# 逆向分析技术

## 实验六文件加壳分析

实验内容：

按照授课计划，6月3日课程为实验课，进行实验6。本次实验需要用到在实验5中完成的PE文件解析器。

1. 安装配置UPX和ZProtect两款加壳工具；
2. 选择一个EXE文件，分别用UPX和ZProtect加壳处理；
3. 分别对原文件，UPX加壳后的文件和ZProtect加壳后的文件使用在实验5中完成的PE文件解析器进行解析。

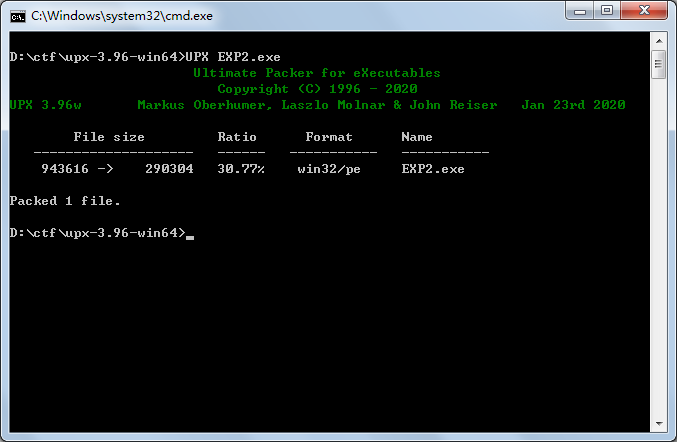
实验要求：

请各位同学通过QQ作业于6月7日前在线提交如下内容：

1. 比较三者（原文件，UPX加壳后的文件和ZProtect加壳后的文件）的分析结果，找出差异（不需要把所有的运行结果都截图，只需要差异之处的截图，然后加上文字描述）；
2. 分析为什么会出现这样的差异（提示：结合UPX和ZProtect的工作原理）。

UPX是一个控制台应用程序，以命令行方式进行操作。

**upx test.exe 加壳**



**upx -d test.exe 脱壳**

<http://blog.nsfocus.net/malicious-sample-analysis-manual-file-encapsulation/>

加壳可执行文件的两个主要目的是缩小程序的大小，阻碍对加壳程序的探测和分析。虽然加壳程序的种类繁多，但它们都遵循相似的模式：将一个可执行文件转换创建一个新的可执行文件，被转换的可执行文件在这个新的可执行文件中作为数据存储，另外新的可执行文件还包括一个供操作系统调用的脱壳存根(stub)。

在讲解加壳后的文件结构前，先要了解一下加壳器的工作原理。

**剖析加壳**

所有的加壳器都是将一个可执行文件作为输入，输出一个新的可执行文件。被加壳的可执行文件经过压缩，加密或者其他转换，目的是使他们难以被识别，难以被逆向分析。

多数加壳器用压缩算法压缩可执行文件，通过加密原始可执行文件并且实施一些反逆向技术实现，如对抗反汇编、反调试和反虚拟机等。加壳器既可以打包整个可执行文件，包括所有的数据与资源节，也可以仅打包代码节和数据节。

要保持原程序的功能，加壳程序需要存储程序中的导入函数表信息。这些信息可以用任何格式存储。

**脱壳存根**

未加壳的可执行程序由操作系统加载，被加壳程序中的脱壳存根也是由操作系统加载，然后它负责加载原始程序。可执行程序的入口点指向脱壳存根，而不是原始代码。原始程序通常存储在加壳程序的一个或多个附加的节中。

我们可以查看脱壳存根，理解脱壳存根的不同部分是脱壳的基本规则。因为脱壳存根不负责执行程序的主体功能，所以它通常很小。另外，它的功能也很简单，只是脱壳原始程序。一般脱壳存根会执行以下三步操作：

* 将原始程序脱壳到内存中
* 解析原始可执行文件的所有导入函数
* 将可执行程序转移到原始的程序入口点（OEP）

**加载可执行程序**

当加载一个可执行文件时，加载器会首先读取硬盘上可执行文件的PE头部信息，然后根据PE头部信息为可执行文件的各个节分配内存。然后，加载器将这些节复制到分配的内存空间中。

