加壳原理

1.加壳器

(1)加载DLL模块

①pData = LoadLibrary(DLL)

②获取DLL模块中.Text段大小和内存地址

1)遍历区段头表（IMAGE_FIRST_SECTION(pNt)），每个区段头表前8个字节均为区段名，查找.TEXT段

2)获取.TEXT段的VirtualAddress，加上pData,得到DLL被加载进内存的地址

3)获取.TEXT段的SizeofRawData(区块大小（文件对齐）)

4)获取导出的变量结构体地址

(2)为文件添加一个区段

①CreateFile打开文件操作，GetFileSize获取文件大小，读取文件到内存ReadFile

②找到文件的区段头表的位置IMAGE_FIRST_SECTION(pNt)，根据文件头中区段数量*0x28(每个区段头表0x28个字节)，在最后添加一个区段头表

1)新区段名（8字节）

2)新区段VirtualSize，赋值为.TEXT段的SizeofRawData

3)新区段VirtualAddress（通过前区段的VirtualAddress + 前区段SizeofRawData（内存对齐））

4)新区段SizeofRawData，赋值为.TEXT段的SizeofRawData

5)新区段PointertoRawData（前区段PointertoRawData + 前区段SizeofRawData）

6)新区段属性0xE00000E0

③文件头中区段数目加1

④可选头中SizeofImage+=新区段SizeofRawData（内存对齐）

⑤申请一段新的内存空间，保存添加新区段的文件（此时并不拷贝.TEXT段）

(3)修复DLL重定位（添加到文件后）

①获取DLL的重定位表（pDLL_Nt->OptionalHeader.DataDirectory[5].VirtualAddress + (DWORD)DLL_File）

②遍历DLL重定位表

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress; //重定位内存页的起始RVA
    DWORD   SizeOfBlock;    //重定位块的大小
// WORD  TypeOffset[1]
} IMAGE_BASE_RELOCATION;
typedef IMAGE_BASE_RELOCATION UNALIGNED * PIMAGE_BASE_RELOCATION;
```

1)重定位表为结构体数组，最后一个以0元素结尾，以此为判断条件

2)创建一个结构体

```
struct _TYPEOFFECT
{
    WORD offset : 12;//偏移值
    WORD Type : 4;//重定位属性（方式）
}*pTypeOffect = NULL;
```

3)每个重定位结构体中，共有(SizeofBlock - 8) / 2 个需要重定位的偏移

4)判断属性是否为3，用重定位的偏移 + VirtualAddress + DLL_File 得到需要重定位的地址

5)修复重定位

a.将需要重定位的地址解引用，减去DLL_TEXT段的地址（va - va）获取距离TEXT段的偏移

b.将获取到的偏移 + 文件中新区段（壳代码段）的VirtualAddress（重定位地址的RVA）

c.RVA + 文件中的加载基址 = 重定位后的地址

(4)修复之后，将DLL的TEXT段拷贝到文件的新区段

(5)修复DLL内导出的全局变量

①DLL全局变量的地址 -DLL内.TEXT段的地址 + 文件内新区段地址RVA = 全局变量在文件中的地址RVA(此时，我们需要将OEP指向此RVA(壳程序入口点)，让壳程序优先运行)

(6)由于被加壳程序可能有随机基址，对于拷贝过来的DLL程序并不友好，所以需要将DLL的重定位表覆盖到被加壳文件中

①判断被加壳文件是否支持重定位

②遍历重定位段，将需要重定位的地址替换成被加壳文件的地址（只需要改重定位表结构体中的VirtualAddress）（VirtualAddress - DLL中TEXT段RVA + 被加壳文件新区段RVA）

③添加一个区段，将重定位表拷贝到新区段，并将加壳文件数据目录表中重定位表地址改成新添加的区段

(7)因为添加了一个区段，此时再次寻找全局变量地址

(8)对INT进行HASH加密填充

①此时，被加壳文件由于是在文件中展开，数据目录表中的导出表地址为RVA，所以要将RVA转换成FOA，再找到被加壳文件导入表位置，获取模块名。

②LoadLibrary此模块，利用GetProcAddress获取函数地址

IMAGE_THUNK_DATA32中，如果最高位为0，则代表名称导入，PIMAGE_THUNK_DATA便是一个RVA,转成FOA，附加文件基地址便是导入函数 名，通过函数名获取函数地址

③利用加载模块，获取函数导出表，遍历导出表，字符串比对，获取函数地址

④两次获取的函数地址进行比对，如果不相同，则不进行操作，如果相同，则将加密后的HASH填入PTHUNK中

注：两种获取函数地址的方式中，由于模块的导出函数中可能会导出其他模块的函数，遍历模块导出表时获取到的地址不正确，但GetProcAddress更智能，会自动帮你获取到其他模块的函数

(9)全区段加密（也正因为全区段加密，所以我们才要手动修复被加壳文件的重定位以及IAT）

①遍历每个区段，数据异或15加密

②区段表信息中，除了资源表和重定位表，其余全部清0

(10)区段压缩（算法压缩，区段位移）

2.壳

(1)合并区段为.Text

```
#include <Windows.h>
#pragma comment(linker,"/merge:.data=.text")
#pragma comment(linker,"/merge:.rdata=.text")
//#pragma comment(linker,"/merge:.tls=.text")
#pragma comment(linker, "/section:.text,RWE")

//#define ZLIB_WINAPI
//#include "zlib.h"
//#include <stdlib.h>
//#pragma comment(lib,"zlibstat.lib")
#include "aplib.h"
#pragma comment(lib,"aPlib.lib")
```

(2)解压缩数据

(3)解密代码

(4)修复文件重定位，重定位的地址 - 0x400000 + 新的基址

(5)修复IAT表，取出HASH值直接解密获取函数地址，没有获取到的，就使用GetProcAddresss获取

(6)跳回原始OEP

1.由于壳作为一个dll，loadlibrary默认加载为0x1000000，拷贝到新区段时，应考虑到exe文件的默认加载基址为0x400000,所以先将壳中的重定位地址内的值修改为被加壳文件中对应基址中的值

(拷贝到新区段之前)

先修复dll内重定位 通过重定位表地址

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress; //重定位内存页的起始RVA
    DWORD   SizeOfBlock;    //重定位块的大小
// WORD  TypeOffset[1];
} IMAGE_BASE_RELOCATION;
typedef IMAGE_BASE_RELOCATION UNALIGNED * PIMAGE_BASE_RELOCATION;
```

将需要重定位的地址解引用 - .Text段va + 新区段的rva + 0x400000 贴到新区段后的默认重定位

如果被加壳文件支持重定位，将dll中的重定位表拷贝至新区段，使程序加载时默认先修复壳内重定位。

壳中手动修复程序中的重定位