



第二讲

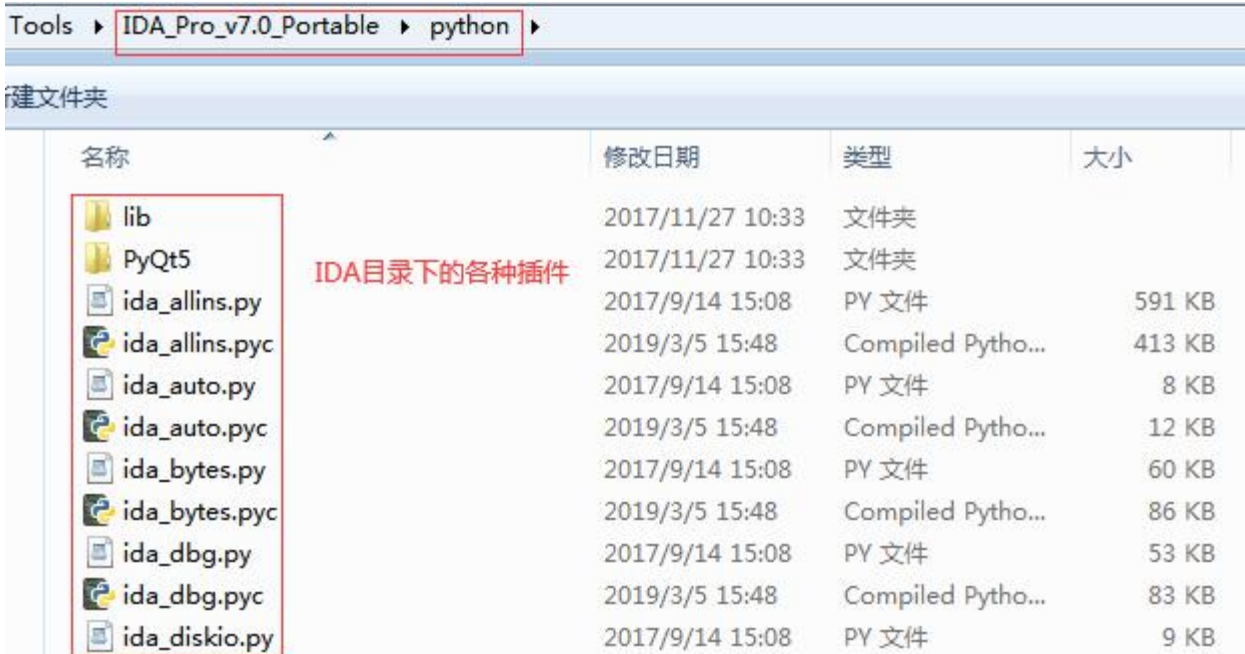
鲁宏伟/luhw@hust.edu.cn

PE和ELF文件格式

Python语言简介

- ◆ 作为一门面向对象的开源编程语言，Python易于理解和扩展，并且使用起来非常方便。
- ◆ Python的创始人为荷兰人吉多·范罗苏姆。1989年圣诞节期间，在阿姆斯特丹，吉多为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，作为ABC语言的一种继承。之所以选中Python（大蟒蛇的意思）作为该编程语言的名字，是取自英国20世纪70年代首播的电视喜剧《蒙提·派森的飞行马戏团》（Monty Python's Flying Circus）。

Python语言简介



Tools > IDA_Pro_v7.0_Portable > python >				
新建文件夹				
名称	修改日期	类型	大小	
lib	2017/11/27 10:33	文件夹		
PyQt5	2017/11/27 10:33	文件夹		
ida_allins.py	2017/9/14 15:08	PY 文件	591 KB	
ida_allins.pyc	2019/3/5 15:48	Compiled Pytho...	413 KB	
ida_auto.py	2017/9/14 15:08	PY 文件	8 KB	
ida_auto.pyc	2019/3/5 15:48	Compiled Pytho...	12 KB	
ida_bytes.py	2017/9/14 15:08	PY 文件	60 KB	
ida_bytes.pyc	2019/3/5 15:48	Compiled Pytho...	86 KB	
ida_dbg.py	2017/9/14 15:08	PY 文件	53 KB	
ida_dbg.pyc	2019/3/5 15:48	Compiled Pytho...	83 KB	
ida_diskio.py	2017/9/14 15:08	PY 文件	9 KB	

.py 与.pyc 文件区别

Python 的程序中，是把原始程序代码放在.py 文件里，而 Python 会在执行.py 文件的时候，将.py 形式的程序编译成字节码形式的.pyc 文件，这么做的目的就是为了加快下次执行文件的速度。

Python语言简介

- ◆ ABC是由Guido参加设计的一种教学语言。
- ◆ Guido认为，ABC 这种语言非常优美和强大，是专门为非专业程序员设计的。但是ABC语言并没有成功，究其原因，Guido 认为是其非开放造成的。Guido 决心在Python 中避免这一错误。同时，他还想实现在ABC 中闪现过但未曾实现的东西
- ◆ 由于Python语言的简洁性、易读性以及可扩展性，在国外用Python做科学计算的研究机构日益增多，一些知名大学已经采用Python来讲授程序设计课程

Python语言简介-风格

- ◆ Python的作者有意地设计限制性很强的语法，使得不好的编程习惯（例如if语句的下一行不向右缩进）都不能通过编译。其中很重要的一项就是Python的缩进规则。
- ◆ 一个和其他大多数语言（如C）的区别就是，一个模块的界限，完全是由每行的首字符在这一行的位置来决定的
- ◆ 通过强制程序员们缩进（包括if，for和函数定义等所有需要使用模块的地方），Python确实使得程序更加清晰和美观

Python语言简介-执行

- ◆ Python在执行时，首先会将.py文件中的源代码编译成Python的字节码，然后再由Python虚拟机来执行这些编译好的字节码。这种机制的基本思想跟Java，.NET是一致的。然而，Python虚拟机与Java或.NET的虚拟机不同的是，Python虚拟机是一种抽象层次更高的虚拟机。
- ◆ Python还可以以交互模式运行，可以直接在命令模式下直接运行Python交互环境。

