看雪安全网站（汇编网站），《windows核心编程》，《Windows API开发详解 函数、接口、编程实例》，《windows程序设计》

工具基本操作

1. OllyICE

OllyICE中C：cpu窗口；M：module窗口；B:断点模式窗口；

Start是启动函数，程序最开始处，先做程序的初始化环境工作，\_main函数是程序的入口（debug模式是\_main\_0），跟c语言对应。窗口程序是WinMain()。

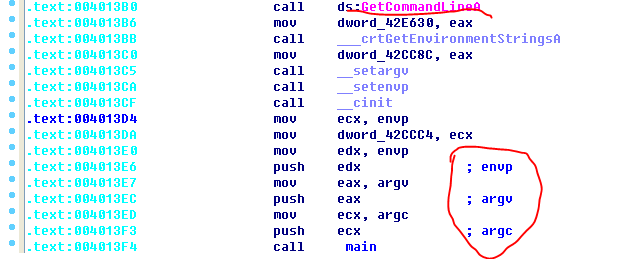
OD中F2设置断点，F9运行，ctrl+g查询地址快捷键，F7进入debug，f8逐条debug，db命令查看内存数据，bp命令添加断点，alt+F9执行debug到函数返回，-回退至上一位置，回车进入函数，ctrl+A是代码分析，分号加注释，

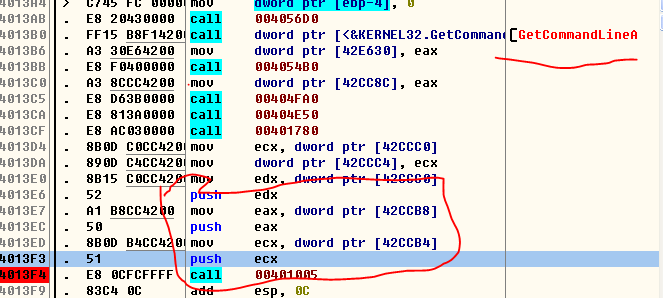
OD中，db ebp是查看地址为ebp的内存空间，db [ebp]是查看地址为ebp的值的内存空间，相当于db (db ebp)。

1. IDA

IDA中空格切换text view和graph view，esc退回前面位置，g跳转至地址，alt+回车进入函数，esc回退，F5还原成C代码，tab从F5切换到汇编码视图，分号加注释，

Getversion和getcommandline是C语言特征。Main函数入口前3个明显的连续push操作，对应main函数的3个基本参数。





MessageBoxA是常用特征函数！

汇编基础：

00401019 |. 8D7D BC lea edi, dword ptr [ebp-44]

1. Lea指令是将 ebp-44的结果赋值到edi（而不是ebp-44地址内存中的值）。
2. Rep是重复执行指令，重复执行rep后面跟随的指令ecx次
3. Stos dword ptr es:[edi]

Stos是串存储指令，将eax的值存入后面跟随的内存地址，同时内存地址+4。

该示例就是[edi]<-eax，edi+4

1. sp指向的数据是合法有用数据，先push，再上移。
2. 函数正常的返回值（高级编程语言的return值）存于eax寄存器。
3. 所有字符串都是存于内存，并且以首字符的地址来引用，00标识字符串的结尾。
4. 从汇编层次看是不区分地址和数据（字符串或者数值），而是根据操作指令来区分所操作的对象，需要字符串就将对象当做字符串来使用（有可能本身是数值）。

函数调用约定：

1. \_cdecl（c标准函数调用）

调用者清理堆栈，参数从右向左入栈

1. \_stdcall（标准调用

调用者不清理堆栈，被调用函数内清理堆栈，参数从右向左入栈

修改程序：

1. 双击寄存器，修改jmp指令的标记位；
2. 右键->二进制->用NOP填充；
3. 双击，修改指令。

保存修改后的程序：

右键->复制到可执行程序->所有修改->全部复制->保存文件。

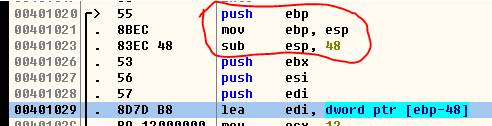
OD中查找字符串：

右键->查找->所有参考文本字串。

函数局部变量存入栈，地址为bp-xx，函数实参也存入栈，但是地址为bp+xx。

Call函数有3种方式，1种需要自己pop栈，即sp+4；第二种是自动pop栈。

每次进入函数都需要对栈进行操作，一般为对bp和sp。将旧bp压栈，sp赋值给bp，sp扩容：



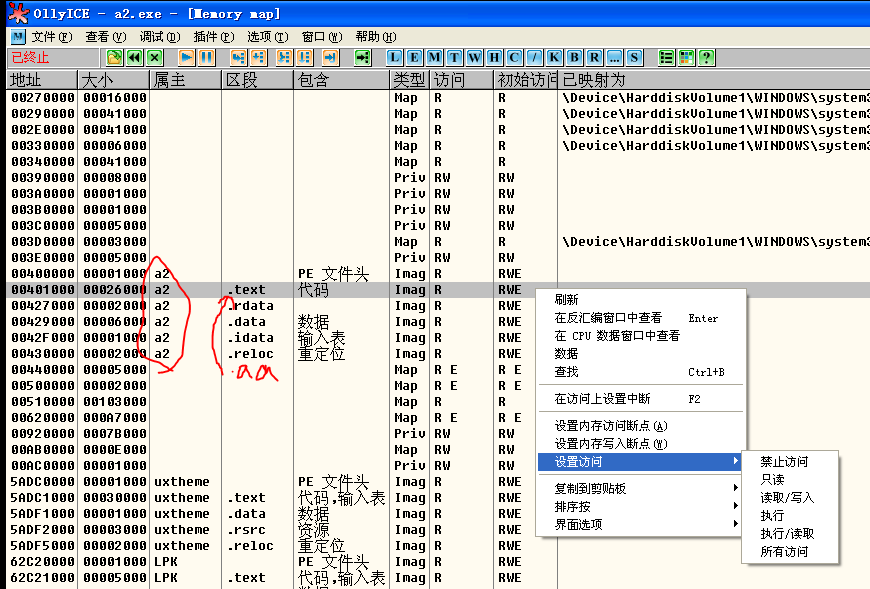
调用函数过程：先push进实参，然后call，call的时候将函数返回地址压入栈。再因为上一条函数初始化时的bp和sp操作，所以bp-xx是局部变量，bp+xx是实参，实参只能读，使用时先拷贝到局部变量。

