**用C++实现的壳（基础版）**

场景：

1. 原程序加一块区段，壳部分代码移植进去。
2. 程序启动前获得控制权，执行完壳代码后再将控制权交还回去。
3. 对代码段进行简单的异或加密。
4. 导入表修复。
5. 原程序开启了随机地址，则对原程序进行重定位修复。

问题一：为什么修复导入表（填充API信息到IAT）

因为先执行的shell部分，原程序的IAT还没有填充需要的API信息。

问题二：为什么“如果原程序开启了随机地址，则对原程序进行重定位修复”

因为原程序重定位表对应的地址是基于默认基址的，如果程序基址是随机的，就需要对其进行修改了。

Dll和exe的重定位表对应的地址都是基于默认基址后的地址。

**第一部分：加壳部分的编写**

具体流程：

1. 读取PE到内存，并保存PE文件关键信息
2. 利用关键信息对PE文件的代码段进行加密
3. 将重要信息保存到shell（pack部分和shell部分的数据交换）。

PE文件操作在两个时候，加壳前pack部分实现和加完壳、程序运行的时候由shell部分实现。前者可在主程序中调用保存的PE关键信息，后者直接让pack部分把需要的信息保存在shell壳。即pack部分和shell部分的数据交换。

实际操作：shell部分导出一个结构体供pack使用，pack将shell加载到内存后，获取结构体的地址，然后将信息传递到结构体，然后生成文件的时候一并保存在被加壳的PE文件中。

《4》将shell部分添加到PE文件

《4.1》之前LoadLibrary(L"Shell.dll")加载了Shell，但为了方便还是申请了一块空间专门存放Shell，同时获取一下Shell的镜像大小，为增加区段做准备。

《4.2》由于shell默认基址为0x10000000，而将dll移植到默认基址为0x00400000的exe文件。又某些程序会有随机基址，所以重定位是必须的。

由于shell部分最先执行，不妨让系统直接给shell重定位，然后在shell部分手动实现原程序的重定位。

《4.3》由于loadlibrary已经重定位，所以首先就是恢复重定位之前的原始信息。

由于我们shell部分是加载到PE文件的末尾，所以加上新的基址后还要+代码基址（PE文件镜像大小）

当然，由于要重定位的信息是我们自己添加的，需要通过修改PE文件目录表中的重定位信息来告诉系统应该去哪找重定位表。

《4.4》修改加壳后的OEP，指向shell，shell部分导出的结构体的第一个变量即是shell部分启动函数的地址。

《5》合并shell到PE，并修改头文件信息，保存时直接dump，并修改文件对齐大小为内存对齐大小。

**第二部分：外壳程序(Shell.dll)的编写**

1. 通过代码实现导入Shell部分所需要的函数，这样没有导入表
2. 解密代码段
3. 修复原PE文件的重定位

只需要遍历原PE文件的重定位表，然后通过一个公式计算出重定位后的地址再填充回去就可以了。

计算公式：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。

1. 修复原程序的导入表(IAT)信息

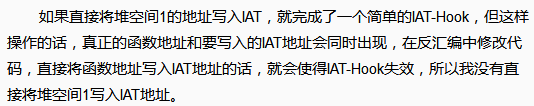
导入表用途是保存该PE文件用到的API信息。由于每次启动程序API函数加载地址不一样，所以无法直接保存一个函数的地址，而是保存这些函数的信息。PE加载器会通过导入表信息将函数地址填充到IAT，这样程序才能正常调用API。

1. 跳转到原程序入口点，将控制权交还给程序。

**用C++实现的壳（扩展版）**

《1》IAT加密，定义自己的导入表数据结构，读取被保护程序的导入表信息并保存到自己的导入表数据结构，然后抹去被加壳程序的导入表数据。

《2》shell部分直接从自己定义的导入表数据结构恢复IAT所用的信息，将得到的真正的函数地址填入一个new出来的堆空间1内保存，最终再调用真正的函数地址。



《3》而是将堆空间1写入了一段又new出来的堆空间2，然后再在别的函数中，将堆空间2地址写入IAT地址。这样保证真正函数地址和IAT地址不同时出现。这样的话，如果脱壳者在不了解我自己定义的导入表数据结构的情况下，是很难修复IAT并成功脱壳的。