Python语言简介-基本语法

- ◆ Python的设计目标之一是让代码具备高度的可阅读性。它设计时尽量使用其它语言经常使用的标点符号和英文单字，让代码看起来整洁美观
- ◆ 它不像其他的静态语言如C、Pascal那样需要重复书写声明语句，也不像它们的语法那样经常有特殊情况 and 意外
- ◆ Python语言利用缩进表示语句块的开始和退出，而非使用花括号或者某种关键字

Python语言简介-示例

◆ #抓包代码

1. `from scapy.sendrecv import sniff`
 2. `from scapy.utils import wrpcap`
 3. `dpkt = sniff(count = 100)` #这里是针对单网卡的机器, 多网卡的可以在参数中指定网卡
 4. `wrpcap("demo.pcap", dpkt)`
- 简单的四行代码就可已完成从网卡抓取数据报文并保存在本地文件中

Python语言简介-示例

```
# This file was automatically generated by SWIG (http://www.swig.org).
# Version 2.0.12
#
# Do not make changes to this file unless you know what you are doing--modify
# the SWIG interface file instead.

from sys import version_info
if version_info >= (2,6,0):
    def swig_import_helper():
        from os.path import dirname
        import imp
        fp = None
        try:
            fp, pathname, description = imp.find_module('_ida_bytes', [dirname(__fi
        except ImportError:
            import _ida_bytes
            return _ida_bytes
        if fp is not None:
            try:
                _mod = imp.load_module('_ida_bytes', fp, pathname, description)
            finally:
                fp.close()
            return _mod
        _ida_bytes = swig_import_helper()
        del swig_import_helper
    else:
        import _ida_bytes
    del version_info
    try:
        _swig_property = property
    except NameError:
        pass # Python < 2.2 doesn't have 'property'.
```

Python的注释语句

定义函数

处理异常

不同的缩进

Python语言简介-示例

```
ida_bytes.pyc x ida_bytes.py
00000000 03 F3 0D 0A DA 2A BA 59 63 00 00 00 00 00 00 00 .....*.Yc.....
00000010 00 05 00 00 00 40 00 00 00 73 FA 14 00 00 64 00 .....@...s....d.
00000020 00 5A 00 00 64 01 00 64 02 00 6C 01 00 6D 02 00 .Z..d..d..l..m..
00000030 5A 02 00 01 65 02 00 64 45 01 6B 05 00 72 3A 00 Z...e..dE.k..r:.
00000040 64 06 00 84 00 00 5A 03 00 65 03 00 83 00 00 5A d.....Z..e.....Z
00000050 04 00 5B 03 00 6E 0C 00 64 01 00 64 07 00 6C 04 ..[..n..d..d..l.
00000060 00 5A 04 00 5B 02 00 79 0A 00 65 05 00 5A 06 00 .Z..[...y...e..Z..
00000070 57 6E 11 00 04 65 07 00 6B 0A 00 72 66 00 01 01 Wn...e..k..rf...
00000080 01 6E 01 00 58 64 08 00 64 09 00 84 01 00 5A 08 .n..Xd..d.....Z.
00000090 00 64 0A 00 84 00 00 5A 09 00 64 0B 00 84 00 00 .d.....Z..d.....
000000a0 5A 0A 00 64 0C 00 84 00 00 5A 0B 00 79 10 00 65 Z..d.....Z..y..e
000000b0 0C 00 5A 0D 00 64 08 00 5A 0E 00 57 6E 2A 00 04 ..Z..d..Z..Wn*..
000000c0 65 0F 00 6B 0A 00 72 CA 00 01 01 01 64 0D 00 64 e..k..r.....d..d
000000d0 46 01 64 0E 00 84 00 00 83 00 00 59 5A 0D 00 64 F.d.....YZ..d
000000e0 05 00 5A 0E 00 6E 01 00 58 64 0F 00 84 00 00 5A ..Z..n..Xd.....Z
000000f0 10 00 79 19 00 64 01 00 64 07 00 6C 11 00 5A 11 ..y..d..d..l..Z.
00000100 00 65 11 00 6A 12 00 5A 13 00 57 6E 10 00 01 01 .e..j..Z..Wn....
```

PE文件格式概述

◆ PE: Portable Executable(可移植的执行体)

- 它是 Win32 环境自身所带的执行体文件格式。
- 它的一些特性继承自UNIX的 COFF (Common Object File Format) 文件格式。

◆ PE文件格式

- PE 文件格式有一系列的字段，固定在一个已知的位置上，定义文件的其他部分。
- PE 表头内含信息包括程序代码和资料区域的大小、位置、适用的OS以及堆栈（stack）的最初大小等。