OD中截取其中部分代码形成新程序：右键->Dump debugged process->dump。

壳包括3类：压缩壳，加密壳，虚拟壳（自带虚拟机，有自己指令的模式，复杂）。可以自动去壳也可以手动去壳。

手动去壳办法：

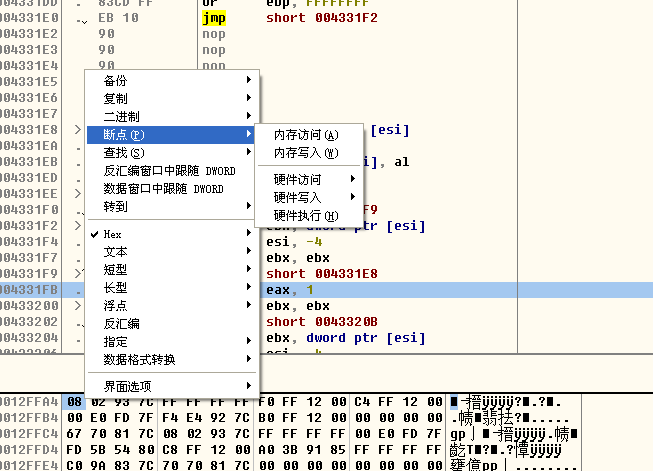
1. 程序启动后会先自动脱壳，所以在OD中查找到MessageBoxA等关键位置设置断点，然后dump程序。（这是进入main）
2. 寻找OEP(original entrance point，整个程序入口，main前面):
   1. 根据跨段指令
   2. 根据内存访问断点。加壳后的程序先运行加密的段，然后执行正常的代码段，在此时设置执行断点。该方法要求存在一个新的加密段。
      1. 在module窗口中找到属于本程序的代码段并设置执行断点（解密时也会访问，所以不能设施访问断点）。右键->设置访问->执行。（在下图中的实例是无效的，加密过程没有写在一个新的段中，程序开始位置已经在.text的代码段中，即已经执行到代码段中，所以设置访问断点无效。）



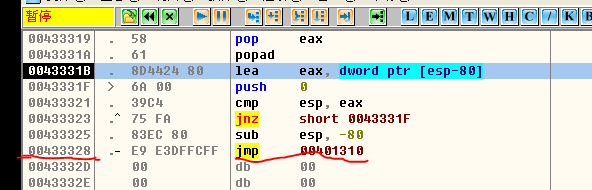
* 1. 根据对战平衡原理（pushad/popad等）。Pushad和popad必须一一对应，所以找到popad的位置就找到了eop。
     1. 执行到pushad（pushad后一句），在esp上右键->数据窗口中跟随，查看数据窗口。

实际上不一定全是pushad命令，只要是找到esp改变命令就是加壳脱壳过程（但是不一定是第一个）。

* + 1. 在数据窗口中esp地址的数据上设置硬件断点，右键->断点->硬件访问->xxx。（在B窗口中看不到该断点）



* + 1. 执行程序，程序会中断在popad后一句。Popad后不远的一个跨段jmp指令即是eop。

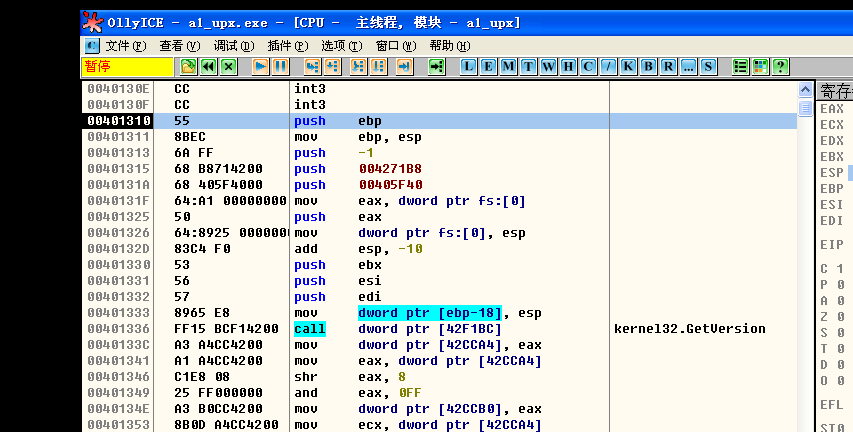


* + 1. 如果是加密壳，这里显示的可能仍然是加密代码，ctrl+A重新分析代码即可。
  1. 根据GetCommandLineA等特征函数（VC特性），操作方式同寻找MessageBoxA。

