**ReverseCore\_NoteBook**



目录

[13 PE文件结构 2](#_Toc493158103)

[13.1 PE头结构 5](#_Toc493158104)

[13.1.1 DOS头 5](#_Toc493158105)

[13.1.2 虚拟地址和文件偏移地址的换算（RVARAW） 5](#_Toc493158106)

[13.2 IAT（导入地址表） 6](#_Toc493158107)

[13.3 notepad.exe 练习 6](#_Toc493158108)

[14 运行时压缩（加壳） 7](#_Toc493158109)

[15 UPX调试脱壳 7](#_Toc493158110)

[15.1 跟踪代码和OEP定律 7](#_Toc493158111)

[15.2 修复IAT 8](#_Toc493158112)

[16 基地址重定位 8](#_Toc493158113)

[17 删除.reloc节 9](#_Toc493158114)

[18 Upack PE文件头详细分析 9](#_Toc493158115)

[19 Upack 调试 查找OEP 10](#_Toc493158116)

[20 内嵌补丁 10](#_Toc493158117)

[21 Windows消息钩取 10](#_Toc493158118)

[22 击键记录器 11](#_Toc493158119)

[23 Dll注入 11](#_Toc493158120)

[24 DLL卸载 14](#_Toc493158121)

[25 通过修改PE加载dll 14](#_Toc493158122)

[26 PE Tools 14](#_Toc493158123)

[27 代码注入 15](#_Toc493158124)

[28 使用汇编语言编写注入代码 20](#_Toc493158125)

[29 20](#_Toc493158126)

[30 记事本WriteFile()API钩取 20](#_Toc493158127)

[31 关于调试器 21](#_Toc493158128)

[32 计算器显示中文数字 21](#_Toc493158129)

[33 隐藏进程（Rookit） 21](#_Toc493158130)

[33.1 dll代码分析 21](#_Toc493158131)

[33.2 全局API钩取 23](#_Toc493158132)

[33.3 热补丁代码修改技术 24](#_Toc493158133)

[34 高级全局API钩取：IE连接控制 25](#_Toc493158134)

[35 25](#_Toc493158135)

[36 25](#_Toc493158136)

[37 26](#_Toc493158137)

[38 26](#_Toc493158138)

[39 26](#_Toc493158139)

[40 26](#_Toc493158140)

[41 26](#_Toc493158141)

[42 26](#_Toc493158142)

[43 26](#_Toc493158143)

[44 26](#_Toc493158144)

[45 TLS回调函数 26](#_Toc493158145)

[45.1 源代码分析 27](#_Toc493158146)

[45.2 TLS回调函数调试 27](#_Toc493158147)

[46 TEB（Thread Environment Block）线程环境块 27](#_Toc493158148)

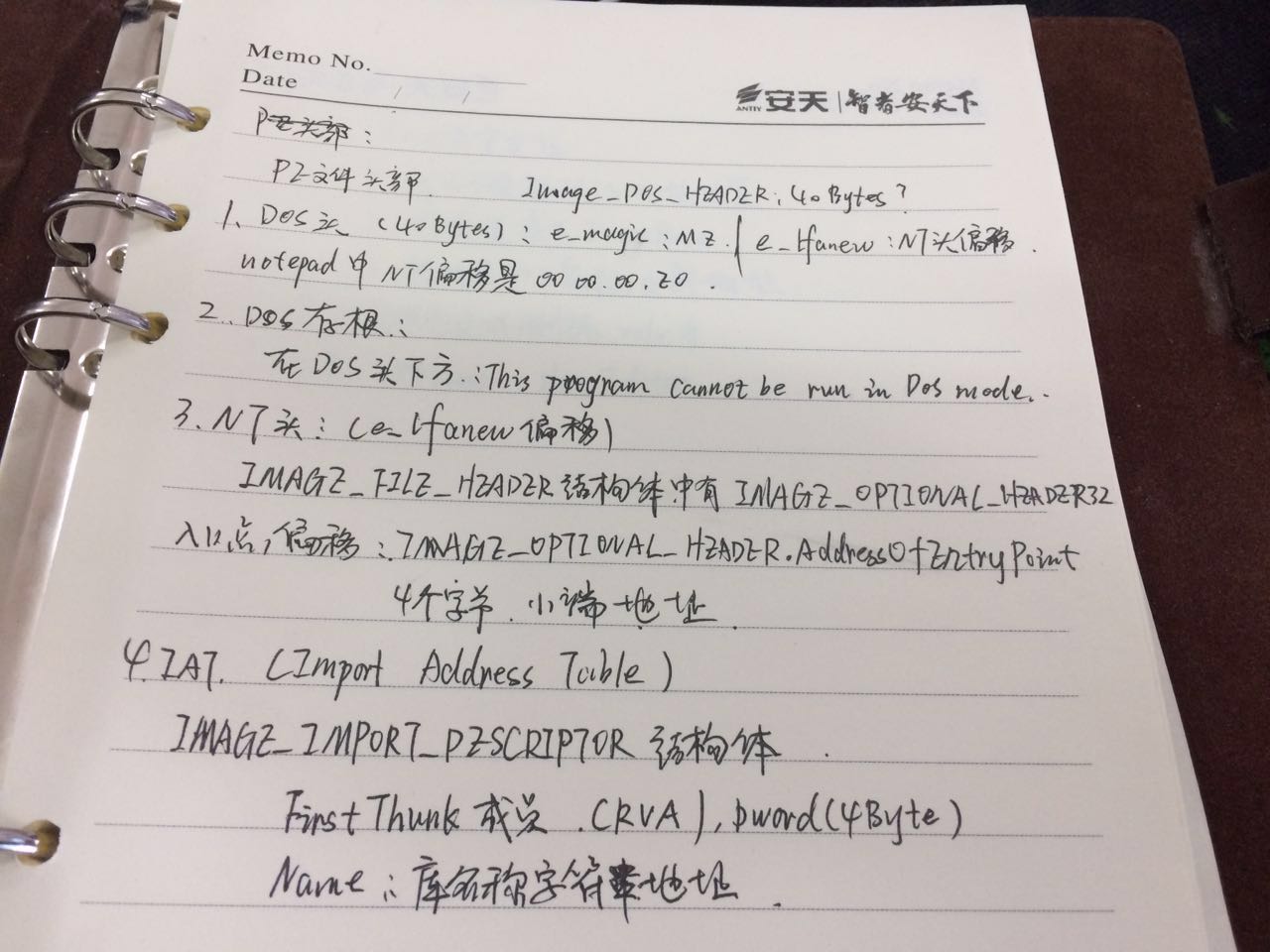
[47 PEB 28](#_Toc493158149)