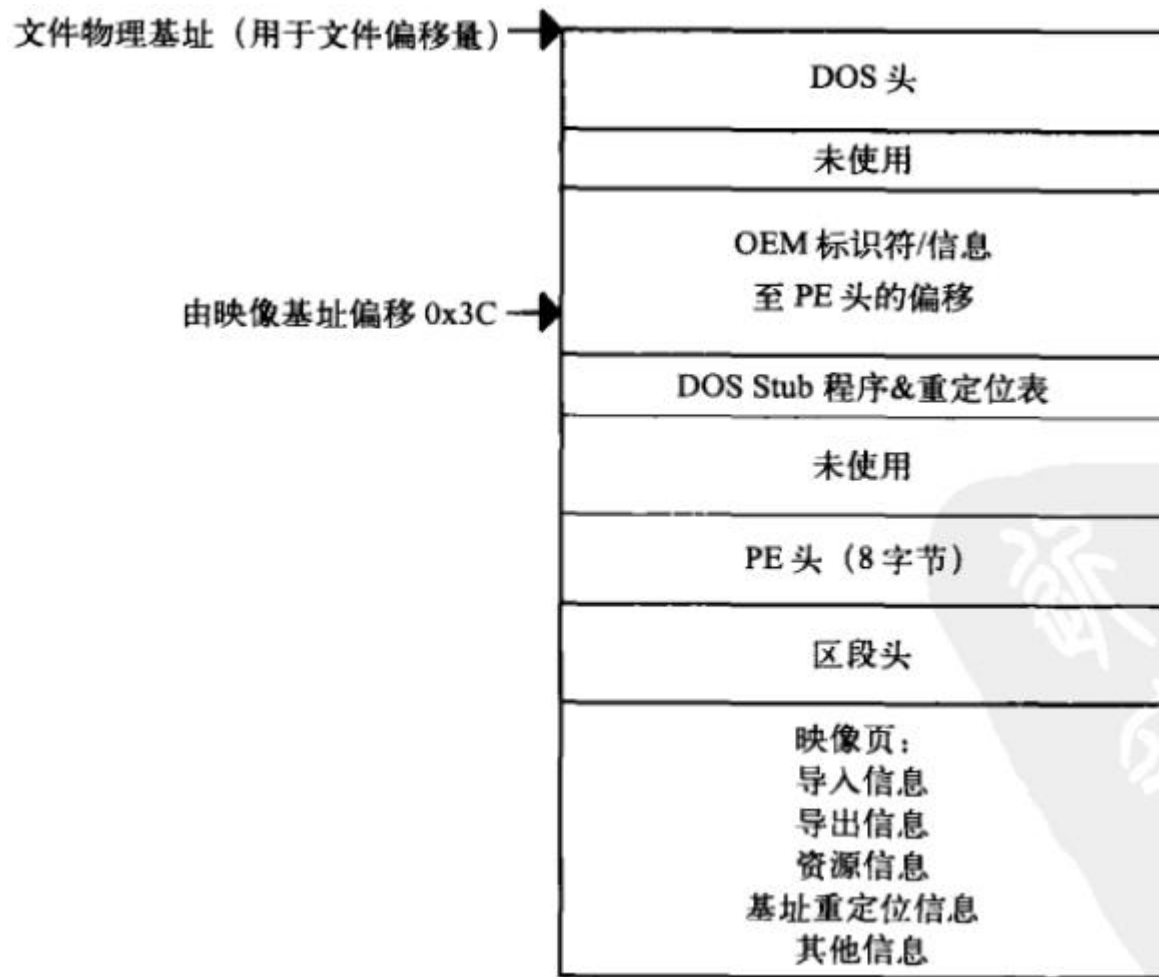
PE文件格式概述

```
AdobePDF.dll x
00000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....
00000010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000030 00 00 00 00 00 00 00 00 00 00 00 00 10 01 00 00 .....
00000040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 .....!..L.!Th
00000050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00000060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00000070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.
00000080 59 F7 3B E9 1D 96 55 BA 1D 96 55 BA 1D 96 55 BA Y.;...U...U...U.
00000090 6B 0B 2E BA 11 96 55 BA 3A 50 28 BA 1F 96 55 BA k....U.:P(...U.
000000a0 3A 50 38 BA 25 96 55 BA 3A 50 3B BA 12 96 55 BA :P8.%U.:P;...U.
000000b0 1D 96 54 BA A1 96 55 BA 3A 50 2E BA 1E 96 55 BA ..T...U.:P...U.
000000c0 6B 0B 38 BA 1C 96 55 BA 3A 50 2F BA 1C 96 55 BA k.8...U.:P/...U.
000000d0 3A 50 24 BA 14 96 55 BA 3A 50 2B BA 1C 96 55 BA :P$.U.:P+...U.
000000e0 3A 50 29 BA 1C 96 55 BA 3A 50 2D BA 1C 96 55 BA :P)...U.:P-...U.
000000f0 52 69 63 68 1D 96 55 BA 00 00 00 00 00 00 00 00 Rich..U.....
00000100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000110 50 45 00 00 64 86 05 00 4B B3 F9 47 00 00 00 00 PE..d...K..G....
00000120 00 00 00 00 F0 00 22 20 0B 02 08 00 00 94 00 00 .....
00000130 00 20 00 00 00 00 00 00 00 7A 00 00 00 10 00 00 . ....Z.....
00000140 00 00 40 00 00 00 00 00 00 10 00 00 00 02 00 00 ..@.....
00000150 06 00 00 00 06 00 00 00 05 00 02 00 00 00 00 00 .....
00000160 00 F0 00 00 00 04 00 00 93 62 01 00 03 00 00 00 .....b.....
00000170 00 00 04 00 00 00 00 00 00 10 00 00 00 00 00 00 .....
00000180 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00 .....
00000190 00 00 00 00 10 00 00 00 20 A2 00 00 72 00 00 00 .....r...
000001a0 E8 94 00 00 A0 00 00 00 00 D0 00 00 60 08 00 00 .....
000001b0 00 C0 00 00 74 04 00 00 00 B2 00 00 58 15 00 00 ....t.....X...
000001c0 00 E0 00 00 5C 00 00 00 60 14 00 00 1C 00 00 00 ....\.....
000001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001f0 00 00 00 00 00 00 00 00 00 10 00 00 38 04 00 00 .....8...
00000200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

PE文件格式概述

- ◆ 这种文件格式是由微软设计的，并于1993年被TIS委员会批准
 - TIS: Tool Interface Standard, 工具接口标准
 - TIS委员会由Microsoft, Intel, Borland, Watcom, IBM等组成
- ◆ Win32 SDK中包含一个名叫<winnt.h>的头文件，其中含有很多用于PE格式的#define和typedef定义

PE 文件布局



DOS头

- ◆ DOS头是用来兼容MS-DOS操作系统的，目的是当这个文件在MS-DOS上运行时提示一段文字，大部分情况下是：
This program cannot be run in DOS mode.
- ◆ DOS，是磁盘操作系统的缩写，是个人计算机上的一类操作系统。从1981年直到1995年的15年间，磁盘操作系统在IBM PC 兼容机市场中占有举足轻重的地位。而且，若是把部分以DOS为基础的Microsoft Windows版本，如Windows 95、Windows 98和Windows Me等都算进去的话，那么其商业寿命至少可以算到2000年。微软的所有后续版本中，磁盘操作系统仍然被保留着。

PE文件-映像

- ◆ 当一个PE文件被加载到内存中以后，我们称之为“映象” (image)
- ◆ 一般来说，PE文件在硬盘上和在内存里是不完全一样的，被加载到内存以后其占用的虚拟地址空间要比在硬盘上占用的空间大一些，这是因为各个节在硬盘上是连续的，而在内存中是按页对齐的，所以加载到内存以后节之间会出现一些“空洞”。

各种地址的概念

- **基址** (Image Base) : PE文件装入内存后的起始地址
- **相对虚拟地址** (Relative Virtual Address,RVA) : 在内存中相对于PE文件装入地址的偏移位置, 是一个相对地址
- **虚拟地址** (Virtual Address,VA) : 装入内存中的实际地址。
 - $\text{虚拟地址(VA)} = \text{基址(Image Base)} + \text{相对虚拟地址(RVA)}$
- **文件偏移** (File Offset) : PE文件存储在磁盘上时, 相对于文件头的偏移位置。16进制文件编辑器打开后的地址为文件偏移地址。

PE文件头结构以及其各个域的取值