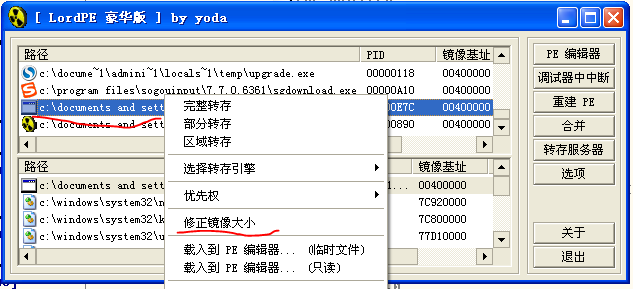
如果在oep位置dump程序，dump出来的程序可以执行，并且是去壳后的版本。

Dump程序修复过程（1.dump还原后的程序;2.如果出错再修复）：

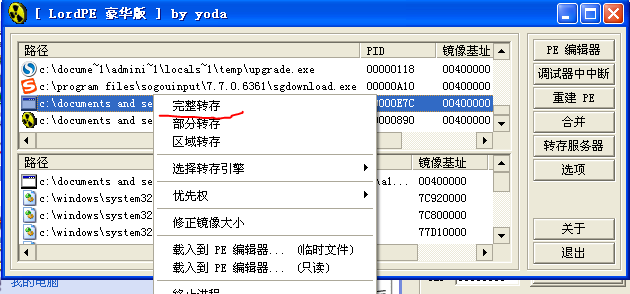
1. 找到oep地址；

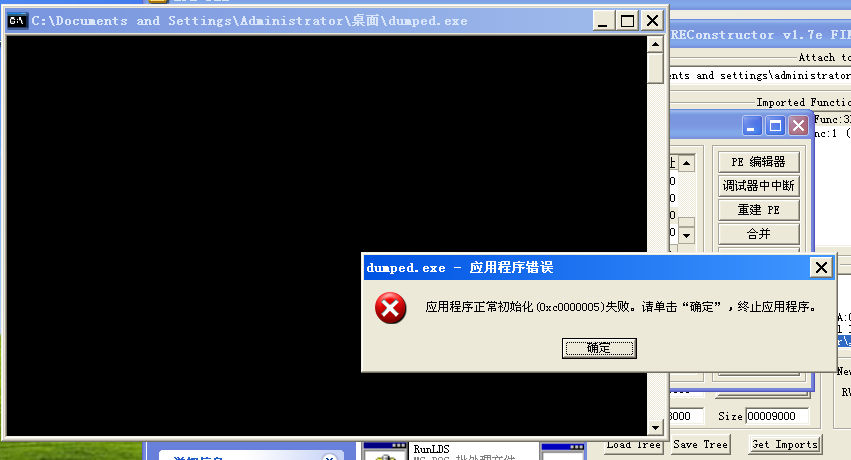


1. 启动loadPe，path中找到a1\_upx.exe.
2. 右键->修正镜像大小；

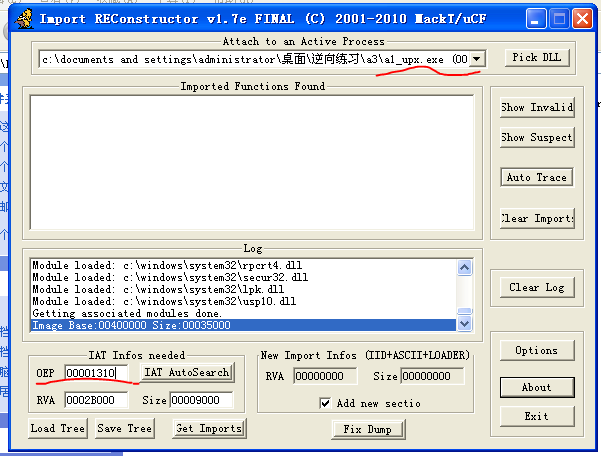


1. 完整转存，此时运行出错；

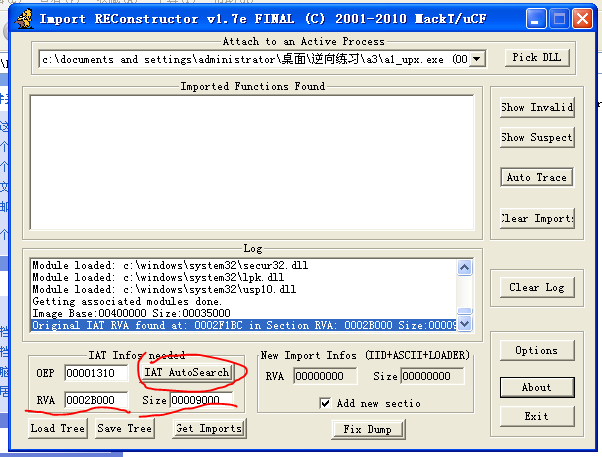




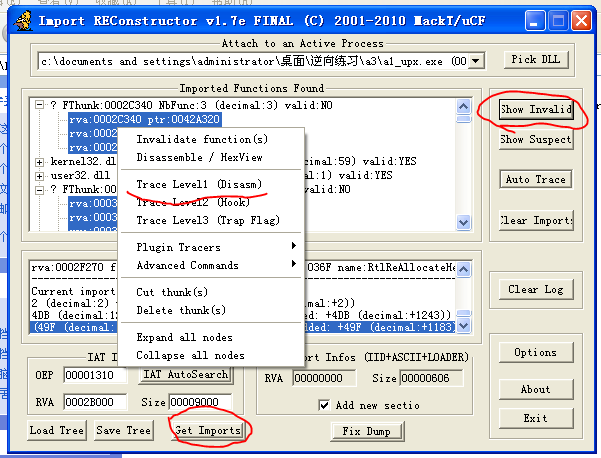
1. 启动importREC，找到对应的程序，根据之前找到的真实OEP修改OEP地址（只需要段内偏移即可）；



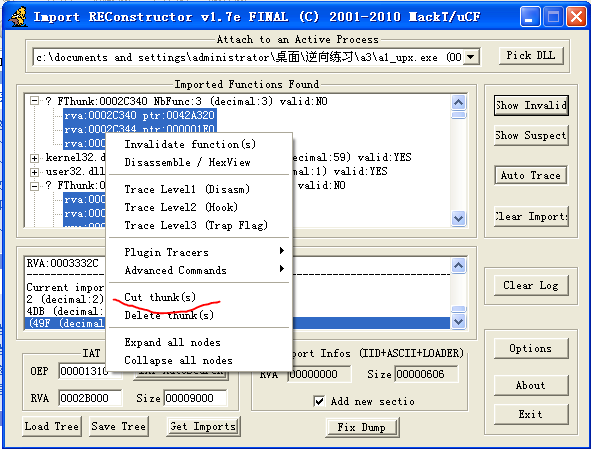
1. 点击IAT AutoSearch，根据提示修改RVA和Size；



1. 点击Get Imports，点击Show Invalid，对错误进行修复，右键->Trace Level1/2/3；



1. 再次点击Show Invalid，剪切掉还错误的部分；



1. 点击Fix Dump，对之前dump出的dump.exe进行修复，新生成的dump\_.exe即可成功运行。

反调试：

IsDebuggerPresent()函数通过检测 struct \_PEB中的UCHAR BeingDebugged字段判断是否属于调试模式。

PEB的地址为fs:[0x30]，BeingDebugged字段位置为fs:[0x30]+2。

1. 直接修改该内存位置数值为00即可。
2. OD中的IsDebugPresent插件。
3. OD中的HideOD插件->Option->HideNtDebugBit->OK。
4. 在汇编代码中找到判断的地方，修改。

