**Window Reverse && Debug – WIP and TODO**

[**Click ME to go to “TODO List”**](#_TODO_List)

由于在文档首部更新TODO List时, 会不可避免地破坏整个文档的结构, 打乱大部分内容, 因此将原TODO List放置于文档末尾, 并设置索引以跳跃至末尾的List中.

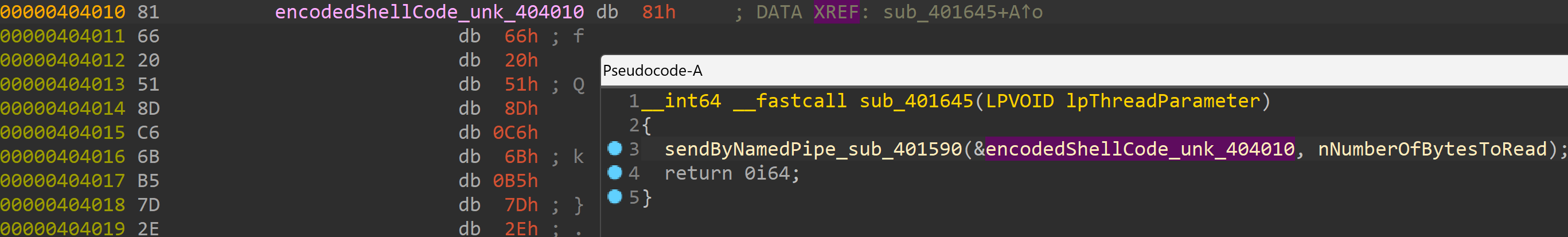
本文档用以暂时记录12月10日初入Window逆向以来调试的所有程式. 由于本档为草案形式, 不会进行特别细致的记录, 仅记录大致调试思路. 粗略记录处由后续逐渐补全.

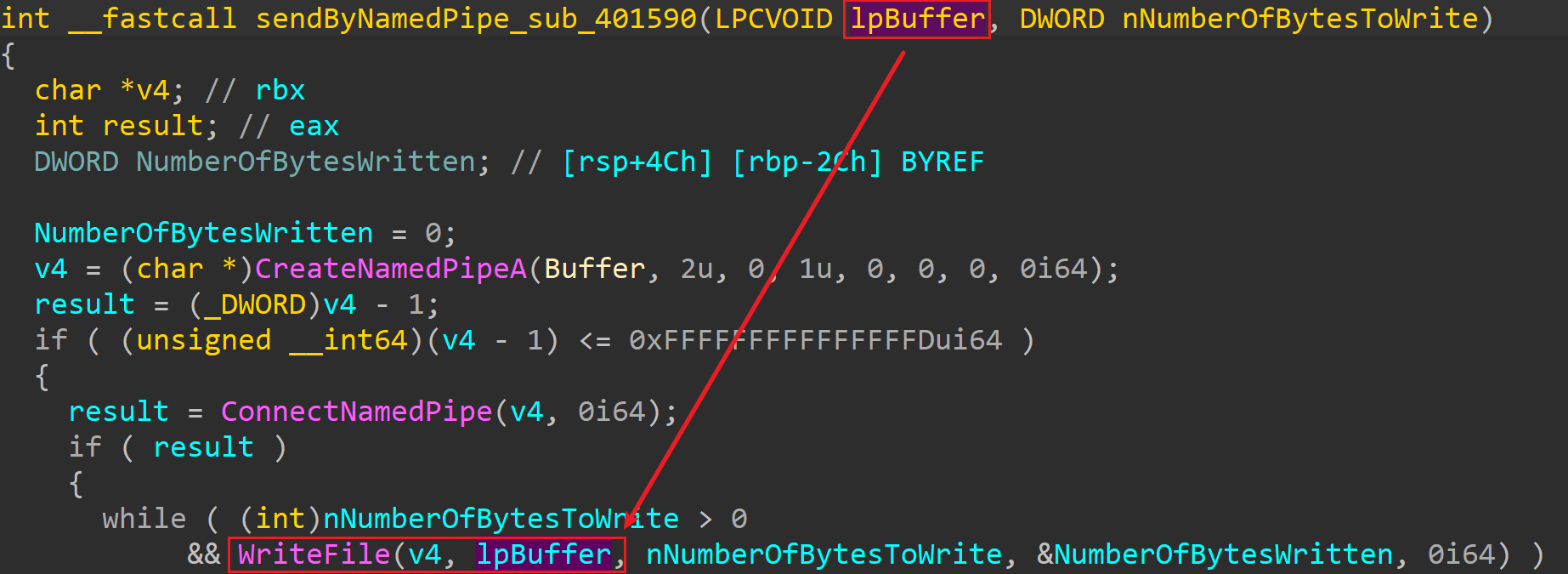
# 调试CS4.1 Beacon

由于Beacon调试时本档暂未创建, 故此本档仅简单记录beacon调试进度.