加壳后的可执行文件会组成PE头部，让加载器为它的节分配空间，它的节要么来自原始程序，要么是脱壳存根创建的节。脱壳存根会脱壳每个节的代码，并将它们复制到分配的内存空间中。具体使用哪种脱壳方法随加壳者的目的而定，一般情况下它们会存放在存根中。

**解析导入函数表**

未加壳的PE文件中有一个节段告诉加载器需要导入哪些函数，同时还有一个节存储了需要导入的函数名字与地址。Windows加载器读取导入信息，确定需要导入哪些函数，然后填入导入函数的地址。

Windows加载器不能读取被加壳可执行程序的导入函数表。对于加壳过的可执行文件，脱壳存根负责解析导入函数表，具体方法取决于使用的壳。

脱壳存根最常使用的方法是仅导入LoadLibrary和GetProcAddress两个函数。脱壳存根在脱壳出原始可执行文件之后，才能读取可执行文件的导入函数信息。为了将每个DLL加载到内存，脱壳存根将调用LoadLibrary函数导入每个函数库。然后使用GetProcAddress获取每个函数的内存地址。

另外一种方法是保持原始导入函数表的完整，让Windows加载器能够加载所有DLL和导入函数。这是最简单的方法，然而静态分析加壳程序就可以发现所有原始导入表，所以这种方法缺乏隐蔽性，此外，导入函数以明文存储在可执行文件中，因此这种方法的压缩性也不理想。

第三种方法是为原始导入表中的每个DLL保留一个函数。分析时，这种方法只能查看每个导入库中的一个函数，因此它比第二种方法的隐蔽性更高，但分析时仍然能够看到原始可执行文件所有的导入库。因为导入库不需要被脱壳存根加载，所以这种加壳方法比第一种方法简单，但脱壳存根仍需要解析大部分导入函数。

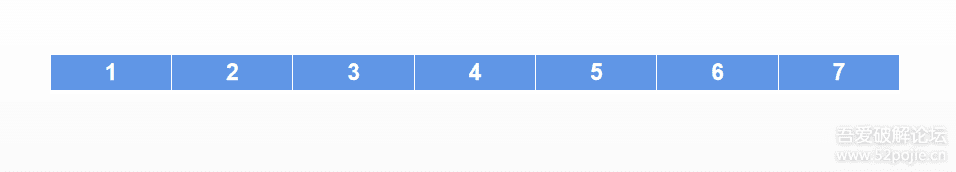
最后一种方法是不导入任何函数。加载程序在不用函数的前提下，自己从库中查找所有需要的函数，或者加壳程序首先找到LoadLibrary函数和GetProcAddress函数，然后用它们定位其他的库。这种方法的好处是加壳程序不导入任何函数，因此具有很高的隐蔽性，然而，为了使用这种方法，脱壳存根必须很复杂。

**尾部跳转**

一旦脱壳存根完成脱壳，它就必须转到OEP运行。转到OEP的指令通常被叫做尾部跳转指令。jump指令是最简单且最流行的转移运行指令。它非常普通，所以多数恶意的加壳程序试图使用ret或者call指令来隐藏这种行为。有时，恶意代码会使用操作系统转移控制的函数来掩盖尾部跳转，例如使用函数NtContinue或者ZwContinue。

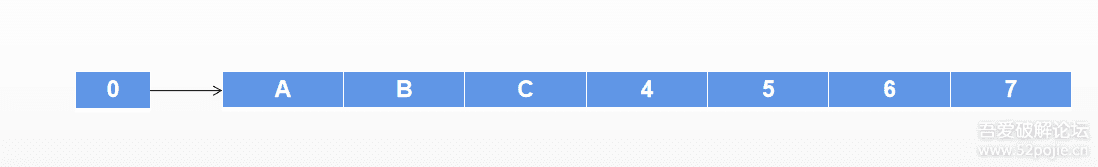
1. **UPX原理**  
   <http://www.doc88.com/p-7975097931214.html>
2. <https://www.52pojie.cn/thread-1161205-1-1.html>  
   虽然在开头讲了（也算打自己的脸吧），但是理论确实能够帮助理解。  
     
