实验四 Shellcode 与 Windows 提权 Shellcode

# 实验目的

1. 学习 shellcode 原理、编写技巧；
2. 熟悉常用的 shellcode 编写工具；
3. 熟悉加载、调试 shellcode 技巧。

# 实验要求

1. 学习 shellcode 原理；
2. 学习 shellcode 编写工具；
3. 掌握 shellcode 加载、调试技巧；
4. 能掌握验证 shellcode 是否有效的技巧。

# 实验原理

## Shellcode 原理

Shellcode 是一段完成特定功能的、构思精巧、实现精简的机器码。一般Shellcode 最主要目的是完成预设的功能，其次为了保证健壮性和通用性。比较在意最终的 shellcode 长度，字节码中是否包含“坏字符”（例如截断字符\x00）。

在可执行程序中，Shellcode 是一段机器码，但是我们仍然可以以“高级”语言编写 Shellcode 功能；常见的使用 C/ASM 编写 Shellcode。

一方面，现在的 C 代码编译器优化的很多，也许在 C 里实现的功能很难在汇编层面比较清楚的显示；另一方面 ASM 相比有更多的灵活性。本节实验将带大家使用 ASM 编写 Shellcode。

## nasm （shellcode 编译工具）

Nasm 是一款用于编译 asm 代码的开源工具，可支持 Linux\Windows 等平台上的多种文件类型。基本可以满足日常 Shellcode 开发的需求。

## x64dbg（用户态调试工具）

X64dbg 是一款开源的 Windows 平台调试工具；支持 32bit 和 64bit。和 OD 和Immunity Debugger 使用相似，但是后二者不再更新，也只能支持 32bits。而且x64dbg 提供使用 python 作为扩展插件编写工具，有较好的社区支持，自定义扩展较好。因此本试验将使用 x64dbg 作为调试工具。

## windbg（Windows Sdk 官方调试工具）

Windbg 是微软官方调试工具，可以调试 Windows 内核，同时可以用于辅助分析 Windows 内部结构。

## IDA（反编译工具）

IDA 是一款逆向必用工具，强大的反编译能力。

# 实验相关资源

实验环境：

* Win10；
* X64dbg （<https://github.com/x64dbg/x64dbg>）
* Nasm （https:/[/www.nasm.us/pub/nasm/snapshots/l](http://www.nasm.us/pub/nasm/snapshots/latest/)a[test/](http://www.nasm.us/pub/nasm/snapshots/latest/)） 相关资源：
* nasm 使用文档（<https://www.csie.ntu.edu.tw/~comp03/nasm/nasmdoc1.html>）

1. x64dbg 使用文档（<https://help.x64dbg.com/en/latest/>）
2. 全球最大 shellcode 仓库（ <https://www.exploit-db.com/>）
3. PE 权威指南（<https://pan.baidu.com/s/1eQndEOq>）

# 实验步骤

## 简单编写弹窗 shellcode

1. 弹窗实现原理：MessageBox（NULL, Title，Message, Type）

在 Shellcode 中实现调用 MessageBox；而 MessageBox 函数在 user32.dll 中，需要首先得到 user32.dll 的基地址。利用 user32.dll 基地址+MessageBox 在其中的偏移得到函数地址。再根据函数调用约定汇编实现即可。

Use32.dll 可以根据 LoadLibraryA(‘user32’) 获取； 其中 LoadLibraryA 在Kernel32.dll 中。所以我们需要获取 Kernel32.dll 、LoadLibraryA 、User32.dll 、MessageBoxA 地址。

