLE hFindFile, //searchhandle
    LPWIN32_FIND_DATA lpFindFileData //databuffer
);

```

参数介绍：

HANDLE hFindFile 搜索的文件句柄 函数执行的时候搜索的是此句柄的下一文件

LPWIN32_FIND_DATA lpFindFileData 指向一个用于保存文件信息的结构体

返回值：

非零表示成功，零表示失败。如不再有与指定条件相符的文件，会将 GetLastError 设置成 ERROR_NO_MORE_FILES

备注：用来搜索文件目录和枚举文件系统的函数

GetModuleFileName

功能：获取当前进程已加载模块的文件的完整路径，该模块必须由当前进程加载。

函数原型：

```
DWORD WINAPI GetModuleFileName(  
    _In_opt_ HMODULE hModule,  
    _Out_ LPTSTR lpFilename,  
    _In_ DWORD nSize  
);
```

参数介绍：

hModule 一个模块的句柄。可以是一个 DLL 模块，或者是一个应用程序的实例句柄。如果该参数为 NULL，

该函数返回该应用程序全路径。

lpFileName 指定一个字串缓冲区，要在其中容纳文件的用 NULL 字符中止的路径名，hModule 模块就是从这个文件装载进来的

nSize 装载到缓冲区 lpFileName 的最大字符数量

返回值：Long，如执行成功，返回复制到 lpFileName 的实际字符数量；零表示失败

备注：返回目前进程装载某个模块的文件名，恶意代码可以使用这个函数，在目前运行进程中修改或复制文件

GetModuleHandle

功能：获取一个应用程序或动态链接库的模块句柄。

函数原型：HMODULE GetModuleHandle(LPCTSTR lpModuleName);

参数介绍：lpModuleName 模块名称

返回值：如执行成功成功，则返回模块句柄。零表示失败

备注：用来获取已装载模块句柄的函数，恶意代码可以使用此函数在一个装载模块中定位和修改代码，或者搜索一个合适位置来注入代码。

GetProcAddress

功能：检索指定的动态链接库 (DLL) 中的输出库函数地址。

函数原型：

```
FARPROC GetProcAddress(  
    HMODULE hModule,  
    LPCSTR lpProcName  
);
```

参数介绍：

hModule

[in] 包含此函数的 DLL 模块的句柄。LoadLibrary、AfxLoadLibrary 或者 GetModuleHandle 函数可以返回此句柄。

lpProcName

[in] 包含函数名的以 NULL 结尾的字符串，或者指定函数的序数值。如果此参数是一个序数值，它必须在一个字的底字节，高字节必须为 0。

返回值：

如果函数调用成功，返回值是 DLL 中的输出函数地址。

如果函数调用失败，返回值是 NULL。

备注：获取装载到内存中一个 DLL 程序的函数地址。用来从其他 DLL 程序中导入函数，以补充在 PE 文件头部中导入的函数。

GetStartupInfo

功能：取得进程在启动时被指定的 STARTUPINFO 结构。

函数原型：

```
VOID GetStartupInfo(  
    LPSTARTUPINFO lpStartupInfo);
```

参数介绍：

lpStartupInfo 一个指向用来存放要获取的 STARTUPINFO 结构的指针。

返回值：无

备注：获取一个包含当前进程如何自动运行配置信息的结构，比如标准句柄指向哪些位置。

GetTempPath

功能：获取为临时文件指定的路径

函数原型：

```
DWORD WINAPI GetTempPath(  
    _In_    DWORD    nBufferLength,  
    _Out_   LPTSTR   lpBuffer  
);
```

参数介绍：

nBufferLength：表示 lpBuffer 的大小

lpBuffer：接收路径的一块内存

返回值：如果成功，返回 lpBuffer 的长度，失败返回 0

备注：返回临时文件路径，如果看到恶意代码使用了这个函数，需要检查他是否咋临时文件路径中读取或写入了一些文件。

GetWindowsDirectory

功能：获取 Windows 目录的完整路径名。

函数原型：UINT GetWindowsDirectory(LPTSTR lpBuffer, UINT uSize)

参数介绍：

lpBuffer，指定一个字串缓冲区，用于装载 Windows 目录名。除非是根目录，否则目录中不会有一个中止用的“\”字符

nSize，lpBuffer 字串的最大长度

返回值：复制到 lpBuffer 的一个字串的长度。如 lpBuffer 不够大，不能容下整个字串，就会返回 lpBuffer 要求的长度，零表示失败

备注：返回 Windows 目录的文件系统路径，恶意代码经常使用这个函数来确定将其他恶意程序安装到哪个目录。

MapViewOfFile

功能：将一个文件映射对象映射到当前应用程序的地址空间

函数原型：

```
LPVOID WINAPI MapViewOfFile(  
    __in HANDLE    hFileMappingObject,  
    __in DWORD     dwDesiredAccess,  
    __in DWORD     dwFileOffsetHigh,  
    __in DWORD     dwFileOffsetLow,  
    __in SIZE_T    dwNumberOfBytesToMap  
);
```

参数介绍：

hFileMappingObject：hFileMappingObject 为 CreateFileMapping() 返回的文件映像对象句柄。

dwDesiredAccess：dwDesiredAccess 映射对象的文件数据的访问方式，而且同样要与 CreateFileMapping() 函数所设置的保护属性相匹配。

dwFileOffsetHigh：dwFileOffsetHigh 表示文件映射起始偏移的高 32 位。

dwFileOffsetLow：dwFileOffsetLow 表示文件映射起始偏移的低 32 位。(64KB 对齐不是必须的)

dwNumberOfBytesToMap：dwNumberOfBytes 指定映射文件的字节数。

返回值：

如果成功，则返回映射视图文件的开始地址值。如果失败，则返回 NULL

备注：映射一个文件到内存，将文件内容变得通过内存地址可访问。启动器，装载器和注入器使用这个函数来读取和修改 PE 文件。通过使用此函数，恶意代码可以避免使用 WriteFile 来修改文件内容。

NtQueryDirectoryFile

功能：返回多种指定的文件信息

函数原型:

NTSTATUS

```
ZwQueryDirectoryFile(  
    __in HANDLE    FileHandle,  
    __in_opt HANDLE    Event,  
    __in_opt PIO_APC_ROUTINE    ApcRoutine,  
    __in_opt PVOID    ApcContext,  
    __out PIO_STATUS_BLOCK    IoStatusBlock,  
    __out PVOID    FileInformation,  
    __in ULONG    Length,  
    __in FILE_INFORMATION_CLASS    FileInformationClass,  
    __in BOOLEAN    ReturnSingleEntry,  
    __in_opt PUNICODE_STRING    FileName,  
    __in BOOLEAN    RestartScan  
);
```

参数介绍:

FileHandle: ZwCreateFile 和 ZwOpenFile 返回的句柄, 代表着被查询信息的文件夹

Event: 调用者创建的一个可选的事件句柄。

ApcRoutine: 调用者提供的一个 APC 例程, 当操作完成时, 调用此例程。

ApcContext: 如果调用者提供了 APC 例程, 则此参数为 APC 例程的上下文

IoStatusBlock: 指向 IO_STATUS_BLOCK 结构体, 返回操作的完成状态和信息。

FileInformation: 接收文件的特定的信息

Length: FileInformation 信息的长度。

FileInformationClass: 需要返回的文件夹信息的类型。

ReturnSingleEntry: 如果要返回单一入口, 就将此值设为 TRUE, 否则设为 FALSE。

FileName: 调用者申请的文件名

RestartScan: 如果第一次访问文件夹, 将此参数设为 TRUE, 否则设为 FALSE。

返回值: 成功返回 STATUS_SUCCESS, 失败返回错误码

备注: 返回一个目录中文件的信息, 内核套件普遍会挂钩这个函数来隐藏文件。

SetFileTime

功能: 设置文件的创建、访问及上次修改时间 函数原型:

```
BOOL WINAPI SetFileTime(  
    _In_ HANDLE    hFile,  
    _In_opt_ const FILETIME *lpCreationTime,  
    _In_opt_ const FILETIME *lpLastAccessTime,  
    _In_opt_ const FILETIME *lpLastWriteTime  
);
```

参数介绍:

hFile Long, 系统文件句柄

lpCreationTime FILETIME, 文件的创建时间

lpLastAccessTime FILETIME, 文件上一次访问的时间

lpLastWriteTime FILETIME, 文件最近一次修改的时间

返回值: 非零表示成功, 零表示失败

备注: 修改一个文件的创建, 访问或者最后修改时间, 恶意代码经常使用这个函数来隐藏恶意行为。

Wow64DisableWow64FsRedirection

功能: 禁用文件系统重定向机制

函数原型:

```
BOOL WINAPI Wow64DisableWow64FsRedirection(  
    _Out_ PVOID *OldValue  
);
```

参数介绍: OldValue: Wow64 文件系统重定向值

返回值: 成功返回非 0, 失败返回 0.

备注: 禁用 32 为文件在 64 位操作系统中装载后发生的文件重定向机制, 如果一个 32 位应用程序在调用这个函数后向 C:\Windows\System32 写数据, 那么它将会直接写到真正的 C:\Windows\System32, 而不是被重定向至 C:\Windows\SysWOW64

0x02 网络类

ws2_32.socket

功能: 用于根据指定的地址族、数据类型和协议来分配一个套接口的描述字及其所用的资源。如果协议 protocol 未指定 (等于 0), 则使用缺省的连接方式

函数原型: int socket(int af, int type, int protocol);

参数介绍: (参数与返回值)

af: 一个地址描述。目前仅支持 AF_INET 格式, 也就是说 ARPA Internet 地址格式。

type: 指定 socket 类型。新套接口的类型描述类型, 如 TCP (SOCK_STREAM) 和 UDP

(SOCK_DGRAM)。常用的 socket 类型有, SOCK_STREAM、SOCK_DGRAM、SOCK_RAW、SOCK_PACKET、SOCK_SEQPACKET 等等。

protocol: 顾名思义, 就是指定协议。套接口所用的协议。如调用者不想指定, 可用 0。常用的协议有, IPPROTO_TCP、IPPROTO_UDP、IPPROTO_STCP、IPPROTO_TIPC 等, 它们分别对应 TCP 传输协议、UDP 传输协议、STCP 传输协议、TIPC 传输协议。

若无错误发生, socket() 返回引用新套接口的描述字。否则的话, 返回 INVALID_SOCKET 错误, 应用程序可通过 WSAGetLastError() 获取相应错误代码。

accept

功能: 用来监听入站网络连接, 此函数预示着程序会在一个套接字上监听入站网络连接。

函数原型:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

参数介绍:

sockfd: 套接字描述符, 该套接口在 listen() 后监听连接。

addr: (可选) 指针, 指向一缓冲区, 其中接收为通讯层所知的连接实体的地址。Addr 参数的实际格式由套接口创建时所产生的地址族确定。

addrlen: (可选) 指针, 输入参数, 配合 addr 一起使用, 指向存有 addr 地址长度的整型数。

返回值: 如果没有错误产生, 则 accept() 返回一个描述所接受包的 SOCKET 类型的值。否则的话, 返回 INVALID_SOCKET 错误, 应用程序可通过调用 WSAGetLastError() 来获得特定的错误代码。

addrlen 所指的整形数初始时包含 addr 所指地址空间的大小, 在返回时它包含实际返回地址的字节长度。

bind

功能: 将一本地地址与一套接口捆绑

函数原型:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t *addrlen);
```

