1. 项目说明

前言

壳作为当前软件保护行业的高端技术，从出现开始就一直战斗在安全对抗的前沿领域，经过多年的发展，出现了各种各样的加密壳，但无论如何，基本思想是不变的，都是通过对PE文件的变形处理，达到防止软件被分析和破解的目的

现在就要实现一个加壳软件框架，有了这个壳框架，就可以将自己熟知的反调试技巧加入项目，这样便有了属于自己的壳

前置知识

下面是要实现这个壳框架所需要知道的知识：

1. PE文件相关知识：PE头，导入表，导出表，重定位表

2. PEB中LDR模块链相关知识

项目目标

1. 掌握壳的基本实现方式

2. 掌握C++壳的实现方式

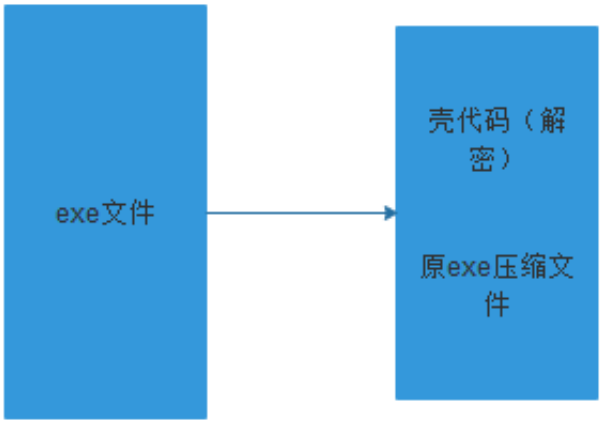
3. 为壳添加一些反调试手段增加壳的强度

2. 基础知识

1. 写壳要写什么？

壳的基本原理是对目标PE文件做一定的处理，添加一段壳代码，目标PE文件执行的时候，先执行壳的部分，之后再回到原始入口点开始执行

加壳前和加壳后如下所示：



通过上面的图示可以看出，把exe文件处理成加壳的样子的

代码和被加到目标程序中的壳代码是两个程序，所以壳要写什么？

1. 加壳程序，对PE文件做移动的处理，然后把解壳代码附到目标可执行文件中

2. 解壳代码，被附着到目标可执行文件上，用于目标文件还可以正常执行

2. 写壳的难点是什么？

1. 如何写壳代码

2. API函数的调用问题

3. 重定位问题

4. 信息交互问题

5. 调试问题

6. 关于目标程序的随机基址

7. 关于目标程序的导入表

8. 关于动态加解密

9. 关于TLS的处理

3.如何写壳代码

壳代码被拷贝到目标程序中时，拷贝的肯定是一段已经编译好的二进制代码，这段代码可以用汇编编写，这里项目用C++编写，所以壳代码也要用C++编写

我们直接把编写好的代码生成一个dll文件，然后将dll文件的代码段拷贝过去即可，考虑到我们的代码不可避免的会使用到一些全局变量，因此将代码段和数据段合并到一起是很有必要的，这样只需要拷贝一个区段就可以了

下面几行代码可以实现合并区段的功能，并将区段属性设置为可读可写可执行

```
#pragma comment (linker,"/merge:.data=.text")
#pragma comment (linker,"/merge:.tdata=.text")
#pragma comment (linker,"/section:.text,RWE")
```

4.API函数的调用问题

原始PE文件中99%的情况下会调用windows系统API，你的壳代码大部分情况也不可避免的也会使用到API函数，但是通常情况下，我们直接调用API的代码都会被编译成call导入表，当我们把壳代码拷贝到目标程序的时候，调用API的地方肯定会失败，解决这个问题有两个办法：

1.动态获取API函数

想要得到一个API函数的地址，最简单的办法就是使用两个API函数：

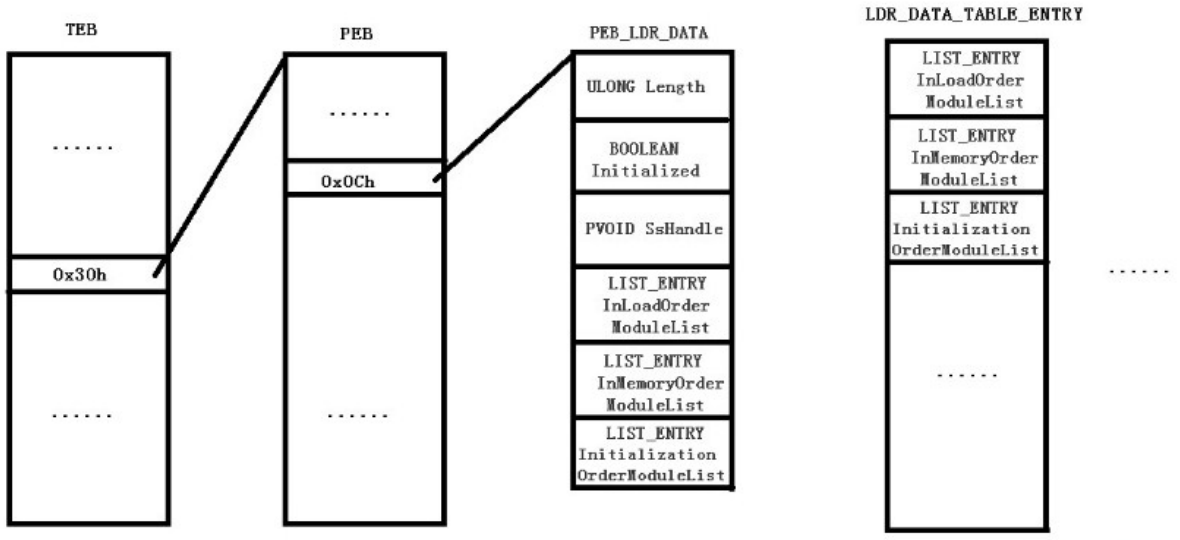
```
LoadLibrary
GetProcAddress
```

先使用GetProcAddress将模块加载进来，不管模块之前有没有被加载，调用完之后，你就能够得到模块句柄，随后使用GetProcAddress就可以获得这个模块所有的导出函数地址