1. 获取 Kernel32.dll 基地址

所有的进程都维护一个 PEB（进程环境块结构）

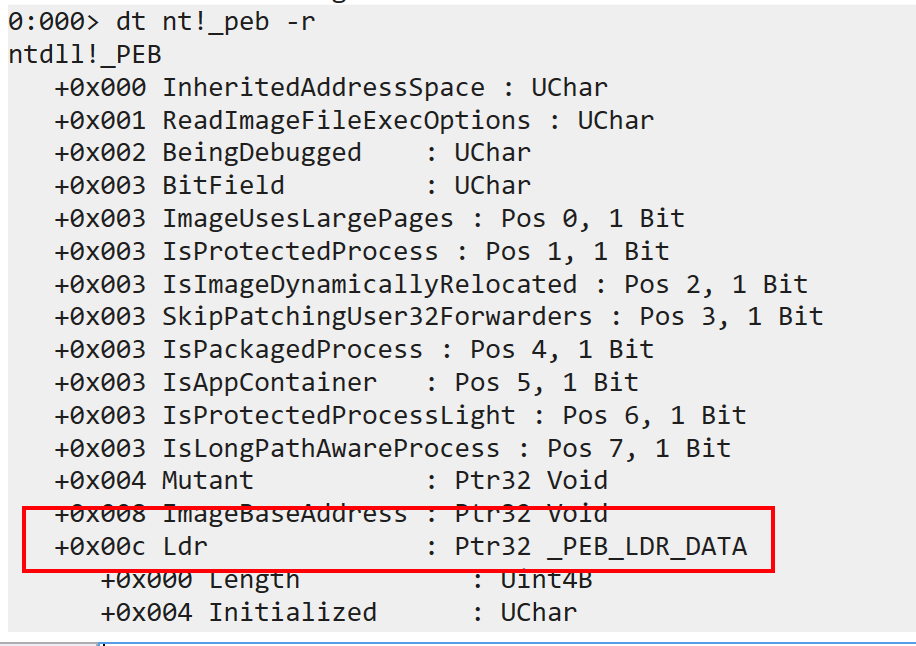


图 4-1 进程环境块 PEB

在 PEB 结构中，维护着\_PEB\_LDR\_DATA 结构保存着进程加载的所有的

Module 信息（包括 Kernel32.dll，基地址）

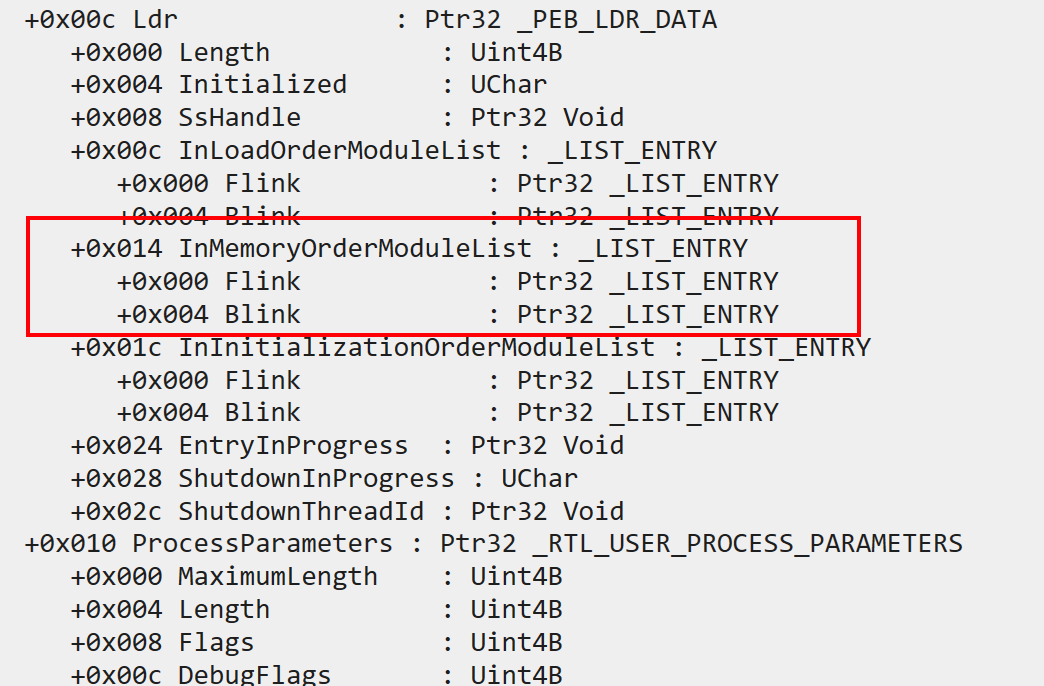


图 4-2 \_PEB\_LDR\_DATA 结构

\_PEB\_LDR\_DATA 其实维护着几个双向链表； 这其中我们主要关注的是InMemoryOrderModuleList，该结构维护了进程加载 Module 的基地址信息。下面我们使用 Windbg 演示一下获取进程 a.exe 下的 Module 基地址：

