第 3 章

加壳与脱壳

3.1 项 目 目 的

理解加壳与脱壳的原理，掌握常用的加壳与脱壳方法。

3.2 项 目 环 境

项目环境为 Windows 操作系统。用到的工具软件包括：Visual Studio 2013 或更高版本的开发环境，工具软件包括 ASPack、UPX、PECompact、PEiD、DiE、IDA、OllyDbg(简称 OD)等。

3.3 名 词 解 释

(1) **壳**：壳是一段执行于原始程序前的代码，其可以将原始程序压缩和加密。当加壳后的程序运行时，壳代码会先被执行，把之前压缩、加密后的代码还原成原始程序代码，然后再把执行权交还给原始程序。加壳的目的主要是为了隐藏程序真正的 OEP(original entry point，原始入口点)，防止程序被破解。

(2) **加壳**：是一种通过一系列数学运算处理程序编码的方法，其能将可执行程序文件或动态链接库文件的编码改变，以达到缩小文件体积或加密程序编码的目的。

(3) **脱壳**：把加在软件上的保护程序去除，将被保护的程序还原成原来的样子。

3.4 预 备 知 识

3.4.1 加壳技术

加壳技术是一把双刃剑，一方面加壳技术可以帮助各种恶意软件躲避杀毒软件的分析和查杀，另一方面，加壳技术又可以保护商业软件的核心算法和机密数据不被他人随意“借鉴”或“窃取”。研究软件加壳技术并实现反调试、反附加等功能，可以增加软件被破解的难度，阻碍逆向技术，其在实现软件保护方面是相当有必要的。

随着技术的发展，单一的技术保护已经无法抵挡破解技术的攻击，因此需要进行多种技术的融合应用。目前最常用的保护技术是加密壳、压缩壳、伪装壳、多层壳和代码虚拟

技术等。其中，代码虚拟技术是通过构造一个虚拟机，产生一些模拟代码来模拟被保护代码的执行，以产生出与被保护代码相同的结果，其可以阻止源代码的泄露。代码虚拟技术是目前最好的保护技术，但是其复杂度和技术难度较大。

为了应对逆向分析给软件带来的安全威胁，目前主要的保护措施有：①加壳，即使用强度比较高的虚拟壳或者自己编写的生僻壳；②代码混淆，即对软件代码进行混淆处理以改变程序的逻辑结构和提升程序的复杂度。

3.4.2 加壳及相关软件

1. 加壳原理

壳是指在一个程序外部再包裹的另外一段代码，是能够保护内部程序的代码不被非法修改或者反编译的外部程序。以压缩壳的演示过程为例，如图 3-1 所示，源文件经过加壳压缩后，加上一段外壳程序(loader)。当加壳后的程序运行时，会先运行 loader，然后再对源程序进行内存映射。可以发现，被加壳的源程序在映射到内存中时，存在一个脱壳还原过程。

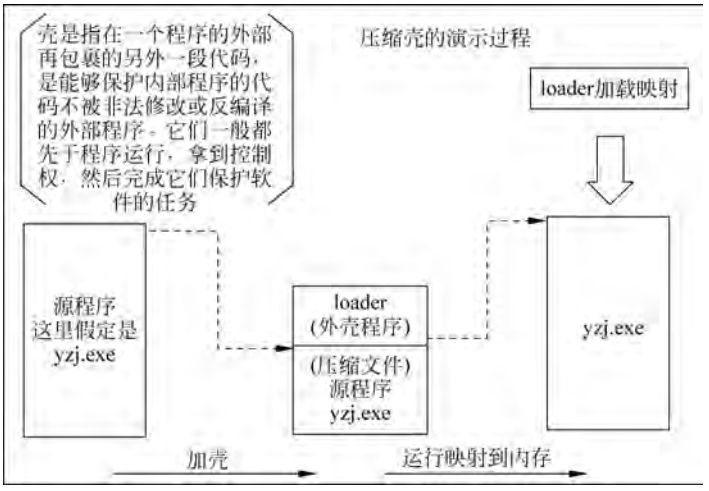


图 3-1 压缩壳的演示过程

图 3-1 中的 loader 同 yzj.exe 文件一样，在其运行时存在一个加载过程。一般壳的加载过程都会经过如图 3-2 所示的步骤：首先获取壳自身所需要使用的 API 地址；然后对原程序加壳的区段进行相应的解压缩和解密；接着，为了确保还原的程序代码可以正常运行，其还需要对脱壳后的原程序进行一次重定向；最后，壳会将控制权交还给原程序，跳转到程序的 OEP 处。

2. 加壳方法

1) 加壳的一般步骤

在实际的操作中，加壳的过程几乎是通用的，如图 3-3 所示，这一过程可以被分解成 6 个步骤。

(1) 采用文件映射的方法将待修改的程序加载进来，获取需要的信息。

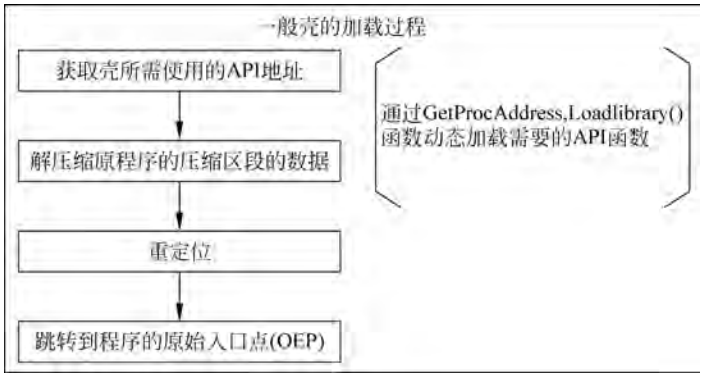


图 3-2　一般壳的加载过程

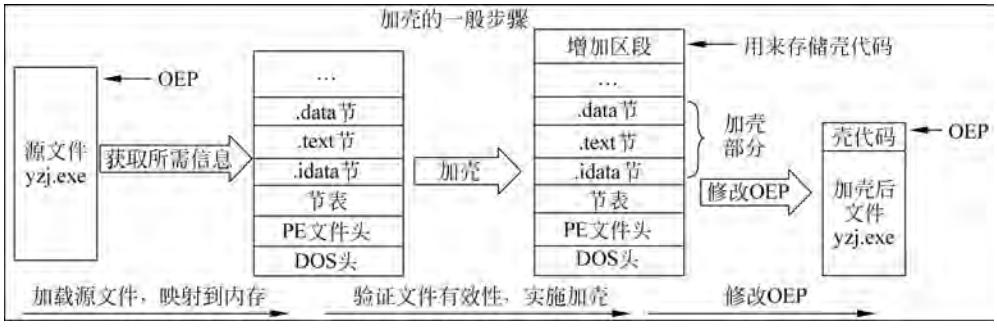


图 3-3　加壳过程

（2）初始化 PE 头信息，验证 PE 的有效性。

（3）添加一个节区，修改属性权限、确定地址、内存映射等，该节区用于存放壳代码。

（4）对目标代码进行加密或压缩。

（5）添加壳代码的解密代码或解压缩代码。

（6）修改程序 OEP。

有些加壳还涉及重定位、导入表等信息。当被加壳的程序运行时，外壳程序先被执行，然后由这个外壳程序负责将用户原有的程序在内存中解压缩，并把控制权交还给脱壳后的真正程序。这些操作过程由外壳程序自动完成，用户并不会知道壳程序是如何运行的，一般情况下，加壳程序和未加壳程序的运行结果是一样的。那么，如何判断一个可执行文件是否被加了壳呢？有一个简单的方法（对中文软件效果较明显），即用记事本打开一个可执行文件，如果能看到软件的提示信息则其一般是未被加壳的，如果完全是乱码，则其多半是被加过壳的。

2）工具加壳

加壳工具软件使用起来极为方便，其可以直接通过图形化的界面操作对目标程序进行加壳，满足用户的各种安全需求。根据功能的不同，可以将加壳工具分为 3 类，即压缩壳、加密壳和虚拟壳。

（1）压缩壳：以隐藏程序代码和数据为目的而设计的壳，其会对隐藏后的代码和数

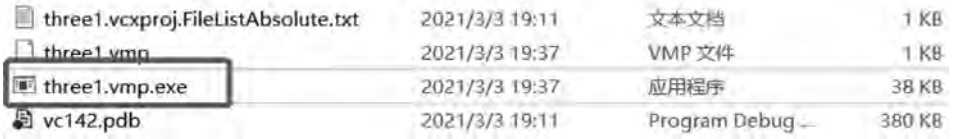
据进行压缩处理。但是，由于压缩壳在运行时会将代码段和数据段还原，所以其安全性较低。

（2）加密壳：可以将代码和数据加密，也可以对单个函数加密的壳，只有函数被执行时其才会将之解密。同样，由于在运行时仍需要解密代码和数据，所以加密壳只能起到辅助的效果。