但是关键在于LoadLibrary和GetProcAddress这两个函数也是API，我们也不能直接使用，首先要获得这两个函数的地址

幸运的是这两个函数存放在kernel.32这个模块中，而kernel.32这个模块又是每一个进程中必然会被加载的

关键就是需要先找到kernel.32这个模块的加载基址，找到基址后通过分析模块的导出表得到想要的函数地址，，方法就是通过模块链表得到：



下面的代码能够找到kernel32.dll的基址

```
int GetKernel32Base()
{
    int nAddress = 0;
    _asm{
        push eax;
        mov eax, fs:[0x30];
        mov eax, [eax + 0xC];
        mov eax, [eax + 0xC];
        mov eax, [eax];
        mov eax, [eax];
```

```
        mov eax, dword ptr ds : [eax + 0x18];
        mov nAddress, eax;
        pop eax;
    }
    return nAddress;
}
```

得到kernel32.dll的基地址后，我们就可以开始获取kernel32.dll模块中华单色GetProcAddress和LoadLibrary的地址

```
#include<windows.h>

typedef FARPROC(WINAPI*GETPROCADDRESS)(HMODULE hMoudle, LPCSTR lpProcName);
typedef HMOUDLE(WINAPI*LOADLIBRARYW) (_In_ LPCSTR lpLibFileName);
GETPROCADDRESS g_GetProcAddress = 0;
LOADLIBRARYW g_LoadLibraryW = 0;

void MyGetProcAddress(){
    char* Kernel32Buf =
```

```
(char*)GetKernel32Base();
    //1.获取kernel32的PE基本信息
    PIMAGE_DOS_HEADER m_pDos =
(PIMAGE_DOS_HEADER)Kernel32Buf;
    PIMAGE_NT_HEADERS m_pNt =
(PIMAGE_NT_HEADERS)(m_pDos->e_lfanew +
Kernel32Buf);
    PIMAGE_OPTIONAL_HEADER m_pOptionalHeader
= &m_pNt->OptionalHeader;
    //2.找到kernel32的导出函数的地址，需要去导出
表中寻找
    PIMAGE_DATA_DIRECTORY pExportDir =
m_pOptionalHeader->DataDirectory + 0;
    //3.导出表有三张表
    PIMAGE_EXPORT_DIRECTORY pExport =
(PIMAGE_EXPORT_DIRECTORY)(pExportDir-
>VirtualAddress + Kernel32Buf);
    PDWORD pEat = (PDWORD)(pExport-
>AddressOfFunctions + Kernel32Buf);
    PWORD pId = (PWORD)(pExport-
>AddressOfNameOrdinals + Kernel32Buf);
    PDWORD pNameRva = (PDWORD)(pExport-
>AddressOfNames + Kernel32Buf);
```

```
21      DWORD dwNameCount = pExport-
>NumberOfNames;
22
23      for (int i = 0; i < dwNameCount; i++){
24          char* pName = (pNameRva[i] +
Kernel32Buf);
25          if (strcmp(pName, "GetProcAddress")
== 0)
26          {
27              DWORD dwId = pId[i];
28              g_GetProcAddress =
(GETPROCADDRESS)(pEat[dwId] +
(DWORD)Kernel32Buf);
29              g_LoadLibraryW =
(LOADLIBRARYW)g_GetProcAddress((HMODULE)Kerne
l32Buf, "LoadLibraryW");
30
31              return ;
32          }
33      }
34  }
```