```
0:000> dt /r1 ntdll!_IMAGE_NT_HEADERS notepad+e0
+0x000 Signature      : 0x4550
+0x004 FileHeader      : _IMAGE_FILE_HEADER
    +0x000 Machine      : 0x14c
    +0x002 NumberOfSections : 3
    +0x004 TimeDateStamp : 0x48025287
    +0x008 PointerToSymbolTable : 0
    +0x00c NumberOfSymbols : 0
    +0x010 SizeOfOptionalHeader : 0xe0
    +0x012 Characteristics : 0x10f
+0x018 OptionalHeader : _IMAGE_OPTIONAL_HEADER
    +0x000 Magic         : 0x10b
    +0x002 MajorLinkerVersion : 0x7
    +0x003 MinorLinkerVersion : 0xa
    +0x004 SizeOfCode      : 0x7800
    +0x008 SizeOfInitializedData : 0x8800
    +0x00c SizeOfUninitializedData : 0
    +0x010 AddressOfEntryPoint : 0x739d
    +0x014 BaseOfCode      : 0x1000
    +0x018 BaseOfData      : 0x9000
    +0x01c ImageBase       : 0x1000000
    +0x020 SectionAlignment : 0x1000
    +0x024 FileAlignment   : 0x200
    +0x028 MajorOperatingSystemVersion : 5
    +0x02a MinorOperatingSystemVersion : 1
    +0x02c MajorImageVersion : 5
    +0x02e MinorImageVersion : 1
    +0x030 MajorSubsystemVersion : 4
    +0x032 MinorSubsystemVersion : 0
    +0x034 Win32VersionValue : 0
    +0x038 SizeOfImage      : 0x13000
    +0x03c SizeOfHeaders    : 0x400
    +0x040 CheckSum         : 0x18ada
    +0x044 Subsystem        : 2
    +0x046 DllCharacteristics : 0x8000
    +0x048 SizeOfStackReserve : 0x40000
    +0x04c SizeOfStackCommit : 0x11000
    +0x050 SizeOfHeapReserve : 0x100000
    +0x054 SizeOfHeapCommit : 0x1000
    +0x058 LoaderFlags      : 0
    +0x05c NumberOfRvaAndSizes : 0x10
    +0x060 DataDirectory    : [16] _IMAGE_DATA_DIRECTORY
```

Signature : 类似于DOS头中的e_magic, 其高16位是0, 低16是0x4550, 用字符表示是'PE '。

IMAGE_FILE_HEADER是PE文件头, c语言的定义是这样的:

```
[cpp]
1. typedef struct _IMAGE_FILE_HEADER {
2.     WORD    Machine;
3.     WORD    NumberOfSections;
4.     DWORD    TimeDateStamp;
5.     DWORD    PointerToSymbolTable;
6.     DWORD    NumberOfSymbols;
7.     WORD    SizeOfOptionalHeader;
8.     WORD    Characteristics;
9. } IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

可以看出, PE文件头定义了PE文件的一些基本信息和属性, 这些属性会在PE加载器加载时用到, 如果加载器发现PE文件头中定义的一些属性不满足当前的运行环境, 将会终止加载该PE。

Python解析PE文件

- 下载安装pefile库
- 简单解析PE文件

```
1  #!/usr/bin/env python
2  # Python 2.7.3
3  # 打印PE文件信息
4  import os, string, shutil, re
5  import pefile ##记得import pefile
6  PEfile_Path = r"C:\Python27\pythonw.exe"
7  pe = pefile.PE(PEfile_Path)
8  # 写入文件
9  f = open('PEInfo.txt', 'w')# 写入文件
10 print >> f, pe
11 f.close()
```

各种块的描述

- PE 文件一般至少都会有两个区块：一个是代码块，另一个是数据块。每一个区块都需要有一个截然不同的名字，这个名字主要是用来表达区块的用途。
- 区块在映像中是按起始地址（RVA）来排列的，而不是按字母表顺序。
- 使用区块名字只是人们为了认识和编程的方便，而对操作系统来说这些是无关紧要的。微软给这些区块取了个有特色的名字，但这不是必须的。
- 在从PE 文件中读取需要的内容时，如输入表、输出表，不能以区块名字作为参考，正确的方法是按照数据目录表中的字段来进行定位。

各种块的描述

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	可执行文件的机制重定位, 基址重定位一般仅D11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IMAGE_RUNTIME_FUNCTION_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

几个主要块的描述

- text

- text 是在编译或汇编结束时候产生的一种块
- 它的内容全是指令代码，对它的加密可以有效地防止对原程序指令代码的静态分析和修改。

```
.text:0000000000401438 ; Segment type: Pure code
.text:0000000000401438 ; Segment permissions: Read/Execute
.text:0000000000401438 _text          segment para public 'CODE' use64
.text:0000000000401438          assume cs:_text
.text:0000000000401438          ;org 401438h
.text:0000000000401438          assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.text:0000000000401438 qword_401438      dq 0                      ; DATA XREF: sub_407328+165↓o
.text:0000000000401440 qword_401440      dq 0                      ; DATA XREF: sub_407328:loc_407486↓o
.text:0000000000401448 qword_401448      dq 0                      ; DATA XREF: sub_407328:loc_40744C↓o
.text:0000000000401450      dq offset sub_4072EC
.text:0000000000401458 qword_401458      dq 0                      ; DATA XREF: sub_407328+12B↓o
.text:0000000000401460 ; Debug Directory entries
.text:0000000000401460      dd 0                      ; Characteristics
.text:0000000000401464      dd 47F9B34Bh          ; TimeDateStamp: Mon Apr 07 05:38:19 2008
.text:0000000000401468      dw 0                      ; MajorVersion
.text:000000000040146A      dw 0                      ; MinorVersion
.text:000000000040146C      dd 2                      ; Type: IMAGE_DEBUG_TYPE_CODEVIEW
.text:0000000000401470      dd 84h                    ; SizeOfData
.text:0000000000401474      dd rva asc_401F98          ; AddressOfRawData
.text:0000000000401478      dd 1398h                 ; PointerToRawData
.text:000000000040147C      align 20h
.text:0000000000401480      dq 0
```