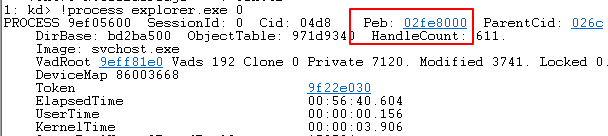


图 4-3 获取 explorer.exe 进程 PEB 地址

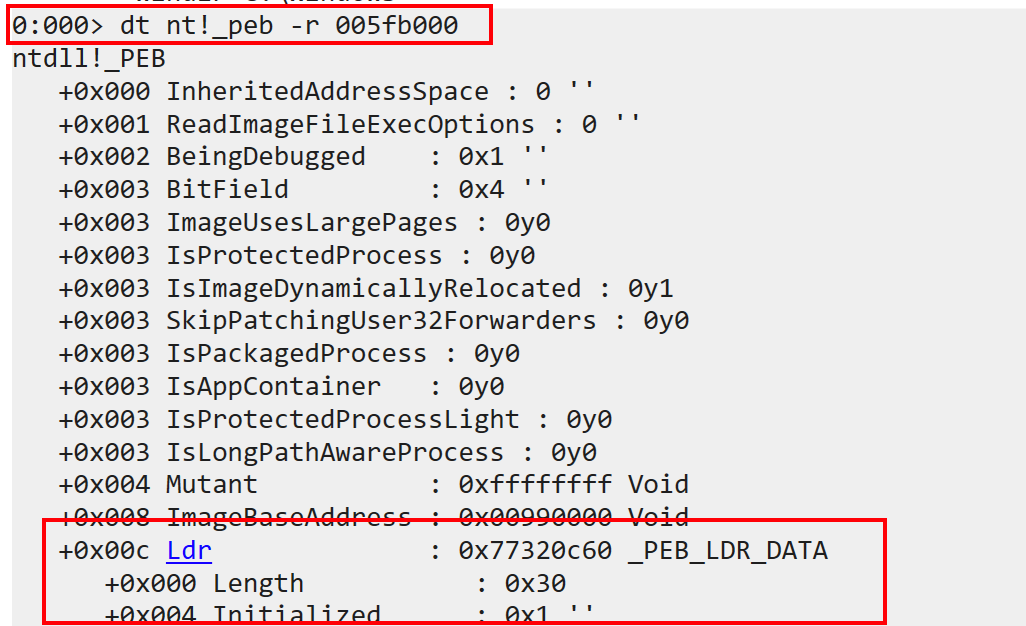


图 4-4 根据 PEB 信息获得 LDR 结构地址

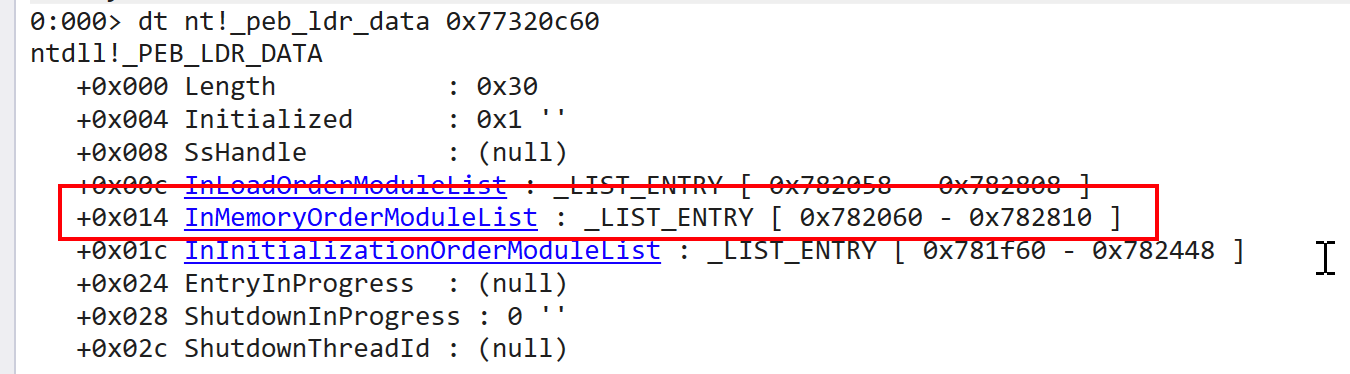


图 4-5 根据 LDR 结构得到其中的双向链表的信息另一种直观的方式是，直接解析 peb：

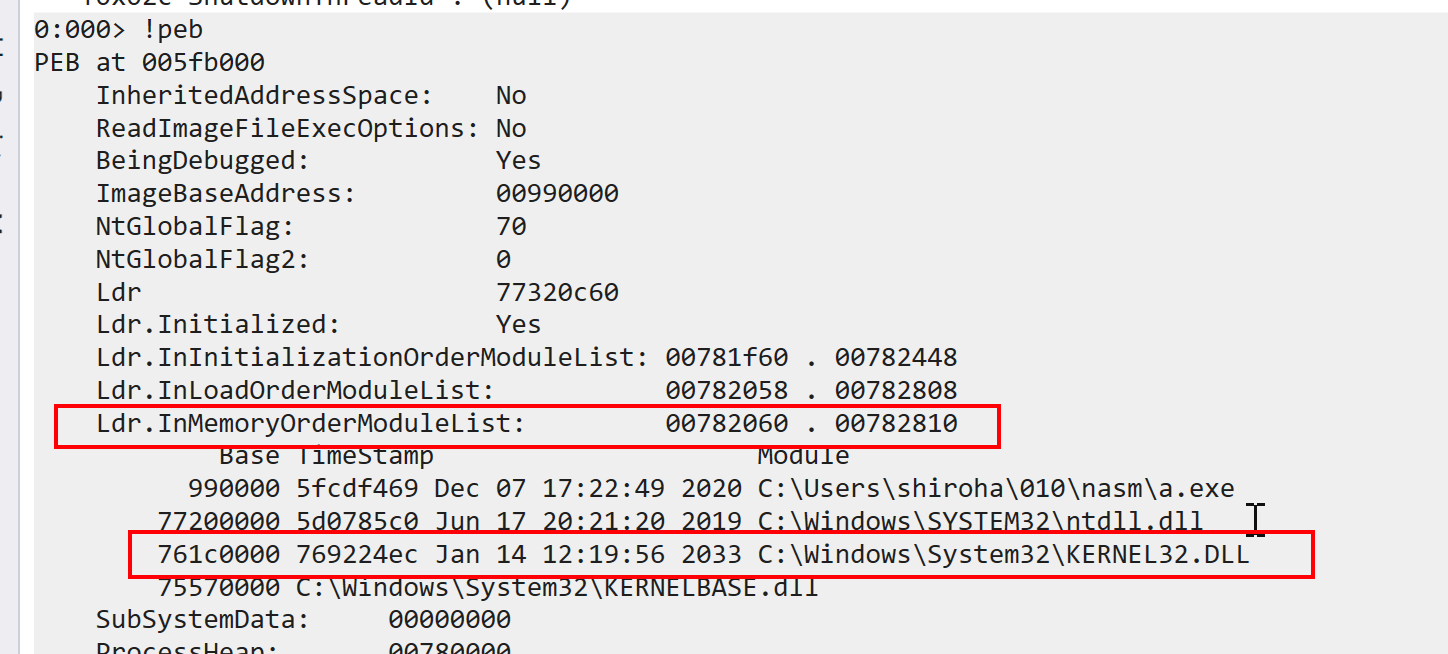
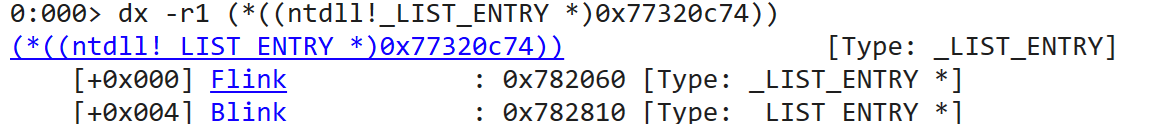


图 4-6 直接根据 PEB 获取 Module 信息

接下来我们需要尝试从 InMemoryOrderModuleList 链表中获取 Kernel32.dll 基地址。第一个节点，对应的 Module 的基地址是 7720000，对照图 4-6 发现是 ntdll.dll



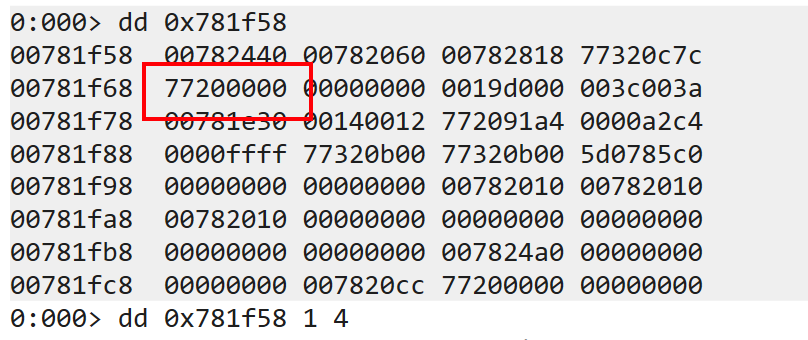


图 4-7 第一个节点

NEXT 指向的下一个节点，对应的 Module 基地址是 75330000，对照图 4-6 发现是

KERNELBASE.dll

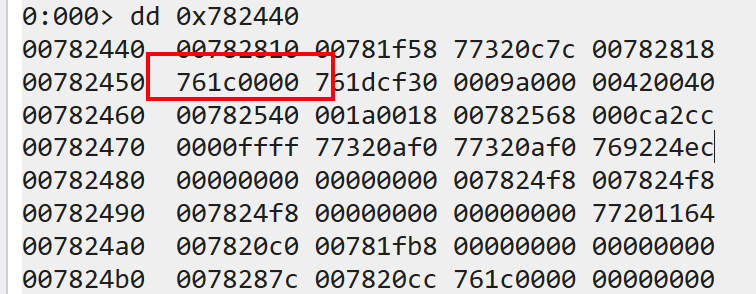


图 4-9 Kernel32.dll 基地址

通过上述过程，我们完成了从 PEB 中获取 Module 基地址的过程，kernel32.dll 在不同 Windows 系统上 InInitializationOrderModuleList 的位置有些区别，上述过程在 Windows 10 32bits 完成，发现 keernel32.dll 在 InInitializationOrderModuleList 的第三个节点的位置。大家需要在自己测试系统上根据上述步骤寻找 kernel32.dll 的位置。（基地址开机随机，节点位置固定）