动态得到这两个函数地址后，就可以动态获得任何模块的

API，并调用它了，比如申请内存，比如创建一个窗口

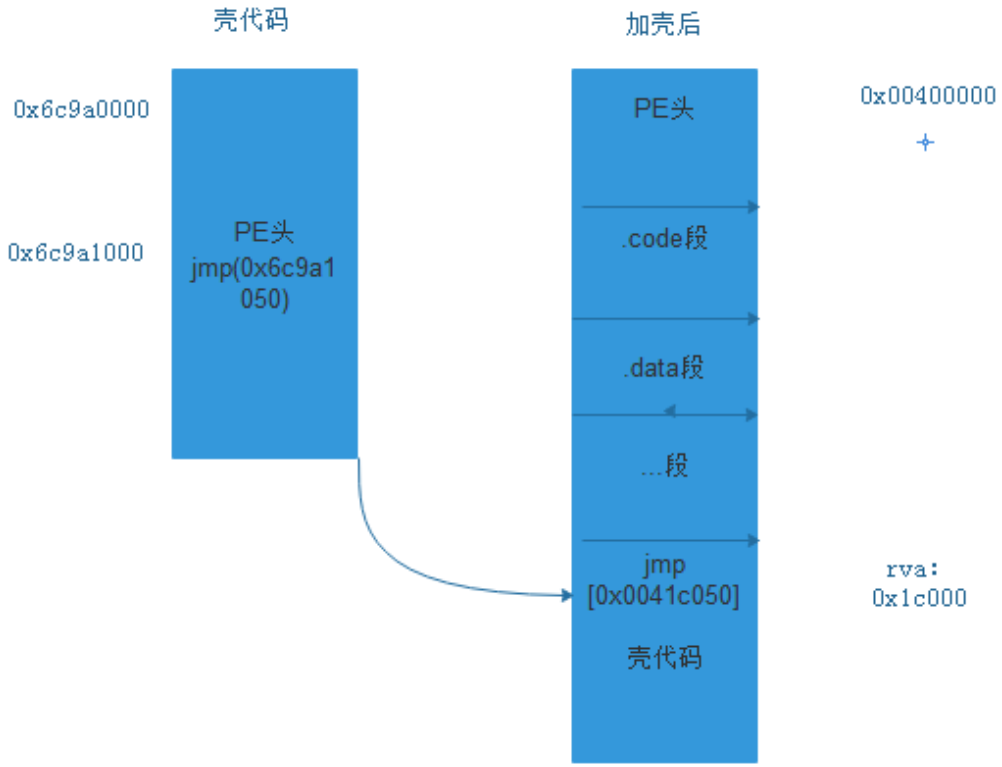
2.在目标程序中构建出壳代码中的导入表，并修改目标程序中的数据目录表

5.重定位问题

在壳代码中，我们不可避免的会使用到全局变量，访问全局变量，在汇编层面都是通过绝对地址（VA）得到数据的，我们把这些代码直接贴在目标程序上是会报错的

在之前学习PE文件的时候，这些地方都可以通过重定位表找到，找到之后修复成正确的即可

正确的位置如何计算？



访问全局变量重定位的公式：

正确位置=原始位置-区段的VA+目标程序ImageBase+壳代码rva

6.信息交互问题

写的壳代码跑起来的时候，可能会用到一些数据，但是这些数据在写壳代码的时候是不知道的，比如：目标程序的原始OEP，被加密区域的大小，被加密的起始位置，或者加了壳之后需要输入账号密码才能运行的账号密码

这些数据都是在加壳程序运行，在加壳过程中才能获得的数

据，需要把这些数据以比较得体的方式放到壳代码中

其中一个方法就是壳部分导出一个全局结构体变量，然后在加壳的时候，使用符号找到这个全局结构体变量，之后把需要用到的数据写入进去

7. 调试问题

写完壳以后，加上了壳，会出现一些比较难以解决的问题：

1. 加壳程序还可以用VS调试，但是壳代码，由于运行在可执行文件上，所以如果出错了，只能用OD调试

2. 假如程序根本就不能运行：

解决方法是：

检查自己的加壳程序代码，是否写错了

使用PE工具打开文件，检查所有修改过的字段

8. 关于目标程序的随机基址

当目标程序有随机地址的时候，事情就会变得很复杂：

1. 目标程序的代码段加密压缩变形，这个时候让系统自己修复就会出错

2. 自己的壳代码仅仅是把他需要重定位的地方修复成适应目标程序固定基址的位置，但是如果目标程序是随机基址了，那么壳代码的VA就会再出问题，毕竟目标程序的重定位表中是没有壳代码中需要重定位的位置的

为了处理以上问题，大致有两种方法：

1. 废掉目标程序的随机基址，这样基址就固定了，只需要去掉目标程序扩展头中的DllCharacteristics去掉

```
DllCharacteristics&=~
IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE;
```

2在目标程序中构造一个重定位表，使得系统能够修复壳代码的重定位，使得壳代码可以正常运行，在壳代码里面把目标程序解密完成后再去修复目标程序的重定位

9. 关于目标程序的导入表

原始IAT中存储的是函数名，通过函数名得到函数地址，将得到的函数地址填充到相应的位置

10. 关于动态加解密

动态加解密可以在程序运行起来以后，动态的对代码进行加解密

这有点类似于hook，在目标程序插入很多hook点，使得程序在运行过程中通过hook点回到壳代码中进行动态加解密

11. 关于TLS的处理

TLS也就是线程局部存储，有的加壳程序由于把很多区段都变了形，导致系统无法找到目标程序的TLS段，也就无法调用其中的TLS回调函数，那么一些依赖TLS回调函数运行的程序就无法运行了

解决的办法是在程序到入口点之前，循环把TLS表中的函数都调用一遍

0. tls表存在rdata段,pe文件中的tls表会被压缩,所以使用stub段中的tls表(stub段融合之后tls段存在.text段中)

1. 程序在线程创建之时会先读取tls段的数据到一块空间,

空间地址保存在FS:[0x2C],之后也都是使用这块空间

所以不要企图在壳代码中修改tls表,想让其使用你提供的内存空间,应该在加壳时就应该处理好这方面的问题

2. index用于在FS:[0x2C]下存的指针找到tls段使用的内存空间指针

3. 函数只需自己循环调用即可

方案:

0. 将pe目录表9指向stub的tls表

1. 不压缩tls数据段[tls数据段的寻找方式:通过tls表中的StartAddressOfRawData在区段中寻找]

2. 将index存入共享信息结构体,计算这个变量的rva(在FixRloc之后设置为rva-0x1000+allensection_rva+pe_imagebase)