（3）虚拟壳：将原始的指令经过虚拟化，翻译成自定义的虚拟机指令的壳。由于虚拟机指令不对外公开，且每次加壳都能产生随机化的虚拟机操作码，因此如果要逆向虚拟化的指令，需要操作者先分析自定义虚拟机，而此壳分析难度极高。

图 3-4 显示了将 three1.exe 导入 VMProtect v1.1 加壳机的过程，加壳机会对 three1.exe 自动加壳如图 3-4(a)所示，并生成 three1.vmp.exe 文件如图 3-4(b)所示。

(a) VMProtect加壳界面



(b) 生成加壳文件

图 3-4 采用 VMProtect 加壳

3. 常用的加壳软件

1）压缩壳软件

（1）ASPack：Win32 平台下的可执行文件压缩软件，其可压缩 Windows 32 位平台下的可执行文件(.exe)以及库文件(.dll、.ocx 等)，压缩的文件压缩比率高达 40%～70%，该软件界面如图 3-5 所示。

（2）PECompact：Windows 平台下能压缩可执行文件的工具(支持.exe、.dll、.ocx 等文件)。相比同类软件，PECompact 提供了多种压缩项目供选择，用户可以根据需要确定哪些内部资源需要被压缩处理。同时，该软件还提供了加解密的插件接口功能，其界面如图 3-6 所示。

图 3-5　ASPack 界面

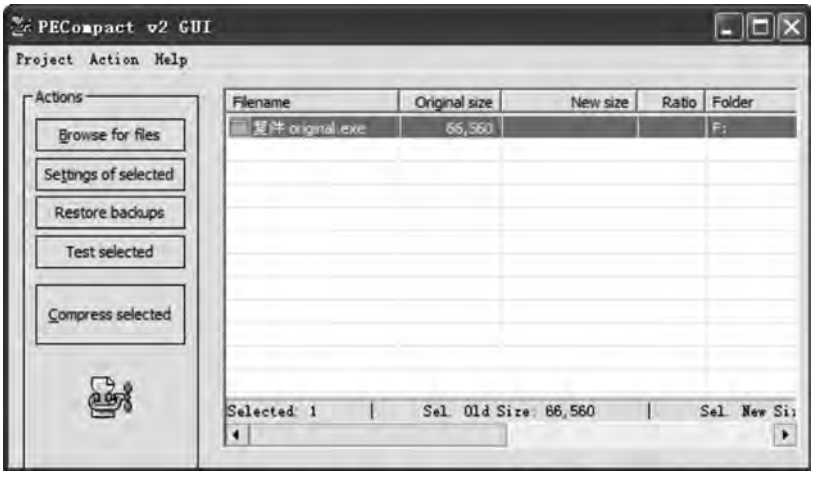


图 3-6　PECompact 界面

2）加密壳软件

（1）Armadillo：也被称为穿山甲，是一款应用面较广的加密壳工具。其可以运用各种手段来保护可执行程序文件，同时也可以为这些文件加上种种限制，包括使用时间、执行次数，启动画面等，该软件界面如图 3-7 所示。

（2）EXECryptor：其特点是 Anti-Debug 功能做得比较隐蔽，采用了虚拟机技术来保护软件的一些关键代码，其软件界面如图 3-8 所示。

（3）Themida：其是 Oreans 公司开发的一款商业壳工具，最大特点是其具有虚拟机保护技术，因此在程序中能够用 SDK 将关键的代码通过虚拟机保护起来。Themida 最大的缺点就是生成的软件容量较大，其软件界面如图 3-9 所示。

图 3-7　Armadillo 界面

图 3-8　EXECryptor 界面

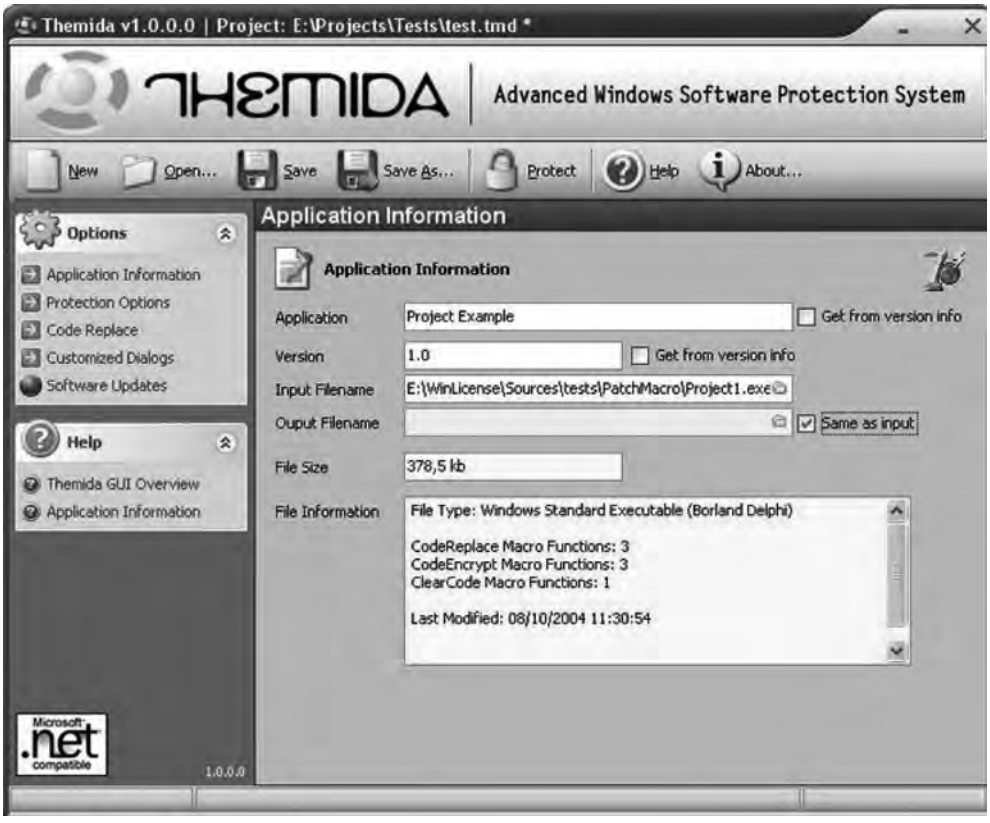


图 3-9　Themida 界面

3.4.3　查壳软件

逆向分析目标软件的第一步是查壳，即分析软件的加壳。常用的查壳方法是将目标软件加载入文件分析工具中，以此来确定其附加的是哪一种壳，然后再对其进行数据跟踪。

1. PEiD

PEiD 的工作原理是利用特征串搜索软件的启动代码，来完成识别工作。每一种开发语言都有固定的启动代码部分，利用这一点就可以识别出软件是用何种语言编译的。同理，不同的壳也有其特征码，利用这一点就可以识别软件是被何种壳加密。PEiD 提供了一个拓展接口文件 userdb.txt，用户可以自定义一些特征码并将之写入其中，这样可以识别出新的文件类型。如图 3-10 所示，PEiD 分析出一个 install.exe 文件是由 masm32 编译的且没有壳。

有些外壳程序的伪装能力很强，会通过伪造启动代码部分欺骗 PEiD 等文件识别软件。例如，外壳程序将入口启动代码部分伪造成 VC++ 7.0 所开发的程序入口处的类似代码，即可达到欺骗的目的，所以文件识别工具所给出的结果只是一个参考，文件是否被加壳处理过，还需要跟踪分析程序代码才能真正得知。

图 3-10 PEiD 界面

2. DiE

Detect It Easy(简称 DiE)是一个多功能的 PE-DIY 工具,其主要用于壳侦测,且其功能正日益完善,是一个不可多得的破解利器。和 PEiD 一样,DiE 可以加载插件。打开 DiE,将目标软件拖曳至其窗口内,就可以看到该文件的各种资源信息,其界面如图 3-11 所示。

