.Net 内存程序集的 DUMP(ProFile 篇)

作者：RZH　　网名：看雪_grassdrago

引言

在 DOTNET 加解密过程中，我们经常会碰到从内存中转贮.NET 程序集的场景。会经常使用那些神奇的 DUMP 工具，特别是分析整体加解密保护的程序集时，感觉很爽，当然这是因为它的保护很弱。于是我们想了解和学习如何完成类似的功能。本文将简单地介绍 Profiling API 的一些概念并通过它完成相同的工作。

Profiling API 简介

.net 为了帮助开发人员进行应用程序的内存、垃圾回收、线程、堆栈、程序集、类、方法、性能等低层分析，提供了我们使用 Profiling API 编写分析器或代码探查器的机制，它是 CLR 的一部分。要求探查器必须被编写为 COM 服务器并实现 IcorProfilerCallback2 接口[.net2.0 环境]或 IcorProfilerCallback[.net1.0 环境]接口，这个 COM 服务器将作为被监视进程的一部分运行并在事件发生时接收通知。

那么怎么启动它？通常我们会写一个 Loder 也可以手动完成，要做以下几点工作：

1. 注册你的 COM，可以通过命令行：regsvr32 XXX.dll 实现。
2. 设置环境变量 COR_PROFILER 为此 COM 的 GUID，告诉.net 由它来完成分析探查工作，可通过命令行：SET COR_PROFILER={xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}完成。 注意 CLR 仅能通过此变量装一个分析器。

3. 设置环境变量 COR_ENABLE_PROFILING 为 1,告诉.net 启动 Profiling 功能，可通过命令行：SET COR_ENABLE_PROFILING=1 完成。
4. 启动要监视的进程。

下面我们来看看 IcorProfilerCallback 接口中我们关注的几个事件，及要在其中完成的相应工作：

```
interface ICorProfilerCallback : IUnknown
{
    HRESULT Initialize( [in] IUnknown     *pICorProfilerInfoUnk);
    // 初始化代码探查器

    //其它略。。。

    HRESULT ModuleLoadFinished([in] ModuleID moduleId,[in] HRESULT  hrStatus);
    // 模块加载完成时，可执行模块的代码已完整地呈现在内存中，此时我们转贮代码

    //其它略。。。
}
```

另外：请留意.net 的版本，并实现相应接口，否则不会如你期望的那样运行。

更多的内容请参考下面几篇文章及 MSDN 帮助文档：

《使用 .NET Profiler API 检查并优化程序的内存使用》

《在 .NET Framework 2.0 中，没有任何代码能够逃避 Profiling API 的分析》

《用 .NET Framework Profiling API 迅速重写 MSIL 代码》

代码实现及说明

首先，我们需要完成一个基本的COM服务器并实现IcorProfilerCallback2 接口，好在这步头疼的工作可以通过修改CLR Profiler for the .NET Framework 2.0源文件来完成。代码实现的主要工作如下：

1. 该源文件的 Profiler.cpp 完成了一个进程内 COM 服务器的所有内容，我们只需要改变一下 GUID 以免和原 com 冲突就可以了。
2. 去掉 ProfilerCallback.h 和 ProfilerCallback.cpp 文件中不必要的部分以提高运行速度。如果你不觉得它太慢了的话，可以什么都不改。而我让它变成了一个实了现 IcorProfilerCallback2 接口的空壳。
3. 在 Initialize 方法中设置我们关心的事件掩码，针对源文件现实，则是在由 Initialize 方法调用的 GetEventMask()方法中。我们只关心 ASSEMBLY_LOADS 和 MODULE_LOADS 系列事件。所以：

```
m_dwEventMask = (DWORD) COR_PRF_MONITOR_MODULE_LOADS     | (DWORD) COR_PRF_MONITOR_ASSEMBLY_LOADS;
```

4. 在 ModuleLoadFinished 方法中完成我们的转贮。
5. 在原 C#编写的 loder 中（Launcher.exe），增加注册和反注册我们的 COM 的功能。

关键代码说明：

转贮的关键是得到模块加载基址，名称则关系不太大，这两者都可以在 ModuleLoadFinished 方法中通过 IcorProfilerInfo 及 IMetaDataImport 接口完成。

ICorProfilerInfo::GetModuleInfo	获取有关指定模块的信息。
ICorProfilerInfo::GetModuleMetaData	获取映射到指定模块的元数据接口实例。
IMetaDataImport::GetScopeProps	获取当前元数据范围内的程序集或模块的名称和版本标识符。