几个主要块的描述

- data

- data 是初始化的数据块。这些数据包括编译时被初始化的 `global` 和 `static` 变量，也包括字符串

- 连接器将 `.data` 数据块放入 `DATA` 段。大的 `.data`

- Local 变量的空间 `ata`

- 数据块是 `以` 有必要将其加密。

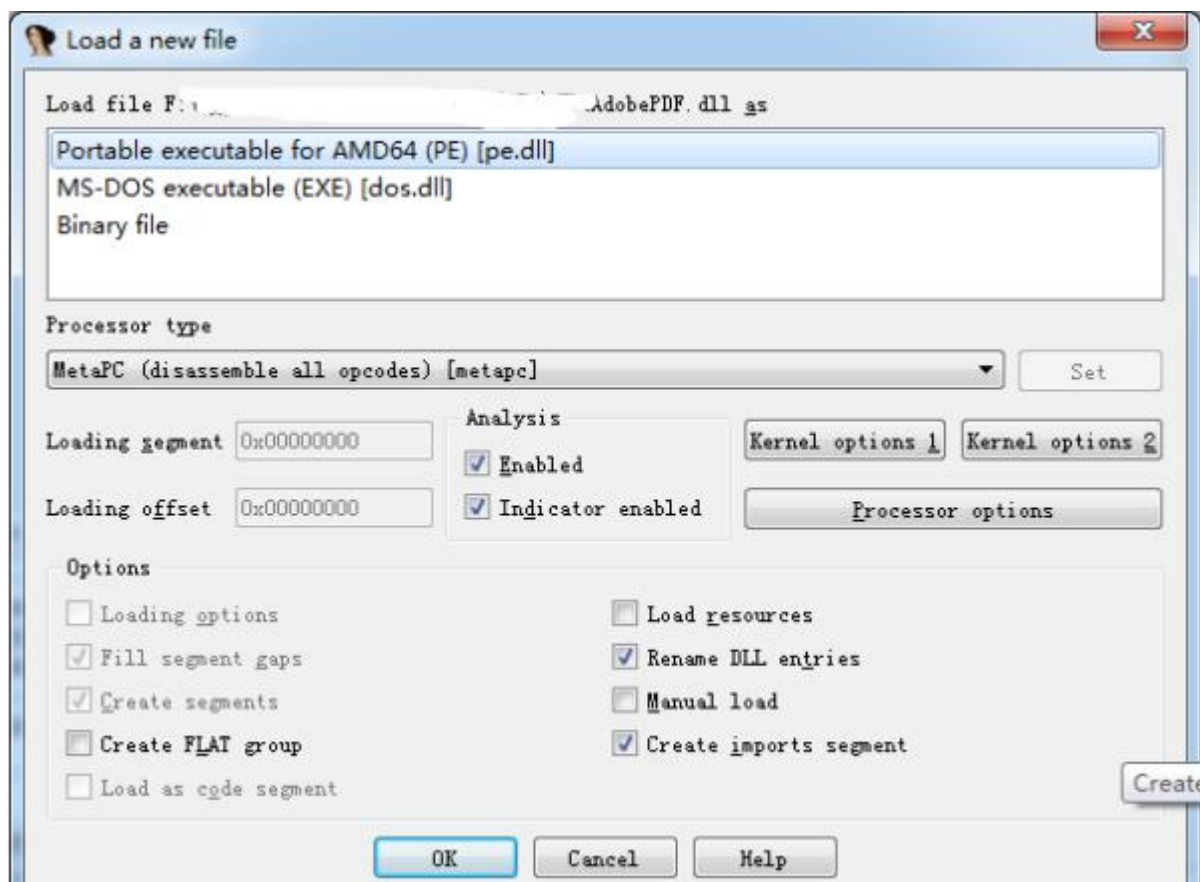
```
.data:000000000040B000 ; Section 2. (virtual address 0000B000)
.data:000000000040B000 ; Virtual size           : 00000D54 ( 3412.)
.data:000000000040B000 ; Section size in file      : 00000800 ( 2048.)
.data:000000000040B000 ; Offset to raw data for section: 00009800
.data:000000000040B000 ; Flags C0000040: Data Readable Writable
.data:000000000040B000 ; Alignment      : default
.data:000000000040B000 ; =====
.data:000000000040B000
.data:000000000040B000 ; Segment type: Pure data
.data:000000000040B000 ; Segment permissions: Read/Write
.data:000000000040B000 _data      segment para public 'DATA' use64
.data:000000000040B000          assume cs:_data
.data:000000000040B000          ;org 40B000h
.data:000000000040B000          db  0
.data:000000000040B001          db  0
.data:000000000040B002          db  0
.data:000000000040B003          db  0
.data:000000000040B004          db  0
.data:000000000040B005          db  1
.data:000000000040B006          db  0
```

几个主要块的描述

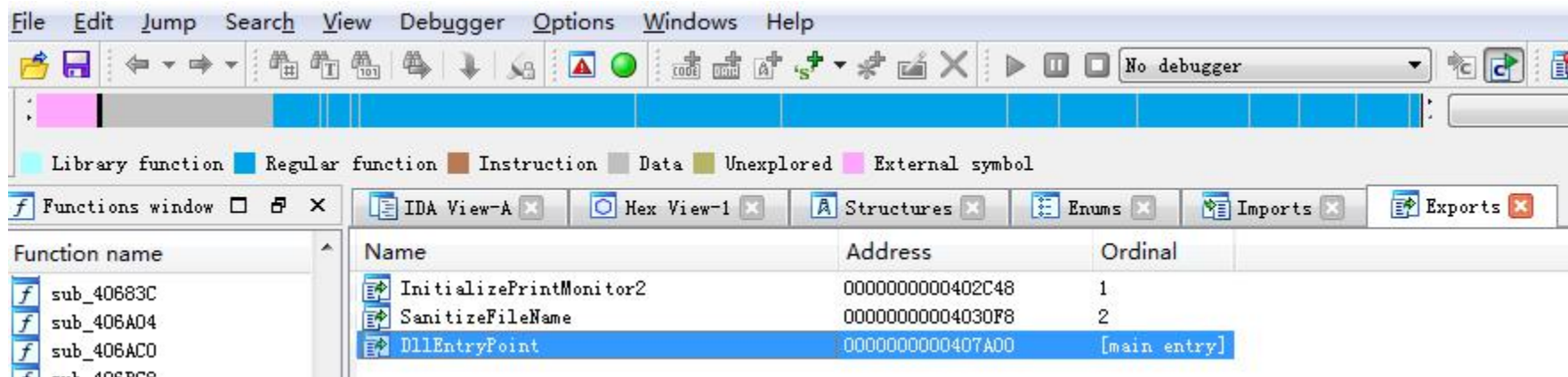
- edata
 - edata 是该 PE 文件输出函数和数据的列表，以供其他模块引用
 - 与文件中的入口表、驻留名表及非驻留名表综合功能相似
 - PE 格式的-execution文件没有必要输出一个函数，因此，通常只是在DLL文件中才可以见到edata块