1. 加载 User32.dll

User32.dll 可能程序中并没有加载， 此时需要 LoadLibrary 加载； 而LoadLibrary 函数位于 kernel32.dll 中；只需要从 Kernel32.dll 中找到 LoadLibrary 函数即可。

虽然在某一 DLL 中函数的偏移是固定的，但是出于通用性的考虑，我们并不直接硬编码偏移，而是 Shellcode 实现在 DLL 中搜索特定的函数。

1. Shellcode 实现

Kernel32.dll 基地址寻找；FS:30 始终指向 PEB 结构（根据上述 windbg 解析过程，实现如下代码）

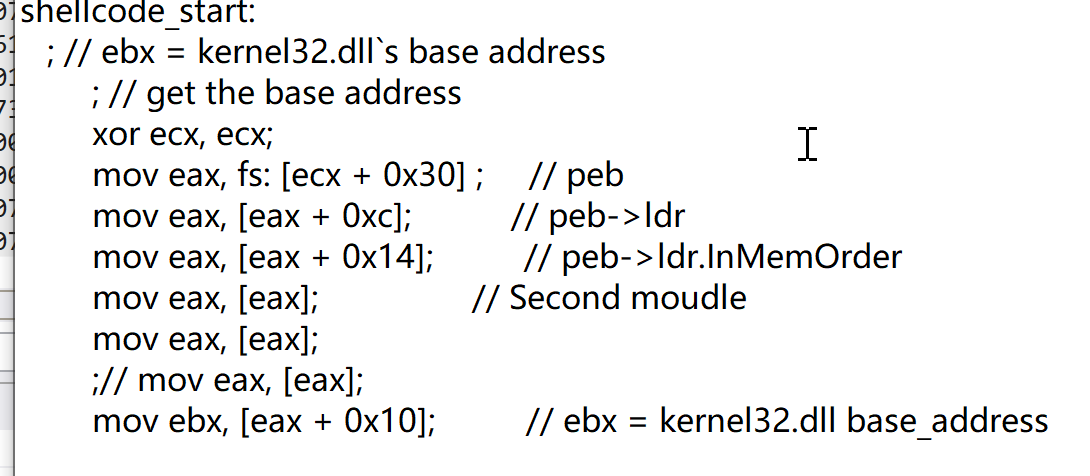
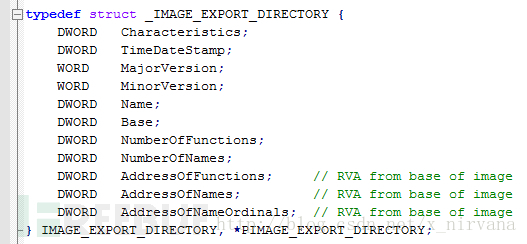
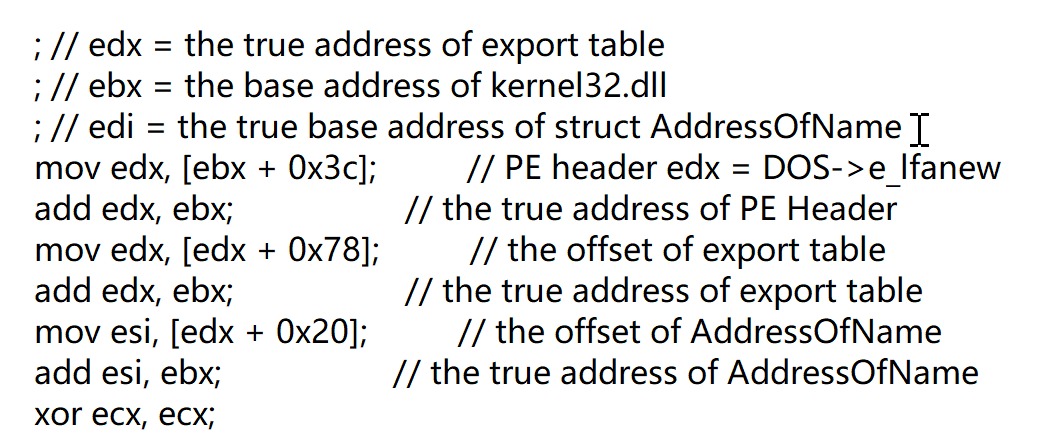