3. stub的tls表前两项同pe的tls表,数值上需要转化(在FixRloc之后设置为和pe的tls表项相同即可)

4. stub的addressOfFunc同pe的tls表,数值上需要转化(在FixRloc之后设置为和pe的tls表项相同即可)

1.加壳原理:

改变原始OEP的指向 指向壳程序 壳程序可以添加其它程序

如弹密码框 只要密码正确就运行

```
DWORD GetKernel32Base()//获取kernel32模块加载基址
DWORD GetGPAFunAddr()//获取GetProcAddress的基址
bool InitializationAPI()//初始化API
LRESULT CALLBACK WindowProc(hwnd,uMsg,wParam,lParam)
void CtrateWin()
```

2. 加密代码段：简单异或

```
void pack::Encode()//对代码段进行加密
void Decode()//解密
```

3. 压缩思路：

除了tls表和资源表与Header头除外，其他的都要压缩，要先于拷贝壳代码和重定位表，不然壳也会被压缩，这样程序就运行不起来了

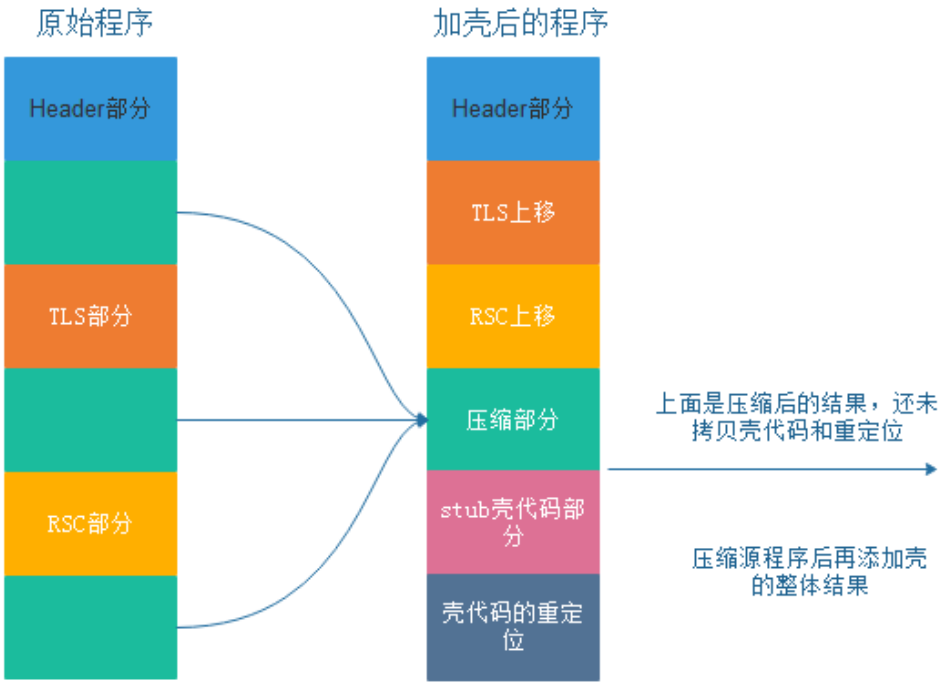
压缩时的问题有：资源段怎么办？ tls段怎么办？

首先运行时会用到资源段，同时当运行壳代码之前会启动线

程，因些就会用到tls数据段

所以这两个都不压缩，其它全部压缩，然后一起放到最后，同时在压缩的时候重建PE(也就是后面的往上移)

最重的一点是在所有操作前：保存相关节与节的前后顺序与在原来大小和压缩后的大小




```
1 void pack::CompressPE(PPACKINFO &pPackInfo)//压缩PE文件
2 void Decompress()  //解压
```

4. 增加随机基址,修复重定位思路:

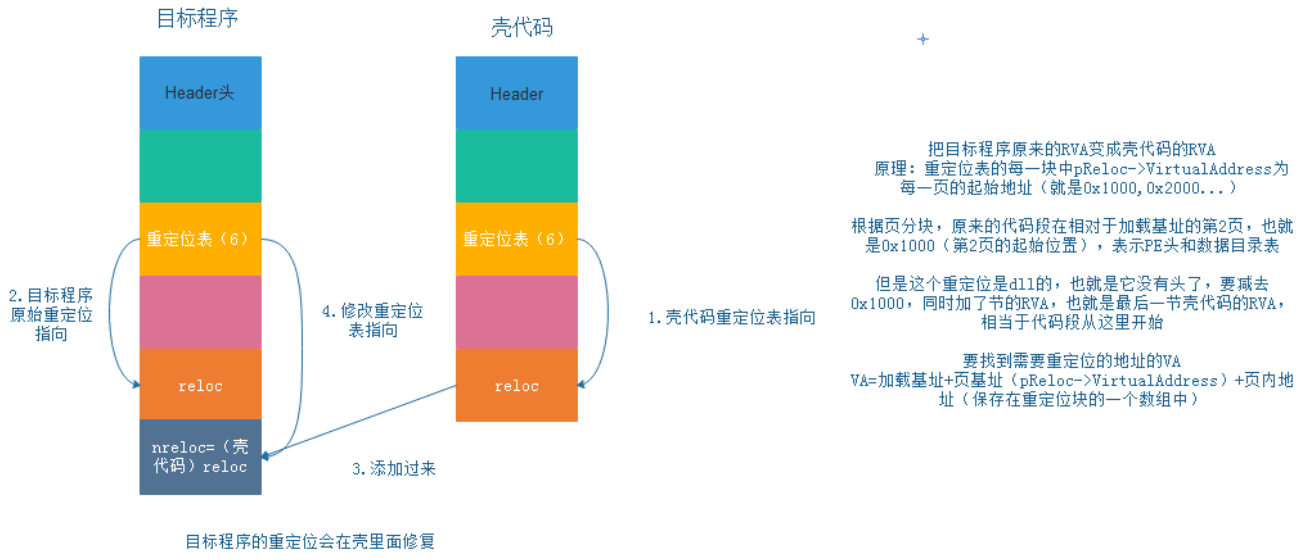
要想实现随机基址 必须解决两个问题:

1. 壳代码的重定位

2. 目标程序（被加壳程序）的重定位

解决第一个问题：由于壳程序区段都在text中，把text区段拷贝到目标程序时，同时也要拷贝重定位，把原来目标程序的重定位目录表的指向改为拷贝的重定位的地址

解决第二个问题：原来的重定位表被压缩了，用壳代码来修复



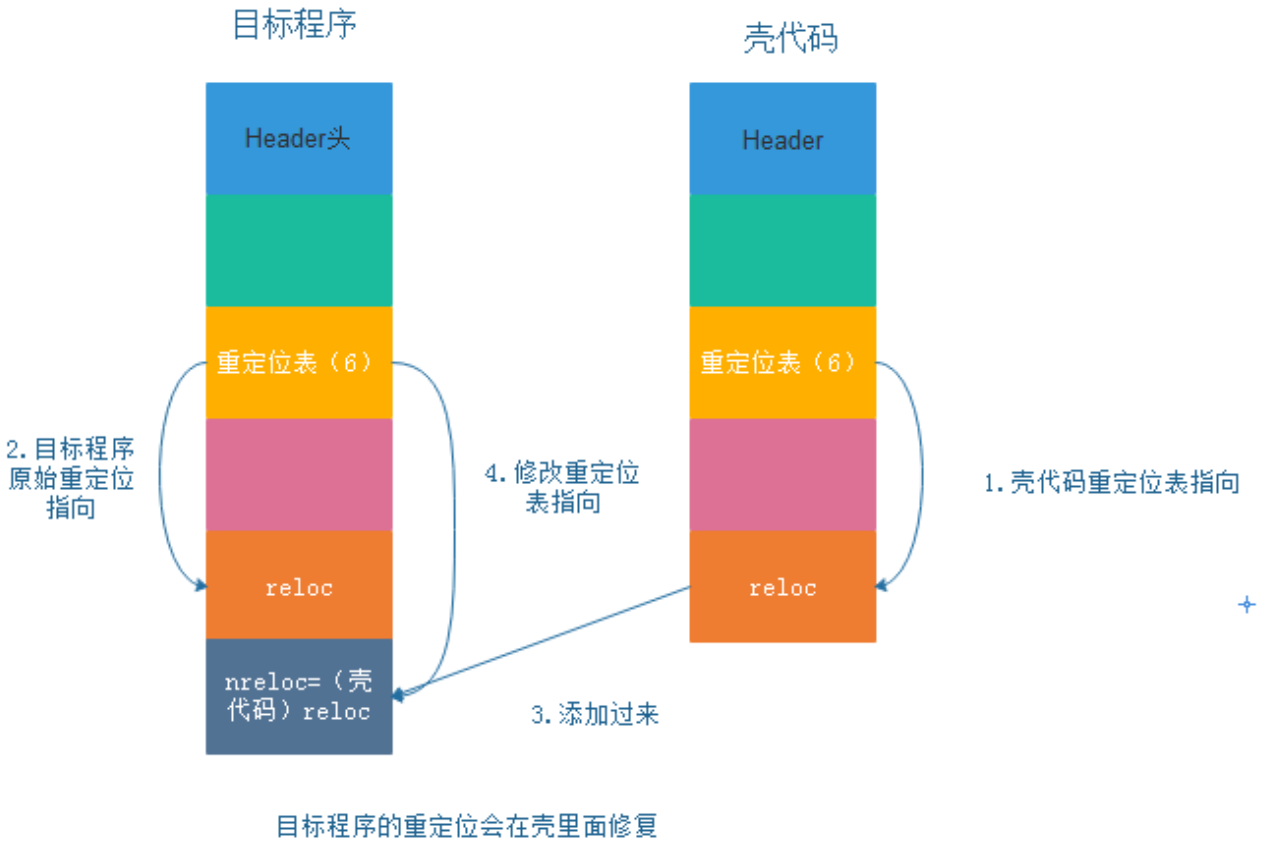

```
1 把目标程序原来的RVA变成壳代码的RVA
2  原理：重定位表的每一块中pReloc->VirtualAddress为每一页的起始地址（就是0x1000,0x2000...）
3
```

根据页分块，原来的代码段在相对于加载基址的第2页，也就是0x1000（第2页的起始位置），表示PE头和数据目录表

但是这个重定位是dll的，也就是它没有头了，要减去0x1000，同时加了节的RVA，也就是最后一节壳代码的RVA，相当于代码段从这里开始

要找到需要重定位的地址的VA

VA=加载基址+页基址（pReloc->VirtualAddress）+页内地址（保存在重定位块的一个数组中）



目标程序的重定位会在壳里面修复

```
void pack::ChangeReloc(PCHAR pBuf)
//对于动态加载基址,需要将stub的重定位区段(.reloc)修改后保存,将PE重定位信息指针指向该地址
void FixReloc()//修复重定位
```

