题目：解析静态库(Lib)文件，提取出所有函数信息，组织成自定义格式文件(flb)

目的：确定函数SIZE的方案还不够严谨，望牛人指教

标准：《Microsoft 可移植可执行文件和通用目标文件格式文件规范》，

简称《PE COFF文件规范》

注意：这里的LIB指的是静态库，要和编写DLL所生成的lib区别开来

《PE COFF文件规范》 中称静态库格式为:

档案（库）文件格式，在WinNT.h中称Archive format.

《PE COFF文件规范》 中称另一种Lib为：

导入库格式-是描述由一个映像导出供其它映像使用的符号的库

具体思路：

1. 解析静态库(Lib)文件，提取出其中的所有目标文件成员(obj)
2. 解析各个目标文件成员(obj)，提取出其中的函数信息(函数名，函数数据，函数数据大小)
3. 把提取出的函数信息，组织成自定义文件格式(flb)

函数库(.flb)文件格式: 签名-函数头表-函数名称段-函数数据段

1. 解析档案（库）文件(.lib)

1. 档案（库）文件(.lib)结构

档案（库）文件(.lib)结构非常简单，就是签名+很多个成员，

详细参见《PE COFF文件规范》第7节。

**签名**，WinNT.h是这么定义的：

#define IMAGE\_ARCHIVE\_START\_SIZE 8

#define IMAGE\_ARCHIVE\_START "!<arch>\n"

**成员**，是由头部和成员内容组成；WinNT.h是这么定义头部的：

typedef struct \_IMAGE\_ARCHIVE\_MEMBER\_HEADER {

BYTE Name[16]; // File member name - `/' terminated.

BYTE Date[12]; // File member date - decimal.

BYTE UserID[6]; // File member user id - decimal.

BYTE GroupID[6]; // File member group id - decimal.

BYTE Mode[8]; // File member mode - octal.

BYTE Size[10]; // File member size – 头部大小不计算在内.

BYTE EndHeader[2]; // String to end header.

} IMAGE\_ARCHIVE\_MEMBER\_HEADER, \*PIMAGE\_ARCHIVE\_MEMBER\_HEADER;

#define IMAGE\_SIZEOF\_ARCHIVE\_MEMBER\_HDR 60

**三个特别成员**：第一链接器成员，第二链接器成员，长名称成员

(签名后面紧跟着的，就是这三个特别成员)

对于两个链接器成员，WinNT.h是这么定义它们头部的Name域的：

#define IMAGE\_ARCHIVE\_LINKER\_MEMBER "/ "

对于长名称成员，WinNT.h是这么定义它们头部的Name域的：

#define IMAGE\_ARCHIVE\_LONGNAMES\_MEMBER "// "

**目标文件成员(重点)：**三个特别成员之后直到文件结束，都是由这种目标文件成员组成。对于它们的头部Name域，有两种形式：

1.“名称/”

2．“/n” 当名称大小大于Name域的16个字节的话，就会存在长名称成员中，

而n(十进制)就给出其名称位于长名称成员中的偏移

目标文件成员的内容，就是标准的目标文件(COFF格式)。

2．定位目标文件成员

整个档案（库）文件的核心就是这些目标文件成员，定位它们，是CLibParser这个类的工作。具体工作分为两步：

1. 定位第一个目标文件成员。
2. 遍历所有目标文件成员。

定位第一个目标文件成员：

PBYTE CLibParser::GetFirstObjSection()

{

int iCtrl=0;

//第一个链接器成员

PBYTE pSect = m\_pLibImage+IMAGE\_ARCHIVE\_START\_SIZE;

if(!pSect)return NULL;

while(pSect)

{

//第二个链接器成员

if(memcmp(((PIMAGE\_ARCHIVE\_MEMBER\_HEADER)pSect)->Name,IMAGE\_ARCHIVE\_LINKER\_MEMBER,16)==0)

{

//Nothing

}

//第三个长名称成员

else if(memcmp(((PIMAGE\_ARCHIVE\_MEMBER\_HEADER)pSect)->Name,IMAGE\_ARCHIVE\_LONGNAMES\_MEMBER,16)==0)//LONG Name