反虚拟机：

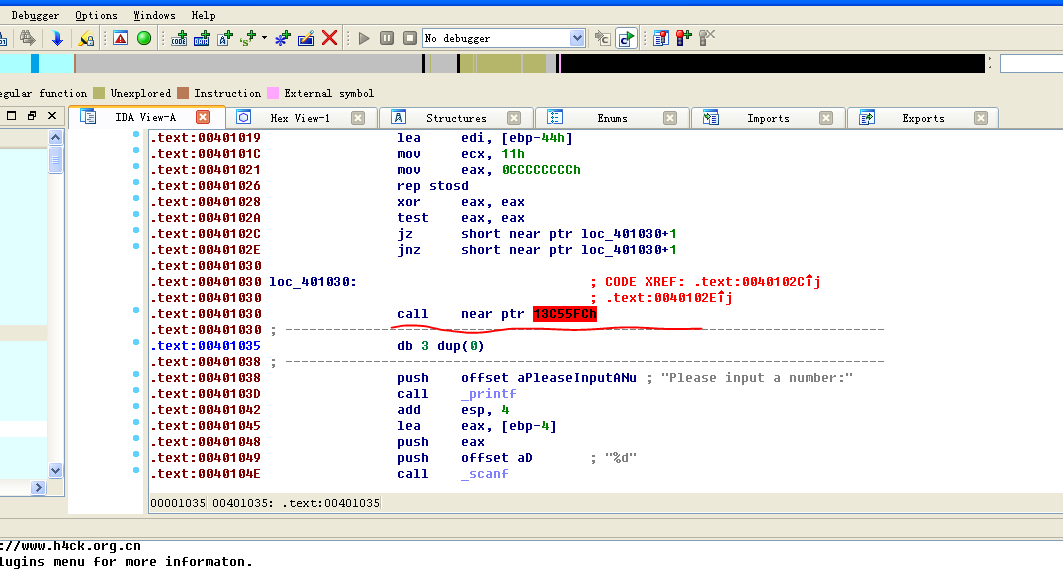
PE文件区段：

|  |  |
| --- | --- |
| 名称 | 描述 |
| .text | 默认的代码区块,它的内容全是指令代码,链接器把所有目标文件的text块连接成一个大的.text块，使用Borland C++,编译器产生的代码存放在CODE的区域里 |
| .data | 默认的读/写数据块,全局变量,静态变量一般放在这个区段 |
| .rdata | 默认只读数据区块,但程序中很少用到该块中的数据，一般两种情况用到,一是MS 的链接器产生EXE文件中用于存放调试目录，二是用于存放说明字符串，如果程序的DEF文件中指定了DESCRIPTION，字符串就会出现在rdata中 |
| .idata | 包含其他外来的DLL的函数及数据信息,即输入表，将.idata区块合并成另一个区块已成为一种惯例，典型的是.rdata区块,默认的，链接器只在创建一个Release模式的可执行文件时才能将idata合并到另外一个区块中 |
| .edata | 输出表，当创建一个输出API或数据的可执行文件时，连接器会创建一个.EXP文件，这个.EXP文件包含一个.edata区块，其会被加载到可执行文件中，经常被合并到.text或.rdata 区块中 |
| .rsrc | 资源,包括模块的全部资源，如图标，菜单，位图等，这个区块是只读的，无论如何不应该吧它命名为.rsrc以外的名字，也不能合并到其他的区块里 |
| .bss | 未初始化的数据,很少在用，取而代之的是执行文件的.data区块的的VirtualSize被扩展大的空间里用来装未初始化的数据. |
| .crt | 用于C++ 运行时(CRT)所添加的数据 |
| .tls | TLS的意思是线程局部存储器，用于支持通过\_declspec(thread)声明的线程局部存储变量的数据，这包括数据的初始化值，也包括运行时所需要的额外变量 |
| .reloc | 可执行文件的基址重定位，重定位一般仅Dll需要的 |
| .sdata | 相对于全局指针的可被定位的 短的读写数据 |
| .pdata | 异常表,包含CPU特定的IAMGE\_RUNTIME\_FUNTION\_ENTRY结构数组，DataDirectory中的IMAGE\_DIRECTORY\_ENTRY\_EXCEPTION指向它. |
| .didat | 延迟装入输入数据，在非Release模式下可以找到 |