参数介绍:

sockfd: 已经建立的 socket 编号(描述符)

addr: 是一个指向 sockaddr 结构体类型的指针;

addrlen: 表示 addr 结构的长度

返回值: 成功返回 0, 失败返回-1

备注:

connect

功能: 用来将参数 sockfd 的 socket 连至参数 serv_addr 指定的网络地址

函数原型:

```
int connect (int sockfd, struct sockaddr * serv_addr, int addrlen);
```

参数介绍:

Sockfd: 套接字描述符

serv_addr: 指向数据结构 sockaddr 的指针, 其中包括目的端口和 IP 地址

addrlen: 参数二 sockaddr 的长度, 可以通过 sizeof(struct sockaddr) 获得

返回值: 成功则返回 0, 失败返回非 0, 错误码 GetLastError()。

备注: 用来连接一个远程套接字。恶意代码经常使用底层功能函数来连接一个命令控制服务器。

wininet.InternetOpenA

功能: 初始化一个应用程序, 以使用 WinINet 函数

函数原型:

```
HINTERNET InternetOpen(  
    _In_ LPCTSTR lpszAgent,  
    _In_ DWORD dwAccessType,  
    _In_ LPCTSTR lpszProxyName,  
    _In_ LPCTSTR lpszProxyBypass,  
    _In_ DWORD dwFlags  
);
```

参数介绍:

lpszAgent: 指向一个空结束的字符串, 该字符串指定调用 WinInet 函数的应用程序或实体的名称。使用此名称作为用户代理的 HTTP 协议。

dwAccessType: 指定访问类型

lpszProxyName: 指针指向一个空结束的字符串, 该字符串指定的代理服务器的名称, 不要使用空字符串; 如果 *dwAccessType* 未设置为 INTERNET_OPEN_TYPE_PROXY, 则此参数应该设置为 NULL。

lpszProxyBypass: 指向一个空结束的字符串, 该字符串指定的可选列表的主机名或 IP 地址。如果 *dwAccessType* 未设置为 INTERNET_OPEN_TYPE_PROXY 的, 参数省略则为 NULL。

dwFlags: 参数可以是下列值的组合:

INTERNET_FLAG_ASYNC: 使异步请求处理的后裔从这个函数返回的句柄

INTERNET_FLAG_FROM_CACHE: 不进行网络请求, 从缓存返回的所有实体, 如果请求的项目不在缓存中, 则返回一个合适的错误, 如 ERROR_FILE_NOT_FOUND。

INTERNET_FLAG_OFFLINE: 不进行网络请求, 从缓存返回的所有实体, 如果请求的项目不在缓存中, 则返回一个合适的错误, 如 ERROR_FILE_NOT_FOUND。

返回值: 成功: 返回一个有效的句柄, 该句柄将由应用程序传递给接下来的 WinInet 函数。

失败: 返回 NULL。

备注: 该函数是第一个由应用程序调用的 WinInet 函数。它告诉 Internet DLL 初始化内部数据结构并准备接收应用程序之后的其他调用。当应用程序结束使用 Internet 函数时, 应调用 InternetCloseHandle 函数来释放与之相关的资源。

应用程序可以对该函数进行任意次数的调用, 不过在一般情况下一次调用就已经足够了。如果要调用多次该函数, 应用程序则有必要定义独立的函数实例的行为, 诸如不同的代理服务器等。

CoCreateInstance

功能: 用指定的类标识符创建一个 Com 对象, 用指定的类标识符创建一个未初始化的对象
函数原型:

```
STDAPI CoCreateInstance(  
    REFCLSID rclsid, //创建的 Com 对象的类标识符(CLSID)  
    LPUNKNOWN pUnkOuter, //指向接口 IUnknown 的指针  
    DWORD dwClsContext, //运行可执行代码的上下文  
    REFIID riid, //创建的 Com 对象的接口标识符  
    LPVOID *ppv //用来接收指向 Com 对象接口地址的指针变量
```

);

参数说明:

rclsid

[in] 用来唯一标识一个对象的 CLSID(128 位), 需要用它来创建指定对象。

pUnkOuter

[in] 如果为 NULL, 表明此对象不是聚合式对象一部分。如果不是 NULL, 则指针指向一个聚合式对象的 IUnknown 接口。

dwClsContext

[in] 组件类别。可使用 CLSCTX 枚举器中预定义的值。

riid

[in] 引用接口标识符, 用来与对象通信。

ppv

[out] 用来接收指向接口地址的指针变量。如果函数调用成功, *ppv 包括请求的接口指针。

返回值:

S_OK: 指定的 Com 对象实例被成功创建。

REGDB_E_CLASSNOTREG: 指定的类没有在注册表中注册。也可能是指定的 dwClsContext 没有注册或注册表中的服务器类型损坏

CLASS_E_NOAGGREGATION: 这个类不能创建为聚合型。

E_NOINTERFACE: 指定的类没有实现请求的接口, 或者是 IUnknown 接口没有暴露请求的接口。

备注: 创建一个 COM 对象, COM 对象提供了非常多样化的功能。类标识 (CLSID) 会告诉你哪个文件包含着实现 COM 对象的代码。

FtpPutFile

功能: 将本地文件上传到 FTP 服务器

函数原型:

```
BOOL WINAPI FtpPutFile( HINTERNET hConnect,
LPCTSTR lpszLocalFile,
LPCTSTR lpszNewRemoteFile,
DWORD dwFlags,
DWORD dwContext);
```

参数介绍:

hConnect: FTP 会话句柄

lpszLocalFile 本地文件路径

lpszNewRemoteFile 上传到 ftp 服务器的文件保存路径

dwFlags 指示文件上传的条件

dwContext 指定应用数据该搜索相关联的应用程序定义的值。此参数仅当应用程序已调用

InternetSetStatusCallback 成立一个状态回调。所有的状态请求都得到同步处理。

返回值：TRUE 表示成功，FALSE 表示失败

备注：一个高层次上的函数，用来向一个远程 FTP 服务器上传文件。

GetAdaptersInfo

功能：获取网卡详细信息

函数原型：

```
DWORD GetAdaptersInfo(  
    _Out_ PIP_ADAPTER_INFO pAdapterInfo,  
    _Inout_ PULONG pOutBufLen  
);
```

参数介绍：

pAdapterInfo：一个缓冲区的指针，用来接收 IP_ADAPTER_INFO 结构的信息。

pOutBufLen：表示 pAdapterInfo 的大小

返回值：成功返回 ERROR_SUCCESS，其他表示失败

备注：用来获取系统上网络适配器的相关信息。后门程序有时会调用此函数来取得关于受感染主机的摘要信息。在某些情况下，这个函数也会被使用来取得主机的 MAC 地址，在对抗虚拟机技术中用来检测 VMware 等虚拟机。

gethostbyname

功能：返回对应于给定主机名的包含主机名字和地址信息的 hostent 结构指针

函数原型：

```
struct hostent *gethostbyname(const char *name);
```

参数介绍：

name：指向主机名的指针。

返回值：如果没有错误发生，gethostbyname() 返回如上所述的一个指向 hostent 结构的指针，否则，返回一个空指针

备注：在向一个远程主机发起 IP 连接之前，用来对一个特定域名执行一次 DNS 查询，作为米宁控制服务器的域名通常可以用来创建很好的网络监测特征码

gethostname

功能：返回本地主机的标准主机名。

函数原型：int PASCAL FAR gethostname(char FAR *name, int namelen);

参数介绍：

name：一个指向将要存放主机名的缓冲区指针。

namelen：缓冲区的长度。

返回值：如果没有错误发生，`gethostname()` 返回 0。否则它返回 `SOCKET_ERROR`

备注：获取计算机主机名。后门程序经常使用此函数来获取受害主机的摘要信息

inet_addr

功能：将一个点分十进制的 IP 转换成一个长整数型数

函数原型：`in_addr_t inet_addr(const char* strptr);`

参数介绍：`strptr`：字符串，一个点分十进制的 IP 地址

返回值：如果正确执行将返回一个无符号长整数型数。如果传入的字符串不是一个合法的 IP 地址，将返回 `INADDR_NONE`。

备注：将一个 IP 地址字符串，如 127.0.0.1，进行转化，使其能够在如 `connect` 等函数中使用。这些字符串有时也可以用作基于网络的特征码。

InternetOpen

功能：初始化一个应用程序，以使用 WinInet 函数。

函数原型：

```
HINTERNET InternetOpen(
    _In_ LPCTSTR lpszAgent,
    _In_ DWORD dwAccessType,
    _In_ LPCTSTR lpszProxyName,
    _In_ LPCTSTR lpszProxyBypass,
    _In_ DWORD dwFlags
);
```

参数介绍：

`lpszAgent`

指向一个空结束的字符串，该字符串指定的应用程序或实体调用 WinInet 函数的名称。使用此名称作为用户代理的 HTTP 协议。

`dwAccessType`

指定访问类型

`lpszProxyName`

指针指向一个空结束的字符串，该字符串指定的代理服务器的名称，不要使用空字符串；如果

`dwAccessType` 未设置为 `INTERNET_OPEN_TYPE_PROXY`，则此参数应该设置为 `NULL`。

`lpszProxyBypass`

指向一个空结束的字符串，该字符串指定的可选列表的主机名或 IP 地址。如果 `dwAccessType` 未设置为 `INTERNET_OPEN_TYPE_PROXY` 的，参数省略则为 `NULL`。

`dwFlags`：网络选项标志位

返回值：

成功：返回一个有效的句柄，该句柄将由应用程序传递给接下来的 WinInet 函数。

失败：返回 `NULL`。

备注：初始化 WinINet 中的一些高层次互联网访问函数，搜索此函数是找到互联网访问功能初始位置的一个好方法。该函数的一个参数是 User-Agent，有时也可以作为基于网络的特征码。