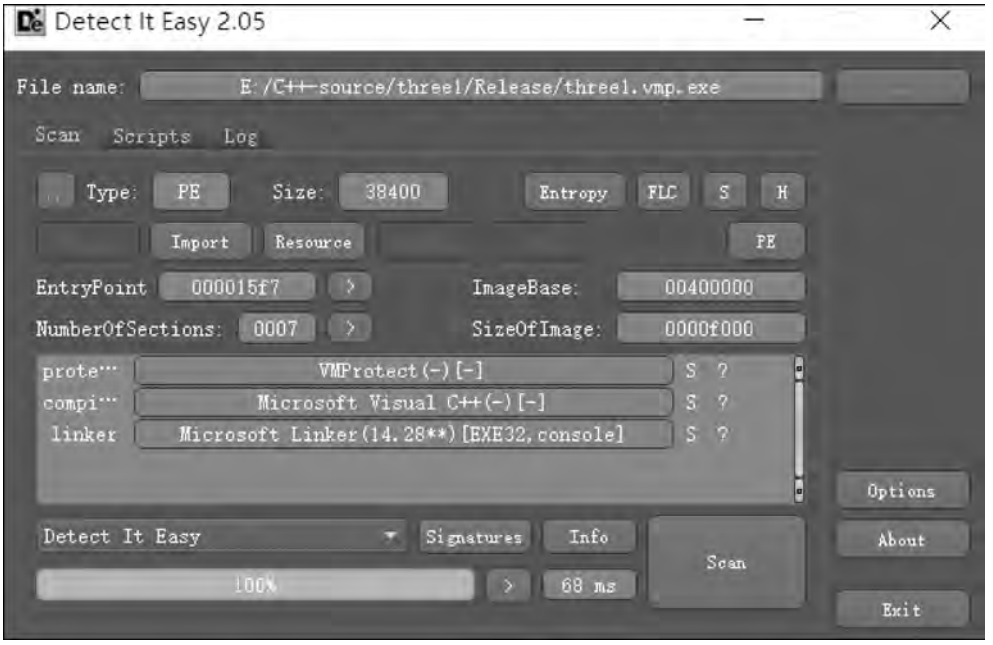


图 3-11 DiE 界面

3.4.4 脱壳及其方法

脱壳是指除掉程序的保护,将原程序本身提取出来。根据具体的表现可以将脱壳的方法分为两种。第一种是硬脱壳,所谓硬脱壳,就是分析和找出加壳软件的加壳算法,然后写出逆向算法。第二种是动态脱壳,由于加壳程序运行时必须将程序还原成原始形态,

即加壳程序会在运行时自行脱落，因此这一脱壳方式就是抓取（dump）内存中的镜像，再将之重构成标准的可执行文件。

在实际操作中，很多壳都会通过“变形”来伪装自己，所以通过硬脱壳的方法实现脱壳就显得很困难，使用动态脱壳的方法应对这种难度很大的壳效果往往很明显。

脱壳是加壳的一个逆过程，其首先要将加密、压缩的部分予以解密和解压缩，然后根据区段头的描述将源程序的区段加载到指定的位置。大多数的加壳都会通过修改 IAT 来获取被加壳程序运行的控制权，所以脱壳时需要修复 IAT。为了确保脱壳后的程序可以正常运行，需要对脱壳后的程序进行重定位。最后，将 OEP 调整到已脱壳程序的入口处。

1. 动态调试脱壳

从壳的定义可知，壳是包裹在一个程序外面的另外一段代码，其本身也是一段程序。在运行的过程中，壳代码会先于原程序运行，将原程序中加密、压缩的部分解密、解压缩，然后再将程序控制权交给原程序。

动态调试的原理就是在调试器中让加壳后的程序运行起来，然后观察壳的具体运行细节，从而发现原程序的入口地址。很多壳会通过变形、伪造来加大脱壳的难度，但是在运行时其都会暴露自己的“庐山真面目”。常用的动态调试工具有 OD、IDA 等。如 OD 动态调试的一般步骤如下所示。

（1）将待脱壳程序载入到 OD 中，如果出现压缩提示，便选择“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，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 指令内，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 指令之内再单步跟踪。

（7）遇到在 popad 指令下的远转移指令时要格处注意，因为这个远转移指令的目的地很可能就是 OEP。

脱壳的方法一般包括 ESP 定律调试方法、内存镜像法和跟踪出口法。

（1）ESP 定律调试方法：ESP 定律是堆栈平衡原理在项目中的一个应用。所谓堆栈平衡原理，即加壳程序运行时需要先保存原程序的初始现场，将现场状态压入堆栈，待壳运行完毕后，将原程序的现场状态出栈，此时开始运行原程序。在具体应用时，通过记录堆栈寄存器中存储原程序初始现场状态的地址，并使程序在为将初始状态出栈而访问以记录的地址时中断，程序中断处便在 OEP 附近。

（2）内存镜像法：程序运行需要先对各个区段进行解码，即将壳的代码写入到原程序的代码段中，然后再从原程序的代码段中读取代码并执行。若找到一个 Code 段已被解压完毕但又未开始执行的地址，就让它中断下来，当再次对 Code 段下内存访问断点时，就可以停在原程序 Code 段的第一条指令位置上，而这个位置正是原程序的 OEP。一

般程序解压都是按照地址从低到高的顺序对原程序的各个段进行解压。所以在实际的应用过程中，是通过首先在内存镜像中地址高于 Code 段的其他段下 INT3 断点，运行程序后再在 Code 段下 INT3 断点，使程序运行后中断在 OEP 处。

（3）跟踪出口法：压栈原程序初始状态后，再一次访问堆栈寄存器，并将所存数据出栈，原程序将运行，程序将跳转到原程序 OEP。

2. 重建输入表

程序脱壳后往往不能正常运行，这时就需要修复输入表。在脱壳中，对输入表的处理是一个很关键的环节，在 PE 文件中，输入表的结构如图 3-12 所示。Windows 系统加载器首先会搜索 OriginalFirstThunk，如果其存在，将加载程序迭代搜索数组中的每个指针，找到每个 IMAGE_IMPORT_BY_NAME 结构所指向的输入函数的地址，然后用函数入口地址来替代由 FirstThunk 指向的 IMAGE_THUNK_DATA 数组里的元素值（即将真实的函数地址填充到 IAT 里）。

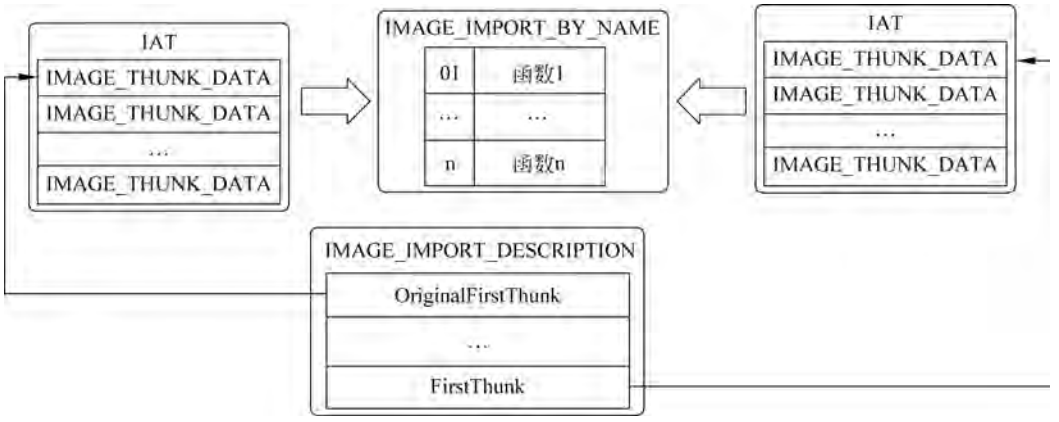


图 3-12　磁盘文件中的输入表结构

当文件被加载到内存并准备被执行时，如图 3-13 所示。输入表中的其他部分就不再重要了。程序依靠 IAT 提供的函数地址就可正常运行。如果程序加壳，那么壳自己会模

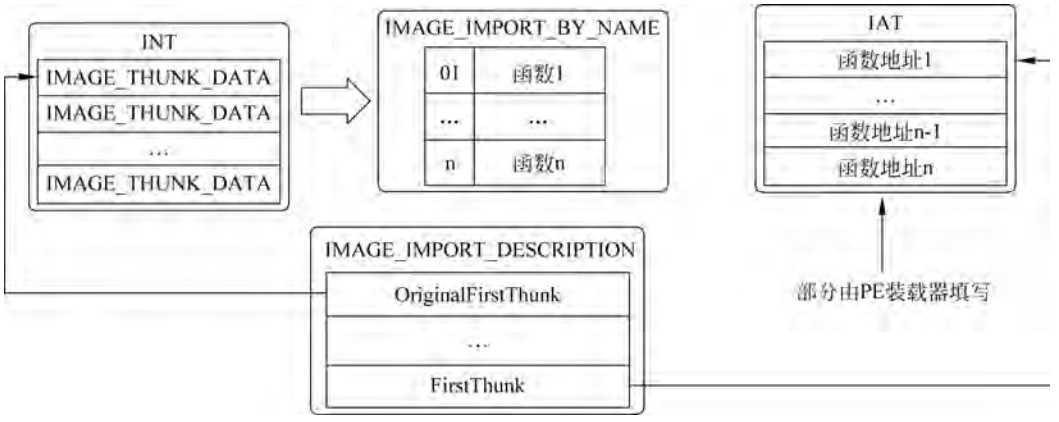


图 3-13　PE 文件装载内存后的输入表结构

仿 Windows 加载器的工作来填充 IAT 中相关的数据，壳中的代码一旦完成了加载工作，在进入原程序的代码之后仍可以间接地获得程序的控制权，进行反跟踪以继续保护软件，同时完成一些特殊任务。有些情况下，壳还会对 IAT 进行加密，用来防止 IAT 被还原。