{

//Nothing

//尽管长名称成员的头部必须存在，但它本身却可以为空。

}

else //第一个目标文件成员

{

return pSect;

}

//注意BYTE Size[10];要用atol((LPSTR)..)这种方法才能得到正确size

PIMAGE\_ARCHIVE\_MEMBER\_HEADER pAME=(PIMAGE\_ARCHIVE\_MEMBER\_HEADER)pSect;

pSect += atol((LPSTR)pAME->Size) + sizeof(IMAGE\_ARCHIVE\_MEMBER\_HEADER);

//两个成员之间有可能是由\n隔开

if(\*pSect=='\n') pSect++;

iCtrl++;//防止遇到错误的Lib文件，而死在这里面

if (iCtrl>3)

{

break;

}

}

return NULL;

}

遍历所有目标文件成员：

BOOL CLibParser::ParseObjs(PBYTE pObjSect)

{

do

{

PIMAGE\_ARCHIVE\_MEMBER\_HEADER pAME=(PIMAGE\_ARCHIVE\_MEMBER\_HEADER)pObjSect;

pObjSect+=sizeof(IMAGE\_ARCHIVE\_MEMBER\_HEADER);//去掉头部，剩下的就是Obj(COFF格式)

//判断是否是导入库格式，以防止错误的把导入库lib当做静态库lib，而程序直接挂掉

if(bImportlibraryFormat(pObjSect))

{

MessageBox(NULL,"This is not a Archive Format File,it's a Import Format File!",

"WARNING",MB\_ICONWARNING);

return FALSE;

}

//解析目标成员(OBJ)

CObjParser objParser;

objParser.Parse(pObjSect,m\_pNameFile,m\_pDataFile,&m\_FuncTable);

//注意：BYTE Size[10];要用atol((LPSTR)..)这种方法才能得到正确size

pObjSect += atol((LPSTR)pAME->Size) ;

//注意：两个成员之间有可能是由\n隔开,《PE COFF 文件格式》中并没有提到

if(\*pObjSect=='\n')

pObjSect++;

} while (pObjSect<m\_pLibImage+m\_fsize);

return TRUE;

}

二．解析目标文件(obj)

1.目标文件(.obj)结构：

**文件头(IMAGE\_FILE\_HEADER)**

typedef struct \_IMAGE\_FILE\_HEADER {

WORD Machine;

WORD NumberOfSections;

DWORD TimeDateStamp;

DWORD **PointerToSymbolTable**;//指向符号表

DWORD **NumberOfSymbols**;//符号表的大小

WORD SizeOfOptionalHeader;

WORD Characteristics;

} IMAGE\_FILE\_HEADER, \*PIMAGE\_FILE\_HEADER;

**节表(IMAGE\_SECTION\_HEADER)**

typedef struct \_IMAGE\_SECTION\_HEADER {

BYTE Name[IMAGE\_SIZEOF\_SHORT\_NAME];

union {

DWORD PhysicalAddress;

DWORD VirtualSize;

} Misc;

DWORD VirtualAddress;

DWORD **SizeOfRawData**;//指定节大小

DWORD **PointerToRawData**;//指向节数据

DWORD **PointerToRelocations**;//指向此节重定位信息表

DWORD PointerToLinenumbers;

WORD **NumberOfRelocations**;//此节重定位信息表的大小

WORD NumberOfLinenumbers;

DWORD Characteristics;

} IMAGE\_SECTION\_HEADER, \*PIMAGE\_SECTION\_HEADER;

。。。

**节数据**

。。。

**重定位表(IMAGE\_RELOCATION)**,几乎每个节都有一张重定位表

typedef struct \_IMAGE\_RELOCATION {

union {

DWORD VirtualAddress;//在相应节中的偏移

DWORD RelocCount; };

DWORD **SymbolTableIndex**;//此重定位信息的符号表索引

WORD Type;

} IMAGE\_RELOCATION;

typedef IMAGE\_RELOCATION UNALIGNED \*PIMAGE\_RELOCATION;