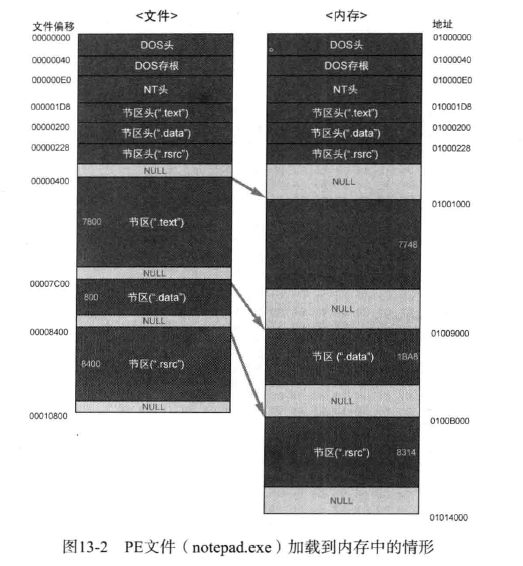
[48 SHE 错误!未定义书签。](#_Toc493158150)

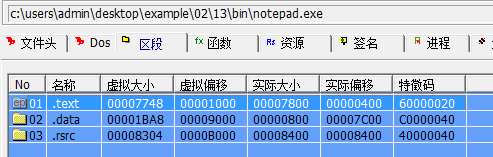
[48.1 代码调试 29](#_Toc493158151)

# PE文件结构

两图为notepad.exe的文件结构







名词解释：

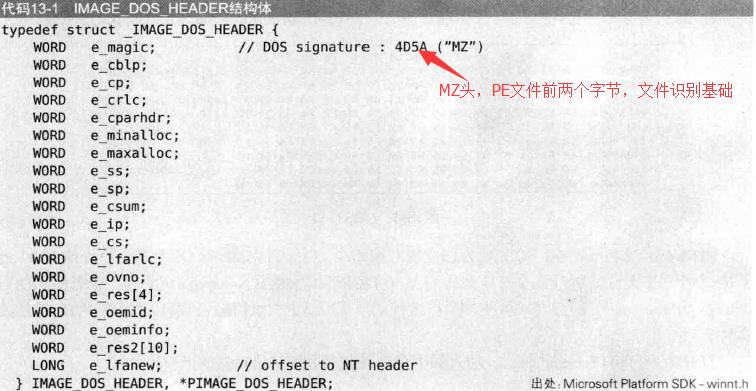
VA：进程虚拟内存的绝对地址

RVA：相对虚拟地址，从基准位置ImageBase开始的相对地址

VA = RVA + ImageBase

## PE头结构

### DOS头



DOS占用40个字节

### 虚拟地址和文件偏移地址的换算（RVARAW）



1. 查找RVA所在的区段

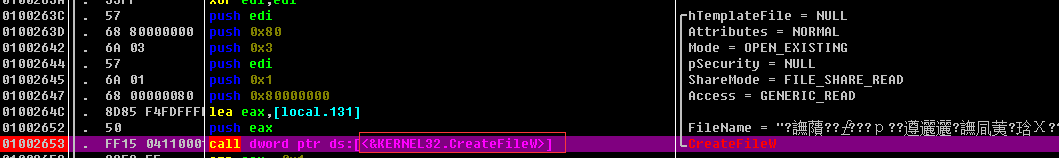
2. 利用公式计算RAW（文件偏移）

RAW – PointerToRawData = RVA – VirtualAddress（虚拟地址相对于虚拟偏移的位置等于文件偏移相当于当前节的文件偏移的位置）。

示例：RVA = 5000，求文件偏移

解： RVA = 5000，在1000和 9000之间，在text节，RAW = 5000 – 1000 + 400 = 4400.

## IAT（导入地址表）

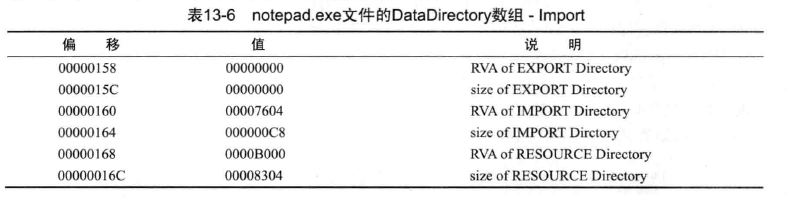


## notepad.exe 练习

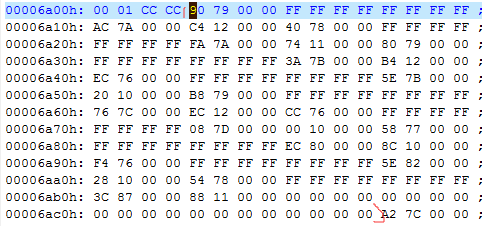
1. 从文件PE体中查找IMAGE\_OPTIONAL\_HEADER32.DataDirectory[1].

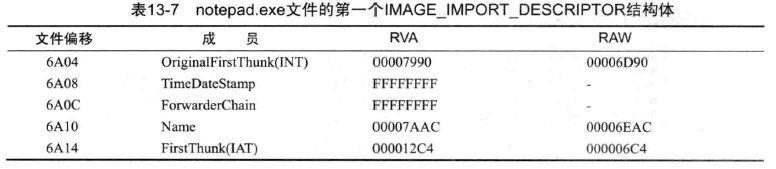
VirtualAddress（结构体数组的起始地址）。



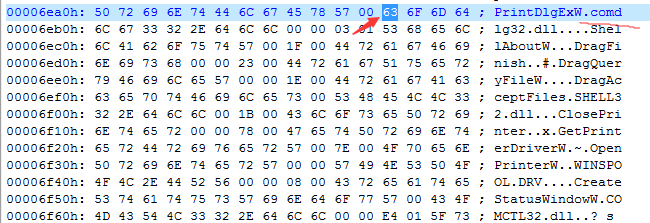


第一个字节是RVA，第二个字节是字节数。RVA = 00007604，size = 0xc8 。RAW = RVA – 1000 + 400（详见之前的公式） = 6A04