重建输入表就是根据图 3-13 中的 IAT 来恢复整个输入表的结构。使用工具软件 ImpREC 重建输入表如图 3-14 所示。重建输入表的关键是获得未加密的 IAT，这需要跟踪加壳程序对 IAT 的处理过程，修改相关指令，不让外壳加密 IAT。

图 3-14　ImpREC 重建输入表

3. 工具脱壳

除了手动脱壳，用户还可以使用自动脱壳机进行脱壳。自动脱壳机需要识别出壳的类型，然后选择对应的脱壳工具，最终实现一键脱壳，如图 3-15 所示的超级巡警虚拟机脱

图 3-15　自动脱壳机

壳机便是如此。自动脱壳机当然也有很多局限性，对于未知的壳如一些较新的壳往往就没有办法，其软件更新也很慢。

3.5 项目实现及步骤

3.5.1 任务一：加壳操作

以一个简单的 Win32 控制台程序 nixiang.exe 为实验对象，其加壳步骤如下。

(1) 首先打开 PEiD 查壳工具，检查可执行文件 nixiang.exe，结果显示该文件无壳，编译语言是 VC++ 8.0，如图 3-16 所示。

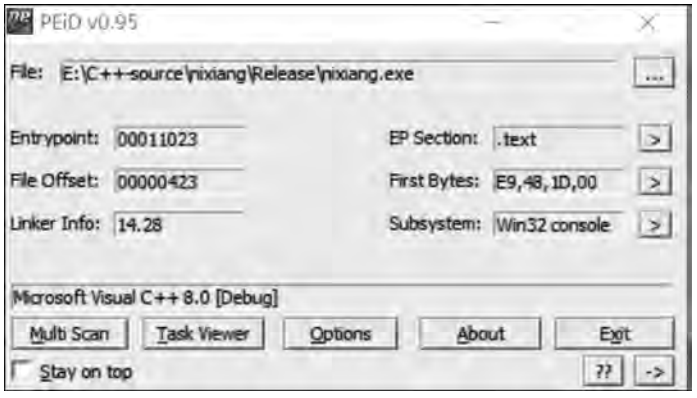


图 3-16 PEiD 查壳

(2) 打开 ASPack 软件，导入可执行文件 nixiang.exe，然后单击 GO! 按钮，即可自动加壳并生成一个新的 nixiang.exe 可执行文件，如图 3-17 所示。