InternetOpenUrl

功能：通过一个完整的 FTP，Gopher 或 HTTP 网址打开一个资源

函数原型：

```
HINTERNET InternetOpenUrl(  
    HINTERNET hInternetSession,  
    LPCTSTR lpszUrl,  
    LPCTSTR lpszHeaders,  
    DWORD dwHeadersLength,  
    DWORD dwFlags,  
    DWORD dwContext  
);
```

参数介绍：

hInternet

当前的 Internet 会话句柄。句柄必须由前期的 InternetOpen 调用返回。

lpszUrl

一个空字符结束的字符串变量的指针，指定读取的网址。只有以 ftp:，gopher:，http:，或者 https: 开头的网址被支持。

lpszHeaders

一个空字符结束的字符串变量的指针，指定发送到 HTTP 服务器的头信息。欲了解更多信息，请参阅 HttpSendRequest 函数里 lpszHeaders 参数的说明。

dwHeadersLength: lpszHeaders 的长度。

dwFlags: 行为标志的位掩码

dwContext

一个指向一个应用程序定义的值，将随着返回的句柄，一起传递给回调函数

返回值：如果已成功建立到 FTP，Gopher，或 HTTP URL 的连接，返回一个有效的句柄，如果连接失败返回 NULL。

备注：使用 FTP, HTTP 或 HTTPS 协议连接来打开一个特定的 URL，如果 URL 固定，则可以作为基于网络的特征码

InternetReadFile

功能：从一个由 InternetOpenUrl, FtpOpenFile, 或 HttpOpenRequest 函数打开的句柄中读取数据

函数原型：

```
BOOL InternetReadFile( __in HINTERNET hFile,  
    __out LPVOID lpBuffer,  
    __in DWORD dwNumberOfBytesToRead,  
    __out LPDWORD lpdwNumberOfBytesRead
```

);

参数介绍:

hFile: 由 InternetOpenUrl, FtpOpenFile, 或 HttpOpenRequest 函数返回的句柄.

lpBuffer: 缓冲区指针

dwNumberOfBytesToRead: 欲读数据的字节量。

lpdwNumberOfBytesRead: 接收读取字节量的变量。该函数在做任何工作或错误检查之前都设置该值为零

返回值: 成功: 返回 TRUE, 失败, 返回 FALSE

备注: 从之前打开的 URL 中读取数据

InternetWriteFile

功能: 向服务器上传文件

函数原型:

```
BOOL InternetWriteFile(  
    _In_     HINTERNET hFile,  
    _In_     LPCVOID   lpBuffer,  
    _In_     DWORD      dwNumberOfBytesToWrite,  
    _Out_    LPDWORD    lpdwNumberOfBytesWritten  
);
```

参数介绍: hFile : 由 InternetOpenUrl, FtpOpenFile, 或 HttpOpenRequest 函数返回的句柄.

lpBuffer: 缓冲区指针

dwNumberOfBytesToWrite : 欲写数据的字节量

lpdwNumberOfBytesWritten: 实际写入的数据的字节量

返回值: 成功: 返回 TRUE, 失败, 返回 FALSE

备注: 写数据到之前打开的一个 URL

NetShareEnum

功能: 返回服务器上共享的资源的信息

函数原型:

```
NET_API_STATUS NetShareEnum(  
    _In_     LPWSTR    servername,  
    _In_     DWORD      level,  
    _Out_    LPBYTE     *bufptr,  
    _In_     DWORD      prefmaxlen,  
    _Out_    LPDWORD    entriesread,  
    _Out_    LPDWORD    totalentries,  
    _Inout_  LPDWORD    resume_handle  
);
```

参数介绍: servername: 函数要执行的远程服务的 DNS 或 NetBIOS 名字

Level: 数据的信息级别

Bufptr: 接收数据的缓冲区, 数据的格式取决于参数 Level。释放此缓冲区需要使用 NetApiBufferFree

Prefmaxlen: 指定返回的数据的最大长度

Entriesread: 接收枚举到的实际的数据的个数。

Totalentries: 接收枚举到的所有函数入口的个数。

resume_handle: 用来接收一个句柄, 此句柄被用来继续进行搜索。当第一次搜索时, 此句柄需要为 0。

返回值: 成功返回 NERR_Success, 失败返回系统的错误码

备注: 用来枚举网络共享的函数

OleInitialize

功能: 是在当前单元 (apartment) 初始化组件对象模型 (COM) 库, 将当前的并发模式标识为 STA (single-thread apartment——单线程单元), 并启用一些特别用于 OLE 技术的额外功能

函数原型: WINOLEAPI OleInitialize(LPVOID pvReserved);

参数介绍: pvReserved 为保留参数, 在使用函数时必须设定为 NULL

返回值:

这个函数除了支持标准的返回值 E_INVALIDARG, E_OUTOFMEMORY 和 E_UNEXPECTED 之外还可能产生以下返回值

S_OK: COM 库和 OLE 技术所特有的额外功能在当前单元被成功初始化。

S_FALSE: COM 库在当前单元已经被初始化过。

OLE_E_WRONGCOMPOBJ: 本机上的文件 COMPOBJ.DLL 和 OLE2.DLL 的版本不相匹配。

RPC_E_CHANGED_MODE: 之前有一个对函数 CoInitializeEx 的调用指明了这个单元的并发模式为 MTA (multithread apartment——多线程单元)。如果当前正在使用 Windows 2000, 这个返回值也能表明发生了 NA (neutral threaded apartment——中立线程单元) 到 STA 的转换。

备注: 用来初始化 COM 库, 使用 COM 对象的程序必须在调用任何其他 COM 功能之前, 调用这个函数。

recv

功能: 从已经连接的 socket 中接收数据

函数原型:

```
int recv( _In_ SOCKET s,
          _Out_ char *buf,
          _In_ int len,
          _In_ int flags
        );
```

参数介绍:

s: 已经连接的 socket

buf: 接收数据的缓冲区

len: buf 缓冲区的长度

flags: 影响此函数行为的标志

返回值: 成功返回接收的数据的长度, 失败返回 0.

备注: 从一个远程主机获取数据, 恶意代码经常使用这个函数来从远程的命令控制服务器获取数据。

send

功能: 向一个已经连接的 socket 发送数据

函数原型: `int PASCAL FAR send(SOCKET s, const char FAR* buf, int len, int flags);`

参数介绍:

s: 一个用于标识已连接套接口的描述字。

buf: 包含待发送数据的缓冲区。

len: 缓冲区中数据的长度。

flags: 调用执行方式。

返回值: 如果无错误, 返回值为所发送数据的总数, 否则返回 SOCKET_ERROR。

备注: 发送数据到远程主机, 恶意代码经常使用这个函数来发送数据到远程的命令控制服务器。

URLDownloadToFile

功能: 从指定 URL 地址读取内容并将读取到的内容保存到特定的文件里

函数原型:

```
HRESULT URLDownloadToFile(  
    LPUNKNOWN pCaller,  
    LPCTSTR szURL,  
    LPCTSTR szFileName,  
    DWORD dwReserved,  
    LPBINDSTATUSCALLBACK lpfnCB  
);
```

参数介绍:

pCaller 控件的接口, 如果不是控件则为 0

szURL 要下载的 url 地址, 不能为空

szFileName 下载后保存的文件名.

dwReserved 保留字段, 必需为 0

lpfnCB 下载进度状态回调

返回值: 成功返回 S_OK, 失败返回相应的错误码。

备注: 一个高层次的函数调用, 来从一个 WEB 服务器下载文件并存储到硬盘上。这个函数在下载器中是非常普遍的, 因为他以一个函数调用便实现了下载器的所有功能。

WSAStartup

功能: 初始化底层级别的网络函数

函数原型: `int WSASStartup (WORD wVersionRequested, LPWSADATA lpWSADATA);`

参数介绍:

wVersionRequested: 一个 WORD (双字节) 型数值, 在最高版本的 Windows Sockets 支持调用者使用, 高阶字节指定小版本(修订本)号, 低位字节指定主版本号。

lpWSADATA 指向 WSADATA 数据结构的指针, 用来接收 Windows Sockets 实现的细节。

返回值: 成功返回 0, 否则返回错误码

备注: 用来初始化底层级别的网络功能, 搜索此函数调用位置, 经常是定义网络相关功能最简单的方法。

0x03 注册表与服务类

CreateService

功能: 创建一个服务对象, 并将其添加到指定的服务控制管理器数据库

函数原型:

```
SC_HANDLE CreateService(  
    SC_HANDLE hSCManager,  
    LPCTSTR lpServiceName,  
    LPCTSTR lpDisplayName,  
    DWORD dwDesiredAccess,  
    DWORD dwServiceType,  
    DWORD dwStartType,  
    DWORD dwErrorControl,  
    LPCTSTR lpBinaryPathName,  
    LPCTSTR lpLoadOrderGroup,  
    LPDWORD lpdwTagId,  
    LPCTSTR lpDependencies,  
    LPCTSTR lpServiceStartName,  
    LPCTSTR lpPassword  
);
```

参数介绍:

hSCManager, //服务控制管理程序维护的登记数据库的句柄, 由系统函数 `OpenSCManager` 返回

lpServiceName, //以 NULL 结尾的服务名, 用于创建登记数据库中的关键字

lpDisplayName, //以 NULL 结尾的服务名, 用于用户界面标识服务

dwDesiredAccess, //指定服务返回类型

dwServiceType, //指定服务类型

dwStartType, //指定何时启动服务

dwErrorControl, //指定服务启动失败的严重程度

lpBinaryPathName, //指定服务程序二进制文件的路径
lpLoadOrderGroup, //指定顺序装入的服务组名
lpdwTagId, //忽略, NULL
lpDependencies, //指定启动该服务前必须先启动的服务或服务组
lpServiceStartName, //以 NULL 结尾的字符串, 指定服务帐号。如是 NULL, 则表示使用

LocalSystem 帐号

lpPassword //以 NULL 结尾的字符串, 指定对应的口令。为 NULL 表示无口令。但使用 LocalSystem 时填 NULL

返回值:

如果函数成功, 返回值将是该服务的句柄。

如果函数失败, 则返回值为 NULL

备注: 创建一个可以再启动时刻运行的服务。恶意代码使用此函数来持久化, 隐藏或者是启动内核驱动。

ControlService

功能: 向服务发送一个控制码

函数原型:

```
BOOL WINAPI ControlService(  
    _In_     SC_HANDLE    hService,  
    _In_     DWORD        dwControl,  
    _Out_    LPSERVICE_STATUS lpServiceStatus  
);
```