花指令：对程序正常执行无用，目的是干扰逆向分析的指令。分为可执行和不可执行两类。反汇编算法分为：线性分析算法和递归下降分析算法。

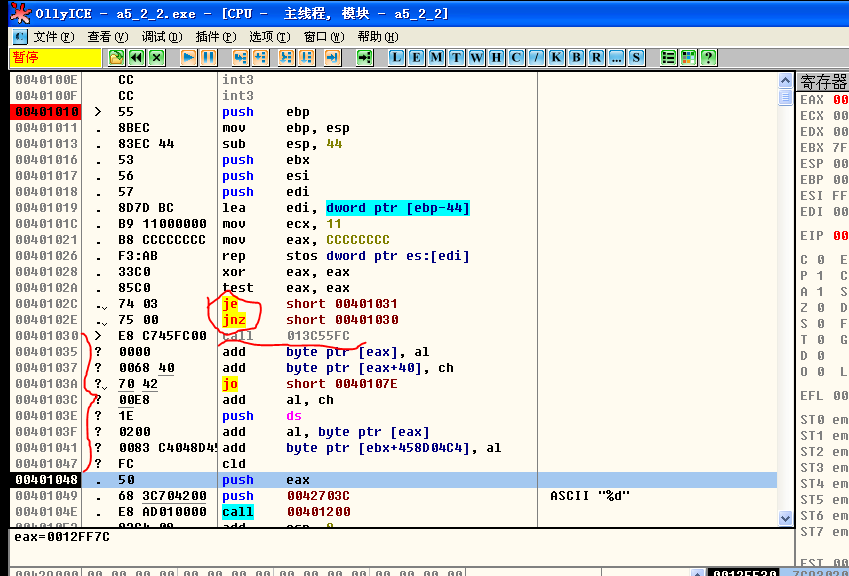
OD中右键->分析代码是递归分析，右键->从模块中删除分析是线性分析方法。

IDA解析错误(花指令)分析：



红色代表错误，表示将该指令解析为call命令时，call的对象解析不正确。解决办法是在OD中将该花指令用nop填充，重新保存新的pe。

线性分析和递归分析都出错的情况：

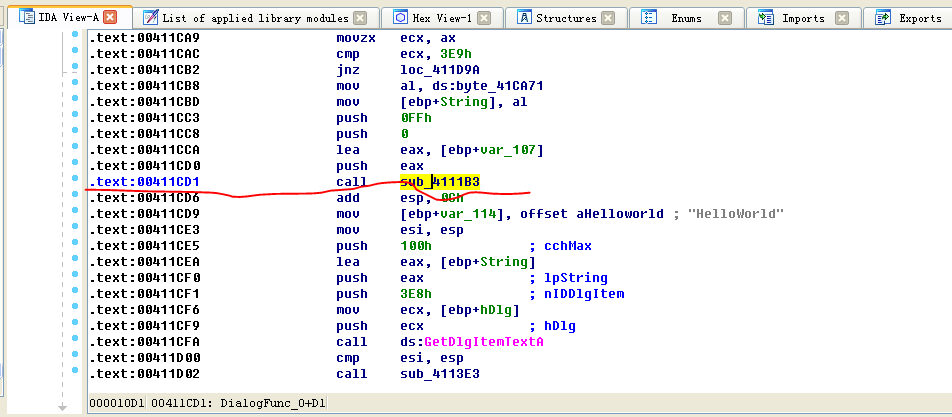


灰色代表永远不会执行，按照0040102c位置je的递归分析是正确的，但是永远不会执行到的jnz的递归分析覆盖了je的结果了。Jnz的递归分析将e8解释为call函数。

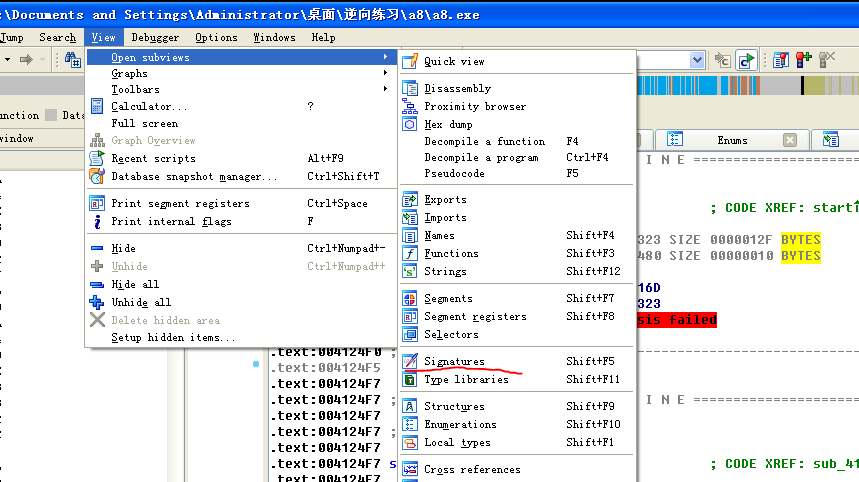
**Pe中常用的资源函数：GetDlgItemTextA(ascii字符)，GetWindowTextW(Unicode字符)，MessageBoxA和MessageBoxW**

IDA添加签名

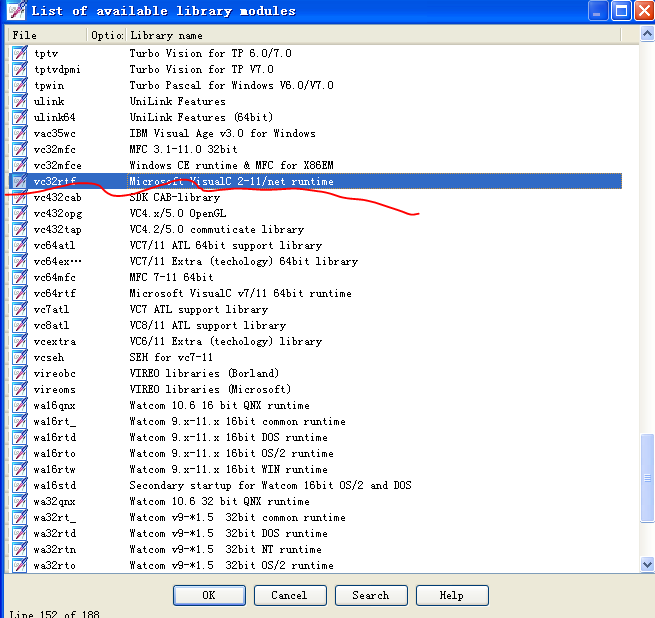
A8实验在IDA中加签名：该call函数的地址未能解析出名称。



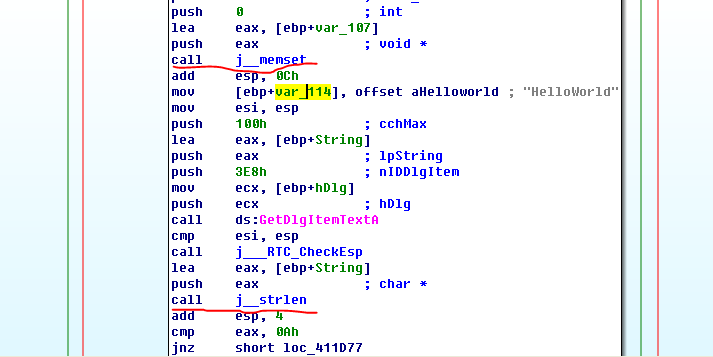
1. 打开signature view。



1. 添加signature，右键->apply new signature->vc runtime.