Name是导入函数的名称。 RVA ：7AAC =》RAW ：6EAC



# 运行时压缩（加壳）

UPX加壳特点：

1. PE头大小不变

2. 节名称有变化：.text => UPX0 | .data => UPX1

3. UPX0节的RawDataSize = 0

4. EP在第二个节区

5. .rsrc节大小几乎没有变化

# UPX调试脱壳

PUSHAD指令：

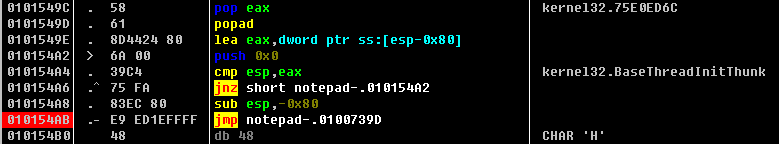
将EAX-EDI寄存器的值保存到栈中。

## 跟踪代码和OEP定律



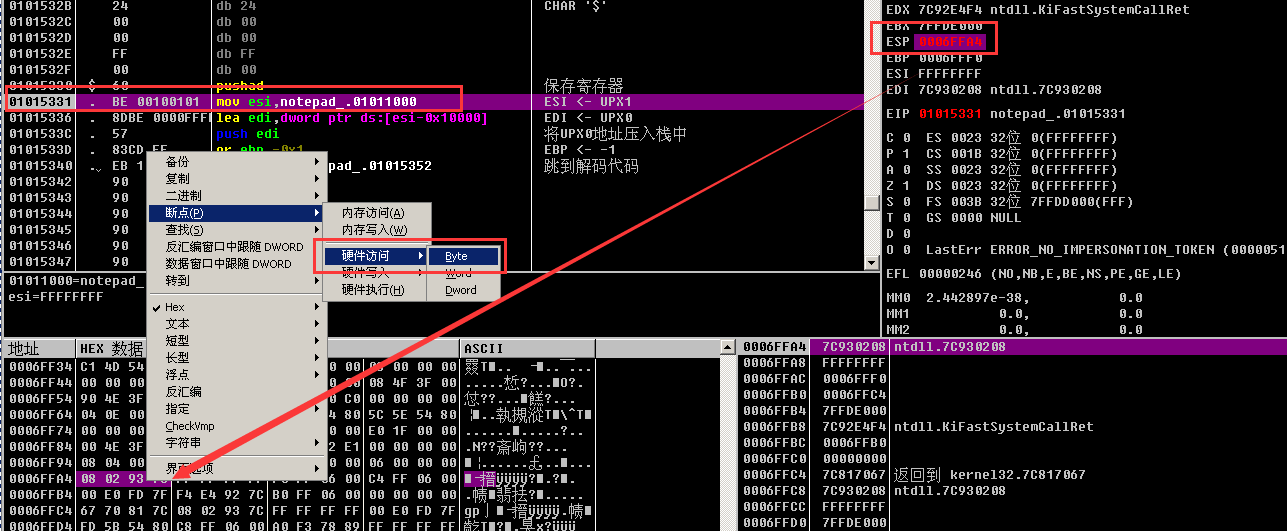
UPX快速脱壳方法：

1 popad 后面的jmp指令下断点



2 硬件断点（但是没有看明白）

在数据窗口找到执行完pushad后在esp对应地址，然后选择该地址开始的1字节，右键设置硬件断点（或者按书p133那样设置硬件断点均可）。



当程序执行popad时会首先从这个地址上读取数据到edi(pop edi)，此时会触发硬件断点，程序断下。其下方即是跳转到OEP的JMP指令。



不过和软件断点(INT 3)断点不同的是，触发断点的指令执行（popad）完成后程序才会停止运行，即程序会断在popad下一条指令。

OD的硬件断点可以在调试->硬件断点里看到。

## 修复IAT

ImportRCE工具和Ollydump

参考链接：<http://www.programlife.net/repair-import-table-through-importrec.html>

主要方法是，手动脱壳以后，用ollydump插件保存脱壳后的进程，然后记住入口点。Zzzzzz……还是看参考链接吧。

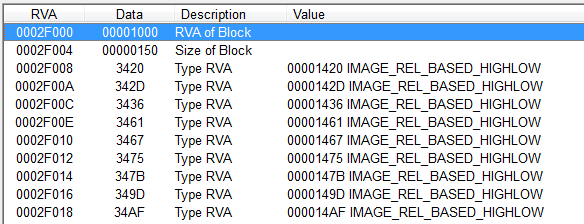
# 基地址重定位

Exe默认的ImageBase 是00400000，dll默认的ImageBase 是00100000 。

基址重定位表在PE头DataDirectory的第六个元素。



基址重定位表地址（RVA）为2f000。PEView 查看2f000地址。



TypeOffset的值是1000，块大小是0x150.



# 删除.reloc节

# Upack PE文件头详细分析

Upack 将MZ文件头和PE文件头进行重叠。

Stud\_PE查看文件：



PE头的地址是00000010，和DOS头重叠。



# Upack 调试 查找OEP

# 内嵌补丁

这章重新看。

# Windows消息钩取

Hookmain.cpp

#include *"stdio.h" //Hookmain.cpp*

#include *"conio.h"*

#include *"windows.h"*

#define DEF\_DLL\_NAME "KeyHook.dll"

#define DEF\_HOOKSTART "HookStart"

#define DEF\_HOOKSTOP "HookStop"

//typedef 关于函数指针的用法，typedef 返回值类型( \*新类型)(参数类型)

**typedef** void (\*PFN\_HOOKSTART)();

**typedef** void (\*PFN\_HOOKSTOP)();

void main()

{

HMODULE hDll = NULL;

PFN\_HOOKSTART HookStart = NULL;

PFN\_HOOKSTOP HookStop = NULL;

char ch = 0;

*// 加载KeyHook.dll*

hDll = LoadLibraryA(DEF\_DLL\_NAME);

**if**( hDll == NULL )

{

printf("LoadLibrary(%s) failed!!! [%d]", DEF\_DLL\_NAME, GetLastError());

**return**;

}

*// 获取导出两个导出函数地址，注意GetProcAddress前的函数指针和前面的typedef定义*

HookStart = (PFN\_HOOKSTART)GetProcAddress(hDll, DEF\_HOOKSTART);

HookStop = (PFN\_HOOKSTOP)GetProcAddress(hDll, DEF\_HOOKSTOP);

*// 开始钩取*

HookStart();

*// 等待用户输入q*

printf("press 'q' to quit!**\n**");

**while**( \_getch() != 'q' ) ;

*// 停止钩取*

HookStop();

FreeLibrary(hDll);

}

GetProcAddress()函数用法：第一个参数是LoadLibrary返回的dll句柄，第二个参数是字符串，要获取的函数名称。返回值与要获取的函数返回值同一类型，参见typedef定义。

# 击键记录器

这章没什么可说的。

# Dll注入

统计dll注入的几种方法：

1. 创建远程线程（调用CreateRemoteThread API）

CreateRemoteThread API用来在目标进程中执行创建出的线程。

1. 利用注册表（AppInit\_DLLs的键值）
2. 利用消息钩子

Dll注入代码示例

#include *"windows.h"*

#include *"tchar.h"*

BOOL SetPrivilege(LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)

{

TOKEN\_PRIVILEGES tp;

HANDLE hToken;

LUID luid;

**if**( !OpenProcessToken(GetCurrentProcess(),

TOKEN\_ADJUST\_PRIVILEGES | TOKEN\_QUERY,

&hToken) )

{

\_tprintf(L"OpenProcessToken error: %u**\n**", GetLastError());

**return** FALSE;

}

**if**( !LookupPrivilegeValue(NULL, *// lookup privilege on local system*

lpszPrivilege, *// privilege to lookup*

&luid) ) *// receives LUID of privilege*

{

\_tprintf(L"LookupPrivilegeValue error: %u**\n**", GetLastError() );

**return** FALSE;

}

tp.PrivilegeCount = 1;

tp.Privileges[0].Luid = luid;

**if**( bEnablePrivilege )

tp.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;

**else**

tp.Privileges[0].Attributes = 0;

*// Enable the privilege or disable all privileges.*

**if**( !AdjustTokenPrivileges(hToken,

FALSE,

&tp,

**sizeof**(TOKEN\_PRIVILEGES),

(PTOKEN\_PRIVILEGES) NULL,

(PDWORD) NULL) )

{

\_tprintf(L"AdjustTokenPrivileges error: %u**\n**", GetLastError() );

**return** FALSE;

}

**if**( GetLastError() == ERROR\_NOT\_ALL\_ASSIGNED )

{

\_tprintf(L"The token does not have the specified privilege. **\n**");