图 4-13 汇编实现寻找 Kernel32.dll 基地址

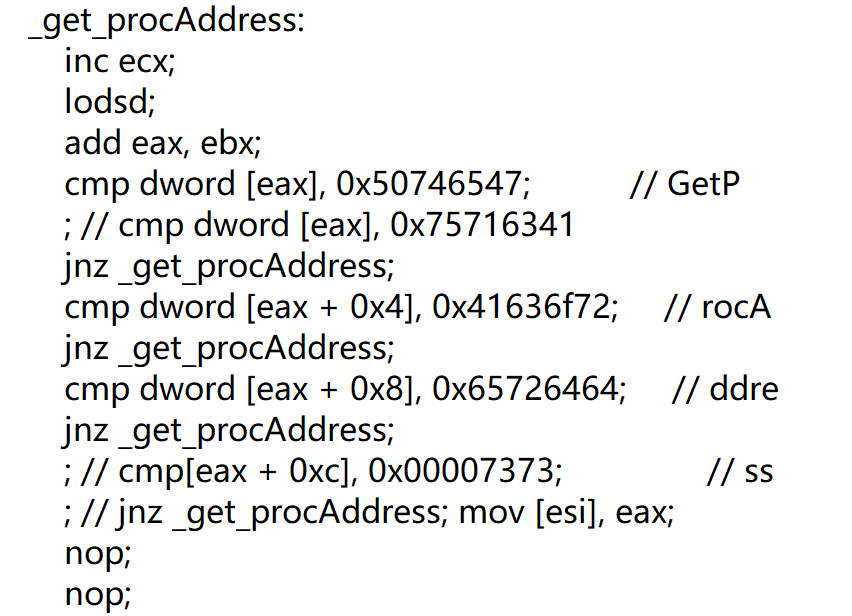
现在去获得导出表



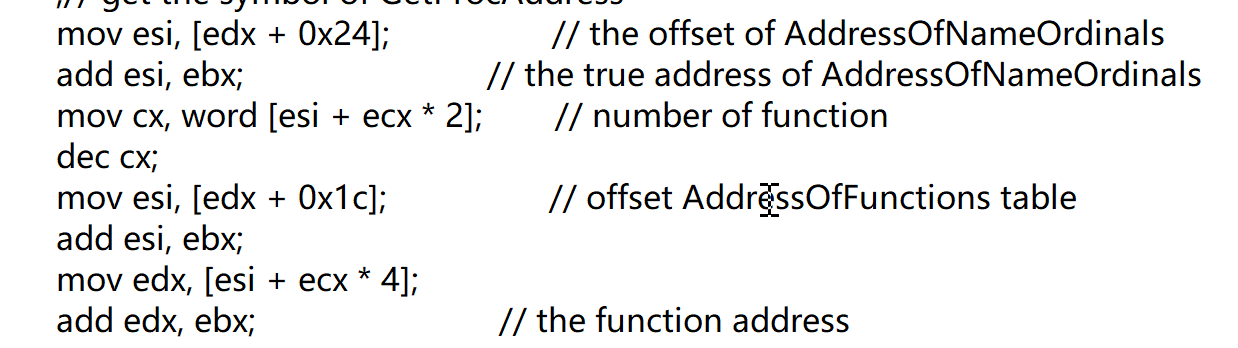
由结构体可得偏移量



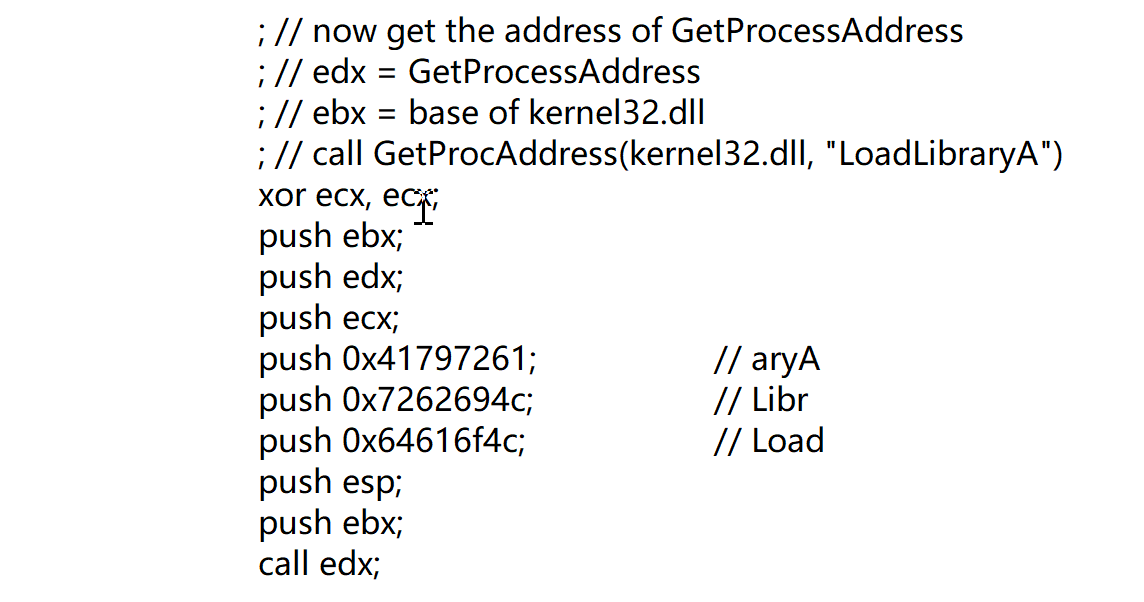
获得AddressOfNames之后，通过遍历名称表，找到GetProcAddress函数的名称所在的序号



然后通过序号，找到函数的地址



得到了GetProcAddress函数的地址，得到之后，调用GetProcAddress(“LoadLibraryA”)



得到 LoadLibraryA 函数后，我们执行 LoadLibraryA(“user32”)

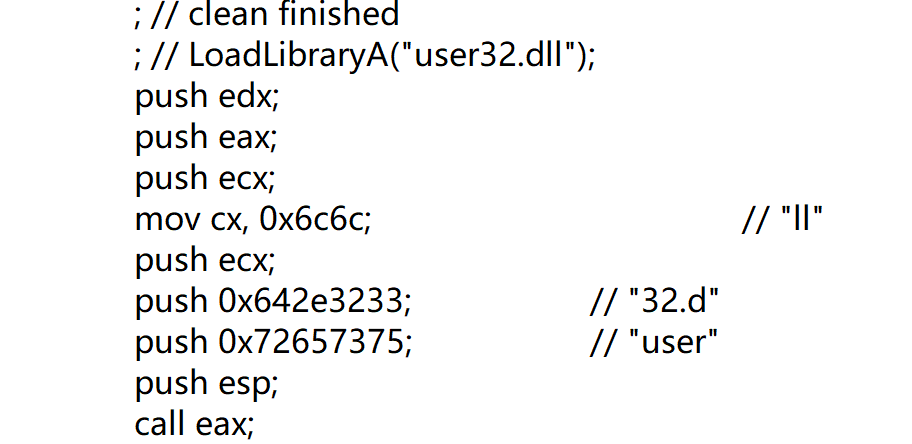


图 4-15 执行 LoadLibraryA(“user32”)