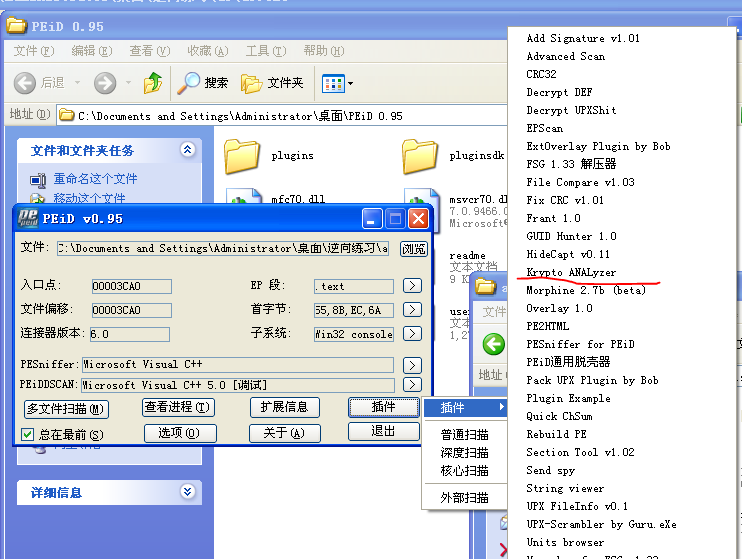


1. 添加signature后可显示vc runtime函数名，便于分析。



汇编中算法的识别：

1. PEid插件识别。



逆向关键函数：

GetWindowTextA //获得文本框中的数据

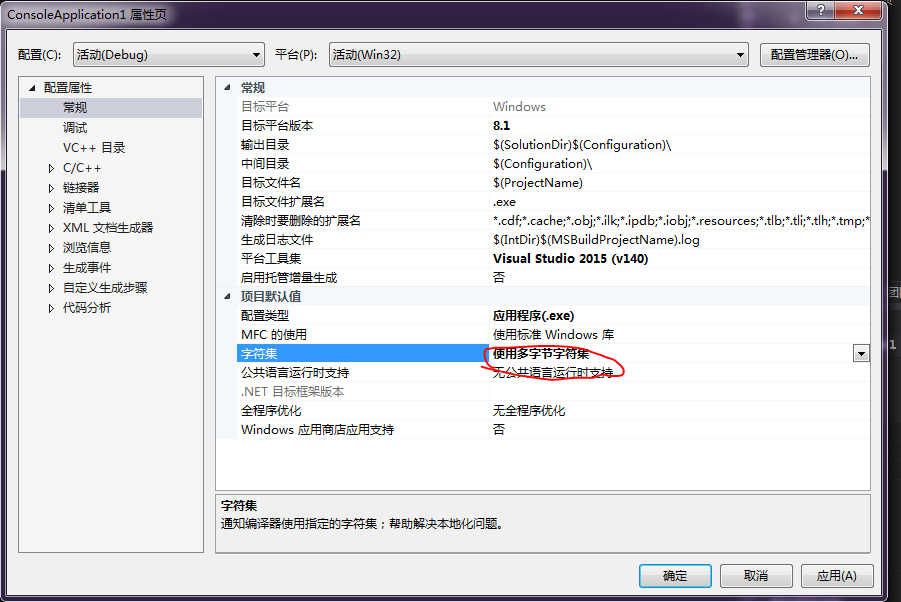
GetDlgItemTextA //获得对话框中的数据

MessageBoxA //弹出消息框显示消息

VC编程：

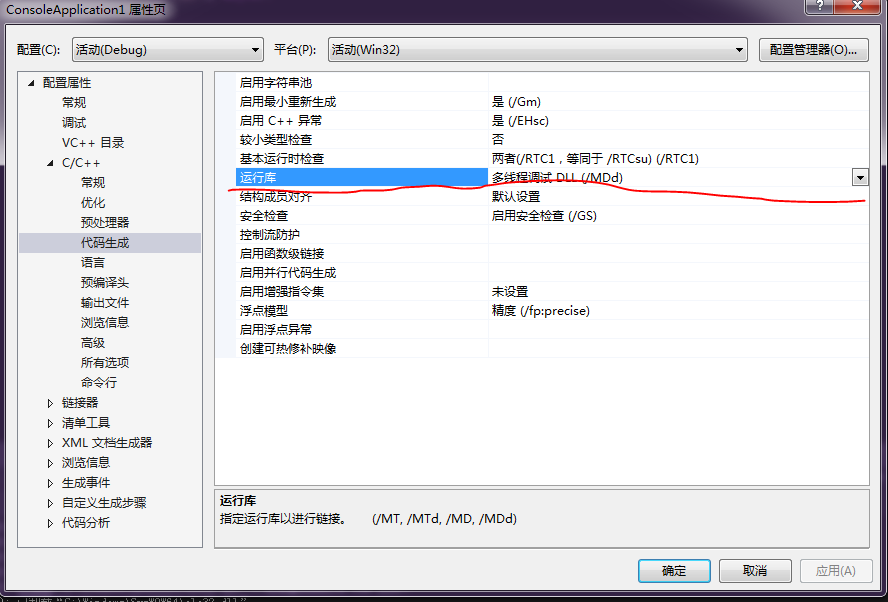
Windows.h包含kernel32.dll，

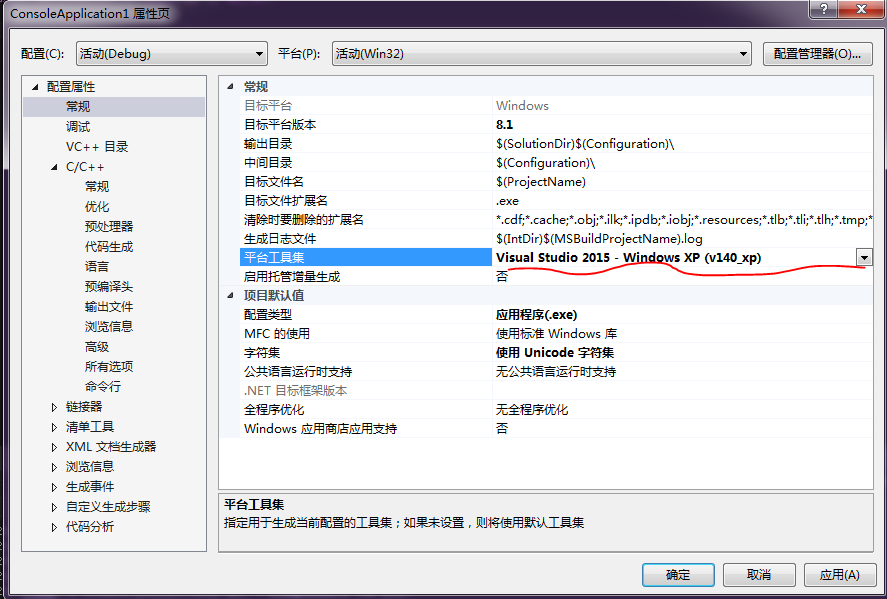
1. 编码字符集错误解决方案：
2. TEXT( "HelloWorld!")，加TEXT()宏定义。
3. 修改工程的字符集。项目->右键->属性->配置属性->常规->字符集。