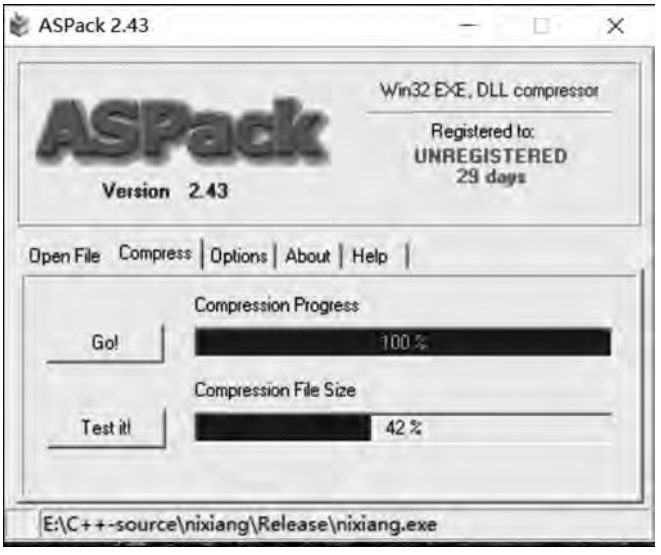


图 3-17 工具加壳

(3) 再次打开 PEiD 查壳工具，检查 nixiang.exe 文件，结果显示其已加的壳为 ASPack 2.12，如图 3-18 所示。

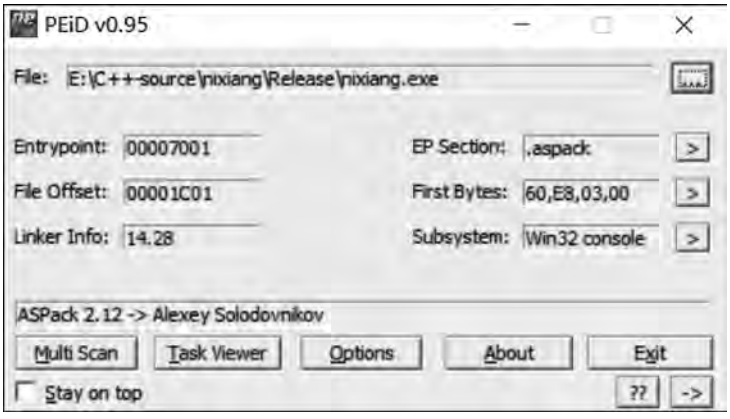


图 3-18　PEiD 查壳

3.5.2　任务二：用工具脱壳

打开自动脱壳机 ASPack unpacker，导入 nixiang.exe 文件，如图 3-19 所示。

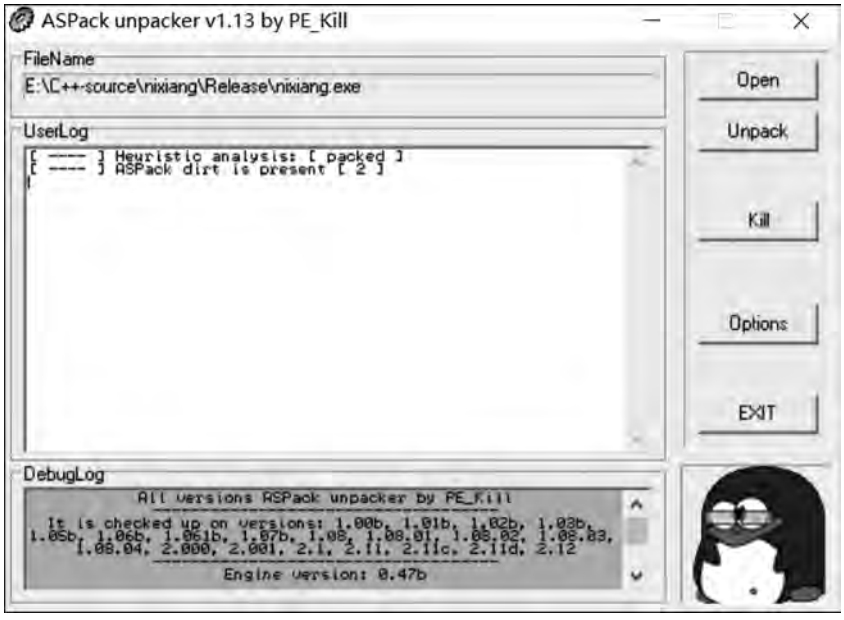


图 3-19　ASPack unpacker

单击 Unpack 按钮，ASPack unpacker 将开始自动调试、跟踪文件并完成脱壳，如图 3-20 所示。

再次将脱壳后的 nixiang.exe 文件载入到 PEiD 中查壳，如图 3-21 所示，结果显示该文件已无壳。

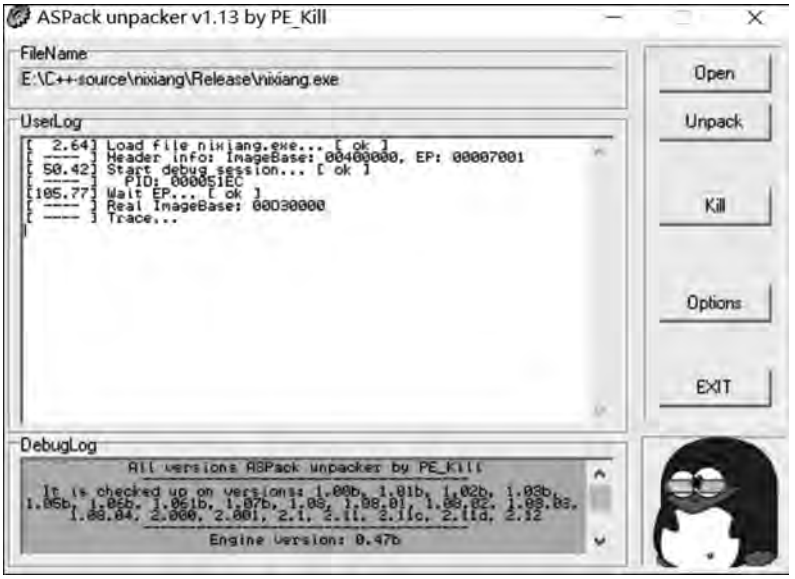


图 3-20 自动脱壳

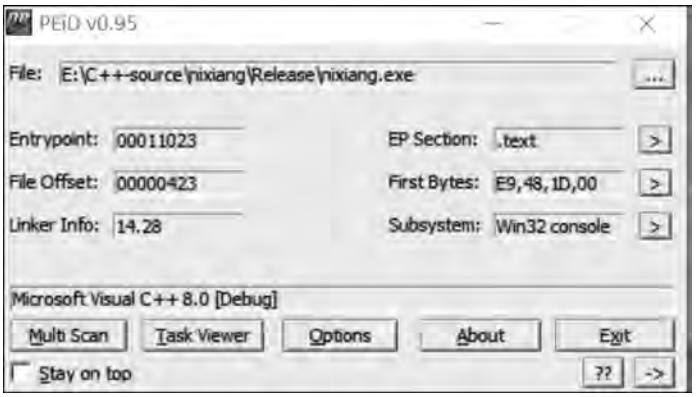


图 3-21 PEiD 再次查壳

3.5.3 任务三：自定义加壳程序及脱壳

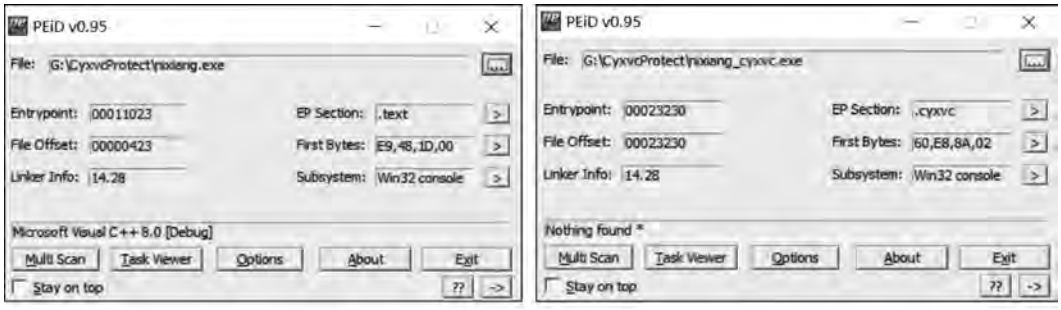
运行自定义加壳程序 CyxvcProtect，单击“选择文件”按钮，然后选择 nixiang. exe 文件，如图 3-22(a)所示，再单击“加壳”按钮，其将弹出“加壳成功!”的提示框如图 3-22(b)所示，并生成加壳的 nixiang_cyxvc. exe 文件。

验证 nixiang_cyxvc. exe 的壳，并比较其与 nixiang. exe 的区别，图 3-23(a)显示 nixiang. exe 文件无壳，图 3-23(b)显示 nixiang_cyxvc. exe 文件查不出任何信息。

具体的执行流程是首先读取目标文件的 PE 信息并保存，然后对其代码段进行加密操作，将必要的信息保存到外壳程序，并将其附加到 PE 文件中，再保存文件，完成加壳，最后释放资源。CyxvcProtect 项目由 Visual Studio 2013 开发，其包含 2 个 Visual Studio 项目。

(a) 加壳程序界面　　(b) 提示框

图 3-22　自定义加壳



(a) 加壳前　　(b) 加壳后

图 3-23　加壳前后的比较

(1) 项目 Shell：生成 Shell 模块(Shell.dll)，其将作为外壳程序完成目标 PE 文件的运行。

(2) 项目 CyxvcProtect：一个基于 MFC 的对话框程序，如图 3-22(a)所示，其实现了类似一个 PE 文件加载器的功能，可将外壳程序(Shell 模块)附加到目标 PE 文件中，完成加壳操作。

1. 读取目标文件

在 CyxvcProtect 项目中的头文件 PE.h 中定义了 CPE 类，其被用于以内存对齐的方式将目标文件读到内存，保存获取到的目标文件关键信息。CPE 类中对应成员函数的代码如下。

```
BOOL CPE::InitPE(CString strFilePath){
    if (OpenPEFile(strFilePath) == FALSE)          //打开文件
        return FALSE;
    //将 PE 以文件分布格式读取到内存
    …
    //将 PE 以内存分布格式读取到内存
    ReadFile(m_hFile, m_pFileBuf, m_dwFileSize, &ReadSize, NULL);
    CloseHandle(m_hFile);
    m_hFile = NULL;
    if (IsPE() == FALSE)                           //判断是否为 PE 文件
```

```
        return FALSE;
    //修正镜像大小没有对齐的情况
    m_dwImageSize = m_pNtHeader->OptionalHeader.SizeOfImage;
    m_dwMemAlign = m_pNtHeader->OptionalHeader.SectionAlignment;
    m_dwSizeOfHeader = m_pNtHeader->OptionalHeader.SizeOfHeaders;
    m_dwSectionNum = m_pNtHeader->FileHeader.NumberOfSections;
    if (m_dwImageSize % m_dwMemAlign)
        m_dwImageSize = (m_dwImageSize / m_dwMemAlign + 1) * m_dwMemAlign;
    LPBYTE pFileBuf_New = new BYTE[m_dwImageSize];
    memset(pFileBuf_New, 0, m_dwImageSize);
    memcpy_s(pFileBuf_New, m_dwSizeOfHeader, m_pFileBuf, m_dwSizeOfHeader); //复制文件头
    PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(m_pNtHeader);    //复制区段
    for (DWORD i = 0; i < m_dwSectionNum; i++, pSectionHeader++) {
        memcpy_s(pFileBuf_New + pSectionHeader->VirtualAddress,
            pSectionHeader->SizeOfRawData,
            m_pFileBuf + pSectionHeader->PointerToRawData,
            pSectionHeader->SizeOfRawData);
    }
    delete[] m_pFileBuf;
    m_pFileBuf = pFileBuf_New;
    pFileBuf_New = NULL;
    GetPEInfo();                                    //获取 PE 信息
    return TRUE;
}
void CPE::GetPEInfo(){                              //获取并保存目标文件的关键信息
    …
    //保存重定位目录信息
    m_PERelocDir =
    IMAGE_DATA_DIRECTORY(m_pNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_
ENTRY_BASERELOC]);
    //保存 IAT 信息目录信息
    m_PEImportDir = IMAGE_DATA_DIRECTORY(m_pNtHeader->OptionalHeader.DataDirectory
[IMAGE_DIRECTORY_ENTRY_IMPORT]);
    //获取 IAT 所在区段的起始位置和大小
    PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(m_pNtHeader);
    for (DWORD i = 0; i < m_dwSectionNum; i++, pSectionHeader++){
        if (m_PEImportDir.VirtualAddress >= pSectionHeader->VirtualAddress&&
            m_PEImportDir.VirtualAddress <= pSectionHeader[1].VirtualAddress){
            //保存该区段的起始地址和大小
            m_IATSectionBase = pSectionHeader->VirtualAddress;
            m_IATSectionSize = pSectionHeader[1].VirtualAddress - pSectionHeader->
VirtualAddress;
            break;
        }
    }
}
```

2. 代码段加密

首先获取目标 PE 文件的缓冲区指针 m_pFileBuf、代码段基址 m_dwCodeBase 和代码段大小 m_dwCodeSize，然后计算得到目标文件代码段在内存中的起始地址 pCodeBase，再依次遍历 pCodeBase，异或加密其每一个字节，最后返回加密的长度，对应代码如下。

```
DWORD CPE::XorCode(BYTE byXOR) {                    //采用异或加密
   PBYTE pCodeBase = (PBYTE)((DWORD)m_pFileBuf + m_dwCodeBase);
   //获取目标文件代码段的起始位置
   for (DWORD i = 0; i < m_dwCodeSize; i++){
        pCodeBase[i] ^= i;                          //遍历代码段,并异或加密
    }
    return m_dwCodeSize;
}
```

3. 加壳操作

对目标文件完成加壳的操作，其可以分为两步：加壳前操作由 CPACK 类实现；加完壳及程序运行时的操作由 Shell.dll 模块实现。CPACK 类通过 LoadLibrary()函数将 Shell.dll 加载到内存后，将获取到的目标文件之 PE 关键信息保存到 Shell.dll 的 ShellData 结构体中，当壳运行的时候，Shell.dll 就可以直接调用这些数据。Shell 项目中的 Shell.h 中定义了 ShellData 结构体。

```
extern"C" typedef struct _SEHLL_DATA
{
      DWORD dwStartFun;                             //启动函数
      DWORD dwPEOEP;                                //程序入口点
      DWORD dwXorKey;                               //解密 KEY
      DWORD dwCodeBase;                             //代码段起始地址
      DWORD dwXorSize;                              //代码段加密大小
      DWORD dwPEImageBase;                          //PE 文件映像基址
      IMAGE_DATA_DIRECTORY   stcPERelocDir;         //重定位表信息
      IMAGE_DATA_DIRECTORY   stcPEImportDir;        //导入表信息
      DWORD                  dwIATSectionBase;      //IAT 所在段基址
      DWORD                  dwIATSectionSize;      //IAT 所在段大小
      BOOL                   bIsShowMesBox;         //是否显示 MessageBox
}SEHLL_DATA, *PSEHLL_DATA;
```

要将 Shell.dll 模块附加到目标文件中，首先需要读取 Shell.dll 模块的二进制代码，然后设置重定位信息，修改程序的 OEP，使其指向 Shell.dll 模块的启动函数 Start()，最后将目标文件和 Shell.dll 模块的二进制代码合并到新的缓冲区。