此时又得到了 user32.dll 基地址，同样的道理可以获得 MessageBoxA 函数地址

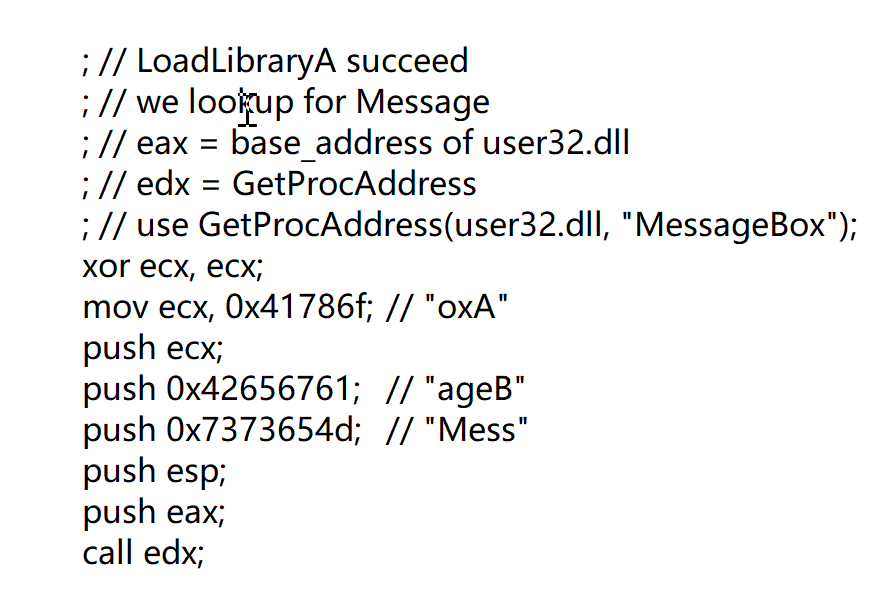


图 4-16 获取 MessageBoxA 函数地址

接 下 来 需 要 在 栈 上 布 置 MessageBoxA 函 数 参 数 ， 最 终 调 用MessageBoxA(NULL, “Shellcode”, “Hacked !”, 1)；注意保持堆栈平衡，以便执行完shellcode 后可以顺利恢复原执行流程。



图 4-17 MessageBoxA 执行

Shellcode 编写完后，使用 nasm 编译得到 obj 文件命令：nasm –f Win32 demo.asm –o demo

编译之后我们得到的是 obj 文件，并不能直接执行，我们需要提取出其中的

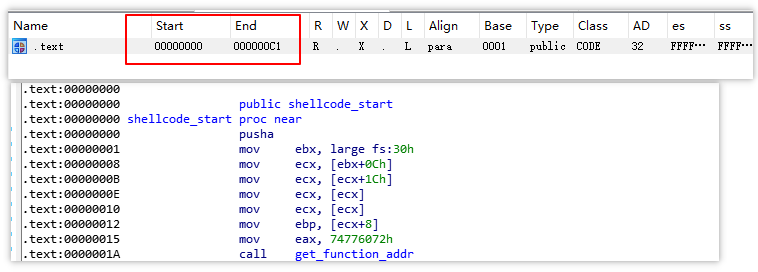
opcode，这里借助 IDA 和 Python 脚本可以提取出需要的 opcode。IDA 打开生成的 obj 文件，shift+f7 查看.text 段的位置及长度。