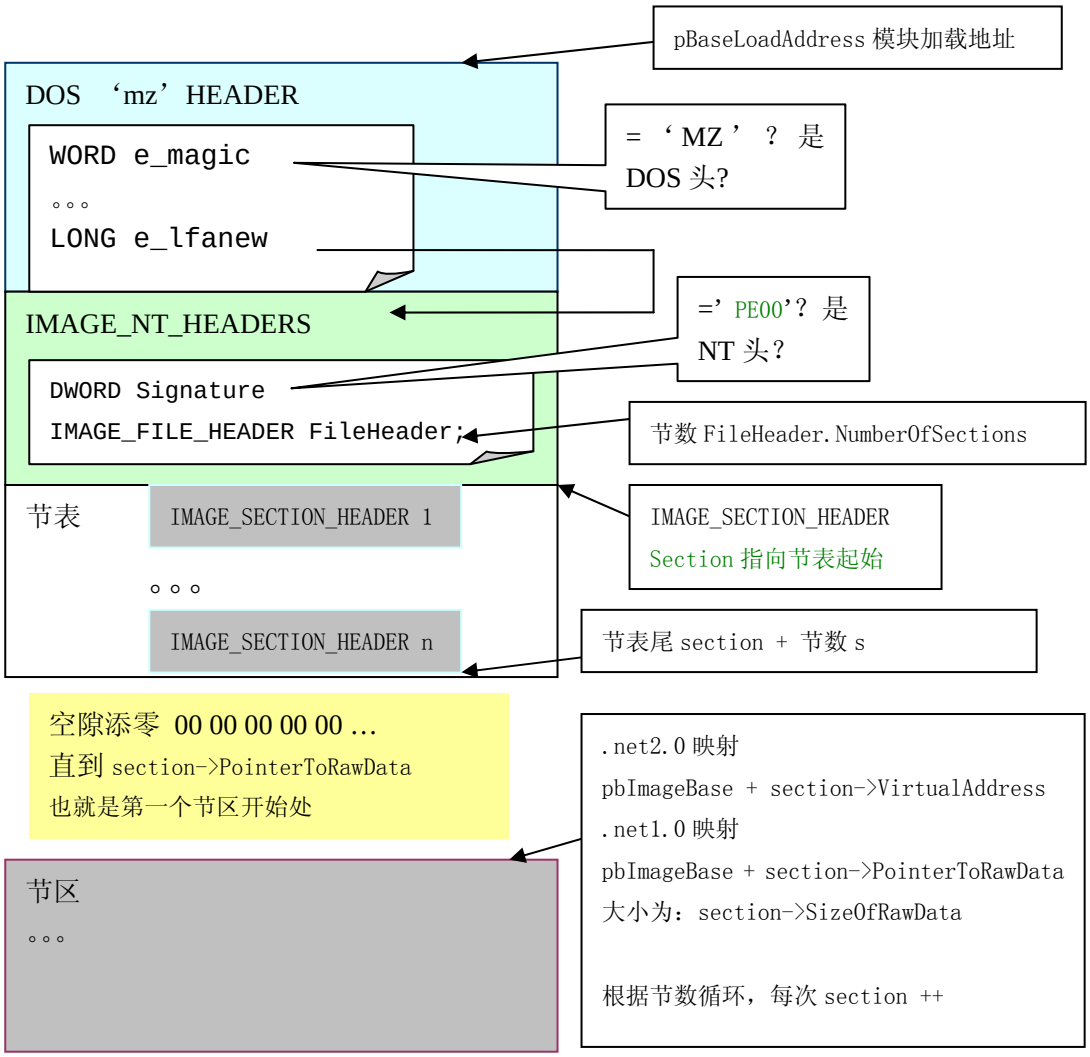
更详细的说明请参见 MSDN 帮助文件或 Profiler 的 Doc 文档。

具体代码如下：

```
HRESULT CProfilerCallback::ModuleLoadFinished(ModuleID moduleId, HRESULT hrStatus)
{
    HRESULT hr=m_pICorProfilerInfo->GetModuleInfo (
                            moduleId, (LPCBYTE *)&pBaseLoadAddress,
```

```
                        2048, &size, name,
                        &assemblyId             );
    __try {
        // let's determine the module name from metadata
        hr = m_pICorProfilerInfo->GetModuleMetaData(moduleId, 0, IID_IMetaDataImport,
(IUnknown**) &pImport);

        if (SUCCEEDED(hr)) {
            GUID     mvid;
            ULONG          nameLen = 0;
            hr = pImport->GetScopeProps(moduleName, 2048, &nameLen, &mvid);
        }
```

