2021/03/01_PE_第10课_重定位表

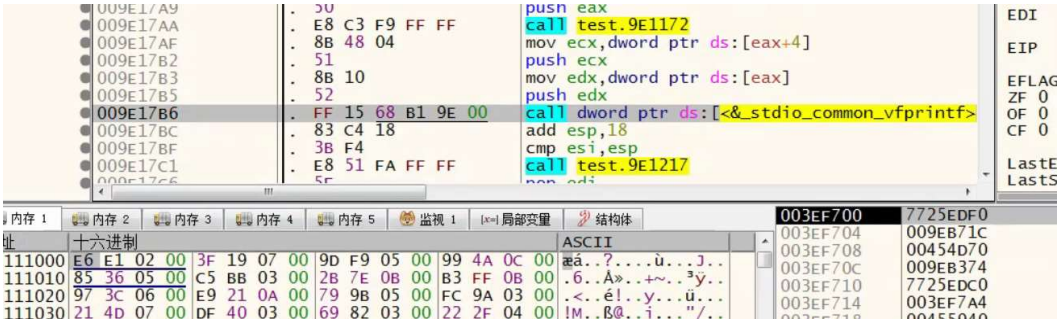
笔记本： PE
创建时间： 2021/3/1 星期一 10:03
作者： ileemi

内存地址随机问题

可执行文件的代码加载到内存中，代码存在重定位问题。

在常量区（.rdata）定位可执行程序要访问的字符串，并下一个内存访问断点，来定位库函数。在CPU窗口可以右键 --> 搜索 --> 当前模块 --> 字符串 来确定目标字符串在内存中的偏移。

通过x64dbg调试随机基址的可执行程序时，API调用的汇编代码下有一条横线的，表示操作系统会重定位代码（在执行入口代码之前进行重定位操作，根据重定位表（编译器生成）进行判断是否进行修正）



重定位表

> 需要记录重定位代码的RVA、修正方式（地址+修正方式）12bit放偏移，4bit放修正方式。

编译器解决代码重定位上的解决方案：

- 代码本身无重定位问题（模块+偏移 --> IAT），比较麻烦
- 生成重定位表

注意：

- 可执行文件，重定位表不是必须的（不是随机基址就不需要重定位表）
- 动态库，必须带重定位表（动态库模块基址经常满足不了）

重定位表的结构

> 重定位表在数据目录的第5项：**IMAGE_BASE_RELOCATION**

```
typedef struct _IMAGE_BASE_RELOCATION {
  DWORD VirtualAddress;
  DWORD SizeOfBlock;
//WORD TypeOffset[1];
} IMAGE_BASE_RELOCATION;
```

微软重定位表的设计：12bit放偏移，4bit放修正方式。

1000（RVA）
4bit 12bit
2000
4bit 12bit
3000
4bit 12bit

> 重定位表会有一个单独的节（.reloc）来进行存放。在对应的节中确定表在文件中的位置可通过文件偏移进行定位。

重定位表的每一项第一个参数为要修正页的偏移，第二个参数表示每一项的字节大小（整个结构体（重定位表）的大小）。

重定位内存的修正

修正方式：

- **IMAGE_REL_BASED_ABSOLUTE** 0 -- 无效
- **IMAGE_REL_BASED_HIGH** 1 -- 修复高16位
- **IMAGE_REL_BASED_LOW** 2 -- 修复低16位
- **IMAGE_REL_BASED_HIGHLOW** 3 -- 高、低位同时修复（32位）

定位修正的内存地址：

- 记录模块基地址 （0x1080000）
- 定位第一个重定位表项的RVA（0x00011000），重定位表每一项后两个字节（0x372F）取出最高位（0x3）当作修正方式，定位偏移（0x72F），确定要修正的内存地址：0x1080000 + 0x00011000 = 0x1091000 + 0x72F = **0x109172F**

使用可执行文件的 ImageBase：0x00400000 进行修正。

修正公式：

- 要修正的内存地址 = 原本汇编代码地址 - 原本模块基址 + 现在模块基址
- 要修正的内存地址 = 现在模块基址 - 原本模块基址 + 原本汇编代码地址

示例：
原本模块基址：00400000
b9 40 00 40 00 mov ecx, 00400040

现在模块基址：00500000
b9 40 00 40 00 mov ecx, 00500040

> 修正方法1：

要修正的内存地址 = 00500040 - 00400040 + 00500000 = 0x00500040
b9 40 00 50 00 mov ecx, 00500040

> 修正方法2：

offset：00500000 - 00400000 = 0x100000 -- 只需要计算一次
要修正的内存地址 = 0x100000 + 00400040 = 0x00500040
b9 40 00 50 00 mov ecx, 00500040

加壳作者会通过一些手段让脱壳的人找不到目标程序的OEP，一般做法：

- 将可执行程序的OEP代码抽走，在壳中模拟OEP代码，保证程序可以正常运行。这种做法通过仔细分析还是可以找到OEP代码的。
- 模拟API执行（所使用API的库不由操作系统去加载，自己加载（自己申请内存，拷贝库函数代码到内存中），IAT表中填写自己加载的库函数的地址。但是自己加载会有代码重定位问题，可通过重定位表对其进行修复），使脱壳者不能通过IAT还原导入表，模块列表也遍历不出来。