在 CyxvcProtect 项目中，源文件 Pack.cpp 中的 CPACK::Pack()函数通过 GetCurrentProcess()方法来获取当前进程的句柄，通过 GetModuleInformation()方法来读取 Shell.dll 模块的信息，然后将之存储在 modinfo 变量中，缓冲区 pShellBuf 是专门用来存储 Shell 的内存空间，其可同时获取 Shell.dll 模块的镜像大小，对应代码如下。

```
MODULEINFO modinfo = { 0 };
```

```
GetModuleInformation(GetCurrentProcess(), shell, &modinfo, sizeof(MODULEINFO));
PBYTE pShellBuf = new BYTE[modinfo, SizeOfImage];
Memcpy_s (pShellBuf , modinfo.SizeOfImage, Shell, modinfo.SizeOfImage);
```

重定位的实现有两种：第一种是系统的 PE 加载器通过重定位表的信息，在加载程序前先重定位好；第二种是通过代码手动重定位，模拟 PE 加载器进行的重定位操作。程序运行过程可以分为两部分：一部分是外壳 Shell.dll 模块的加载过程；另一部分是原程序的加载过程。从外壳 Shell 跳转到原程序是一个重定位过程，Shell.dll 模块默认的加载基地址是 0x10000000，而 EXE 文件默认的加载基地址是 0x00400000。

在自定义加壳程序中，通过系统重定位 Shell.dll 模块的代码可以保证其函数可以正常执行。然后在 Shell.dll 模块中手动重定位原程序代码，让原程序能够执行。由于 Shell.dll 模块的代码是通过 LoadLibrary(L"Shell.dll")的方式加载的，说明在其被加载到内存中之前，PE 加载器已经完成重定位了。当 Shell.dll 模块再次访问其重定位表信息时，其是已经修复过的、正确的重定位信息，而不是原始的重定位信息，所以需要将原始的重定位信息写入到加壳后的原程序中。当 PE 加载器运行到原程序时，才能通过原始的重定位信息给 Shell.dll 模块进行正确的重定位，过程如图 3-24 所示。

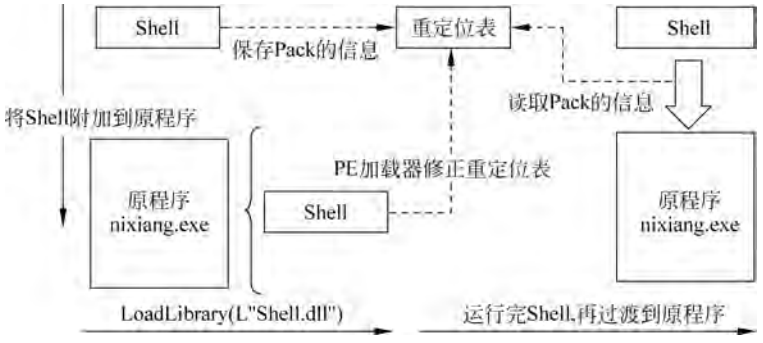


图 3-24 重定位表的改变

重定位原始地址是一个内存相对偏移(RVA)的过程，需要把这个 RVA 加上目标原程序文件默认的加载基址，然后再写回重定位表中的数据，这样才能让系统正确地进行重定位。它们之间的关系如下。

(1) 重定位原始地址＝重定位后的地址－加载时的镜像基址

(2) 新的重定位地址＝重定位原始地址＋新的镜像基址

Shell.dll 模块是加载到 PE 文件的末尾，所以 RVA 地址还需要加上目标 PE 文件的镜像大小，即以下关系。

新的重定位地址＝重定位后的地址－加载时的镜像基址＋新的镜像基址＋代码基址。

将地址信息写入到加壳后的 PE 文件中，之后在系统加载时就可以正确地进行重定位，对应代码如下。

```
BOOL CPE::SetShellReloc(LPBYTE pShellBuf, DWORD hShell){
   typedef struct _TYPEOFFSET {
```

```
    WORD offset : 12;                    //偏移值
    WORD Type : 4;                       //重定位属性(方式)
}TYPEOFFSET, * PTYPEOFFSET;
//1.获取被加壳 PE 文件的重定位目录表指针信息
PIMAGE_DATA_DIRECTORY pPERelocDir =
 &(m_pNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC]);
//2.获取 Shell 的重定位表指针信息
PIMAGE_DOS_HEADER pShellDosHeader = (PIMAGE_DOS_HEADER)pShellBuf;
PIMAGE_NT_HEADERS pShellNtHeader = (PIMAGE_NT_HEADERS)(pShellBuf + pShellDosHeader->e_
lfanew);
PIMAGE_DATA_DIRECTORY pShellRelocDir =
  &(pShellNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC]);
PIMAGE_BASE_RELOCATION pShellReloc =
 (PIMAGE_BASE_RELOCATION)((DWORD)pShellBuf + pShellRelocDir->VirtualAddress);
  /*3.还原修复重定位信息,由于 Shell.dll 模块是通过 LoadLibrary()函数加载的,所以系
统会对其进行一次重定位,需要把 Shell.dll 模块的重定位信息恢复到系统没加载前的样子,然后
在写入被加壳文件的末尾*/
PTYPEOFFSET pTypeOffset = (PTYPEOFFSET)(pShellReloc + 1);
DWORD dwNumber = (pShellReloc->SizeOfBlock - 8) / 2;
for (DWORD i = 0; i < dwNumber; i++) {
     if ( *(PWORD)(&pTypeOffset[i]) == NULL)
         break;
     //新的重定位地址 = 重定位后的地址 - 加载时的镜像基址 + 新的镜像基址 + 代码基址
     //(PE 镜像文件大小)
     DWORD AddrOfNeedReloc = *(PDWORD)((DWORD)pShellBuf + dwRVA);
     //4.修改 PE 重定位目录指针,指向 Shell 的重定位表信息
     pPERelocDir->Size = pShellRelocDir->Size;
     pPERelocDir->VirtualAddress = pShellRelocDir->VirtualAddress + m_dwImageSize;
     return TRUE;
 }
```

然后再修改被加壳程序的 OEP,使其指向 Shell.dll 模块,通过函数 SetNewOEP()来实现。

目前内存中有两个缓冲区,一个是原目标程序的缓冲区,另一个则是 Shell.dll 模块的缓冲区。重新申请一块连续的空间,大小与这两个缓冲区的大小相同,然后将它们复制过去。这一操作在逻辑上虽是一次合并操作,但需要深入理解 PE 文件格式。函数 CPE::MergeBuf()读取最后一个区段的信息,然后修改头节表的区段数量,编辑区段表头结构体信息,修改区段属性,使之可读可写可执行,接着修改 PE 头文件大小,最后申请合并所需要的空间,代码如下。

```
void CPE::MergeBuf(LPBYTE pFileBuf, DWORD pFileBufSize,
     LPBYTE pShellBuf, DWORD pShellBufSize,
     LPBYTE& pFinalBuf, DWORD& pFinalBufSize){
     //获取最后一个区段的信息
     PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)pFileBuf;
     PIMAGE_NT_HEADERS pNtHeader = (PIMAGE_NT_HEADERS)(pFileBuf + pDosHeader->e_
lfanew);
     PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(pNtHeader);
```

```
    PIMAGE_SECTION_HEADER pLastSection = &pSectionHeader[pNtHeader->FileHeader.
NumberOfSections-1];
    //1.修改区段数量
    pNtHeader->FileHeader.NumberOfSections += 1;
    //2.编辑区段表头结构体信息
    PIMAGE_SECTION_HEADER AddSectionHeader =
     &pSectionHeader[pNtHeader->FileHeader.NumberOfSections - 1];
    memcpy_s(AddSectionHeader->Name, 8, ".cyxvc", 7);
    //VOffset(1000对齐)
    DWORD dwTemp = 0;
    dwTemp = (pLastSection->Misc.VirtualSize / m_dwMemAlign) * m_dwMemAlign;
    if (pLastSection->Misc.VirtualSize % m_dwMemAlign) {
     dwTemp += 0x1000;
    }
    AddSectionHeader->VirtualAddress = pLastSection->VirtualAddress + dwTemp;
    //Vsize(实际添加的大小)
    AddSectionHeader->Misc.VirtualSize = pShellBufSize;
    //ROffset(旧文件的末尾)
    AddSectionHeader->PointerToRawData = pFileBufSize;
    //RSize(200对齐)
    dwTemp = (pShellBufSize / m_dwFileAlign) * m_dwFileAlign;
    if (pShellBufSize % m_dwFileAlign) {
     dwTemp += m_dwFileAlign;
    }
    AddSectionHeader->SizeOfRawData = dwTemp;
    //区段属性标志(可读可写可执行)
    AddSectionHeader->Characteristics = 0XE0000040;
    //3.修改PE头文件大小属性,增加文件大小
    dwTemp = (pShellBufSize / m_dwMemAlign) * m_dwMemAlign;
    if (pShellBufSize % m_dwMemAlign) {
     dwTemp += m_dwMemAlign;
    }
    pNtHeader->OptionalHeader.SizeOfImage += dwTemp;
    //4.申请合并所需要的空间
    pFinalBuf = new BYTE[pFileBufSize + dwTemp];
    pFinalBufSize = pFileBufSize + dwTemp;
    memset(pFinalBuf, 0, pFileBufSize + dwTemp);
    memcpy_s(pFinalBuf, pFileBufSize, pFileBuf, pFileBufSize);
    memcpy_s(pFinalBuf + FileBufSize, wTemp, pShellBuf, dwTemp);
}
```