在得到了模块基址及名称的情况下，要做的就是根据 PE 结构写文件了，代码流程如下：



1. 模块基址指向 DOS 头，基址+ e_lfanew 指向 NT 头。

   FileHeader.NumberOfSections 是节数也是节表项的数。NT 头结构+1 指向节表。节数量知道了，也就知道了节表的尾部地址。好从模块起始地址一直到此，先写入文件。

2. 节表尾到第一个节区开始处填充零。
3. 内存中第一个节区的位置起，每次一字节，向文件中写入节数据，每个节数据大小为 section->SizeOfRawData

下面为具体代码：

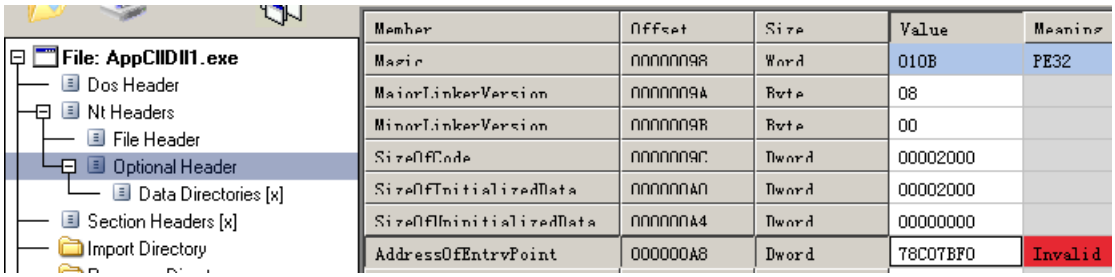
```
PIMAGE_NT_HEADERS pNTHeader = NULL;

    // only dump executable images
    __try {
        PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)pbImageBase;
        if (pDosHeader->e_magic != IMAGE_DOS_SIGNATURE) {
            return false;
        }

        pNTHeader = (PIMAGE_NT_HEADERS)((PBYTE)pDosHeader + pDosHeader->e_lfanew);
        if (pNTHeader->Signature != IMAGE_NT_SIGNATURE) {
            return false;
        }
    } __except(EXCEPTION_EXECUTE_HANDLER) {

       return false;

         }


     。。。

    int numSections = pNTHeader->FileHeader.NumberOfSections;
     PIMAGE_SECTION_HEADER section = (PIMAGE_SECTION_HEADER)(pNTHeader + 1);

     PBYTE pLastSectionEnd = (PBYTE)(section + numSections);


     。。。

     int headerLen = pLastSectionEnd-pbImageBase;
```

```
int numwritten = fwrite( pbImageBase, 1, headerLen, stream );

。。。

char zero = 0;
for (int i=headerLen; i<section->PointerToRawData; i++) {
    fwrite( &zero, 1, 1, stream );

    }

。。。

 if (isMapped)
    {
        buf = pbImageBase + section->VirtualAddress;
    } else {
        buf = pbImageBase + section->PointerToRawData; //第一个节区的指针
    }

numwritten = fwrite(buf, 1, section->SizeOfRawData, stream);
```

请参见随文档提供的代码及项目文件。

运行情况

这里仍旧使用上篇文章中《**{samartassembly}4.1.39 分析(加解密)**》提供的样例代码（包括原始无压缩/{sa}程序集打包/{sa}整体压缩）进行测试。被精简了的COM运行速度令人满意，每个可执行模块正确转贮成功。但转贮完成的exe并不能直接运行，经对比发现NT头中的AddressOfEntryPoint所指向的RVA错误，这对DLL并没有影响。



修正（二种方法）：

1. 对比原执行程序，修改 RVA 值。
2. 用 ILASM/ILDASM 对 dump 出的 exe 进行重新编译。

结语

利用 Profiling API 可以完成很多工作，微软的样列、文档及 MSDN 提供了较丰富的内容。而在.net 解密方面这已是一种陈旧的技术，但并不妨碍我们学习和在必要时使用它。如果有时间我们会继续 Hook mscoree.dll 及 Hook mscorjit.dll 的旅程。文章中所引用的知识、代码、甚至文档风格全部学习和来源于互联网，在此向所有具有知识共享精神的网友们表示谢意！