5. tls处理思路：

1. PE目录表第10项为tls目录它指向tls表，tls表保存在rdata段，表的一些地址又指向另外的地址就是:tls数据段

2. 有了这个知识点，如果全部压缩，有几个问题：

第一 rdata肯定要压缩，但压缩完后，表也就没了

第二 有的程序本就没有线程，也就对应着没有索引数组

第三 tls被我处理了但是加载器不知道了，怎么处理

解决问题：

解决第一个 rdata段压缩了没了tls表：那就用stub的（壳对应的tls，由于它的段都被合成text了，但是要做一些地址转换问题）

解决第二个 有的程序没有，那我就在壳程序里创建一个线程局部变量，对应着 它就给我生成一个tls表

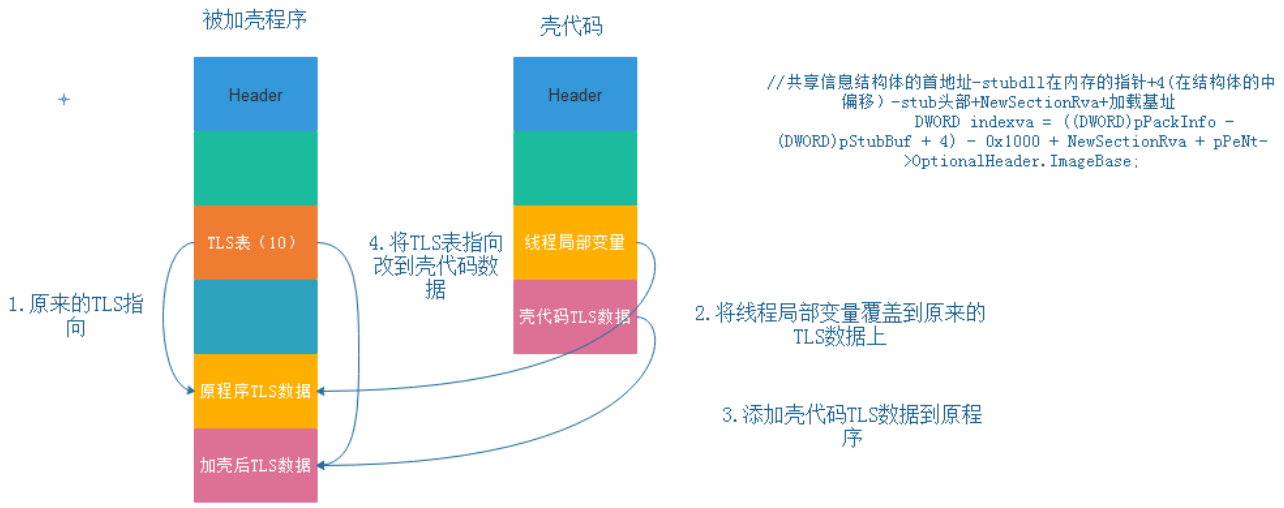
解决第三个 tls表不就是保存一些回调函数嘛，加载器也就是判断你有没有回调，有调用那处理了tls表，加载器不会调用（处理tls也就是把回调地址变为0），那就壳代码来调用

总结：

在加壳之前我先做相关处理：处理tls表

然后加壳之后设置tls表，就是设置目录表第10项tls表的指向

还有一个最重要的就是：不管你有没有，先创建一个线程（索引数组）也就是创建一个线程局部变量，这个是在壳代码里创建的：所以放在共享结构体中，tls表中保存的都是VA




```
1  void PEpack::ReadTargetFile(char* pPath,
   PPACKINFO& pPackInfo)//读取要加密文件到内存   同
   时获取相关的区段信息
2          1 打开文件
3          2 获取文件大小
4          3 申请空间
5          4 把文件内容读取到申请出的空间中
6             获取OEP
7             获取tls的信息
8          5 关闭文件
9
10 BOOL PEpack::DealwithTLS(PPACKINFO &
   pPackInfo) //处理tls
11
```

```
12 void PEpack::SetTls(DWORD NewSectionRva,
   PCHAR pStubBuf, PPACKINFO pPackInfo)//压缩后
   设置tls
13
14 void IATReloc()//IAT修复 压缩后加载器不会修复了
   （因为设置为0了）要在壳里修复，主要是有一个地址，
   就是自己申请一块内存空间,构造一段硬编码,将原函数
   地址填到这个硬编码的指定位置,然后将内存空间首地址
   写到IAT表
```

[区段表]

名称	VOffset	VSize	ROffset	RSize	标志
.rdata	0001C000	000027CC	0000A600	00002800	40000040
.data	0001F000	0000077C	0000CE00	00000400	C0000040
.idata	00020000	00000BE4	0000D200	00000C00	40000040
.rsrc	00021000	0000043C	0000DE00	00000600	40000040
.reloc	00022000	000006E3	0000E400	00000800	42000040

[区段表]

名称	VOffset	VSize	ROffset	RSize	标志
.rsrc	00021000	0000043C	00000400	00000600	C0000040
.reloc	00022000	000006E3	00000000	00000000	C2000040
.pack	00023000	000031FA	00000A00	00003200	C0000040
.stub	00027000	00003E00	00003C00	00003E00	E0000020
.nreloc	0002B000	00000400	00007A00	00000400	C2000040