PE文件结构详解-导出表



PE文件结构详解-导出表



各种块的描述

- idata

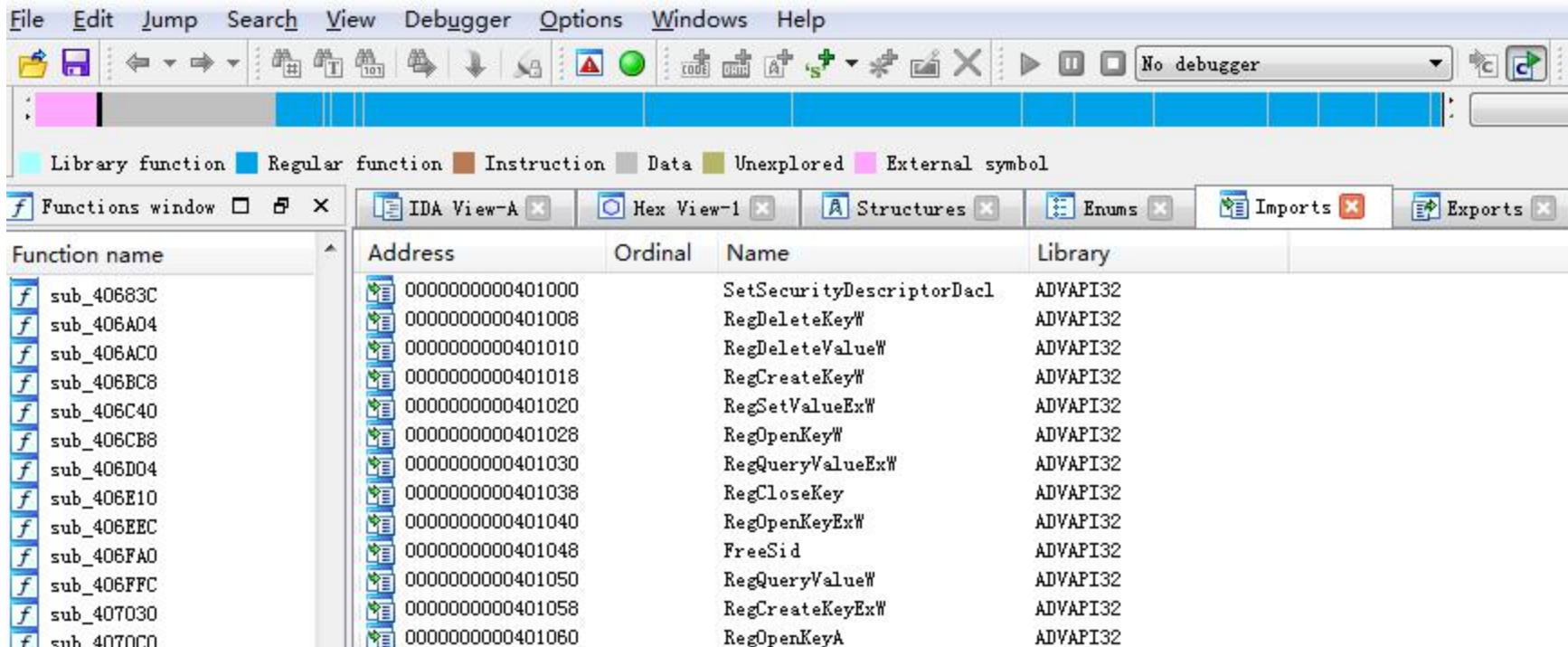
- idata 节是输入数据，包括输入目录和输入地址名字表。
- idata 包含其他外来 DLL 的函数及数据信息

```
idata:0000000000401000 ; Segment type: Externs
idata:0000000000401000 ; _idata
idata:0000000000401000 ; BOOL __stdcall SetSecurityDescriptorDacl(PSECURITY_DESCRIPTOR pSecurityDescriptor, BOOL bDaclI
                        extrn SetSecurityDescriptorDacl:qword
                        ; CODE XREF: sub_4036B4+1F4↓p
                        ; DATA XREF: sub_4036B4+1F4↓r ...
idata:0000000000401008 ; LSTATUS __stdcall RegDeleteKeyW(HKEY hKey, LPCWSTR lpSubKey)
                        extrn RegDeleteKeyW:qword
                        ; CODE XREF: sub_403B04+8C↓p
                        ; sub_403B04+D3↓p
                        ; DATA XREF: ...
```

PE文件结构详解-导入表

- 装载程序判定目标函数的地址并将该函数插补到执行文件的映像中，所需要的信息都是放在 PE 文件的 idata 中，也就是 import 块
- 由于外壳程序需要在原程序执行前进行一系列的工作，需要用到大量的Windows API调用，所以，我们必须深入地了解import块的结构和装入机制