参数介绍:

hService: 服务的句柄, 此句柄由 OpenService 或者 CreateService 返回。

dwControl: 控制码

lpServiceStatus: 指向 SERVICE_STATUS 的指针, 用来接收最近的服务状态信息。

返回值: 成功返回非 0, 失败返回 0。

备注: 用来启动, 停止, 修改或发送一个信号到运行服务。如果恶意代码使用了他自己的恶意服务, 你就需要分析实现服务的代码, 来确定出调用的用意。

OpenSCManager

功能: 建立了一个到服务控制管理器的连接, 并打开指定的数据库。

函数原型:

```
SC_HANDLE WINAPI OpenSCManager(  
    _In_opt_ LPCTSTR lpMachineName,  
    _In_opt_ LPCTSTR lpDatabaseName,  
    _In_     DWORD    dwDesiredAccess
```


);

参数介绍:

lpMachineName

指向零终止字符串, 指定目标计算机的名称。如果该指针为 NULL, 或者它指向一个空字符串, 那么该函数连接到本地计算机上的服务控制管理器。

lpDatabaseName

指向零终止字符串, 指定将要打开的服务控制管理数据库的名称。此字符串应被置

为 SERVICES_ACTIVE_DATABASE。如果该指针为 NULL, 则打开默认的 SERVICES_ACTIVE_DATABASE 数据库。

dwDesiredAccess

指定服务访问控制管理器的权限

返回值:

如果函数成功, 返回值是一个指定的服务控制管理器数据库的句柄。

如果函数失败, 返回值为 NULL。

备注: 打开一个到服务控制管理器的句柄。任何想要安装, 修改或是控制一个服务的程序, 都必须调用这个函数, 才能使用其他服务操纵函数。

RegisterHotKey

功能: 定义一个系统范围的热键

函数原型:

```
BOOL RegisterHotKey (HWND hWnd,  
int id,  
UINT fsModifiers,  
UINT vk  
);
```

参数介绍:

hWnd: 接收热键产生 WM_HOTKEY 消息的窗口句柄。若该参数 NULL, 传递给调用线程的 WM_HOTKEY 消息必须在消息循环中进行处理。

id: 定义热键的标识符。调用线程中的其他热键, 不能使用同样的标识符

fsModifoers: 定义为了产生 WM_HOTKEY 消息而必须与由 nVirtKey 参数定义的键一起按下的键。

vk: 定义热键的虚拟键码。

返回值:

若函数调用成功, 返回一个非 0 值。若函数调用失败, 则返回值为 0

备注: 用来注册一个热键, 当用户任意时刻输入一个特定键值组合时, 注册热键句柄将会被通知, 无论当用户输入键值组合时哪个窗口是活跃的, 这个函数通常被间谍软件使用, 使其在键值组合中输入前对用户保持隐藏。

RegOpenKey

功能：打开给定键

函数原型：

```
LONG RegOpenKey( HKEY hKey,  
LPCTSTR lpSubKey,  
PHKEY phkResult);
```

参数介绍：

HKEY hKey：要打开键的句柄
LPCTSTR lpSubKey：要打开子键的名字的地址
PHKEY phkResult：要打开键的句柄的地址

返回值：

如果调用成功，返回 ERROR_SUCCESS。

如果调用失败，返回一个非零错误码

备注：打开一个注册表键值的句柄，来进行读写。修改注册表键值通常是软件在主机上进行持久化保存的一种方法。这册表也包含了完整的操作系统和应用程序配置信息。

StartServiceCtrlDispatcher

功能：将服务进程的主线程连接到服务控制管理器

函数原型：

```
BOOL WINAPI StartServiceCtrlDispatcher(  
_In_ const SERVICE_TABLE_ENTRY *lpServiceTable  
);
```

参数介绍：lpServiceTable：指向 SERVICE_TABLE_ENTRY 结构体数组，包含进程可以运行的服务的入口。

返回值：成功返回非 0，失败返回 0。

备注：由服务使用来连接到服务管理控制进程的主线程。任何以服务方式运行的进程必须在启动后 30 秒内调用这个函数。在恶意代码中找到这个函数，可以知道他的功能应该以服务方式运行。

0x04 进程类

AttachThreadInput

功能：把一个线程的输入消息连接到另外的线程

函数原型：

```
BOOL WINAPI AttachThreadInput(  
__in DWORD idAttach,
```

```
__in DWORD idAttachTo,  
__in BOOL fAttach  
);
```

参数介绍:

idAttach:指定要连接到另外一个线程的线程。该线程不能是系统线程。

idAttachTo:要连接其他线程的线程,该线程不能是系统线程。且线程不能自己连接到自己。

fAttach:为 TRUE: 连接; 为 FALSE: 释放连接

返回值: 如果调用成功则返回非零值。

备注:

在一些情况下,自己的窗口没有输入焦点但是想要当前焦点窗口的键盘输入消息,可以使用 Win32 API 函数 `AttachThreadInput()` 来解决这个问题

CheckRemoteDebuggerPresent

功能: 检查一个特定进程是否被调试。这个函数通常在一个反调试技术中被使用。

函数原型:

```
BOOL WINAPI CheckRemoteDebuggerPresent(  
_In_ HANDLE hProcess,  
_Inout_ PBOOL pbDebuggerPresent  
);
```

参数介绍:

hProcess: 进程句柄。

pbDebuggerPresent: 如果 hProcess 所代表的进程被调试,此变量返回 TRUE,否则返回 FALSE。

返回值: 成功返回非 0,失败返回 0。

备注:

ConnectNamedPipe

功能: 指示一台服务器等待下去,直至客户机同一个命名管道连接。

函数原型:

```
BOOL WINAPI ConnectNamedPipe(  
_In_ HANDLE hNamedPipe,  
_Inout_opt_ LPOVERLAPPED lpOverlapped  
);
```

参数介绍:

hNamedPipe: 管道的句柄。

lpOverlapped, 如设为 NULL (传递 `ByVal As int`), 表示将线程挂起,直到一个客户同管道连接为止。否则就立即返回;此时,如管道尚未连接,客户同管道连接时就会触发 lpOverlapped 结构中的事件对象。随后,可用一个等待函数来监视连接

返回值:

Long, 如 lpOverlapped 为 NULL, 那么:

如管道已连接, 就返回 Ture (非零); 如发生错误, 或者管道已经连接, 就返回零 (GetLastError 此时会返回 ERROR_PIPE_CONNECTED)

lpOverlapped 有效, 就返回零; 如管道已经连接, GetLastError 会返回 ERROR_PIPE_CONNECTED; 如重叠操作成功完成, 就返回 ERROR_IO_PENDING。在这两种情况下, 倘若一个客户已关闭了管道, 且服务器尚未用 DisconnectNamedPipe 函数同客户断开连接, 那么 GetLastError 都会返回 ERROR_NO_DATA
备注: 用来为进程间通信创建一个服务端管道, 等待一个客户端管道连接进来。后门程序和反向 shell 经常使用此函数来简单的连接到一个命令控制服务器。

CreateProcess

功能: 创建一个新的进程和它的主线程, 这个新进程运行指定的可执行文件。

函数原型:

```
BOOL CreateProcess
(
    LPCTSTR lpApplicationName,
    LPCTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

参数说明:

lpApplicationName

指向一个 NULL 结尾的、用来指定可执行模块的字符串。

这个字符串可以是可执行模块的绝对路径, 也可以是相对路径, 在后一种情况下, 函数使用当前驱动器和目录建立可执行模块的路径。

这个参数可以被设为 NULL, 在这种情况下, 可执行模块的名字必须处于 lpCommandLine 参数最前面并由空格符与后面的字符分开。

lpCommandLine

指向一个以 NULL 结尾的字符串, 该字符串指定要执行的命令行。

这个参数可以为空, 那么函数将使用 lpApplicationName 参数指定的字符串当做要运行的程序的命令行。

如果 lpApplicationName 和 lpCommandLine 参数都不为空，那么 lpApplicationName 参数指定将要被运行的模块，lpCommandLine 参数指定将被运行的模块的命令行。新运行的进程可以使用 GetCommandLine 函数获得整个命令行。C 语言程序可以使用 argc 和 argv 参数。

lpProcessAttributes

指向一个 SECURITY_ATTRIBUTES 结构体，这个结构体决定是否返回的句柄可以被子进程继承。如果 lpProcessAttributes 参数为空（NULL），那么句柄不能被继承。

lpThreadAttributes

同 lpProcessAttribute，不过这个参数决定的是线程是否被继承。通常置为 NULL。

bInheritHandles

指示新进程是否从调用进程处继承了句柄。

如果参数的值为真，调用进程中的每一个可继承的打开句柄都将被子进程继承。被继承的句柄与原进程拥有完全相同的值和访问权限。

dwCreationFlags

指定附加的、用来控制优先类和进程的创建的标志。还用来控制新进程的优先类，优先类用来决定此进程的线程调度的优先级。

lpEnvironment

指向一个新进程的环境块。如果此参数为空，新进程使用调用进程的环境。

lpCurrentDirectory

指向一个以 NULL 结尾的字符串，这个字符串用来指定子进程的工作路径。这个字符串必须是一个包含驱动器名的绝对路径。如果这个参数为空，新进程将使用与调用进程相同的驱动器和目录。这个选项是一个需要启动应用程序并指定它们的驱动器和工作目录的外壳程序的主要条件。

lpStartupInfo

指向一个用于决定新进程的主窗体如何显示的 STARTUPINFO 结构体。

lpProcessInformation

指向一个用来接收新进程的识别信息的 PROCESS_INFORMATION 结构体。

返回值：

如果函数执行成功，返回非零值。

如果函数执行失败，返回零

备注：创建并启动一个新进程。如果恶意代码创建了一个新进程，你需要同时分析这个新进程。

CreateRemoteThread

功能：创建一个在其它进程地址空间中运行的线程(也称:创建远程线程)

函数原型：

```
HANDLE WINAPI CreateRemoteThread(  
    __in HANDLE hProcess,  
    __in LPSECURITY_ATTRIBUTES lpThreadAttributes,
```