图 4-18 IDA 查看 obj 文件

脚本读取.text 段字节码（思考一下 start 为什么是 0x3c）：



图 4-19 从 obj 文件提取 opcode

得到 shellcode：

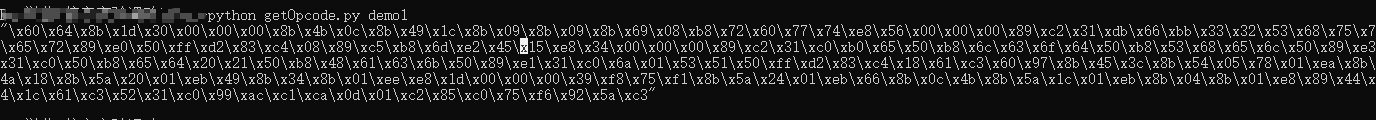


图 4-20 shellcode

接下来需要做的就是如何执行这些 shellcode，最简单的方式就是写一个 C 代码加载器（用的最多的）；

主要是修改缓冲区为 Execute，强制转换字节码为函数。

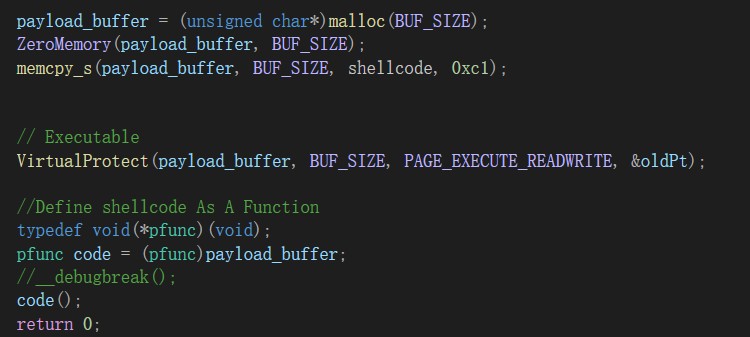


图 4-21 C 加载 Shellcode

1. 调试 Shellcode

在 shellcode 执行前（code()函数处）断下

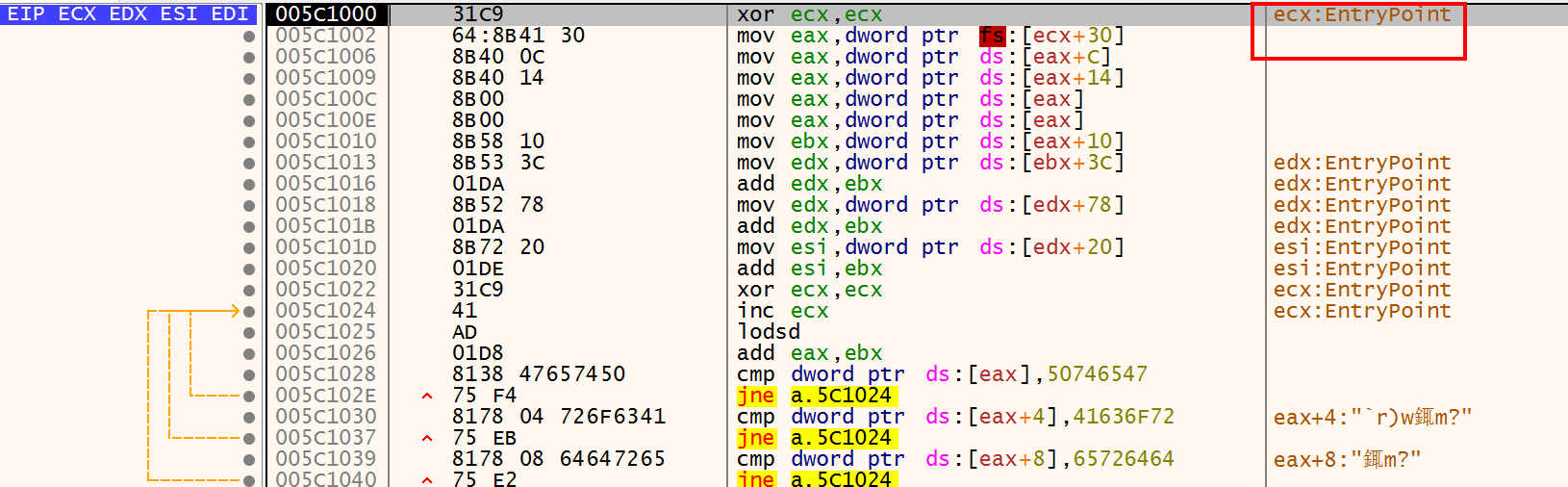
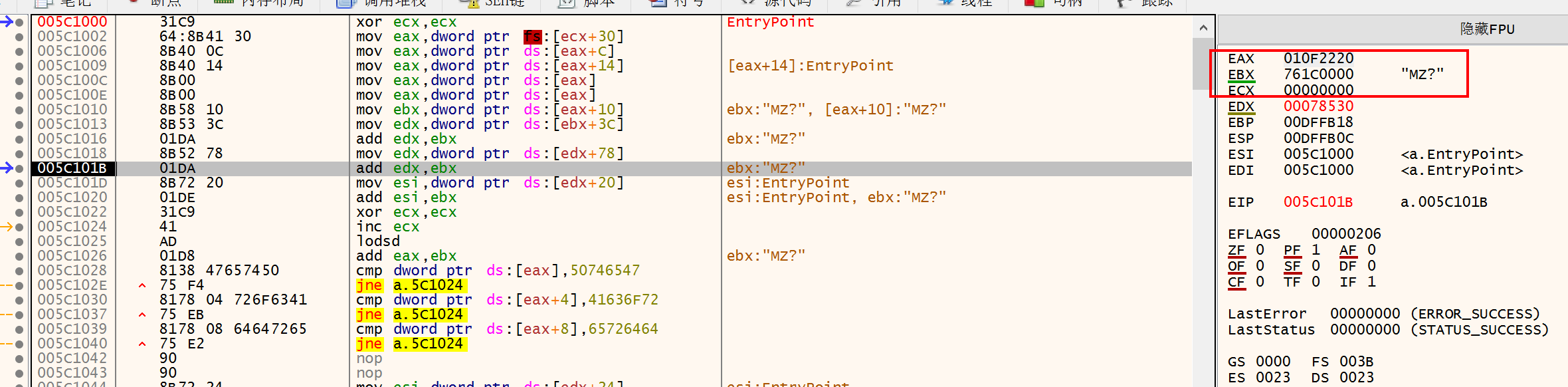


图 4-22 调试 shellcode 入口

F7 单步跟进，检查是否获取 kernel32.dll 基地址



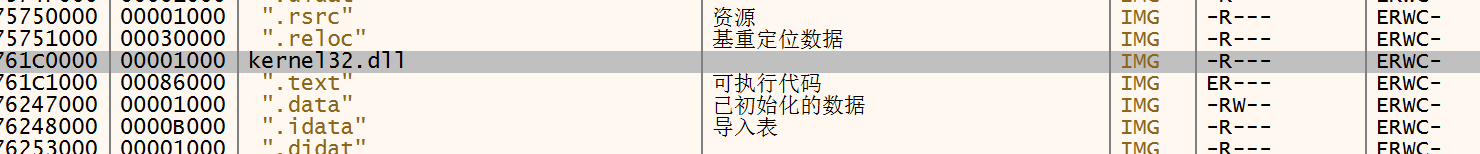
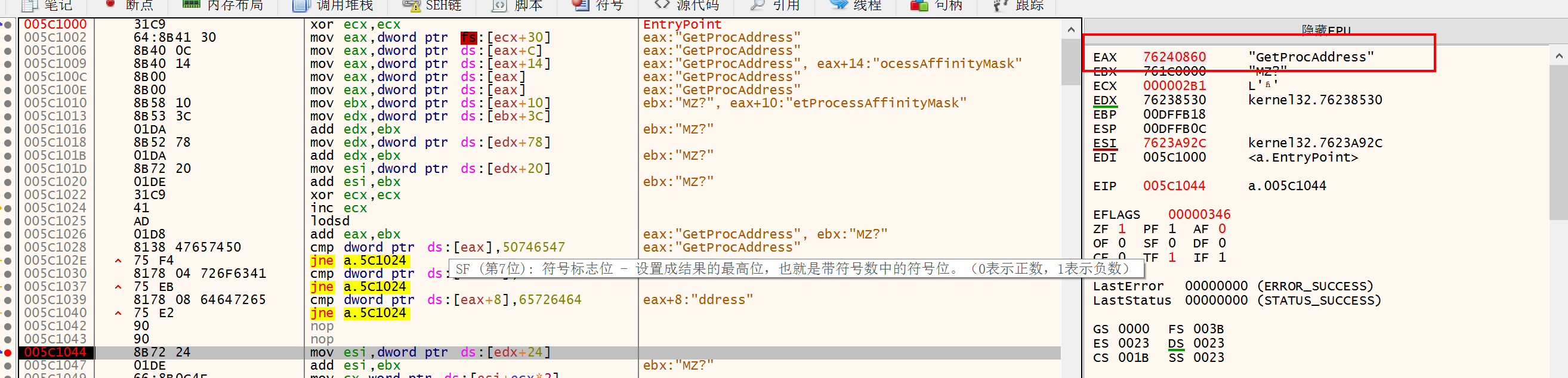