   UPX这个壳的产生初始的目的起始是为了压缩，它产生的那个年代，机器的内存容量还很小，为了便于文件的传输，压缩文件是一个很有效的手段，UPX应运而生。到了后来才演变为为了安全为目的，当然对现在的一些分析工具，UPX的安全效能已经很低了

UPX的原理其实比较容易理解。假设原始程序是分为以下几个段



那么UPX会在程序入口点加上自己的壳代码，同时原来的1、2、3被UPX压缩算法所压缩（假设压缩之后变为A、B、C）

之后程序开始运行，这个时候会执行0处的代码，这个代码的功能是将A、B、C的代码解压，恢复成1、2、3，这样程序的执行与源程序是一样的。



差异：

1. IMAGE\_FILE\_HEADER里

Section数目



2. IMAGE\_OPTIONAL\_HEADER里

SizeOfInitializedData：已初始化数据块大小，编译时所构成的块的大小。

SizeOfImage：映象装入内存后总尺寸

SizeOfHeaders：MS头，PE头，SECTION表大小



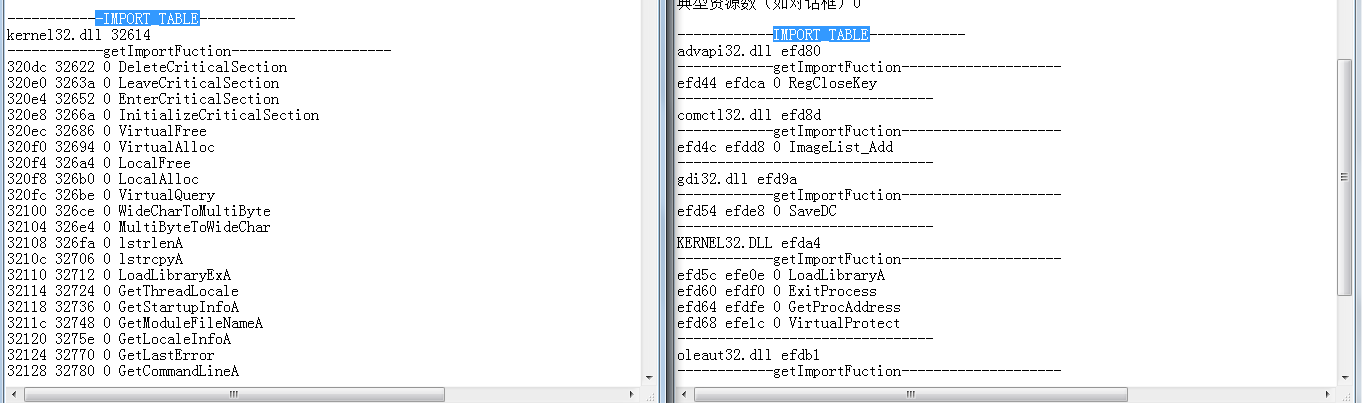