```
__in SIZE_T dwStackSize,  
__in LPTHREAD_START_ROUTINE lpStartAddress,  
__in LPVOID lpParameter,  
__in DWORD dwCreationFlags,  
__out LPDWORD lpThreadId  
);
```

参数介绍:

hProcess [in]

线程所属进程的进程句柄.

该句柄必须具有 PROCESS_CREATE_THREAD, PROCESS_QUERY_INFORMATION, PROCESS_VM_OPERATION, PROCESS_VM_WRITE, 和 PROCESS_VM_READ 访问权限.

lpThreadAttributes [in]

一个指向 SECURITY_ATTRIBUTES 结构的指针, 该结构指定了线程的安全属性.

dwStackSize [in]

线程初始大小, 以字节为单位, 如果该值设为 0, 那么使用系统默认大小.

lpStartAddress [in]

在远程进程的地址空间中, 该线程的线程函数的起始地址.

lpParameter [in]

传给线程函数的参数.

dwCreationFlags [in]

线程的创建标志.

lpThreadId [out]

指向所创建线程句柄的指针, 如果创建失败, 该参数为 NULL.

返回值:

如果调用成功, 返回新线程句柄.

如果失败, 返回 NULL.

备注: 用来在一个远程进程中启动一个线程. 启动器和隐蔽性恶意代码通常使用这个函数, 将代码注入到其他进程中执行.

CreateToolhelp32Snapshot

功能: 通过获取进程信息为指定的进程、进程使用的堆[HEAP]、模块[MODULE]、线程建立一个快照.

函数原型:

```
HANDLE WINAPI CreateToolhelp32Snapshot(  
    DWORD dwFlags,  
    DWORD th32ProcessID  
);
```

参数说明:

dwFlags, 用来指定“快照”中需要返回的对象, 可以是 TH32CS_SNAPPROCESS 等

th32ProcessID 一个进程 ID 号, 用来指定要获取哪一个进程的快照, 当获取系统进程列表或获取当前进程快照时可以设为 0

返回值:

调用成功, 返回快照的句柄, 调用失败, 返回 INVALID_HANDLE_VALUE 。

备注:

用来创建一个进程, 堆空间, 线程和模块的快照。恶意代码经常使用这个函数, 在多个进程或线程之间传播感染。

EnumProcesses

功能: 检索进程中的每一个进程标识符。

函数原型:

```
BOOL WINAPI EnumProcesses (
    _Out_ DWORD * pProcessIds,
    _In_ DWORD CB,
    _Out_ DWORD * pBytesReturned
);
```

参数介绍:

pProcessIds 接收进程标识符的数组

cb 数组的大小。

pBytesReturned 数组返回的字节数。

返回值: 成功返回非零数, 失败返回零, 可以使用函数 GetLastError 获取错误信息。

备注: 用来在系统上枚举运行进程的函数, 恶意代码经常枚举进程来找到一个可以注入的进程。

EnumProcessModules

功能: 枚举进程模块

函数原型:

```
BOOL WINAPI EnumProcessModules(
    _In_ HANDLE hProcess,
    _Out_ HMODULE *lphModule,
    _In_ DWORD cb,
    _Out_ LPDWORD lpcbNeeded
);
```

参数介绍:

hProcess: 要枚举的进程的句柄

lphModule: 该进程所包含的模块句柄的数组，我们可以定义一个数组来接受该进程所含有的//模块句柄

cb: lphModule 数组的大小

lpcbNeeded: 该进程实际的模块的数量，以字节来计数

返回值: 成功返回非 0，失败返回 0

备注: 用来枚举给定进程的已装载模块（可执行文件和 DLL 程序），恶意代码在进行注入时经常枚举模块

IsWow64Process

功能: 确定指定进程是否运行在 64 位操作系统的 32 环境（Wow64）下

函数原型: BOOL WINAPI IsWow64Process(__in HANDLE hProcess, __out PBOOL Wow64Process);

参数介绍:

hProcess: 进程句柄。该句柄必须具有 PROCESS_QUERY_INFORMATION 或者 PROCESS_QUERY_LIMITED_INFORMATION 访问权限

Wow64Process: 指向一个 bool 值，如果该进程是 32 位进程，运行在 64 操作系统下，该值为 true，否则为 false。如果该进程是一个 64 位应用程序，运行在 64 位系统上，该值也被设置为 false。

返回值:

如果函数成功返回值为非零值。如果该函数失败，则返回值为零

备注: 由一个 32 位进程使用，来确定它是否运行在 64 位操作系统上

ZwQueryInformationProcess

功能: 返回特定进程的信息

函数原型:

```
NTSTATUS WINAPI ZwQueryInformationProcess(
    _In_          HANDLE          ProcessHandle,
    _In_          PROCESSINFOCLASS ProcessInformationClass,
    _Out_         PVOID           ProcessInformation,
    _In_          ULONG           ProcessInformationLength,
    _Out_opt_     PULONG          ReturnLength
);
```

参数介绍:

ProcessHandle: 需要查询的进程的句柄。

ProcessInformationClass: 需要查询的进程信息的类型。

ProcessInformation: 接收进程信息的缓冲区

ProcessInformationLength: ProcessInformation 内容的大小。

ReturnLength: 返回信息的大小。

返回值: 成功返回 STATUS_SUCCESS，失败返回错误码

备注: 返回关于一个特定进程的不同信息。这个函数通常在反调试技术中被使用。

OpenProcess

功能：用来打开一个已存在的进程对象，并返回进程的句柄

函数原型：

```
HANDLE OpenProcess(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    DWORD dwProcessId  
);
```

参数介绍：

dwDesiredAccess：渴望得到的访问权限（标志）

bInheritHandle：是否继承句柄

dwProcessId：进程标示符

返回值：

如成功，返回值为指定进程的句柄。

如失败，返回值为空

备注：打开系统上运行其他进程的句柄。这个句柄可以被用来向其他进程内存中读写数据，或是注入代码到其他进程中。

PeekNamedPipe

功能：预览一个管道中的数据，或取得与管道中的数据有关的信息。

函数原型：

```
BOOL WINAPI PeekNamedPipe(  
    __in HANDLE hNamedPipe, //管道句柄  
    __out_opt LPVOID lpBuffer, //读取输出缓冲区，可选  
    __in DWORD nBufferSize, //缓冲区大小  
    __out_opt LPDWORD lpBytesRead, //接收从管道中读取数据的变量的指针，可选  
    __out_opt LPDWORD lpTotalBytesAvail, //接收从管道读取的字节总数  
    __out_opt LPDWORD lpBytesLeftThisMessage  
);
```

参数介绍：

hNamedPipe：管道句柄。这个参数可以是一个命名管道实例句柄，返回，由 CreateNamedPipe 或 CreateFile 函数，或者它可以是一个匿名管道的读端句柄，返回由 CREATEPIPE 功能。句柄必须有 GENERIC_READ 权限的管道。

lpBuffer：接收从管道读取数据的缓冲区的指针。如果没有数据要读取，此参数可以为 NULL。

nBufferSize：lpBuffer 参数以字节为单位，由指定的缓冲区大小。如果 lpBuffer 是 NULL，则忽略此参数。

lpBytesRead : 接收从管道中读取的字节数的变量的指针。此参数可以为 NULL, 如果没有数据要读取。

lpTotalBytesAvail : 一个指针变量, 接收从管道读取的字节总数。此参数可以为 NULL, 如果没有数据要读取。

lpBytesLeftThisMessage : 指向剩余的字节数的变量的指针消息。此参数将是零字节类型的命名管道或匿名管道。此参数可以为 NULL, 如果没有数据要读取。

返回值: 非零表示成功, 零表示失败

备注: 用来从一个命名管道中复制数据, 而无须从管道中移除数据。这个函数在反向 shell 中很常用。

Process32First

功能: 是一个进程获取函数, 当我们利用函数 `CreateToolhelp32Snapshot()` 获得当前运行进程的快照后, 我们可以利用 `process32First` 函数来获得第一个进程的句柄。

函数原型:

```
BOOL WINAPI Process32First(  
    HANDLE hSnapshot,  
    LPPROCESSENTRY32 lppe  
)
```

参数介绍:

hSnapshot: 调用 `CreateToolhelp32Snapshot` 返回的快照句柄。

lppe: 指向 `PROCESSENTRY32` 结构体

返回值: 成功返回 `true`, 失败返回 `FALSE`

备注: 在调用 `CreateToolhelp32Snapshot` 之后, 使用此函数和 `Process32Next` 来枚举进程。恶意代码通常枚举进程, 来找到一个可以注入的进程。

Process32Next

功能: 当我们利用函数 `CreateToolhelp32Snapshot()` 获得当前运行进程的快照后, 我们可以利用 `Process32Next` 函数来获得下一个进程的句柄。

函数原型:

```
BOOL WINAPI Process32Next(  
    __inHANDLE hSnapshot,  
    __outLPPROCESSENTRY32 lppe  
)
```

参数介绍:

hSnapshot: 从 `CreateToolhelp32Snapshot` 返回的句柄。

lppe: 指向 `PROCESSENTRY32` 结构的指针。

返回值: 成功返回 `TRUE`, 失败返回 `FALSE`。

备注: 在调用 `CreateToolhelp32Snapshot` 之后, 使用此函数和 `Process32First` 来枚举进程。恶意代码通常枚举进程, 来找到一个可以注入的进程。

ReadProcessMemory

功能：根据进程句柄读入该进程的某个内存空间

函数原型：

```
BOOL ReadProcessMemory(  
    HANDLE hProcess,  
    PVOID pvAddressRemote,  
    PVOIDpvBufferLocal,  
    DWORD dwSize,  
    PDWORDpdwNumBytesRead  
);
```

参数介绍：

hProcess [in] 远程进程句柄。 被读取者

pvAddressRemote [in] 远程进程中内存地址。 从具体何处读取

pvBufferLocal [out] 本地进程中内存地址。 函数将读取的内容写入此处

dwSize [in] 要传送的字节数。要写入多少

pdwNumBytesRead [out] 实际传送的字节数。函数返回时报告实际写入多少

返回值：成功返回 1，失败返回 0。

备注：用来从远程进程中读取内存。

ResumeThread

功能：恢复挂起的线程

函数原型：DWORD WINAPI ResumeThread(__in HANDLE hThread);

参数介绍：hThread：需要恢复的线程的句柄。

返回值：成功返回线程先前的挂起次数，失败返回-1。

备注：继续之前挂起的线程。此函数在几种注入技术中都会被使用。

SetThreadContext

功能：修改给定线程的上下文

函数原型：

```
BOOL WINAPI SetThreadContext(  
    __In_ HANDLE hThread,  
    __In_ const CONTEXT *lpContext  
);
```

参数介绍：

hThread：目标线程的句柄

lpContext：目标线程的上下文

返回值：修改成功，返回非 0，失败返回 0

备注：一些注入技术会使用这个函数。

ShellExecute

功能：运行一个外部程序（或者是打开一个已注册的文件、打开一个目录、打印一个文件等等），并对外部程序有一定的控制

函数原型：