**return** FALSE;

}

**return** TRUE;

}

BOOL InjectDll(DWORD dwPID, LPCTSTR szDllPath)

{

HANDLE hProcess = NULL, hThread = NULL;

HMODULE hMod = NULL;

LPVOID pRemoteBuf = NULL;

DWORD dwBufSize = (DWORD)(\_tcslen(szDllPath) + 1) \* **sizeof**(TCHAR);//strlen函数的宽字符定义

LPTHREAD\_START\_ROUTINE pThrea dProc;

*// #1. 使用dwPID 获取目标进程notepad.exe的句柄.*

**if** ( !(hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, dwPID)) )

{

\_tprintf(L"OpenProcess(%d) failed!!! [%d]**\n**", dwPID, GetLastError());

**return** FALSE;

}

*// #2. 在目标进程notepad.exe的内存中分配内存，大小是szDllName*

pRemoteBuf = VirtualAllocEx(hProcess, NULL, dwBufSize, MEM\_COMMIT, PAGE\_READWRITE);

*// #3. 将myhack.dll路径写入分配的内存*

WriteProcessMemory(hProcess, pRemoteBuf, (LPVOID)szDllPath, dwBufSize, NULL);

*// #4. LoadLibraryA() 地址获取*

hMod = GetModuleHandle(L"kernel32.dll");

pThreadProc = (LPTHREAD\_START\_ROUTINE)GetProcAddress(hMod, "LoadLibraryW");

*// #5. 在notepad.exe中远程注入线程。*

hThread = CreateRemoteThread(hProcess, NULL, 0, pThreadProc, pRemoteBuf, 0, NULL);

WaitForSingleObject(hThread, INFINITE);

CloseHandle(hThread);

CloseHandle(hProcess);

**return** TRUE;

}

int \_tmain(int argc, TCHAR \*argv[])

{

**if**( argc != 3)

{

\_tprintf(L"USAGE : %s <pid> <dll\_path>**\n**", argv[0]);

**return** 1;

}

*// change privilege*

**if**( !SetPrivilege(SE\_DEBUG\_NAME, TRUE) )

**return** 1;

*// inject dll*

**if**( InjectDll((DWORD)\_tstol(argv[1]), argv[2]) )

\_tprintf(L"InjectDll(**\"**%s**\"**) success!!!**\n**", argv[2]);

**else**

\_tprintf(L"InjectDll(**\"**%s**\"**) failed!!!**\n**", argv[2]);

**return** 0;

}

进程代码注入的方法步骤如下：

1. 利用OpenProcess() 获得目标进程的句柄（PROCESS\_ALL\_ACCESS权限）
2. 将要注入的DLL路径写入目标进程内存，利用VirtualAllocEx ()在目标进程申请一块缓冲区
3. 利用WreProcessMemory()向要注入的dll路径写入缓冲区当中。
4. 获取LoadLibraryW()的地址，由于kernel32.dll在所有进程中加载的地址都是相同的，所以可以这么写。
5. 利用CreateRemoteThread()创建远程线程

# DLL卸载

DLL卸载技术的核心原理是驱使目标进程调用FreeLibrary()函数。

# 通过修改PE加载dll

# PE Tools

# 代码注入

*// CodeInjection.cpp*

*// reversecore@gmail.com*

*// http://www.reversecore.com*

#include *"windows.h"*

#include *"stdio.h"*

**typedef** **struct** \_THREAD\_PARAM //自定义结构体，存放要注入的代码。

{

FARPROC pFunc[2]; *// LoadLibraryA(), GetProcAddress()typedef int (FAR WINAPI \*FARPROC)()，参见typedef 的函数指针用法;*

char szBuf[4][128]; *// "user32.dll", "MessageBoxA", "www.reversecore.com", "ReverseCore"*

} THREAD\_PARAM, \*PTHREAD\_PARAM;

**typedef** HMODULE (WINAPI \*PFLOADLIBRARYA)

(

LPCSTR lpLibFileName

);

**typedef** FARPROC (WINAPI \*PFGETPROCADDRESS)

(

HMODULE hModule,

LPCSTR lpProcName

);

**typedef** int (WINAPI \*PFMESSAGEBOXA)

(

HWND hWnd,

LPCSTR lpText,

LPCSTR lpCaption,

UINT uType

);

DWORD WINAPI ThreadProc(LPVOID lParam)

{

PTHREAD\_PARAM pParam = (PTHREAD\_PARAM)lParam;

HMODULE hMod = NULL;

FARPROC pFunc = NULL;

*// LoadLibrary()*

hMod = ((PFLOADLIBRARYA)pParam->pFunc[0])(pParam->szBuf[0]); *// "user32.dll"*

**if**( !hMod )

**return** 1;

*// GetProcAddress()*

pFunc = (FARPROC)((PFGETPROCADDRESS)pParam->pFunc[1])(hMod, pParam->szBuf[1]); *// "MessageBoxA"*

**if**( !pFunc )

**return** 1;

*// MessageBoxA()*

((PFMESSAGEBOXA)pFunc)(NULL, pParam->szBuf[2], pParam->szBuf[3], MB\_OK);

**return** 0;

}

BOOL InjectCode(DWORD dwPID)

{

HMODULE hMod = NULL;

THREAD\_PARAM param = {0,};

HANDLE hProcess = NULL;

HANDLE hThread = NULL;

LPVOID pRemoteBuf[2] = {0,};

DWORD dwSize = 0;

hMod = GetModuleHandleA("kernel32.dll");

*// set THREAD\_PARAM*

param.pFunc[0] = GetProcAddress(hMod, "LoadLibraryA");

param.pFunc[1] = GetProcAddress(hMod, "GetProcAddress");

strcpy\_s(param.szBuf[0], "user32.dll");

strcpy\_s(param.szBuf[1], "MessageBoxA");

strcpy\_s(param.szBuf[2], "www.reversecore.com");

strcpy\_s(param.szBuf[3], "ReverseCore");

*// Open Process*

**if** ( !(hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, *// dwDesiredAccess*

FALSE, *// bInheritHandle*

dwPID)) ) *// dwProcessId*

{

printf("OpenProcess() fail : err\_code = %d**\n**", GetLastError());

**return** FALSE;

}

*// Allocation for THREAD\_PARAM*

dwSize = **sizeof**(THREAD\_PARAM);

**if**( !(pRemoteBuf[0] = VirtualAllocEx(hProcess, *// hProcess*

NULL, *// lpAddress*

dwSize, *// dwSize*

MEM\_COMMIT, *// flAllocationType*

PAGE\_READWRITE)) ) *// flProtect*

{

printf("VirtualAllocEx() fail : err\_code = %d**\n**", GetLastError());

**return** FALSE;

}

**if**( !WriteProcessMemory(hProcess, *// hProcess*

pRemoteBuf[0], *// lpBaseAddress*

(LPVOID)&param, *// lpBuffer*

dwSize, *// nSize*

NULL) ) *// [out] lpNumberOfBytesWritten*

{

printf("WriteProcessMemory() fail : err\_code = %d**\n**", GetLastError());