图 4-23 成功获取 Kernel32.dll 基地址

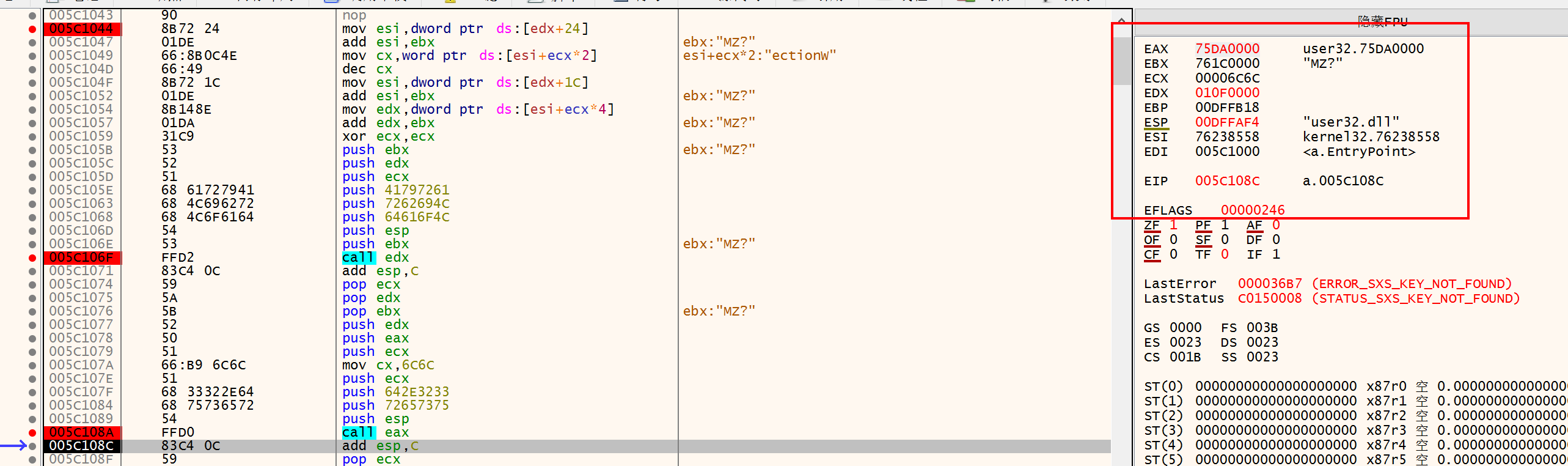
继续调试，得到GetProcAddress



得到LoadLibrary



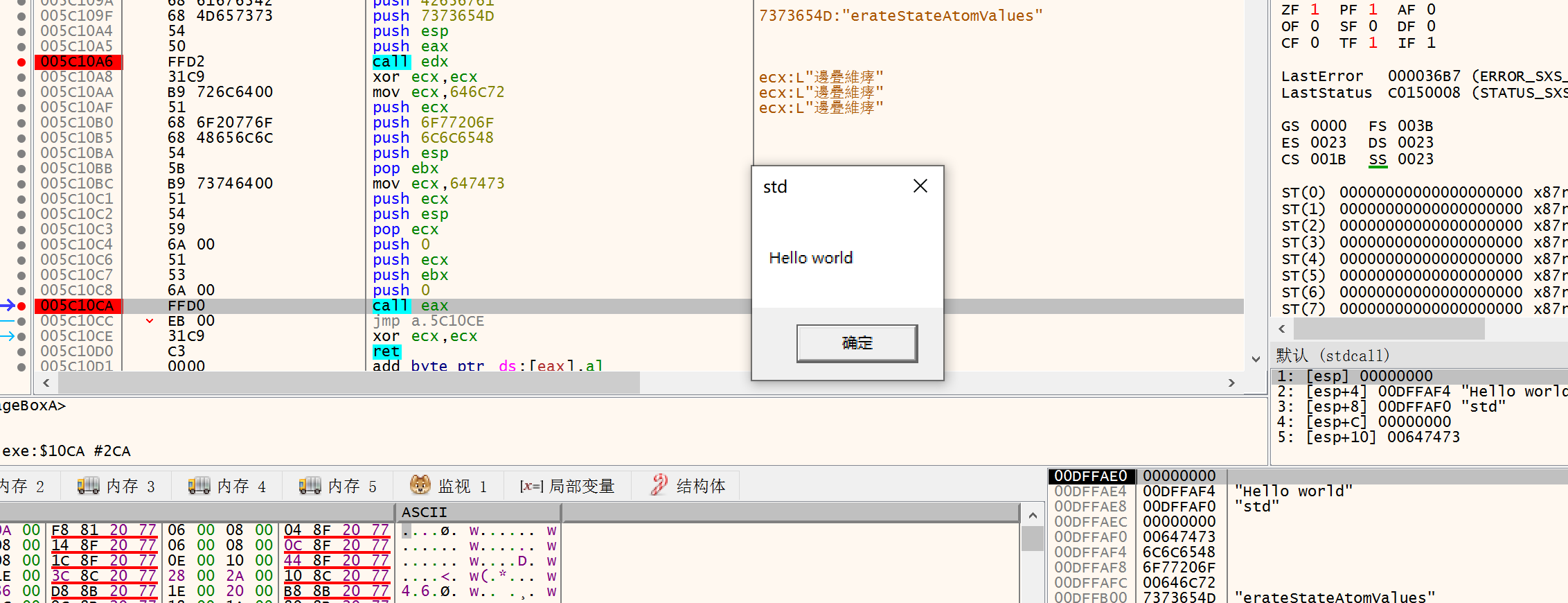
加载user32.dll



找到MessageBoxA



调用MessageBoxA



成功弹窗

# 思考题

1. 如何遍历 DLL 导出函数寻找目标函数？

从AddressOfName表中遍历函数名，找到函数名对应的序号，从AddressOfNameOrdinals找到对应的下标，从AddressOfFunctions 找到函数的地址。

1. 如何改进 Hash 使得查找 API 时不必考虑 DLL（不指定 DLL）？

没有使用hash函数（

个人觉得，遍历所有的DLL的导出表，就可以不必考虑DLL了（即遍历InMemoryOrderModuleList，找到所有的DLL，然后遍历所有DLL的导出表

采取hash是为了隐藏代码和压缩shellcode，但是在效率方面有所下降。