```
HINSTANCE ShellExecute(  
    _In_opt_ HWND        hwnd,  
    _In_opt_ LPCTSTR lpOperation,  
    _In_      LPCTSTR lpFile,  
    _In_opt_ LPCTSTR lpParameters,  
    _In_opt_ LPCTSTR lpDirectory,  
    _In_      INT         nShowCmd  
);
```

参数介绍：

Hwnd：指定父进程句柄

lpOperation：指定动作，例如：open、runas、print、edit、explore、find

lpFile：指定要打开的文件或程序

lpParameters：给要打开的程序指定的参数，如果打开的是文件，这里应该是 null

lpDirectory。缺省目录

nShowCmd：打开选项

返回值：成功返回应用程序句柄。失败返回相应的错误码

备注：用来执行另一个程序，如果恶意代码创建了一个新的进程，需要分析这个新进程。

SuspendThread

功能：暂停指定的线程

函数原型：

```
DWORD WINAPI SuspendThread(  
    _In_ HANDLE hThread  
);
```

参数介绍：hThread：需要暂停的线程的句柄

返回值：如果成功，返回线程先前的挂起计数，失败返回-1

备注：挂起一个线程，使得他停止运行。恶意代码有时会挂起一个线程，铜鼓代码注入技术来修改它。

Thread32First

功能：返回进程中第一个线程的信息

函数原型：

```
BOOL WINAPI Thread32First(  
    _In_ HANDLE hSnapshot,  
    _Inout_ LPTHREADENTRY32 lpte  
);
```

参数介绍:

hSnapshot 句柄从先前的调用返回的快照 CreateToolhelp32Snapshot 功能。

lpte 一个指向一个 THREADENTRY32 结构。

返回值: 如果第一个条目的线程列表复制到缓冲区返回 TRUE, 否则返回 FALSE。

备注: 用来轮询一个进程的所有线程。注入器会使用这些函数来找出可供注入的合适线程。

Thread32Next

功能: 返回进程中下一个线程的信息

函数原型:

```
BOOL WINAPI Thread32Next(  
    _In_ HANDLE hSnapshot,  
    _Out_ LPTHREADENTRY32 lpte  
);
```

参数介绍:

hSnapshot 句柄从先前的调用返回的快照 CreateToolhelp32Snapshot 功能。

lpte 一个指向一个 THREADENTRY32 结构。

返回值: 如果下一个条目的线程列表复制到缓冲区返回 TRUE, 否则返回 FALSE。

备注: 用来轮询一个进程的所有线程。注入器会使用这些函数来找出可供注入的合适线程。

WinExec

功能: 运行指定的程序

函数原型:

```
UINT WINAPI WinExec(  
    _In_ LPCSTR lpCmdLine,  
    _In_ UINT uCmdShow  
);
```

参数介绍:

lpCmdLine: 指向一个空结束的字符串, 串中包含将要执行的应用程序的命令行(文件名加上可选参数)。

uCmdShow: 定义 Windows 应用程序的窗口如何显示, 并为 CreateProcess 函数提供 STARTUPINFO 参数的 wShowWindow 成员的值。

返回值: 返回值大于 31 表示成功, 否则返回相应的错误码

备注:

WriteProcessMemory

功能：写入某一进程的内存区域。入口区必须可以访问，否则操作将失败

函数原型：

```
BOOL WriteProcessMemory(  
    HANDLE hProcess,  
    LPVOID lpBaseAddress,  
    LPVOID lpBuffer,  
    DWORD nSize,  
    LPDWORD lpNumberOfBytesWritten  
);
```

参数介绍：

hProcess 由 OpenProcess 返回的进程句柄。

如参数传数据为 INVALID_HANDLE_VALUE 【即-1】目标进程为自身进程

lpBaseAddress 要写的内存首地址，再写入之前，此函数将先检查目标地址是否可用，并能容纳待写入的数据。

lpBuffer 指向要写的数据的指针。

nSize 要写入的字节数。

返回值：非零代表成功，零代表失败

备注：用来向远程进程写数据的函数，恶意代码在进程注入中会用到此函数。

0x05 注入类

GetThreadContext

功能：获取目标线程的上下文。

函数原型：

```
BOOL GetThreadContext(HANDLE hThread, LPCONTEXT lpContext);
```

参数介绍：

hThread：获取信息目标进程的线程句柄。（用 OpenThread 获取）

lpContext：一个用于接收信息的 CONTEXT 结构指针

返回值：如果成功，返回值不为零. 如果不成功，返回值为零

备注：返回一个给定线程的上下文结构。线程上下文结构中存储了所有线程信息，比如寄存器值和目前状态。

QueueUserAPC

功能：把一个 APC 对象加入到指定线程的 APC 队列中。

函数原型：

```
DWORD QueueUserAPC(  
    PAPCFUNC pfnAPC,  
    HANDLE hThread,  
    ULONG_PTR dwData  
);
```

参数介绍：

pfnAPC：指向一个用户提供的 APC 函数的指针

hThread：指定特定线程的句柄

dwData：指定一个被传到 pfnAPC 参数指向的 APC 函数的值。

返回值：成功返回非 0，失败返回 0

备注：用来在其他线程中执行代码，恶意代码有时会使用这个函数注入代码到其他进程。

VirtualAllocEx

功能：在指定进程的虚拟空间保留或提交内存区域，除非指定 MEM_RESET 参数，否则将该内存区域置 0。

函数原型：

```
LPVOID VirtualAllocEx(  
    HANDLE hProcess,  
    LPVOID lpAddress,  
    SIZE_T dwSize,  
    DWORD flAllocationType,  
    DWORD flProtect  
);
```

参数介绍：

hProcess：

申请内存所在的进程句柄。

lpAddress：

保留页面的内存地址；一般用 NULL 自动分配。

dwSize：

欲分配的内存大小，字节单位；注意实际分配的内存大小是页内存大小的整数倍

flAllocationType：申请空间的类型。

flProtect：申请空间的保护属性。

返回值：执行成功就返回分配内存的首地址，不成功就是 NULL

备注：一个内存分配的例程，支持在远程进程中分配内存。恶意代码有时会在进程注入中使用此函数。

VirtualProtectEx

功能：改变在特定进程中内存区域的保护属性。

函数原型：

```
BOOL VirtualProtectEx(  
    HANDLE hProcess, // 要修改内存的进程句柄  
    LPVOID lpAddress, // 要修改内存的起始地址  
    DWORD dwSize, // 页区域大小  
    DWORD flNewProtect, // 新访问方式  
    PDWORD lpfOldProtect // 原访问方式 用于保存改变前的保护属性 易语言要传址  
);
```

参数介绍：

hProcess：目的进程的句柄。

lpAddress：要修改内存的起始地址。

dwSize：要修改内存的大小

flNewProtect：保护选项

lpfOldProtect：必须指向一个有效的变量，如果是 NULL 或者无效的变量，函数将失败

返回值：如果成功，返回非零。失败返回零

备注：修改一个内存区域的保护机制，恶意代码可能会使用这个函数来将一块只读的内存节修改为可执行代码。

0x06 驱动类

DeviceIoControl

功能：直接发送控制代码到指定的设备驱动程序，使相应的移动设备以执行相应的操作。

函数原型：

```
BOOL WINAPI DeviceIoControl(  
    _In_ HANDLE hDevice,  
    _In_ DWORD dwIoControlCode,  
    _In_opt_ LPVOID lpInBuffer,  
    _In_ DWORD nInBufferSize,  
    _Out_opt_ LPVOID lpOutBuffer,  
    _In_ DWORD nOutBufferSize,  
    _Out_opt_ LPDWORD lpBytesReturned,  
    _Inout_opt_ LPOVERLAPPED lpOverlapped);
```

参数介绍：

hDevice，设备句柄

`dwIoControlCode`, 应用程序调用驱动程序的控制命令, 就是 `IOCTL_XXX` `IOCTLs`。

`lpInBuffer`, 应用程序传递给驱动程序的数据缓冲区地址。

`nInBufferSize`, 应用程序传递给驱动程序的数据缓冲区大小, 字节数。

`lpOutBuffer`, 驱动程序返回给应用程序的数据缓冲区地址。

`nOutBufferSize`, 驱动程序返回给应用程序的数据缓冲区大小, 字节数。

`lpBytesReturned`, 驱动程序实际返回给应用程序的数据字节数地址。

`lpOverlapped`, 这个结构用于重叠操作。针对同步操作, 请用 `ByVal As Long` 传递零值

返回值: 非零表示成功, 零表示失败。

备注: 从用户空间向设备驱动发送一个控制消息。此函数在驱动级的恶意代码中是非常普遍使用的, 因为他是一种最简单和灵活的方式, 在用户空间和内核空间之间传递信息。

LsaEnumerateLogonSessions

功能: 枚举当前系统上的登录会话

函数原型:

```
NTSTATUS NTAPI LsaEnumerateLogonSessions(  
    _Out_ PULONG LogonSessionCount,  
    _Out_ PLUID *LogonSessionList  
);
```

参数介绍: `LogonSessionCount`: 指向一个 `long` 型的指针, 返回 `LogonSessionList` 链表中的参数个数。

`LogonSessionList`: 指向 `LUID` 的指针, 返回登录会话标识符数组中的第一个成员的地址。

返回值: 成功返回 `STATUS_SUCCESS`, 失败返回相应的错误码

备注: 枚举当前系统上的登录会话, 往往是一个登录凭证窃取器使用的部分功能。

MmGetSystemRoutineAddress

功能: 返回特定函数的地址

函数原型:

```
PVOID MmGetSystemRoutineAddress(  
    _In_ PUNICODE_STRING SystemRoutineName  
);
```

参数介绍: `SystemRoutineName` : 函数名称

返回值: 成功则返回指定函数的地址, 否则返回 `NULL`。

备注: 与 `GetProcAddress` 类似, 但是这个函数是内核代码使用的。这个函数从另外一个模块中获取函数的地址, 但仅仅可以用来获得 `ntoskrnl.exe` 和 `hal.dll` 的函数地址。

0x07 其他

AdjustTokenPrivileges

功能：启用或禁止指定访问令牌的特权。

函数原型：