**return** FALSE;

}

*// Allocation for ThreadProc()*

dwSize = (DWORD)InjectCode - (DWORD)ThreadProc;

**if**( !(pRemoteBuf[1] = VirtualAllocEx(hProcess, *// hProcess*

NULL, *// lpAddress*

dwSize, *// dwSize*

MEM\_COMMIT, *// flAllocationType*

PAGE\_EXECUTE\_READWRITE)) ) *// flProtect*

{

printf("VirtualAllocEx() fail : err\_code = %d**\n**", GetLastError());

**return** FALSE;

}

**if**( !WriteProcessMemory(hProcess, *// hProcess*

pRemoteBuf[1], *// lpBaseAddress*

(LPVOID)ThreadProc, *// lpBuffer*

dwSize, *// nSize*

NULL) ) *// [out] lpNumberOfBytesWritten*

{

printf("WriteProcessMemory() fail : err\_code = %d**\n**", GetLastError());

**return** FALSE;

}

**if**( !(hThread = CreateRemoteThread(hProcess, *// hProcess*

NULL, *// lpThreadAttributes*

0, *// dwStackSize*

(LPTHREAD\_START\_ROUTINE)pRemoteBuf[1], *// dwStackSize*

pRemoteBuf[0], *// lpParameter*

0, *// dwCreationFlags*

NULL)) ) *// lpThreadId*

{

printf("CreateRemoteThread() fail : err\_code = %d**\n**", GetLastError());

**return** FALSE;

}

WaitForSingleObject(hThread, INFINITE);

CloseHandle(hThread);

CloseHandle(hProcess);

**return** TRUE;

}

BOOL SetPrivilege(LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)

{

TOKEN\_PRIVILEGES tp;

HANDLE hToken;

LUID luid;

**if**( !OpenProcessToken(GetCurrentProcess(),

TOKEN\_ADJUST\_PRIVILEGES | TOKEN\_QUERY,

&hToken) )

{

printf("OpenProcessToken error: %u**\n**", GetLastError());

**return** FALSE;

}

**if**( !LookupPrivilegeValue(NULL, *// lookup privilege on local system*

lpszPrivilege, *// privilege to lookup*

&luid) ) *// receives LUID of privilege*

{

printf("LookupPrivilegeValue error: %u**\n**", GetLastError() );

**return** FALSE;

}

tp.PrivilegeCount = 1;

tp.Privileges[0].Luid = luid;

**if**( bEnablePrivilege )

tp.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;

**else**

tp.Privileges[0].Attributes = 0;

*// Enable the privilege or disable all privileges.*

**if**( !AdjustTokenPrivileges(hToken,

FALSE,

&tp,

**sizeof**(TOKEN\_PRIVILEGES),

(PTOKEN\_PRIVILEGES) NULL,

(PDWORD) NULL) )

{

printf("AdjustTokenPrivileges error: %u**\n**", GetLastError() );

**return** FALSE;

}

**if**( GetLastError() == ERROR\_NOT\_ALL\_ASSIGNED )

{

printf("The token does not have the specified privilege. **\n**");

**return** FALSE;

}

**return** TRUE;

}

int main(int argc, char \*argv[])

{

DWORD dwPID = 0;

**if**( argc != 2 )

{

printf("**\n** USAGE : %s <pid>**\n**", argv[0]);

**return** 1;

}

*// change privilege*

**if**( !SetPrivilege(SE\_DEBUG\_NAME, TRUE) )

**return** 1;

*// code injection*

dwPID = (DWORD)atol(argv[1]);

InjectCode(dwPID);

**return** 0;

}

代码注入示例代码。

# 使用汇编语言编写注入代码

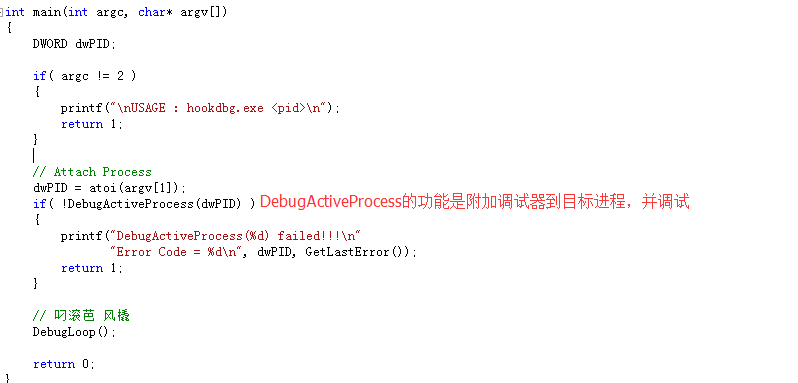
没看。

# 

# 记事本WriteFile()API钩取

这个API钩取的实际行为就是，在进程调用WriteFile()时，利用0xCC(INT 3)触发中断，然后通过相关进程操作函数修改buffer内容，达到挂钩子的效果。

调试过程。



总体步骤如下：

1. 通过调用DebugActiveProcess()将调试器附加到进程。
2. 调用WaitDebugEvent()等待调试事件，类似于WaitForSingleObject()。
3. 获取挂钩函数的地址（WriteFile()），并修改首地址为0xCC，INT 3，将进程的控制权交给调试器。
4. 进行钩子函数：
   1. 删除API钩子，具体方法为 将此前修改的进程函数首地址恢复原状
   2. 获取线程上下文，调用GetThreadContext() API将线程的上下文存储到ctx结构体当中。
   3. 获取挂钩函数（WriteFile）的第二和第三个参数（缓冲区地址和缓冲区大小）。
   4. 修改缓冲区内容，具体为malloc申请内存，修改缓冲区内容，释放申请的缓冲区。
   5. 修改线程上下文，将EIP修改为挂钩函数（WriteFile），然后调用SetThreadContext API
   6. 调用ContinuteDebugEvent继续运行进程。
   7. 设置钩子，方便下次钩取。

# 关于调试器

# 计算器显示中文数字

这一章比较繁琐，抽时间看。

# 隐藏进程（Rookit）

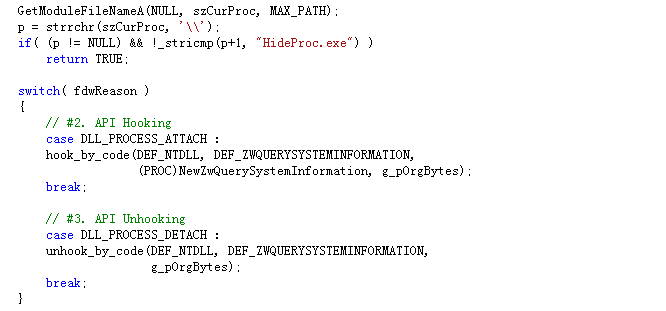
API钩取的原理：通过修改API代码的前五个字节，修改为JMP XXXXXXXX（JMP的指令是 E9），来转向Hook函数，从而实现API的钩取。

主要过程如下：

1. 向Process Explorer 进程注入dll文件。
2. 运行dll文件，文件代码钩取ntdll.ZwQuerySystemInformation（），具体方法是修改ZwQuerySystemInformation入口代码的前5个字节。Jmp到dll的代码区域，jmp的地址是dll文件中ZwQuerySystemInformation函数的地址。