4. 外壳程序

Shell项目生成的外壳程序要实现的功能是解密原程序的代码段,修复原程序的重定位和IAT信息,最后跳到程序的OEP,将控制权交还给程序。

利用Kernel32.dll模块中的GetProcAddress()函数获取Shell.dll模块导出函数的地址。首先需要获取Kernel32.dll模块的加载基址,然后获取GetProcAddress()函数的地址。获取Kernel32.dll模块加载基址的代码如下。

```
HMODULE GetKernel32Addr(){
    HMODULE dwKernel32Addr = 0;
    __asm{
        push eax
        mov eax, DWORD ptr fs : [0x30]     // eax = PEB 的地址
        mov eax, [eax + 0x0C]              // eax = 指向 PEB_LDR_DATA 结构的指针
        mov eax, [eax + 0x1C] // eax = 模块初始化链表的头指针 InInitializationOrderModuleList
        mov eax, [eax]                     // eax = 列表中的第二个条目
        mov eax, [eax]                     // eax = 列表中的第三个条目
        mov eax, [eax + 0x08]              // eax = 获取到的 Kernel32.dll 模块基址
        mov dwKernel32Addr, eax
        pop eax
    }
    return dwKernel32Addr;
}
```

解密代码段的代码如下。

```
void DeXorCode(){
     PBYTE pCodeBase = (PBYTE)g_stcShellData.dwCodeBase + dwImageBase;
     DWORD dwOldProtect = 0;
     g_pfnVirtualProtect(pCodeBase, g_stcShellData.dwXorSize, PAGE_EXECUTE_READWRITE,
&dwOldProtect);
     for (DWORD i = 0; i < g_stcShellData.dwXorSize; i++)
          CodeBase[i] ^= i;              //再次异或解码
     g_pfnVirtualProtect(pCodeBase, g_stcShellData.dwXorSize, dwOldProtect, &dwOldProtect);
}
```

由于加壳程序的重定位指针指向了 Shell.dll 模块的重定位，且 PE 加载器在加载目标文件的时候对 Shell.dll 模块的代码进行了重定位，所以本应给原程序进行的重定位就需要在 Shell.dll 模块上实现，此项目原程序的重定位表并没有遭到破坏（有些壳会对重定位表进行破坏或加密）。

重定位表最终指向的是一个需要重定位的地址，这个地址是基于原目标 PE 文件默认基址（一般为 0x00400000）的地址，原 PE 文件的默认基址已经被保存过，所以只需要遍历原 PE 文件的重定位表，计算出重定位后的地址再将之填充回去就可以了，其计算公式为重定位后的地址＝需要重定位的地址－默认加载基址＋当前真实的加载基址，对应代码如下。

```
void RecReloc(){
     typedef struct _TYPEOFFSET{
          WORD offset : 12;              //偏移值
          WORD Type : 4;                 //重定位属性(方式)
     }TYPEOFFSET, *PTYPEOFFSET;
     //1.获取重定位表的结构体指针
     PIMAGE_BASE_RELOCATION pPEReloc = (PIMAGE_BASE_RELOCATION)(dwImageBase + g_
stcShellData.stcPERelocDir.VirtualAddress);
     //2.开始修复重定位
     while (pPEReloc->VirtualAddress){
```

```
        //2.1 修改内存属性为可写
        DWORD dwOldProtect = 0;
        g_pfnVirtualProtect((PBYTE)dwImageBase + pPEReloc->VirtualAddress,
            0x1000, PAGE_EXECUTE_READWRITE, &dwOldProtect);
        //2.2 修复重定位
        PTYPEOFFSET pTypeOffset = (PTYPEOFFSET)(pPEReloc + 1);
        DWORD dwNumber = (pPEReloc->SizeOfBlock - 8) / 2;
        for (DWORD i = 0; i < dwNumber; i++){
            if ( *(PWORD)(&pTypeOffset[i]) == NULL)
                break;
            //RVA
            DWORD dwRVA = pTypeOffset[i].offset + pPEReloc->VirtualAddress;
            //FAR 地址
            DWORD AddrOfNeedReloc = *(PDWORD)((DWORD)dwImageBase + dwRVA);
            *(PDWORD)((DWORD)dwImageBase + dwRVA) =
                AddrOfNeedReloc - g_stcShellData.dwPEImageBase + dwImageBase;
        }
        //2.3 恢复内存属性
        g_pfnVirtualProtect((PBYTE)dwImageBase + pPEReloc->VirtualAddress,
            0x1000, dwOldProtect, &dwOldProtect);
        //2.4 修复下一个区段
        pPEReloc = (PIMAGE_BASE_RELOCATION)((DWORD)pPEReloc + PEReloc->SizeOfBlock);
    }
```

修复 IAT 表的具体过程是通过导入表的指针遍历导入表信息(里面保存着需要导入的函数的名称和所在模块),然后加载这些模块,并从中获取函数地址,将之填到正确的 IAT 位置,对应代码如下。

```
void RecIAT(){
    //1.获取导入表结构体的指针
    PIMAGE_IMPORT_DESCRIPTOR pPEImport =
        (PIMAGE_IMPORT_DESCRIPTOR)(dwImageBase + g_stcShellData.stcPEImportDir.
VirtualAddress);
    //2.修改内存属性为可写
    DWORD dwOldProtect = 0;
    g_pfnVirtualProtect(
        (LPBYTE)(dwImageBase + g_stcShellData.dwIATSectionBase), g_stcShellData.
dwIATSectionSize,
        PAGE_EXECUTE_READWRITE, &dwOldProtect);
    //3.开始修复 IAT
    while (pPEImport->Name){
        //获取模块名
        DWORD dwModNameRVA = pPEImport->Name;
        char* pModName = (char*)(dwImageBase + dwModNameRVA);
        HMODULE hMod = g_pfnLoadLibraryA(pModName);
        //获取 IAT 信息(有些 PE 文件 INT 是空的,最好用 IAT 解析,也可两个都解析并做对比)
        PIMAGE_THUNK_DATA pIAT = (PIMAGE_THUNK_DATA)(dwImageBase + pPEImport->
FirstThunk);
        //获取 INT 信息(同 IAT 一样,可将 INT 看作是 IAT 的一个备份)
```

```
            //PIMAGE_THUNK_DATA pINT = (PIMAGE_THUNK_DATA)(dwImageBase + pPEImport->
OriginalFirstThunk);
            //通过 IAT 循环获取该模块下的所有函数信息(这里只获取了函数名)
            while (pIAT->u1.AddressOfData){  //判断是输出函数名还是序号
                  if (IMAGE_SNAP_BY_ORDINAL(pIAT->u1.Ordinal)){ //输出序号
                        DWORD dwFunOrdinal = (pIAT->u1.Ordinal) & 0x7FFFFFFF;
                        DWORD dwFunAddr = g_pfnGetProcAddress(hMod, (char*)dwFunOrdinal);
                        *(DWORD*)pIAT = (DWORD)dwFunAddr;}
                  else{
                        //输出函数名
                        DWORD dwFunNameRVA = pIAT->u1.AddressOfData;
                        PIMAGE_IMPORT_BY_NAME pstcFunName = (PIMAGE_IMPORT_BY_NAME)
(dwImageBase + dwFunNameRVA);
                        DWORD dwFunAddr = g_pfnGetProcAddress(hMod, pstcFunName->Name);
                        *(DWORD*)pIAT = (DWORD)dwFunAddr;
                  }
                  pIAT++;
             }
             pPEImport++;                        //遍历下一个模块
      }
      g_pfnVirtualProtect(                       //4.恢复内存属性
            (LPBYTE)(dwImageBase + g_stcShellData.dwIATSectionBase), g_stcShellData.
dwIATSectionSize, dwOldProtect, &dwOldProtect);
}
```