```
BOOL AdjustTokenPrivileges(  
    HANDLE TokenHandle,  
    BOOL DisableAllPrivileges,  
    PTOKEN_PRIVILEGES NewState,  
    DWORD BufferLength,  
    PTOKEN_PRIVILEGES PreviousState,  
    PDWORD ReturnLength  
);
```

参数介绍：

TokenHandle, 包含特权的句柄

DisableAllPrivileges, 禁用所有权限标志

NewState, 新特权信息的指针(结构体)

BufferLength, 缓冲数据大小, 以字节为单位的 *PreviousState* 的缓存区(sizeof)

PreviousState, 接收被改变特权当前状态的 Buffer

ReturnLength 接收 *PreviousState* 缓存区要求的大小

如果这个函数成功, 返回非 0. 失败返回 0. 为了确定这个函数是否修改了所有指定的特权, 可以调用 *GetLastError* 函数, 当这个函数返回下面的值之一时就代表函数成功:

ERROR_SUCCESS: 这个函数修改了所有指定的特权。

ERROR_NOT_ALL_ASSIGNED: 这个令牌没有参数 *NewState* 里指定一个或多个的权限。(一个或多个没有修改成功). 即使权限没有被修改。这个函数也可能成功(返回这个 error 值)表明 参数 *PreviousState* 被修改。

BitBlt

功能：对指定的源设备环境区域中的像素进行位块(bit_block)转换, 以传送到目标设备环境。

函数原型：

```
BOOL BitBlt(  
    int x,  
    int y,  
    int nWidth,  
    int nHeight,
```

```
CDC* pSrcDC,
int xSrc,
int ySrc,
DWORD dwRop
);
```

参数介绍:

X: 指定目标矩形的左上角的逻辑 x 坐标。

Y: 指定目标矩形的左上角的逻辑 y 坐标。

nWidth: 指定宽度(以逻辑单位)的目标矩形和源位图。

nHeight: 指定高度(以逻辑单位)目标矩形和源位图。

pSrcDC: 设置为标识设备上下文位图要复制的 CDC 对象的指针。它必须是 NULL, 如果

dwRop 指定不包括一个源的光栅操作。

xSrc: 指定源位图的左上角的逻辑 x 坐标。

ySrc: 指定源位图的左上角的逻辑 y 坐标。

dwRop: 指定要执行的光栅操作。光栅操作代码定义 GDI 如何组合在涉及一个当前画笔、一个可能的源位图和一个目标位图的输出操作的颜色

返回值: 返回非 0 表示成功, 0 表示失败

备注: 用来将图形数据从一个设备复制到另一个设备。间谍软件有时候会使用这个函数来捕获屏幕。这个函数也经常被编译器作为共享代码而添加。

CallNextHookEx

功能: 可以将钩子信息传递到当前钩子链中的下一个子程, 一个钩子程序可以调用这个函数之前或之后处理钩子信息。

函数原型 LRESULT WINAPI CallNextHookEx(

_In_opt_ HHOOK hhk,

In int nCode,

In WPARAM wParam,

In LPARAM lParam):

参数介绍:

hhk[可选]

说明: 当前钩子的句柄

nCode [in]

说明: 钩子代码; 就是给下一个钩子要交待的

传递给当前 Hook 过程的代码。下一个钩子程序使用此代码, 以确定如何处理钩的信息。

wParam[in]

说明: 要传递的参数; 由钩子类型决定是什么参数

wParam 参数值传递给当前 Hook 过程。此参数的含义取决于当前的钩链与钩的类型。

lParam[in]

说明: 要传递的参数; 由钩子类型决定是什么参数

lParam 的值传递给当前 Hook 过程。此参数的含义取决于当前的钩链与钩的类型。

返回值：返回这个值链中的下一个钩子程序。当前 Hook 过程也必须返回该值。返回值的含义取决于钩型。

备注：在由 SetWindowsHookEx 函数设置了挂钩时间的代码中使用。CallNextHookEx 函数会调用链上的下一个挂钩函数。分析调用 CallNextHookEx 的函数可以确定出 SetWindowsHookEx 设置挂钩的用意。

CertOpenSystemStore

功能：用来访问本地系统中的证书。

函数原型：

```
HCERTSTORE WINAPI CertOpenSystemStore(  
    HCRYPTPROV hProv,  
    LPCTSTR szSubsystemProtocol  
);
```

参数介绍：

hProv：CSP 句柄，NULL 为默认句柄，或者由 CryptAcquireContext 返回

szSubsystemProtocol：打开的系统存储区的名字。假如名字不为 CA，MY，ROOT，SPC 则新建一个证书存储区域，可以使用 CertEnumSystemStore 列出所有的已存在的系统存储区

返回值：NULL 表示失败，成功则返回证书句柄。

备注：用来访问在本地系统中的证书。

CreateMutex

功能：用来创建一个有名或无名的互斥量对象

函数原型：

```
HANDLE CreateMutex(  
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // 指向安全属性的指针  
    BOOL bInitialOwner, // 初始化互斥对象的所有者  
    LPCTSTR lpName // 指向互斥对象名的指针  
);
```

参数介绍：

lpMutexAttributes SECURITY_ATTRIBUTES，指定一个 SECURITY_ATTRIBUTES 结构，或传递零值（将参数声明为 ByVal As Long，并传递零值），表示使用不允许继承的默认描述符

bInitialOwner Long，如创建进程希望立即拥有互斥体，则设为 TRUE。一个互斥体同时只能由一个线程拥有

lpName String，指定互斥体对象的名字。用 vbNullString 创建一个未命名的互斥体对象。如已经存在拥有这个名字的一个事件，则打开现有的已命名互斥体。这个名字可能不与现有的事件、信号机、可等待计时器或文件映射相符

返回值：

Long，如执行成功，就返回互斥体对象的句柄；零表示出错。会设置 GetLastError。即使返回的是一个有效句柄，但倘若指定的名字已经存在，GetLastError 也会设为 ERROR_ALREADY_EXISTS

备注：创建一个互斥对象，可以被恶意代码用来确保在给定时刻只有一个实例在系统上运行。恶意代码经常使用固定名字为互斥对象命名，这样他们就可以成为一个很好的主机特征，来检测系统是否感染了这个恶意样本。

CryptAcquireContext

功能：连接 CSP，获得指定 CSP 的密钥容器的句柄；

函数原型：

```
BOOL WINAPI CryptAcquireContext(  
    __out    HCRYPTPROV *phProv,  
    __in     LPCTSTR pszContainer,  
    __in     LPCTSTR pszProvider,  
    __in     DWORD dwProvType,  
    __in     DWORD dwFlags  
);
```

参数介绍：

*phProv, CSP 句柄指针

pszContainer, 密钥容器名称，指向密钥容器的字符串指针；如果 dwFlags 为 CRYPT_VERIFYCONTEXT, pszContainer 必须为 NULL

pszProvider, 指向 CSP 名称的字符串指针；为 NULL，表示使用默认的 CSP

dwProvType, CSP 类型

dwFlags 标志位

返回值：操作成功返回 TRUE，否则返回 FALSE。

备注：这经常是恶意代码用来初始化使用 Windows 加密库的第一个函数。还有跟多其他函数是和加密相关的，绝大多数的函数都以 Crypt 开头。

EnbaleExecuteProtectionSupport

备注：一个未经文档化的 API 函数，用来修改宿主上的数据执行保护（DEP）设置，是的系统更容易被攻击。

FindResource

功能：确定指定模块中指定类型和名称的资源所在位置。

函数原型：

```
HRSRC WINAPI FindResource(  
    __In_opt_ HMODULE hModule,  
    __In_     LPCTSTR lpName,  
    __In_     LPCTSTR lpType  
);
```

参数介绍：

hModule: 处理包含资源的可执行文件的模块。NULL 值则指定模块句柄指向操作系统通常情况下创建最近过程的相关位图文件。

lpName: 指定资源名称。若想了解更多的信息，请参见注意部分。

lpType: 指定资源类型。若想了解更多的信息，请参见注意部分。作为标准资源类型。这个参数的含义同 EnumResLangProc\lpType。

返回值: 如果函数运行成功，那么返回值为指向被指定资源信息块的句柄。为了获得这些资源，将这个句柄传递给 LoadResource 函数。如果函数运行失败，则返回值为 NULL

备注: 用来在可执行文件和装在 DLL 程序中寻找资源的函数。恶意代码有时会使用资源来存储字符串，配置信息或者其他恶意文件。如果看到使用了这个函数，需要进一步检查恶意代码 PE 头中的.rsrc 节

FindWindow

功能: 检索处理顶级窗口的类名和窗口名称匹配指定的字符串

函数原型:

```
HWND FindWindow  
(  
    LPCSTR lpClassName,  
    LPCSTR lpWindowName  
);
```

参数介绍:

lpClassName

指向一个以 NULL 字符结尾的、用来指定类名的字符串或一个可以确定类名字符串的原子。如果这个参数是一个原子，那么它必须是一个在调用此函数前已经通过 GlobalAddAtom 函数创建好的全局原子。这个原子（一个 16bit 的值），必须被放置在 lpClassName 的低位字节中，lpClassName 的高位字节置零。

如果该参数为 null 时，将会寻找任何与 lpWindowName 参数匹配的窗口。

lpWindowName

指向一个以 NULL 字符结尾的、用来指定窗口名（即窗口标题）的字符串。如果此参数为 NULL，则匹配所有窗口名。

返回值:

如果函数执行成功，则返回值是拥有指定窗口类名或窗口名的窗口的句柄。

如果函数执行失败，则返回值为 NULL

备注: 在桌面上搜索打开窗口的函数，有时这个函数会在反调试技术中使用，来搜索 011yDbg 工具的窗口。

GetAsyncKeyState

功能: 用来判断函数调用时指定虚拟键的状态

函数原型:

```
SHORT GetAsyncKeyState(int nVirtKey);
```

参数介绍: nVirtKey:指定 256 个可能的虚拟键盘值中的一个.

返回值: 指定虚拟键瞬时的状态值, 它有四种返回值:

0——键当前未处于按下状态, 而且自上次调用 GetAsyncKeyState 后改键也未被按过;

1——键当前未处于按下状态, 但在此之前 (自上次调用 GetAsyncKeyState 后) 键曾经被按过;

-32768 (即 16 进制数&H8000) —— 键当前处于按下状态, 但在此之前 (自上次调用 GetAsyncKeyState 后) 键未被按过;

-32767 (即 16 进制数&H8001) —— 键当前处于按下状态, 而且在此之前 (自上次调用

GetAsyncKeyState 后) 键也曾经被按过。

备注: 用来确定一个特定键是否被输入。恶意代码有时会使用这个函数来实现一个击键记录器

GetDC

功能: 该函数检索一指定窗口的客户区域或整个屏幕的显示设备上下文环境的句柄, 以后可以在 GDI 函数中使用该句柄来在设备上下文环境中绘图。

函数原型: : HDC GetDC(HWND hWnd);

参数介绍: hWnd: 设备上下文环境被检索的窗口的句柄, 如果该值为 NULL, GetDC 则检索整个屏幕的设备上下文环境。

返回值: 如果成功, 返回指定窗口客户区的设备上下文环境; 如果失败, 返回值为 Null。

备注: 返回一个窗口或者是整个屏幕的设备上下文句柄。间谍软件经常使用这个函数来抓取桌面截屏

GetForegroundWindow

功能: 获取一个前台窗口的句柄

函数原型: HWND GetForegroundWindow(void);

参数介绍: 无

返回值: 返回值是一个前台窗口的句柄。在某些情况下, 如一个窗口失去激活时, 前台窗口可以是 NULL。

备注: 返回桌面目前正在处于前端的窗口句柄, 击键记录器普遍使用这个函数, 来确定用户正在往哪个窗口输入

GetKeyState

功能: 该函数检取指定虚拟键的状态。该状态指定此键是 UP 状态, DOWN 状态, 还是被触发的

函数原型: SHORT GetKeyState (int nVirtKey);

参数介绍: nVrtKey: 定义一虚拟键。若要求的虚拟键是字母或数字 (A~Z, a~z 或 0~9), nVirtKey 必须被置为相应字符的 ASCII 码值, 对于其他的键, nVirtKey 必须是一虚拟键码。若使用非英语键盘布局, 则取值在 ASCIIa~z 和 0~9 的虚拟键被用于定义绝大多数的字符键

返回值: 返回虚拟键的状态, 若高序位为 1, 则键处于 DOWN 状态, 否则为 UP 状态。

若低序位为 1, 则键被触发

备注：被击键记录器使用，来获取键盘上一个特定键的状态。

GetSystemDefaultLangId

功能：返回系统默认的语言设置

函数原型：LANGID GetSystemDefaultLangID(void);

参数介绍：无

返回值：返回本地系统的语言标识符

备注：返回系统默认语言设置的函数。这可以用来定制显示与文件名，作为对感染主机摘要信息的获取，这个函数也会被一些由“爱国主义”所驱动的恶意代码使用，从而对一些特定区域的系统进行感染。

GetTickCount

功能：返回从操作系统启动所经过的毫秒数

函数原型：DWORD GetTickCount(void);

参数介绍：无

返回值：返回从操作系统启动到当前所经过的毫秒数

备注：返回从启动后到当前时间的微秒数。这个函数有时在反调试技术中被使用来获取时间信息。此函数也经常有编译器添加并包含在许多可执行程序中，因此简单的在导入函数列表中看到这个函数只能提供少量的线索信息

GetVersionEx

功能：

函数原型：

```
BOOL WINAPI GetVersionEx(  
    _Inout_ LPOSVERSIONINFO lpVersionInfo  
);
```

参数介绍：lpVersionInfo：用于装载版本信息的结构。在正式调用函数之前，必须先将这个结构的 dwOSVersionInfoSize 字段设为结构的大小（148）

返回值：成功返回非 0，失败返回 0

备注：返回目前正在运行的 Windows 操作系统版本信息。可以被用来获取受害主机的摘要信息，或是在不同 Windows 版本中选择未经文档化结构的一些偏移地址。

IsDebuggerPresent

功能：确定调用进程是否由用户模式的调试器调试

函数原型：BOOL WINAPI IsDebuggerPresent(void);

参数介绍：无

返回值:

如果当前进程运行在调试器的上下文, 返回值为非零值。

如果当前进程没有运行在调试器的上下文, 返回值是零。

备注: 检查当前进程是否被调试, 经常在反调试技术中使用, 这个函数经常有编译器添加并包含在许多可执行程序中, 因此简单的在导入函数列表中看到这个函数, 只能提供少量的线索信息。

LoadLibrary

功能: 装载一个 dll 程序到进程中

函数原型: HMODULE WINAPI LoadLibrary(_In_ LPCTSTR lpFileName);

参数介绍: lpLibFileName 指定要载入的动态链接库的名称。采用与 CreateProcess 函数的 lpCommandLine 参数指定的同样的搜索顺序

返回值: 成功则返回库模块的句柄, 零表示失败

备注: 装载一个 dll 程序到进程中, 而这个程序在程序启动时还没有被装载。几乎每一个 Win32 程序都会导入这个函数。

LoadResource

功能: 该函数装载指定资源到全局存储器。

函数原型: HGLOBAL LoadResource (HMODULE hModule, HRSRC hResInfo);

参数介绍:

hModule: 处理包含资源的可执行文件的模块句柄。若 hModule 为 NULL, 系统从当前过程中的模块中装载资源。

hResInfo: 将被装载资源的句柄。它必须由函数 FindResource 或 FindResourceEx 创建。

返回值: 如果函数运行成功, 返回值是相关资源的数据的句柄。如果函数运行失败, 返回值为 NULL

备注: 从 PE 文件中装载资源进入到内存中。恶意代码有时候会使用资源来存储字符串, 配置信息或者其他恶意软件。

MapVirtualKey

功能: 该函数将一虚拟键码翻译(映射)成一扫描码或一字符值, 或者将一扫描码翻译成一虚拟键码

函数原型: UINT MapVirtualKey (UINT uCode, UINT uMapType)

参数介绍:

uCode: 定义一个键的扫描码或虚拟键码。该值如何解释依赖于 uMapType 参数的值。

uMapType: 定义将要执行的翻译。该参数的值依赖于 uCode 参数的值。取值如下:

0: 代表 uCode 是一虚拟键码且被翻译为一扫描码。若一虚拟键码不区分左右, 则返回左键的扫描码。若未进行翻译, 则函数返回 0。

1: 代表 uCode 是一扫描码且被翻译为一虚拟键码, 且此虚拟键码不区分左右。若未进行翻译, 则函数返回 0。

2: 代表 uCode 为一虚拟键码且被翻译为一未被移位的字符值存放于返回值的低序字中。死键 (发符号) 则通过设置返回值的最高位来表示。若未进行翻译, 则函数返回 0。

3: 代表 uCode 为一扫描码且被翻译为区分左右键的一虚拟键码。若未进行翻译, 则函数返回 0。

返回值: 返回值可以是一扫描码, 或一虚拟键码, 或一字符值, 这完全依赖于不同的 uCode 和 uMapType 的值。若未进行翻译, 则函数返回 0。

备注: 将一个虚拟键值代码转化成字符值, 经常被击键记录器恶意代码所使用

Module32First

功能: 检索与进程相关联的第一个模块的信息

函数原型:

```
BOOL WINAPI Module32First(  
    HANDLE hSnapshot,  
    LPMODULEENTRY32 lpme  
);
```

参数介绍:

hSnapshot

调用 CreateToolhelp32Snapshot 函数返回的快照句柄

lpme

MODULEENTRY32 结构的指针。用来返回数据

返回值: 成功返回 TRUE 失败返回 FALSE

备注: 用来枚举装载到进程中的模块, 注入器通常使用这个函数来确定注入代码的位置。

Module32Next

功能: 用来枚举装载到进程中的模块

函数原型:

```
BOOL WINAPI Module32Next(  
    _In_ HANDLE hSnapshot,  
    _Out_ LPMODULEENTRY32 lpme  
);
```

参数介绍:

hSnapshot

调用 CreateToolhelp32Snapshot 函数返回的快照句柄

lpme

MODULEENTRY32 结构的指针。用来返回数据

返回值: 如果模块链中的下一个模块拷贝到了缓冲区中, 就返回 TRUE, 否则返回 FALSE。

备注：用来枚举装载到进程中的模块，注入器通常使用这个函数来确定注入代码的位置。

NetScheduleJobAdd

功能：

函数原型：

```
NET_API_STATUS NetScheduleJobAdd(  
    _In_opt_ LPCWSTR Servername,  
    _In_      LPBYTE  Buffer,  
    _Out_     LPDWORD JobId  
);
```

参数介绍：

Servername：函数要执行的远程服务的 DNS 或 NetBIOS 名字

Buffer：执行 AT_INFO 结构的指针，描述了要提交的任务

JobId：新提交的任务 ID

返回值：成功返回 NERR_Success，失败返回系统错误码

备注：提交一个计划安排，来让一个程序在某个特定日期时间运行。恶意代码可以使用这个函数来运行一个其他程序，作为一位恶意代码分析师，需要定位与分析会在未来某个时间运行的程序。

OpenMutex

功能：为现有的一个已命名互斥体对象创建一个新句柄。

函数原型：

```
函数原型： HANDLE OpenMutex(  
    DWORD dwDesiredAccess, // access  
    BOOL bInheritHandle, // inheritance option  
    LPCTSTR lpName // object name  
);
```

参数介绍：

dwDesiredAccess：

MUTEX_ALL_ACCESS 请求对互斥体的完全访问

MUTEX_MODIFY_STATE 允许使用 ReleaseMutex 函数

SYNCHRONIZE 允许互斥体对象同步使用

bInheritHandle ：如希望子进程能够继承句柄，则为 TRUE

lpName ：要打开对象的名字

返回值：如执行成功，返回对象的句柄；零表示失败

备注：打开一个双向互斥对象的句柄，可以被恶意代码用来确保在任意给定时间在系统上只能有一个运行实例。恶意代码通常使用固定名字来为互斥对象命名，因此可以用作很好的主机特征。

OutputDebugString

功能：输出字符串到一个附加的调试器上。

函数原型：void WINAPI OutputDebugString(__in_opt LPCTSTR lpOutputString);

参数介绍：lpOutputString：需要输出的字符串

返回值：无

备注：可以被使用在反调试技术上。

SetWindowsHookEx

功能：设置一个挂钩函数，使其在某个特定时间触发时被调用。

函数原型：

```
HHOOK WINAPI SetWindowsHookEx(  
    __in int idHook, \\钩子类型  
    __in HOOKPROC lpfn, \\回调函数地址  
    __in HINSTANCE hMod, \\实例句柄  
    __in DWORD dwThreadId); \\线程 ID
```

参数介绍：

idHook：钩子类型

lpfn：回调函数地址

hMod：实例句柄

dwThreadId：线程 ID

返回值：若此函数执行成功，则返回值就是该挂钩处理过程的句柄；若此函数执行失败，则返回值为 NULL

备注：此函数普遍被击键记录器和间谍软件使用，这个函数也提供了一种轻易的方法，将一个 DLL 程序装载入系统的所有 GUI 进程中。这个函数有时也会被编译器添加。