## dll代码分析





发生DLL\_PROCESS\_ATTACH事件时使用hook\_by\_code函数。



函数参数：

LPCSTR szDLLName，包含要钩取API的dll名称，这里是ntdll；

LPCSTR szProcName 是要钩取的API名称，这里是ZwQuerySystemInformation

PROC pfnNew 是开发者希望程序跳转的 替代原有API的新函数的地址，这里是NewZwQuerySystemInformation。

PBYTE pOrgBytes，修改API时被覆盖的五个字节的缓冲区，用来脱钩使用。

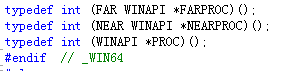
有个问题，在备份原有5字节代码时，为什么是memcpy(P0rgBytes, pByte, 5) 呢？

以下是问题解答：

GetProcAddress MSDN的定义

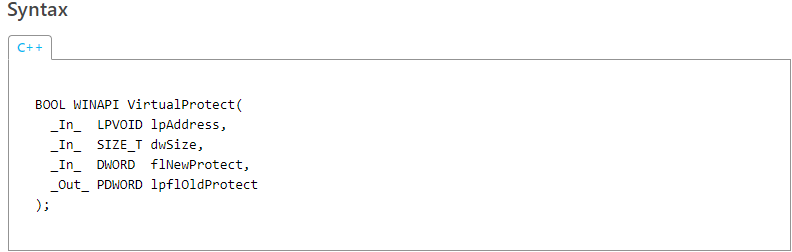


关于FARPROC函数类型的定义，第一个参数是GetModuleHandle 获取的dll文件句柄，第二个参数是字符串，要获取地址的函数名称。



参见函数指针的定义方法：typedef 返回值类型 ( \*新类型) (参数类型)

VirtualProtect函数相关的内容：



## 全局API钩取

Windows系统中，几乎所有新进程的父进程都是Explorer.exe。如果对Explorer.exe进程的CreateProcess API进行钩取，则可以钩取所有的新进程。

钩取CreateProcess() API，需要钩取CreateProcessA() , CreateProcessW()。也可以钩取比他们更为低级的的 ntdll.ZwResumeThread()。

钩取代码与之前分析过的相类似。

## 热补丁代码修改技术

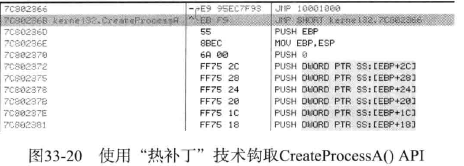
常用API起始代码的相似点：

1. 以MOV EDI， EDI 开始（指令代码8BFF）
2. API代码上方有5个NOP指令。

这7个字节的指令不进行任何操作，没有任何意义。

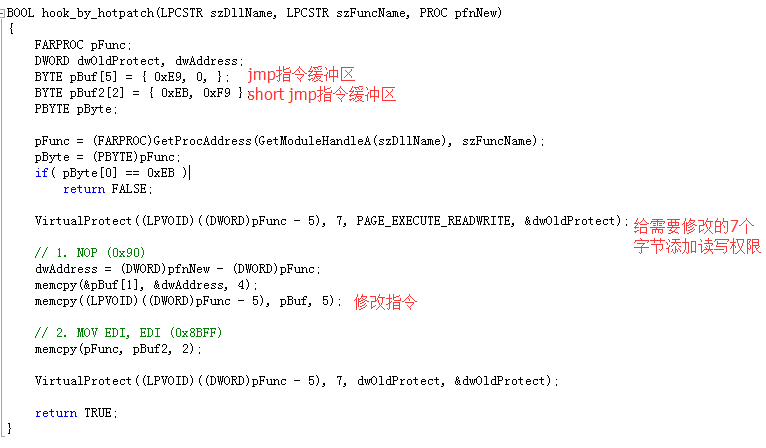
七字节代码钩取方法：

1. 首先将API起始代码前的5个字节修改为JMP指令（E9 XXXXXXXX），然后将MOV EDI， EDI指令修改为SHORT JMP（EB F9）、



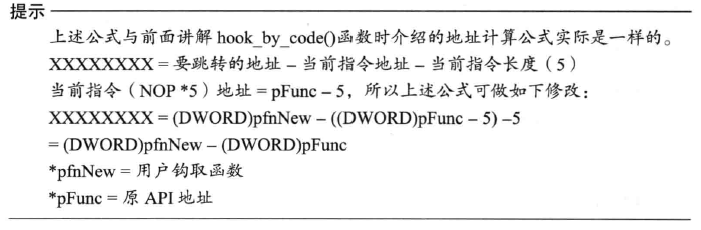
SHORT JMP指令只能在当前EIP -128~127范围内进行跳转。EB F9 即EIP向前跳5个字节。

1. 只需在API起始阶段的无意义代码做修改即可完成钩取，。



热补丁钩取API的过程：

1. 通过GetProcAddress 获取被钩取API的地址（CreateProcess）
2. 在进程空间中向要修改的7个字节添加“写”属性。内存块的起始地址是API地址 – 5，内存块大小是7个字节。
3. 计算跳转距离：dwaddress = pfnNew – pFunc，并且复制给5字节缓冲区



1. 复制指令。
2. 恢复内存属性

热补丁脱钩的过程：

七个字节修改成 5个NOP 和MOV EDI，EDI即可。



# 高级全局API钩取：IE连接控制

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# TLS回调函数

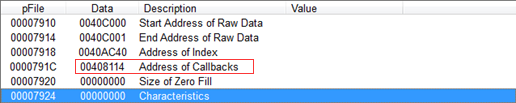
TLS技术，可以在线程内部独立使用或修改进程中的全局数据或者静态数据。当编程时使用TLS功能时，PE头文件中会存在TLS表。、



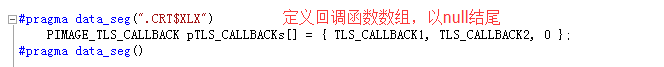
复习一下RAW⬄RVA，9310 在8000和A000之间，在.rdata节，TLS表的文件偏移为：