TestBird	2018/12/5 11:36	应用程序	59 KB
TestBird.exe_pack	2018/12/5 11:36	应用程序	32 KB
TestBird	2018/12/5 11:36	Incremental Link...	304 KB
TestBird	2018/12/5 11:36	Program Debug...	748 KB

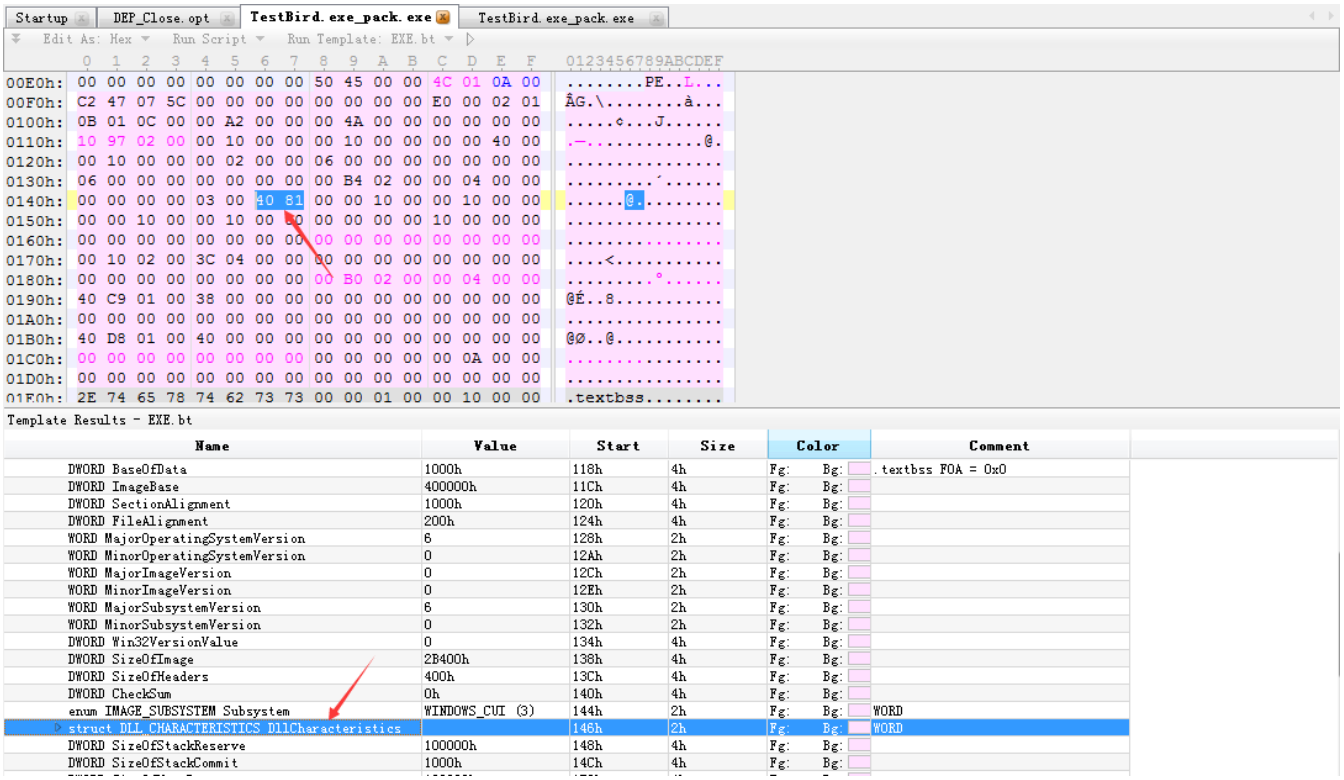
Startup | DEP_Close.opt | TestBird.exe_pack.exe | TestBird.exe_pack.exe