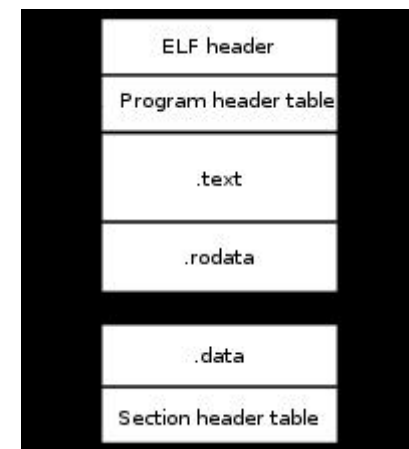
PE文件结构详解-导入表



Function name	Address	Ordinal	Name	Library
sub_40683C	0000000000401000		SetSecurityDescriptorDacl	ADVAPI32
sub_406A04	0000000000401008		RegDeleteKeyW	ADVAPI32
sub_406AC0	0000000000401010		RegDeleteValueW	ADVAPI32
sub_406BC8	0000000000401018		RegCreateKeyW	ADVAPI32
sub_406C40	0000000000401020		RegSetValueExW	ADVAPI32
sub_406CB8	0000000000401028		RegOpenKeyW	ADVAPI32
sub_406D04	0000000000401030		RegQueryValueExW	ADVAPI32
sub_406E10	0000000000401038		RegCloseKey	ADVAPI32
sub_406EEC	0000000000401040		RegOpenKeyExW	ADVAPI32
sub_406FA0	0000000000401048		FreeSid	ADVAPI32
sub_406FFC	0000000000401050		RegQueryValueW	ADVAPI32
sub_407030	0000000000401058		RegCreateKeyExW	ADVAPI32
sub_4070C0	0000000000401060		RegOpenKeyA	ADVAPI32

ELF文件概述

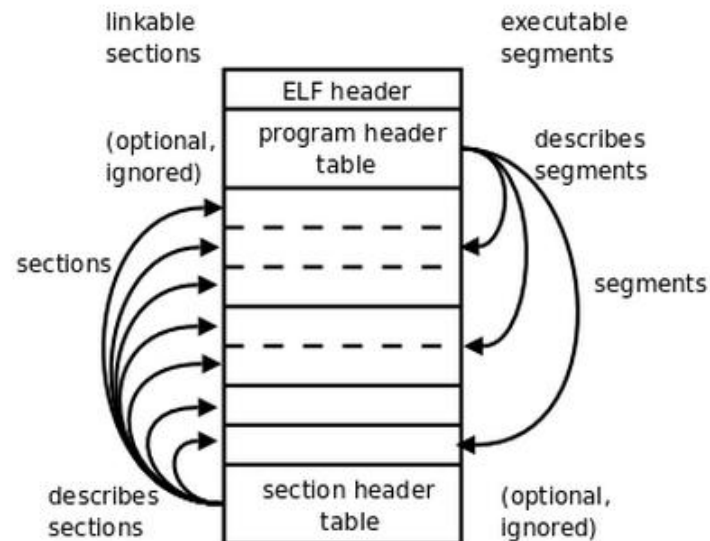
- ARM的可执行文件的格式是ELF格式文件 (Executable and Linkable Format)
- ELF文件由4部分组成, 分别是**ELF头** (ELF header)、**程序头表** (Program header table)、**节** (Section) 和**节头表** (Section header table)。
- 实际上, 一个文件中不一定包含全部内容, 而且他们的位置也未必如同所示这样安排, 只有ELF头的位置是固定的, 其余各部分的位置、大小等信息由ELF头中的各项值来决定。



参考: <https://blog.csdn.net/feglass/article/details/51469511>

ELF header

- ◆ ELF文件格式提供了两种视图，分别是**链接视图**和**执行视图**
- ◆ 链接视图是以节(section)为单位
- ◆ 执行视图是以段(segment)为单位
- ◆ **链接视图**是在链接时用到的视图
- ◆ **执行视图**则是在执行时用到的视图



ELF header

- ◆ ELF header: 描述整个文件的组织。
- ◆ Program Header Table: 描述文件中的各种segments, 用来告诉系统如何创建进程映像的。

ELF header

- ◆ 节(sections) 或者 段(segments)
- ◆ 段是从运行的角度来描述ELF文件，节是从链接的角度来描述ELF文件，也就是说，在链接阶段，我们可以忽略program header table来处理此文件，在运行阶段可以忽略section header table来处理此程序（所以很多加固手段删除了section header table）。
- ◆ 从图中我们也可以看出，segments与sections是包含的关系，一个segment包含若干个section。
- ◆ Section Header Table: 包含了各个section的属性信息。

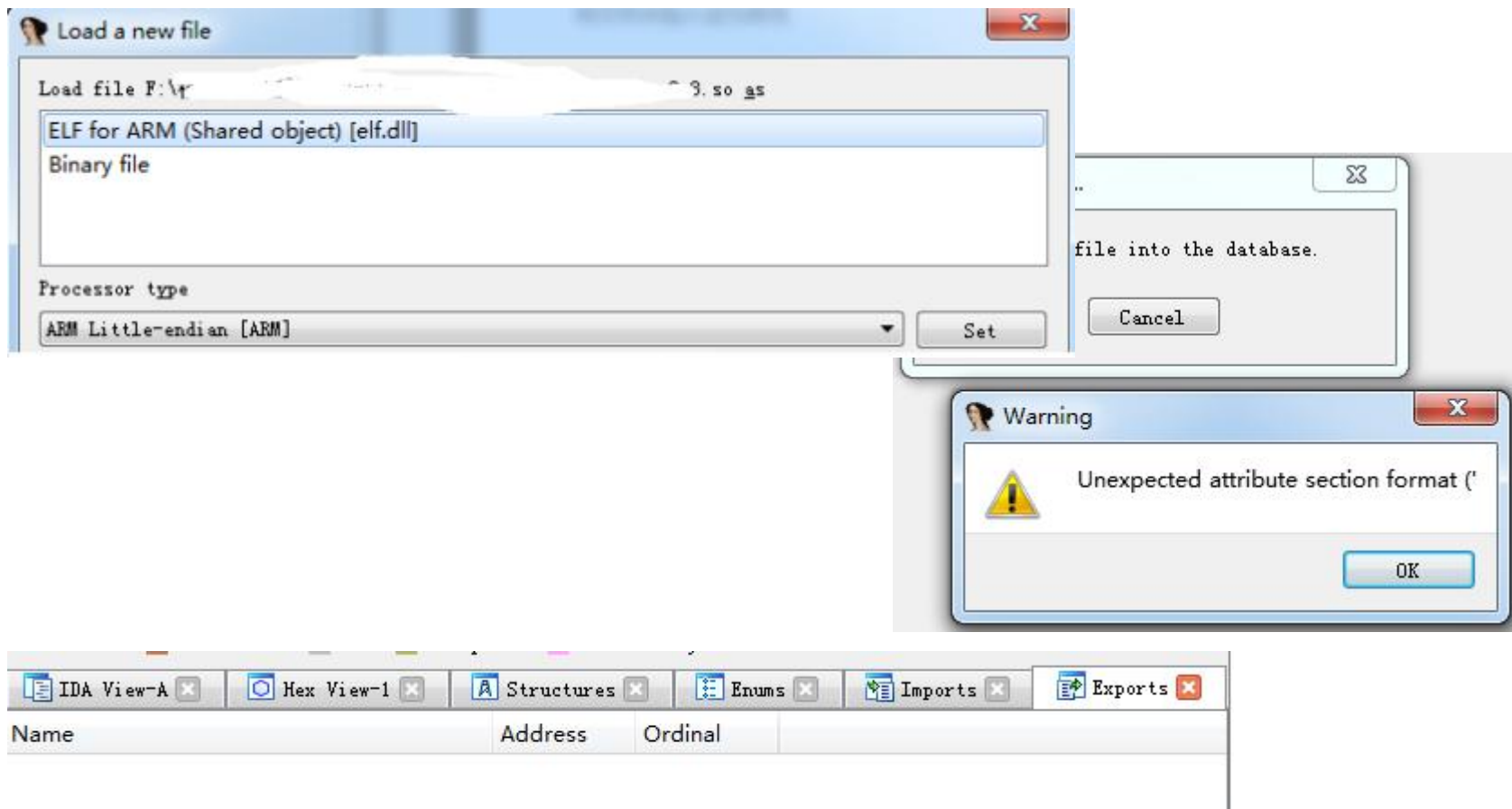
链接视图	执行视图
ELF 头部	ELF 头部
程序头部表（可选）	程序头部表
节区 1	段 1
...	
节区 n	段 2
...	
...	...
节区头部表	节区头部表（可选）