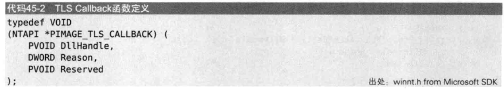
9310 (RVA) - 8000+6000 = 7910（RAW）





## 源代码分析





## TLS回调函数调试

# TEB（Thread Environment Block）线程环境块

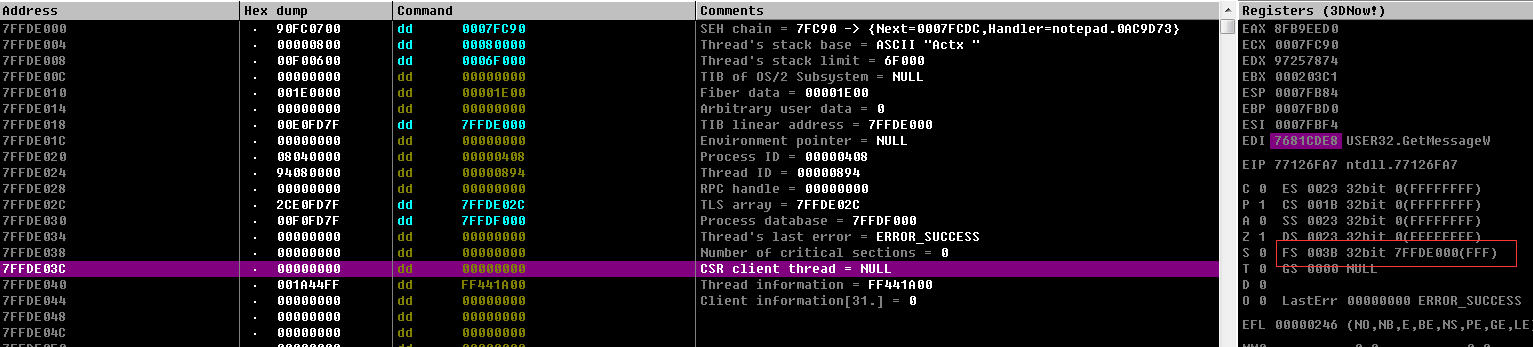
TEB是线程环境块，是一个用来保存线程的各种信息。进程中的每个线程都对应着一个结构体。

关于FS寄存器：FS寄存器存放的地址，是TEB结构体的地址。

MOV EAX, DWORD PTR FS:[18]// TEB地址

RET

上面的汇编代码 就是NtCurrentTeb的汇编代码。



# PEB

PEB 进程环境块，位于进程的FS：0x30位置。



在TEB结构体中含有PEB的地址信息。

可以使用多种方法访问PEB

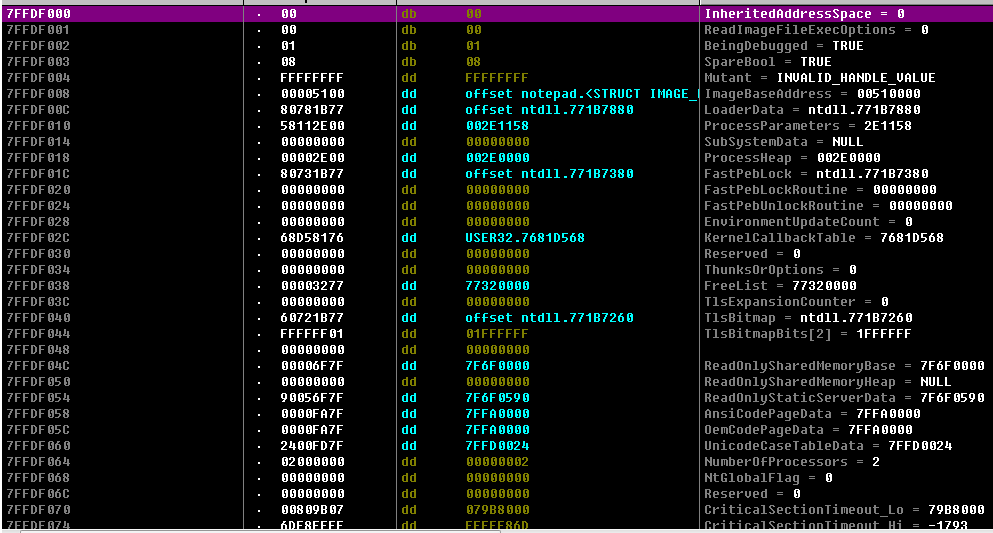
MOV EAX, DWORD PTR FS:[30]

\*\*\*\*\*

或者

MOV EAX, DWORD PTR FS:[18]// TEB地址

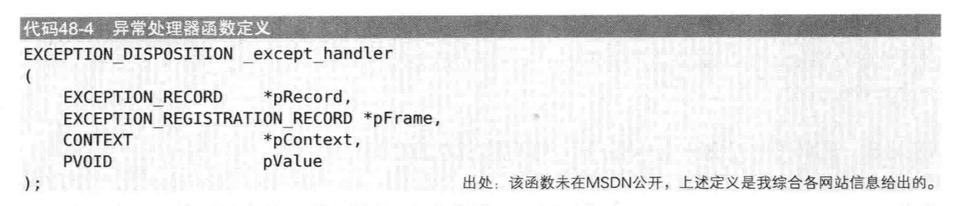
MOV EAX，DWORD PTR DS:[EAX+30] //PEB地址

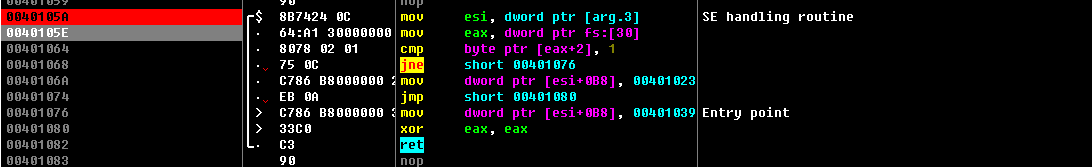


# SEH

SEH是Windows系统默认的异常处理机制。

## 代码调试



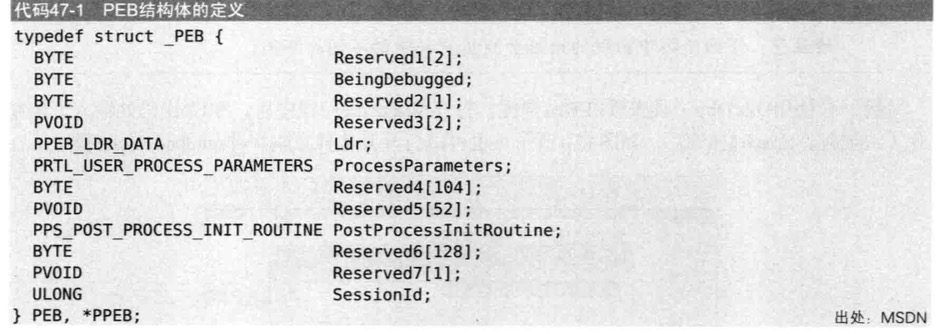


上面的图是异常处理器函数的参数，第一个参数是PRecord的指针，指向EXCEPTION\_RECORD结构体。第二个参数是SHE链的起始地址。第三个参数是指向pContext的指针，指向结构体。