Edit As: Hex | Run Script | Run Template: EXE.bt

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
79E0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
79F0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
7A00h: D8 AC 57 00 B4 00 00 00 04 30 08 30 0C 30 18 30   Ø¬W.´....0.0.0.0
7A10h: 1C 30 F0 35 D0 37 D4 37 D8 37 E0 37 F8 37 24 38   .0ð5Ð7Ô7Ø7à7ø7$8
7A20h: 50 38 74 38 D8 38 DC 38 E4 38 E8 38 D4 3B D8 3B   P8t8Ø8Ü8ä8è8Ô;Ø;
7A30h: DC 3B E0 3B E4 3B E8 3B EC 3B F0 3B F4 3B F8 3B   Ü;à;ä;è;ì;ð;ô;ø;
7A40h: FC 3B 00 3C 04 3C 08 3C 0C 3C 10 3C 14 3C 18 3C   ü;.<.<.<.<.<.<.<.<
7A50h: 1C 3C 20 3C 24 3C 28 3C 2C 3C 30 3C 34 3C 38 3C   .< <$<(<,<0<4<8<
7A60h: 3C 3C 40 3C 84 3C 90 3C 94 3C 98 3C 9C 3C 1C 3D   <<@<„<.<"<˜<œ<.=
7A70h: 20 3D 30 3D 34 3D 3C 3D 54 3D 64 3D 68 3D 80 3D    =0=4=<=T=d=h=€=
7A80h: 90 3D 94 3D 98 3D 9C 3D B0 3D B4 3D B8 3D D0 3D   .=”=˜=œ=°=´=¸=Ð=
7A90h: D4 3D EC 3D FC 3D 00 3E 10 3E 14 3E 18 3E 20 3E   Ô=ì=ü=.>.>.>.> >
7AA0h: 38 3E 48 3E 58 3E 5C 3E 60 3E 64 3E 78 3E 7C 3E   8>H>X>\>`>d>x>|>
7AB0h: 80 3E 84 3E D8 BC 57 00 44 01 00 00 3A 32 21 33   €>„>Ø¼W.D...:2!3
7AC0h: 31 33 41 33 6C 33 75 33 11 34 31 34 3E 34 A1 34   13A313u3.414>4¡4
7AD0h: DA 34 01 35 21 35 2E 35 9A 35 AD 35 B9 35 4C 36   Ú4.5!5.5š5-5¹5L6
7AE0h: BA 36 DC 36 E5 36 FE 36 F3 36 FE 36 03 37 09 37   º6Ü6å6i666b6.7.7
```

Template Results - EXE.bt

Name	Value	Start	Size	Color	Comment
struct IMAGE_DOS_STUB DosStub		40h	A8h	Fg: Bg:	
struct IMAGE_NT_HEADERS NtHeader		E8h	F8h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[10]		1E0h	190h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[0]	.textbss	1E0h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[1]	.text	208h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[2]	.rdata	230h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[3]	.data	258h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[4]	.idata	280h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[5]	.rsrc	2A8h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[6]	.reloc	2D0h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[7]	.pack	2F8h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[8]	.stub	320h	28h	Fg: Bg:	
struct IMAGE_SECTION_HEADER SectionHeaders[9]	.nreloc	348h	28h	Fg: Bg:	
struct IMAGE_SECTION_DATA Section[0]	.rsrc	400h	600h	Fg: Bg:	
struct IMAGE_SECTION_DATA Section[1]	.pack	A00h	3200h	Fg: Bg:	
struct IMAGE_SECTION_DATA Section[2]	.stub	3C00h	3E00h	Fg: Bg:	
struct IMAGE_SECTION_DATA Section[3]	.nreloc	7A00h	400h	Fg: Bg:	
struct BASE_RELOCATION_TABLE RelocTable		7A00h	3F4h	Fg: Bg:	4



随机基址：x32拽进去地址不一样，010editor是4081

弹框：加壳后输密码弹框

花指令：加花和没有加花的同样位置汇编对比（OD）

重定位：010editor加上了重定位段，并且区段里面有数据，再就是看代码

加密：看代码说

压缩：程序变小了就压缩好了

反调试：加壳后会打印回调函数

修复IAT：去遍历每一个IAT项，在每一个IAT表里面遍历找

到函数地址和函数名

IAT加密：把IAT表清0就加密了

重定位：把壳代码重定位区段加到程序上，然后更改重定位（因为加上dll后头没有了），最后把指向改了

压缩：TLS和资源段上移，其他的压缩

TLS：数据目录表指向壳代码STL表，拿到信息结构体里面的VA，然后把向共享信息结构体中传入tls回调函数指针

加密：就是循环遍历代码段，把所有代码段加密

花指令：

```asm
// 花指令
_asm
{
    PUSH - 1
        PUSH 0
        PUSH 0
        MOV EAX, DWORD PTR FS : [0]
        PUSH EAX
        MOV DWORD PTR FS : [0], ESP
        SUB ESP, 0x68
        PUSH EBX
        PUSH ESI
        PUSH EDI
        POP EAX
        POP EAX
        POP EAX
        ADD ESP, 0x68
        POP EAX
        MOV DWORD PTR FS : [0], EAX
        POP EAX
        POP EAX
        POP EAX
        POP EAX
        MOV EBP, EAX
}
```

```
EIP EDX  00429710 <testbird.exe_pack.EntryPoi   6A FF                  push 0xFFFFFFFF                 EntryPoint
         00429712                               6A 00                  push 0x0
         00429714                               6A 00                  push 0x0
         00429716                               64 A1 00 00 00 00      mov eax,dword ptr fs:[0]
         0042971C                               50                     push eax
         0042971D                               64 89 25 00 00 00 00   mov dword ptr fs:[0],esp
         00429724                               83 EC 68               sub esp,0x68
         00429727                               53                     push ebx
         00429728                               56                     push esi
         00429729                               57                     push edi
         0042972A                               58                     pop eax
         0042972B                               58                     pop eax
         0042972C                               58                     pop eax
         0042972D                               83 C4 68               add esp,0x68
         00429730                               58                     pop eax
         00429731                               64 A3 00 00 00 00      mov dword ptr fs:[0],eax
         00429737                               58                     pop eax
         00429738                               58                     pop eax
         00429739                               58                     pop eax
         0042973A                               58                     pop eax
         0042973B                               8B E8                  mov ebp,eax
```

```
         004296F0 <testbird.exe_pack.sub_4296  $  55                  push ebp                        sub_4296F0
         004296F1                              .  8B EC               mov ebp,esp
         004296F3                              .  51                  push ecx
         004296F4                              .  A1 0C 76 42 00      mov eax,dword ptr ds:[0x42760C]
         004296F9                              .  89 45 FC            mov dword ptr ss:[ebp-0x4],eax
         004296FC                              .  8B 45 FC            mov eax,dword ptr ss:[ebp-0x4]
         004296FF                              .v FF E0               jmp eax
         00429701                              .  8B E5               mov esp,ebp
         00429703                              .  5D                  pop ebp
         00429704                              L. C3                  ret
```