最后外壳程序运行结束,根据重定位表跳到程序 OEP,将控制权交还给程序。

3.5.4　任务四：调试脱壳

用 ASPack 对 nixiang.exe 加壳,再将加壳后的 nixiang.exe 加载到调试器 OD 中,如图 3-25 所示。

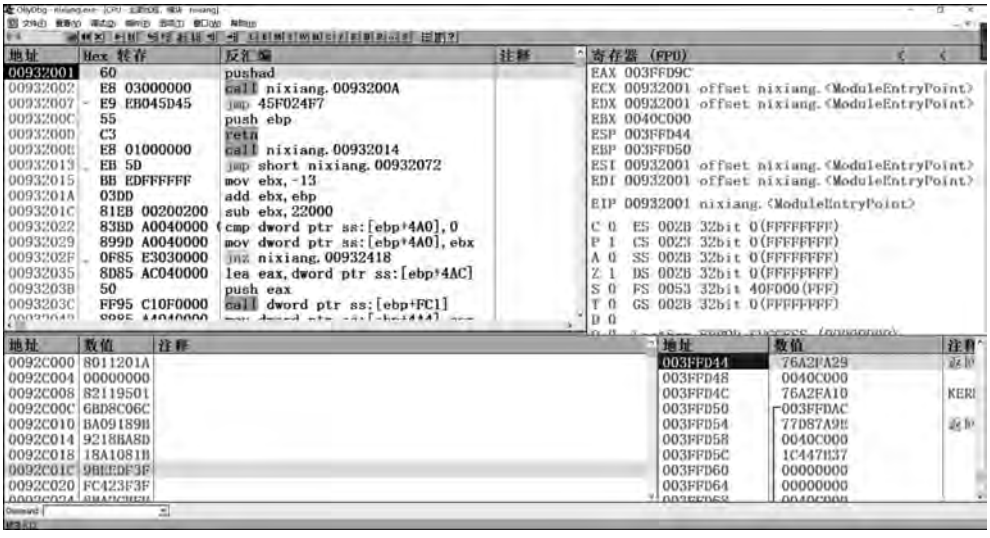


图 3-25　OD 加载 nixiang.exe

显示加壳后的 nixiang. exe，第 1 个汇编指令是 pushad，寄存器 ESP 的值为 003FFD44，然后按 F8 键单步调试，此时 ESP 的值变为 003FFD24，如图 3-26 所示。

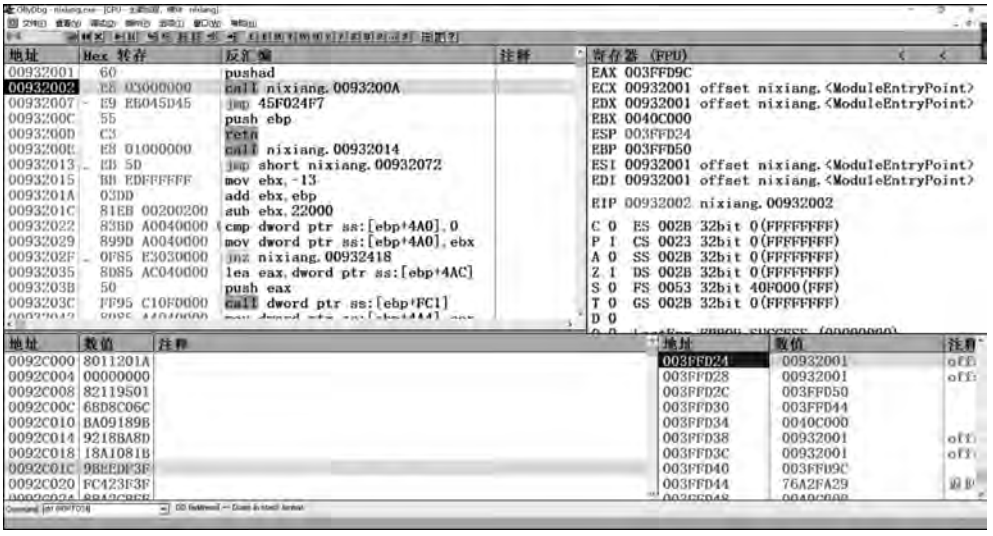


图 3-26　使用 F8 键单步调试

在命令行中，输入 dd 003FFD24。在数据窗口显示地址 003FFD24 中的数据，右击地址 003FFD24 的上一条地址(即 003FFD20)，选择“断点”→“硬件访问”→Byte 菜单项，如图 3-27 所示。

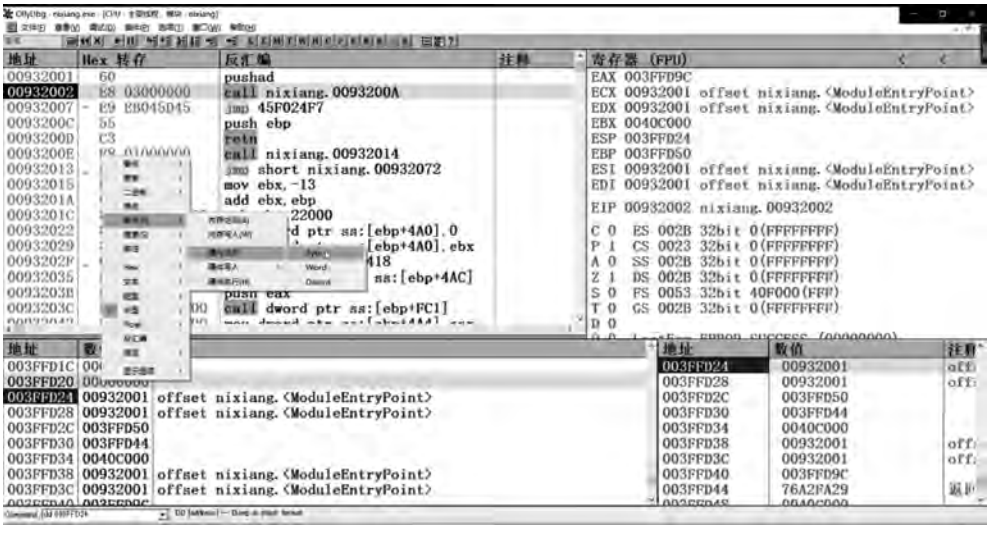


图 3-27　设置硬件访问断点

按下 F9 键运行程序，程序会在断点处暂停。选择工具栏上的 E 按钮，确定程序基址为 00910000，如图 3-28 所示。

然后单击工具栏上的 C 按钮，返回反汇编窗口并右击，选择“脱壳在当前调试的进

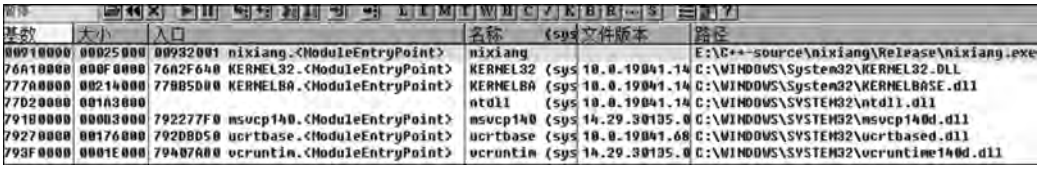
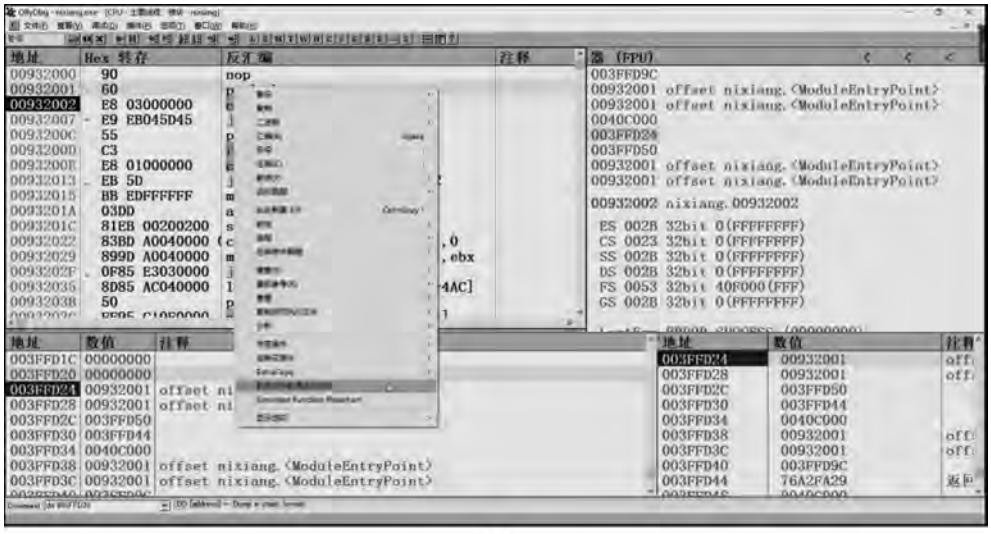


图 3-28　程序基址(见彩插)

程”菜单项,如图 3-29(a)所示。在弹出的窗口中先修改起始地址为程序基址,然后单击“获取 EIP 值作为 OEP”按钮,最后单击“脱壳之”按钮,保存文件为 nixiang_dump.exe,如图 3-29(b)所示。



(a)

(b)

图 3-29　脱壳在当前调试的进程

检测 nixiang_dump.exe,可发现其已无壳且运行正常,这表示脱壳成功。

3.6 思 考 题

1. 软件加壳对软件有什么保护作用？
2. 软断点和硬断点被访问时的区别是什么？