调试生产的exe在xp虚拟机中无法运行解决方案：

1. 项目->右键->属性->配置属性->C/C++->代码生成->运行库->多线程(MT)；
2. 项目->右键->属性->配置属性->常规->平台工具集->win xp。





Socket编程中server端的汇编代码中接受功能使用函数recv，

窗口程序过程：

主程序WinMain():

LoadString();

LoadString();

MyRegisterClass();(自定义的窗口注册程序，定义主窗口的样子和响应功能！)

InitInstance();(自定义的窗口初始化、实例化程序，创建窗口实例，显示窗口，更新窗口)

分支1：（另一个分支是程序结束）

LoadAccelerators();

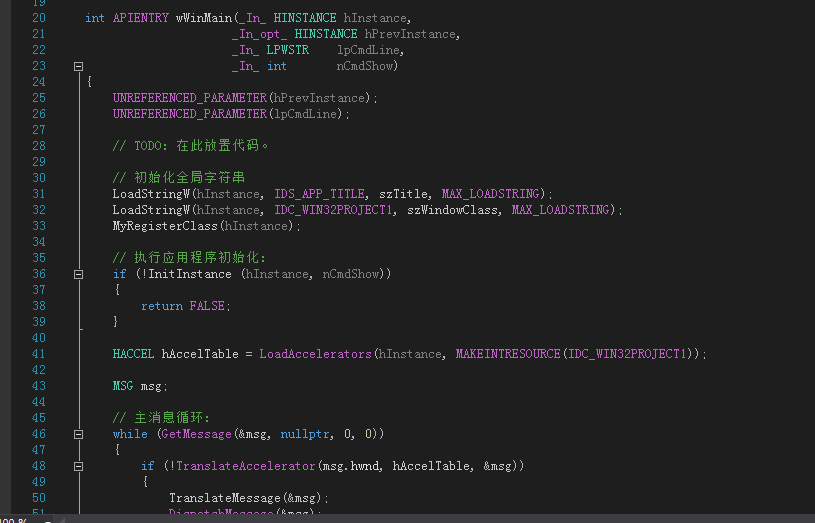
While循环GetMessage();（循环在GetMessage()之前，分支在GetMessage()之后，另一分支是程序结束）

TranslateAccelerator();

分支1:(另一分支是回到循环)

TranslateMessage();

DispatchMessage();



MyRegisterClass()是**自定义的函数**实现窗口注册程序，里面定义了各种消息的响应函数，所以对应的响应函数地址都在该函数中可以找到:

首先定义窗口类WNDCLASSEXW，该类有多个重要属性：

WNDCLASSEXW.lpfnWndProc关键属性定义了消息响应函数地址，是回调函数！

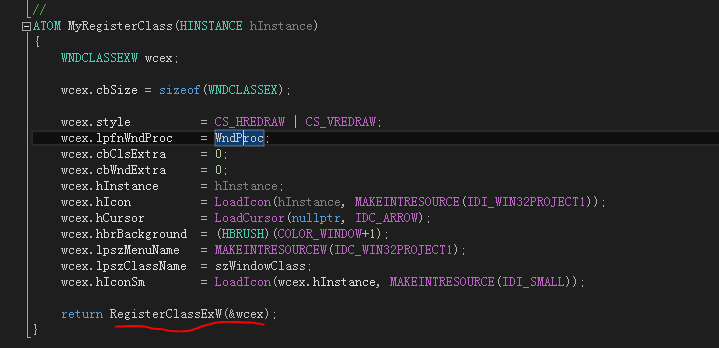
.hIcon=LoadIcon();

. hCursor=LoadCursor();

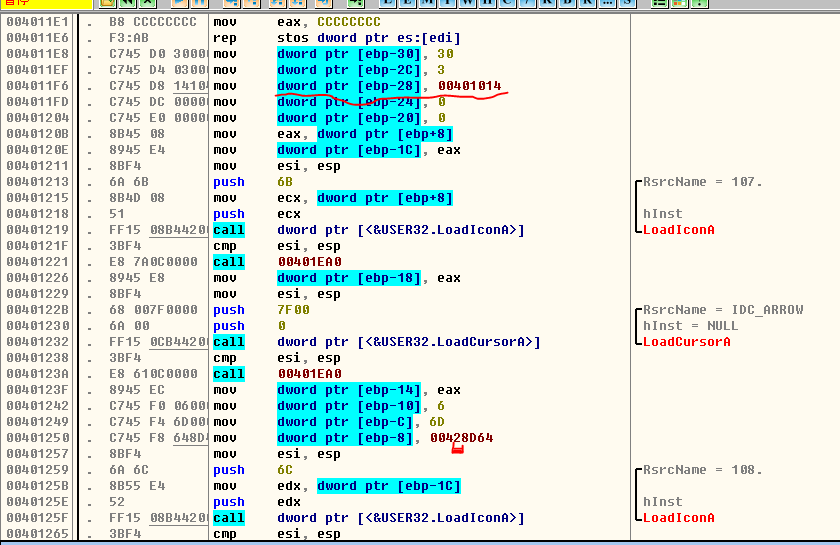
. lpszMenuName=MAKEINTRESOURCEW();

. hIconSm=LoadIcon();

然后RegisterClassExW()，即注册窗口函数，参数为&WNDCLASSEXW。这是系统函数RegisterClassA()和RegisterClassExW()的调用，实现回调函数的注册，因此应该重点关注该函数！