汇编代码执行流程：

首先把第三个参数，也就是pContext结构体的指针传给ESI寄存器。

然后传PEB结构体的指针到EAX。

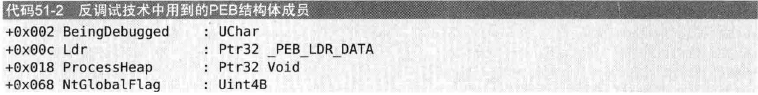


然后用PEB第二个成员，也就是BeingDebugged和1比较，判断是不是出于调试状态。

489页 SEH详细说明，好好看一下。

# 静态反调试技术

## PEB相关的调试技术



PEB结构体中与反调试相关的成员

复习：访问PEB结构体代码：

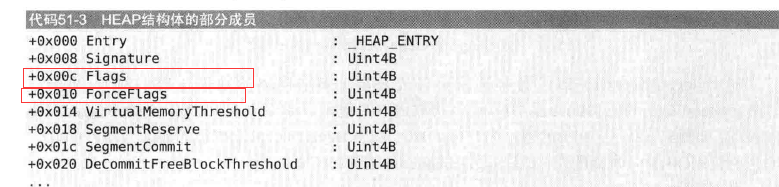
MOV EAX, DWORD PTR FS:[0X18] //TEB地址

MOV EAX, DWORD PTR DS[EAX+0X30]//PEB 地址

BeingDebugged：PEB结构体的第二个成员，当置为1时表示当前进程正在被调试。

Ldr：PEB结构体中成员，是一个指向堆区域的指针，当进程被调试时，未被调试的区域会出现0XFEEEFEEE。

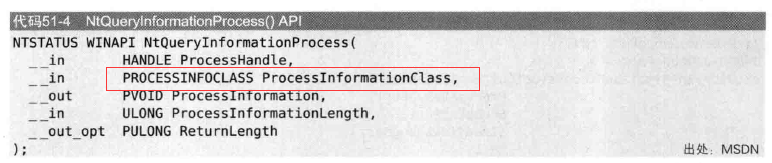
ProcessHeap：指向HEAP结构体的指针。



当进程被调试时，flags 和forceflags会被改变，进程正常运行时，flags=0x2，forceflags = 0x0.

NtGlobalflag ：在被调试的时候，会被设置成0x70，只要设置成0即可破解调试过程。

## NtQueryInformationProcess（）——调试器探测的API



ProcessInformation是一个枚举类型，是函数的输出结果，其中有特定值与调试器的探测相关。ProcessDebugPort （0x7），ProcessDebugObject-Handle（0x1E），ProcessDebugFlags（0x1F）。

ProcessDebugPort：非调试时设置为0，调试时设置为0xFFFFFFFF。

ProcessDebugObject-Handle：由于调试时会声称调试对象，所以该成员会在调试时存放调试句柄，非调试时句柄对象是NULL。

ProcessDebugFlags：调试状态下为0，非调试状态下为1.

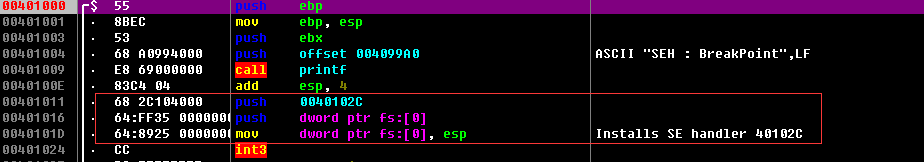
## NtQuerySystemInformation——调试环境检测的API

心里有点乱，冷静下来再看。

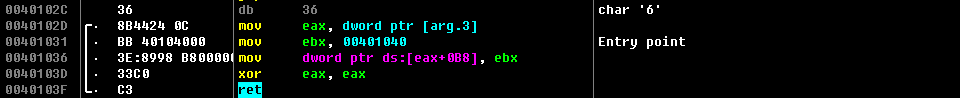
# 动态反调试技术

## 针对异常的调试

### SEH的调试



创建SEH，异常处理函数在40102C



上图为异常处理函数，arg.3是异常处理函数的第三个参数 \*pContext指针。



所以上一段代码的意思是吧401040赋值给pContext结构体的0xB8偏移位置，也就是EIP成员。

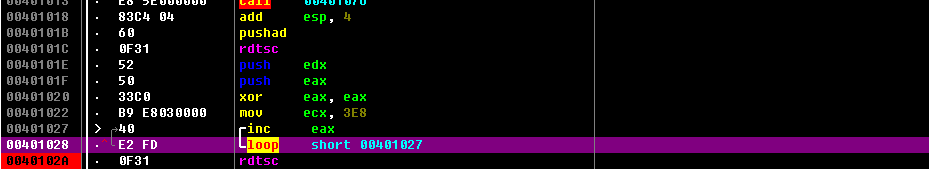
### SetUnhandledExceptionFilter——SEH未注册或不存在时的调试

好好重新看。

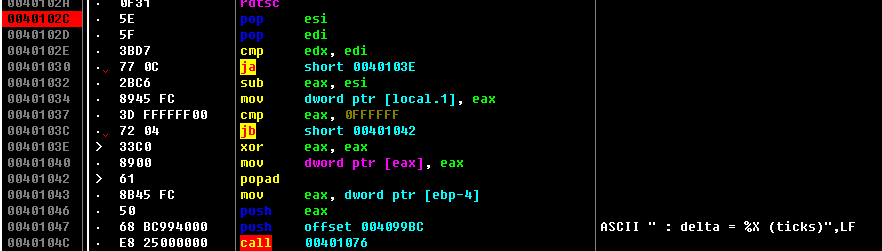
## Timing Check

RDTSC指令；获取时钟周期技术，并存放到TSC寄存器下。

TSC寄存器：64为寄存器，高32位保存在EDX，低32位保存在EAX。



抗调试代码，不断执行循环函数

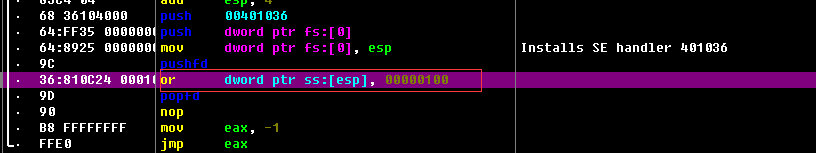


第二次RDTSC指令后，比较两次时间差。

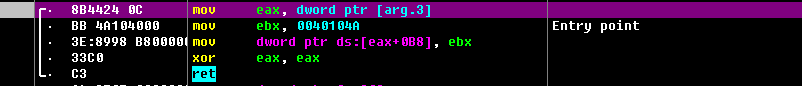
## 陷阱标志

### TF标志位

TF标志位：当值为1时，进入单步执行模式，但是CPU执行一条指令后将处罚一个EXCEPTION\_SINGLE\_STEP异常。可和SEH配合做反调试。



[ESP]和00000100或运算，就是把TF标志位置1.



异常处理函数，将40104A置换原有的eip，函数执行。

### INT 2D

INT 2D在调试模式下不会触发异常。但是会出现几个特征：

1. 忽略下条指令的第一个字节，将下一个指令变成新的指令
2. 会执行程序直到断点

### 0xCC检测

调试时会给API下断点，这时会有0xCC，如果API起始位置是0XCC则被调试。避开这种反调试的方法是：

1. 不在API首字节设置断点
2. 利用硬件断点，硬件访问是停止

另外一种是求校验和检测。如果进程的一段代码被下断点，校验和就会变化。

# 高级反调试技术

扰乱代码对齐。解密、PE保护器 remove OEP，API重定向。。。全都不明白。。。

# 注解：

1. 第七章：栈帧

EBP的作用：当程序运行时，ESP的值会随时变化，所以需要在调用函数时用栈帧去保存函数起始地址，作为基准点。所以在函数调用过程中，第一条指令基本上都是PUSH EBP（55），函数RET之前会POP EBP（5D）

1. 生成栈帧（EBP）的代码有两行

PUSH EBP

MOV EBP， ESP