**破解扩展版的IAT加密**

采用已知明文破解的方法

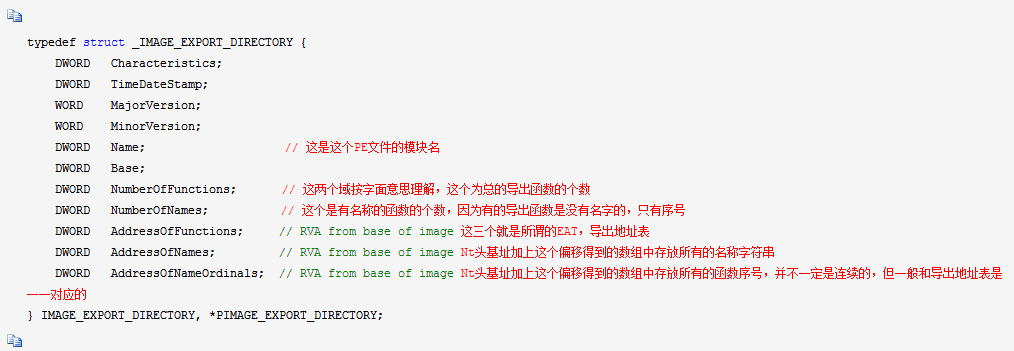
1. 用vc编写一个静态加载自己dll的pe程序
2. Pe编辑工具打开自己的dll，查看导出表在文件的地址，根据对齐值计算加载到内存时的位置并记录
3. 对pe程序进行加壳处理，用OD加载被加壳的程序，拿刚才计算好的IAT地址加上dll模块的加载地址，就可以知道IAT地址。

说的好像静态加载的dll的导出表的地址就是PE程序的IAT地址一样。

在PE文件中，所有DLL对应的导入地址数组是被排列在一起的，全部这些数组的组合也被称为导入地址表(Import Address Table)，导入表中第一个IMAGE\_IMPORT\_DESCRIPTOR结构的FirstThunk字段指向的就是IAT的起始地址。

导出表是由DataDirectory[0]中的目录项指出：

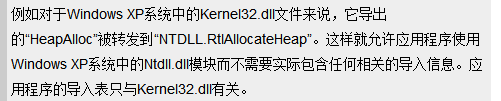
DataDirectory每一项都由一个地址和大小组成，这就告诉我们导出表的基地址和其大小。

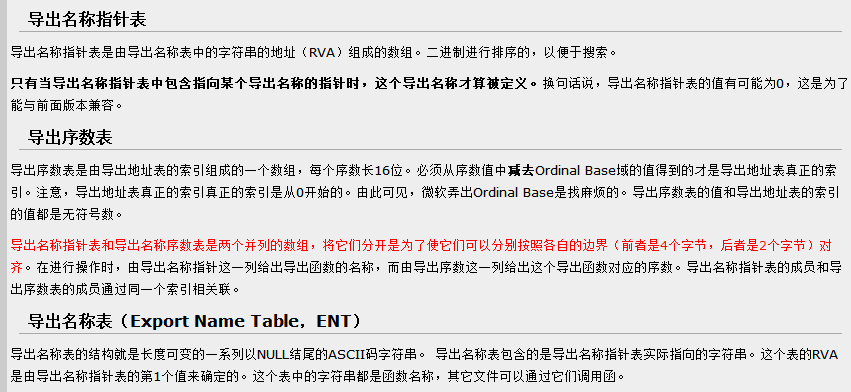


导出地址表：如果指定的地址不是位于导出节（其地址和长度由NT头给出）中，那么这个域就是一个Export RVA；否则这个域是一个Forwarder RVA，它给出了一个位于其它DLL中的符号的名称。



Forwarder RVA导出了其它镜像中定义的函数，使它看起来好像是当前镜像导出的一样。因此对于当前镜像来说，这个符号同时既是导入函数又是导出函数。





动态库多处调用只拷贝一份，静态库（lib）多处调用拷贝多份；动态库可以静态链接（含有导入库lib文件，开源了），使用时直接使用函数名就行；动态库动态链接（仅有dll即未开源），则只能由loadlibrary和getprocaddress调用函数。

<http://blog.csdn.net/iw1210/article/details/29615569>