各种段的起始地址



DATA\_DIRECTORY里的元素种类，大小和地址

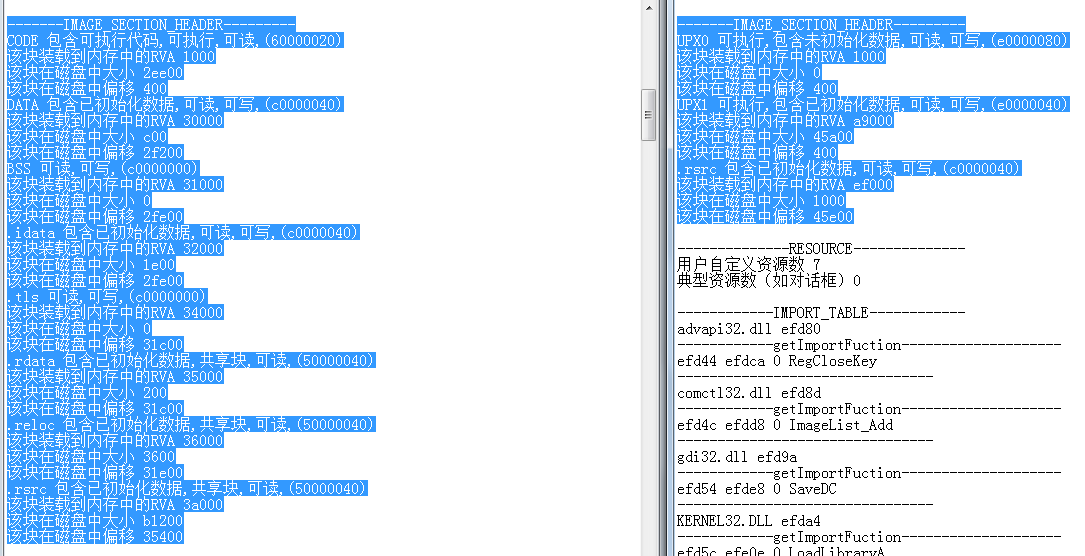


比如导入表，压缩后导入表的函数全部被隐藏了



3. IMAGE\_SECTION\_HEADER里

节数和每节的大小都不同：可以看出被压缩后的exe文件各块的名称已经被修改。大小和数量也有所不同。



Zprotect壳

可以看出Zprotect主要是对导入表加密

1. IMAGE\_FILE\_HEADER里

Section数目



2. IMAGE\_OPTIONAL\_HEADER里

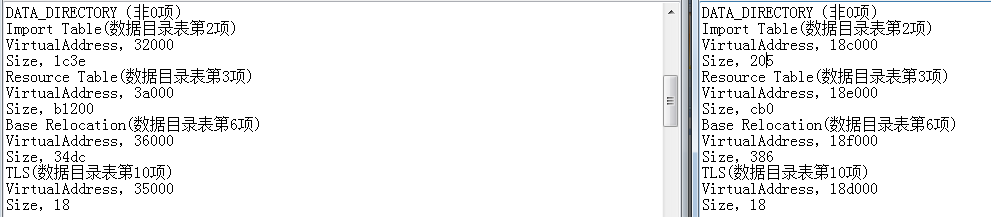
SizeOfImage：映象装入内存后总尺寸



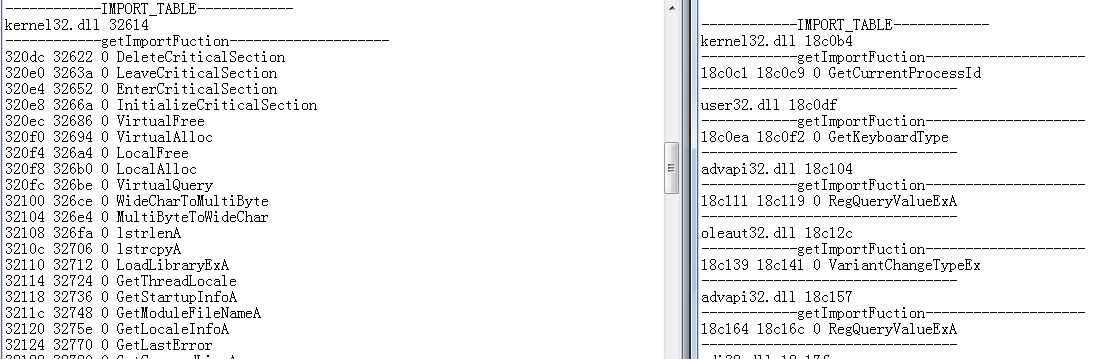
程序入口相对地址



DATA\_DIRECTORY里的元素大小和地址



比如导入表，压缩后导入表的函数全部被隐藏了



3. IMAGE\_SECTION\_HEADER里

每节的虚拟地址不同，各节的顺序也不同：