**符号表(IMAGE\_SYMBOL)目标文件的中心，所有工作都围绕这张表展开**

typedef struct \_IMAGE\_SYMBOL {

union {

BYTE ShortName[8];

struct {

DWORD Short; // 如果是长名称，这四个字节为0

DWORD Long; // 指定长名称位于字符串表中的偏移

} Name;

DWORD LongName[2]; // PBYTE [2]

} N;

DWORD Value;//如果此符号是函数符号，给出函数数据在相应节中的偏移

SHORT SectionNumber;//此符号相应节位于节表的索引，从1开始

WORD Type;// 如果是函数符号，此值为0x20

BYTE StorageClass;//如果是函数符号，此值为//IMAGE\_SYM\_CLASS\_EXTERNAL(2)

BYTE NumberOfAuxSymbols;//辅助符号表的个数，如果有辅助符号表, //紧跟着每个符号表后面，一般是0-1张辅助符号表

} IMAGE\_SYMBOL;

typedef IMAGE\_SYMBOL UNALIGNED \*PIMAGE\_SYMBOL;

**字符串表**，一些符号的名称太长(超过8个字节)，其名称就会存在这里

2.解析目标文件（obj）

目标文件的核心是符号表，整个解析工作就是围绕它展开。CObjParser负责解析目标文件(.obj)

具体步骤：

1. 定位符号表，过滤出其中的函数符号
2. 根据函数符号获得函数名称，输出到.nam中间文件，并记录下相应的偏移和大小
3. 根据函数符号获得函数数据，输出到.dat中间文件，并记录下相应的偏移和大小
4. 根据函数符号对重定位信息进行标志，在以后检测的时候，会跳过这些信息
5. 定位符号表，过滤出其中的函数符号

《PE COFF文件规范》中对函数符号式这么描述的：

存储类别为EXTERNAL（2）、Type 域的值为（0x20）以及SectionNumber 域的值大于0，

表明它是一个函数

具体方法：

//遍历符号表，过滤出其中的函数符号

for (DWORD i=0;i<m\_pFileHeader->NumberOfSymbols;i++)

{

pSymbol=m\_pSymbol+i;

// 存储类别为EXTERNAL（）、Type 域的值表明它是一个函数（x20）

// 以及SectionNumber 域的值大于，它就标志着函数的开头

if(ISFCN(pSymbol->Type)&&pSymbol->SectionNumber>0

&&pSymbol->StorageClass==IMAGE\_SYM\_CLASS\_EXTERNAL)

{

memset(&funcHeader,0,sizeof(funcHeader));

GetNameofSymb(pSymbol,funcHeader);

GetDataofSymb(pSymbol,funcHeader);

m\_pFuncTable->push\_back(funcHeader);

}

//直接跳过辅助符号表

i+=pSymbol->NumberOfAuxSymbols;

}

1. 根据函数符号获得函数名称，输出到.nam中间文件，并记录下相应的偏移和大小

void CObjParser::GetNameofSymb(PIMAGE\_SYMBOL pSymbol,FuncHeader& funcHeader)

{

PCHAR pName=NULL;

CHAR shortNam[9]={0};

//如果符号名称长度不超过个字节，那么符号表的ShortName 域

//就是包含符号名本身的一个字节长的数组；

if (pSymbol->N.Name.Short)

{

//pName= (PCHAR)pSymbol->N.ShortName;

//注意：符号名可能正好占满个字节，那就没有NULL结束符了，

//所以不能简单的用上面的方法

memcpy\_s(shortNam,9,pSymbol->N.ShortName,8);

pName=shortNam;

}

// 否则的话，它给出了字符串表中的一个偏移地址

else

{

pName= m\_pStrings+pSymbol->N.Name.Long;

}

//记录偏移

if (m\_pFuncTable->size()==0)

{

funcHeader.NameOff=0;

}

else

{

FuncHeader& funcHeadPrev=m\_pFuncTable->at(m\_pFuncTable->size()-1);

funcHeader.NameOff=funcHeadPrev.NameOff+funcHeadPrev.NameSize;

}

//记录大小

funcHeader.NameSize=strlen(pName)+1;