ELF header解析

- ◆ 有专门的Python库用以解析ELF文件
- ◆ pyelftools (
<https://github.com/eliben/pyelftools/wiki/User's-guide>)
- ◆ ELF
 - ✓ Extended numbering of segment and section headers.
 - ✓ The current focus of the library is on Intel's x86 and x64 architectures, with some ARM support added recently.
 - ✓ Special handling of .tbss sections .

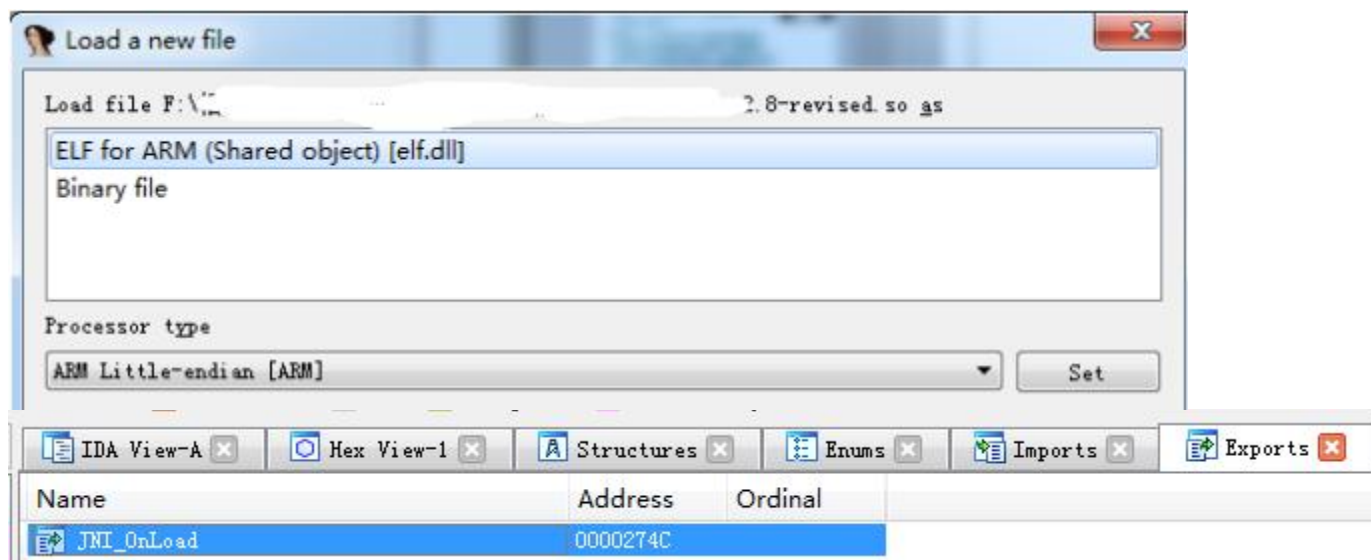
搞清楚这些结构有什么用？

- 采用IDA_Pro打开一个加壳文件



搞清楚这些结构有什么用？

- 修改一下加壳文件




```
EXPORT JNI_OnLoad
JNI_OnLoad
; CODE XREF: sub_898+8C↑p
; DATA XREF: LOAD:0000420C↓o
SVCLE 0x9E04E5
STRMIT R0, [R8], LR, LSR#19
STRVS R2, [R6], LR, ROR#25
; -----
DCD 0xF7D41ADF, 0xF9BE6C01, 0x2260B16A, 0xA0686A25, 0xC4D4DCEF
DCD 0x82280FE2, 0xDD0507A5, 0xD5710877, 0x6E61C0D7, 0x73C37386
DCD 0xDEAC5D34, 0xA028A6D9, 0x447CF4EB, 0x5D9D3A58, 0x66316A27
DCD 0x55C52348, 0xD738249B, 0xCC2CFD36, 0x8D40085D, 0xEF5DF377
DCD 0x57CCA241, 0x2212219D, 0x37FDDECD, 0x23A3AFAE, 0x547EDEFB
```



思考题

1. 了解**PE**文件结构的意义是什么？
2. 为什么**Windows**后来的版本还要添加**DOS**头？
3. **PE**文件与**ELF**文件结构上有哪些不同？
4. 为什么各个系统不采用相同的文件结构？



luhw@hust.edu.cn