MyRegisterClass()中对WNDCLASSEXW.lpfnWndProc的赋值比较明显(如下图)，因为是回调函数，所以设置的是函数地址，而只有00401014距离不远，似乎是代码段地址。00428D64其实是数据段地址，是设置WNDCLASSEXW.lpszClassName字段属性。（在IDA中可以明显看出区别）



InitInstance()是**自定义的函数**实现窗口实例化，创建、显示、更新窗口：

CreateWindowW();

ShowWindow();

UpdateWindow();

WNDCLASSEXW.lpfnWndProc定义了窗口响应函数的回调地址，该函数有4个参数，回调函数定义如下：

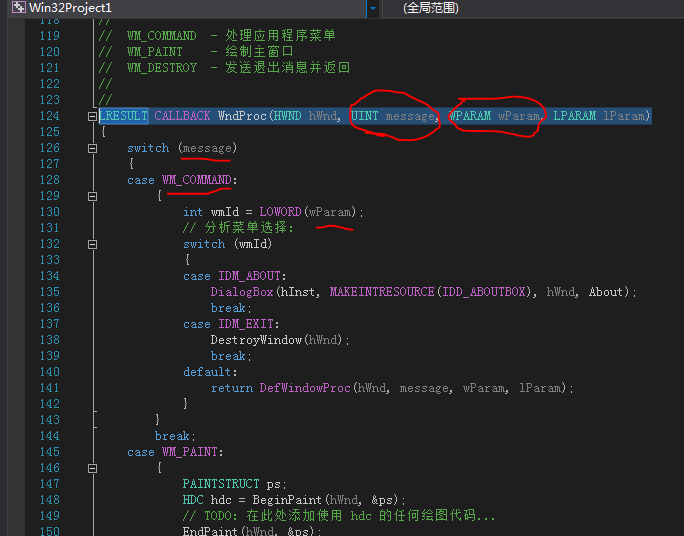
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

注意参数的压栈顺序！从右向左！

主要针对其中两个参数（消息类型message和消息附加参数wParam）定义响应过程，全函数是很明显的分支结构：

消息类型大致分为3类：WM\_COMMAND（处理应用程序菜单,0x0111），WM\_PAINT（绘制主窗口，0x000F），WM\_DESTROY（关闭主窗口，0x0002）。

再根据消息附加参数自定义响应。



如果是非主窗口回调响应函数，则消息类型可能还有其他类型，消息类型和消息附加参数需查询windows 消息大全。

逆向解题顺序：应该先简单看下pe头中是否有flag，没有再做逆向工程。

系统标准MD5和SHA1加密

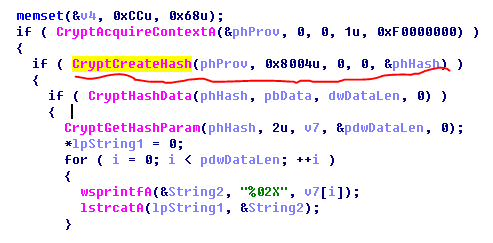
C中标准的md5和sha1加密的基本流程相似，CryptAcquireContextA-->CryptCreateHash-->CryptHashData-->CryptGetHashParam

CryptAcquireContextA：是加密前的初始化函数。

CryptCreateHash：是创建hash加密算法的句柄，第二个参数决定了加密的算法（0x8004u是md5，0x8003u是sha1），第四个参数是返回的hash加密句柄。

CryptHashData：是向hash加密句柄中传入待加密原数据。

CryptGetHashParam：将加密的结果反馈，第二个参数决定反馈的结果类型（加密算法id，hash加密后的长度，hash加密后的值）（2u是hash加密后的值），第三个参数是输出的数据，第四个参数是输出数据的长度。



Andriod逆向

1. 逆向工具：

Smali2Java\Smali2JavaUI.exe 该工具直接打开apk

ApkIDE3.2 该工具功能更强大，可以修改并重新编译

1. Apk伪加密解密

解密工具；apk伪加密\apk伪加密\ZipCenOp.bat，使用方法在readme.txt中。

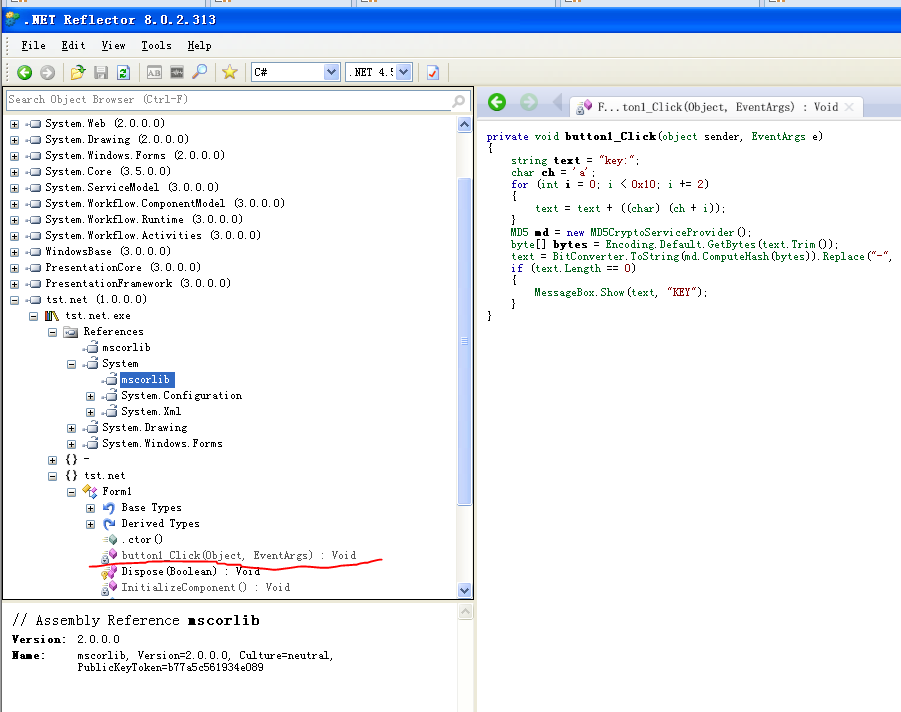
加密完后改为zip，即可解压。

.net逆向

1. 逆向工具

Reflector

该工具安装在xp环境中。



VC编程深入