//写入nam文件

fwrite(pName,funcHeader.NameSize,1,m\_pNamFile);

fflush(m\_pNamFile);

}

3.根据函数符号获得函数数据，输出到.dat中间文件，并记录下相应的偏移和大小

这就是我上面提到的那个问题，函数数据的size确定问题，我现在的方法是：直接断定pSymbol所在节从pSymbol->Value偏移处开始到节结束都是pSymbol所对应的函数数据，但如果一个节中包含多个函数数据，这种方法就有问题了，现在的一些测试还没遇到问题。但这种方法显然不严谨，《PE COFF文件规范》里提到一种：

“5.5.1 辅助符号表记录格式之一：函数定义”；里面可以拿到size；

但悲剧的是每个函数符号后面一张辅助符号表都没有，所以这种方法流产了

我到现在也找不到好方法，望有牛人提点！

下面是具体方法：

//获得函数数据

void CObjParser::GetDataofSymb(PIMAGE\_SYMBOL pSymbol,FuncHeader& funcHeader)

{

PIMAGE\_SECTION\_HEADER pISH = m\_pSectionHeader+(pSymbol->SectionNumber-1);//SectionNumber从开始的索引

if(!pISH)

{

MessageBox(NULL,"Get SectionHeader Error!","Error",MB\_ICONWARNING);

return;

}

//记录偏移

if (m\_pFuncTable->size()==0)

{

funcHeader.DataOff=0;

}

else

{

FuncHeader& funcHeadPrev=m\_pFuncTable->at(m\_pFuncTable->size()-1);

funcHeader.DataOff=funcHeadPrev.DataOff+funcHeadPrev.DataSize;

}

//记录大小

//这种记算函数大小的方法并准确，这样做是默认这个pSymbol所在节从pSymbol->Value偏移处开始到

//节结束都是pSymbol所对应的函数数据，但有可能此节还包括其他函数数据

//拿节大小SizeOfRawData-函数在此节的偏移Value

funcHeader.DataSize=pISH->SizeOfRawData-pSymbol->Value;

//标志重定位位置

MarkRelocatePos(pISH);

//获取函数数据

PBYTE funData=m\_pObjImage+pISH->PointerToRawData+pSymbol->Value;

//写入dat文件

fwrite(funData,funcHeader.DataSize,1,m\_pDatFile);

fflush(m\_pDatFile);

}

1. 根据函数符号对重定位信息进行标志，在以后检测的时候，会跳过这些信息

关于重定位信息，我是这么理解的，比如在一个函数数据里有这样一条命令

Call 0x0040124a,而obj里此数据可能是这样的Call 0x00000000，要到链接的时候，根据重定位信息再进行修订

所以我直接用0x00000000标志这里是重定位信息，以后检测的时候，直接跳过这些重定位信息，不比较。

//标志重定位信息

void CObjParser::MarkRelocatePos(PIMAGE\_SECTION\_HEADER pISH)

{

//用四个字节0标志重定位信息位置

DWORD pReloMark=0;

DWORD modifyOff=0;

//获得重定位表

PIMAGE\_RELOCATION pIR = (PIMAGE\_RELOCATION)(m\_pObjImage + pISH->PointerToRelocations);

//重定位表大小

DWORD RefCount = pISH->NumberOfRelocations;

//遍历重定位表

for(DWORD i =0;i<RefCount;i++)

{

//待重定位偏移

modifyOff=pISH->PointerToRawData+pIR[i].VirtualAddress;

//修订

memcpy\_s(m\_pObjImage+modifyOff,4,&pReloMark,4);

}

}

三.自定义文件格式函数库文件(.flb)

自动义函数库文件(.flb)结构非常简单：

**签名**：

#define IMAGE\_FLIB\_START\_SIZE 8

#define IMAGE\_FLIB\_START "!<flib>\n"

**函数头表**：

typedef struct \_FlibFuncHeader//flib文件中的函数头结构

{

DWORD NameOff;//函数名称的文件偏移

DWORD DataOff;//函数数据的文件偏移

DWORD DataSize;//函数数据的大小

}FlibFuncHeader,\*PFlibFuncHeader;

**函数名称段**，里面存的是所有函数名，是一张以null结尾的字符串表

**函数数据段**，里面存的是所有函数的二进制数据

组织函数库文件

1. 定义一个全局的函数头表，在解析每个函数符号的时候，记录相应的偏移和大小
2. 将所有的函数名写入一个.nam中间文件
3. 将所有的函数数据写入一个.dat中间文件
4. 最后修订好偏移，按照顺序将 函数头表，.nam文件内容，.dat文件内容，写入最终的flb文件中去

至此，所有解析工作完成，剩下的就是测试flb文件是否正确!

Flb文件结构非常简单，所以解析它非常容易：

BOOL CLibScanner::ScanLib(LPCSTR szLib)

{

//加载flb文件

FILE\* pFile;

fopen\_s(&pFile,szLib,"rb");

if (!pFile)

{

CHAR ErrorStr[MAX\_PATH];

sprintf\_s(ErrorStr,MAX\_PATH,"Can't Open %s",szLib);

MessageBox(NULL,ErrorStr,"Error",MB\_ICONWARNING);

return FALSE;

}

int len=\_filelength(\_fileno(pFile));

m\_pImage=(PBYTE)malloc(len);

fread\_s(m\_pImage,len,len,1,pFile);

fclose(pFile);

//检测签名

if (memcmp(m\_pImage,IMAGE\_FLB\_START,IMAGE\_FLB\_START\_SIZE)!=0)

{

return FALSE;

}

//定位函数头表

m\_pFuncHeader=(PFuncHeader)(m\_pImage+IMAGE\_FLB\_START\_SIZE);

fclose(pFile);

return TRUE;

}

使用flb文件，只要遍历函数头表就可以了

PCSTR CLibScanner::CheckIfLibFunc(PCBYTE pFuncDat)

{

PFuncHeader pFuncHeader=m\_pFuncHeader;

PCSTR pNam=NULL;

//遍历函数库

do

{

if(CheckFunc(pFuncHeader,pFuncDat))

{

pNam=(PCSTR)(m\_pImage+pFuncHeader->NameOff);

return pNam;

}

pFuncHeader++;

} while (pFuncHeader->DataSize);//函数头表以一个完全为空的函数头成员结尾

return NULL;

}

BOOL CLibScanner::CheckFunc(PFuncHeader pFuncHeader,PCBYTE pFuncDat)

{

PBYTE pImpFuncDat=m\_pImage+pFuncHeader->DataOff;

//进行逐字节比对

for (DWORD i=0;i<pFuncHeader->DataSize;i++)

{

//跳过重定位信息

//只要是连续四个字节为，就当作重定位信息处理

//注意while:有可能两个重定位信息相邻，比如两个操作数的情况

while(\*(PDWORD)(pImpFuncDat+i)==0)

{

i+=4;

}

if (pImpFuncDat[i]!=pFuncDat[i])

{

return FALSE;

}

}

return TRUE;

}

要想测试flb文件，你还得拿到函数数据，我使用的是PVDasm，下面说一下我的方法：

1. 解析PE文件，获得代码段
2. 使用反汇编引擎，PVDasm，监控Call指令(只做0xe8)，获得函数数据
3. 加载相应的flb文件，检测每个获得函数数据，是否属于此flb文件中的函数

我测试了几个例子，跟ida做了比较，占时还没问题，但那个函数数据size的问题，就像锅里的一坨X，望牛人帮忙！

最后再提一下我的问题，希望有前辈指点一下！

这就是我上面提到的那个问题，函数数据的size确定问题，我现在的方法是：直接断定pSymbol所在节从pSymbol->Value偏移处开始到节结束都是pSymbol所对应的函数数据，但如果一个节中包含多个函数数据，这种方法就有问题了，

《PE COFF文件规范》里提到一种：

“5.5.1 辅助符号表记录格式之一：函数定义”；里面可以拿到size；

但悲剧的是每个函数符号后面一张辅助符号表都没有，所以这种方法流产了

我到现在也找不到好方法，望有牛人提点！