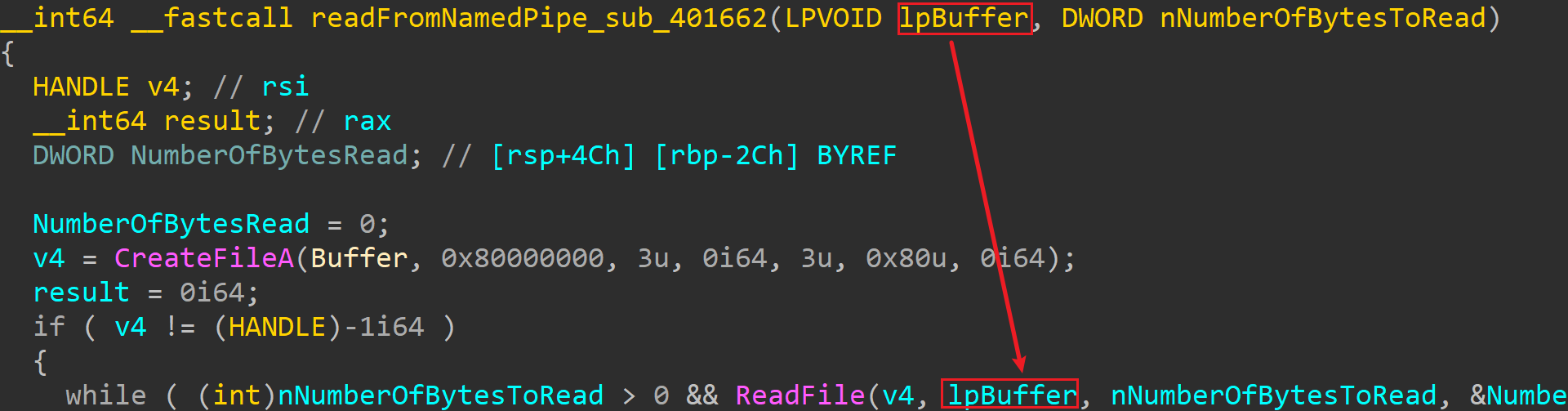
## Beacon初步加载细节

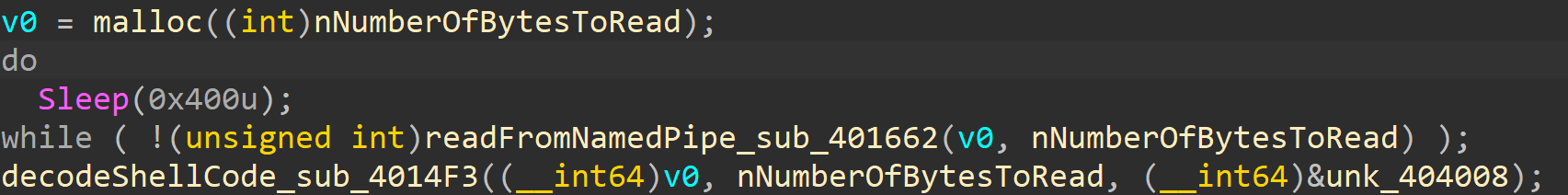
beacon会在载荷初始运行阶段从rdata段读取加密后的shellcode并进行解密, 解密脚本已经编写完毕. 在解密完毕后, 载荷会立即VirtualAlloc一片内存并设置其权限为可执行, 随后正式进入其shellcode头部.



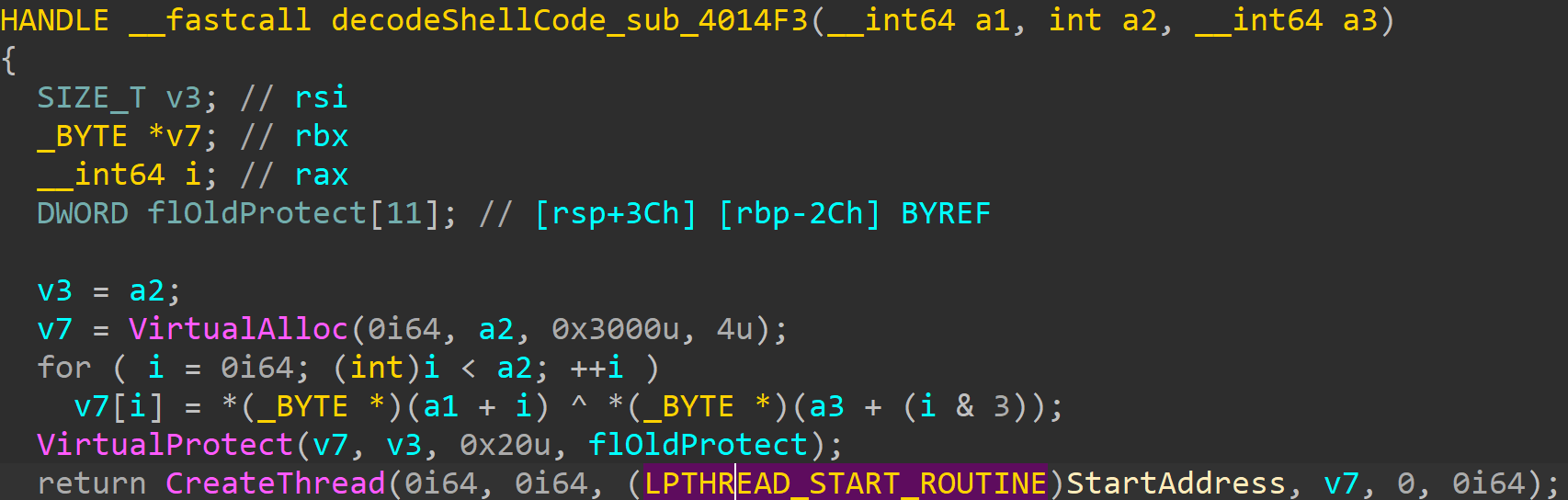


在分析beacon初始载荷的过程中, 了解到Windows特有的一种RPC手段, 即为管道通讯, beacon在初期通过命名管道来与协调其它子线程, 并通知其何时解密shellcode并加载. 如上伪代码所示, 载荷从rdata段取出特定符号下长度为0x379的数据, 这段数据即为加密后的Shellcode. 同时, 代码构建命名管道, 通过WriteFile接口将加密数据向管道内传输.



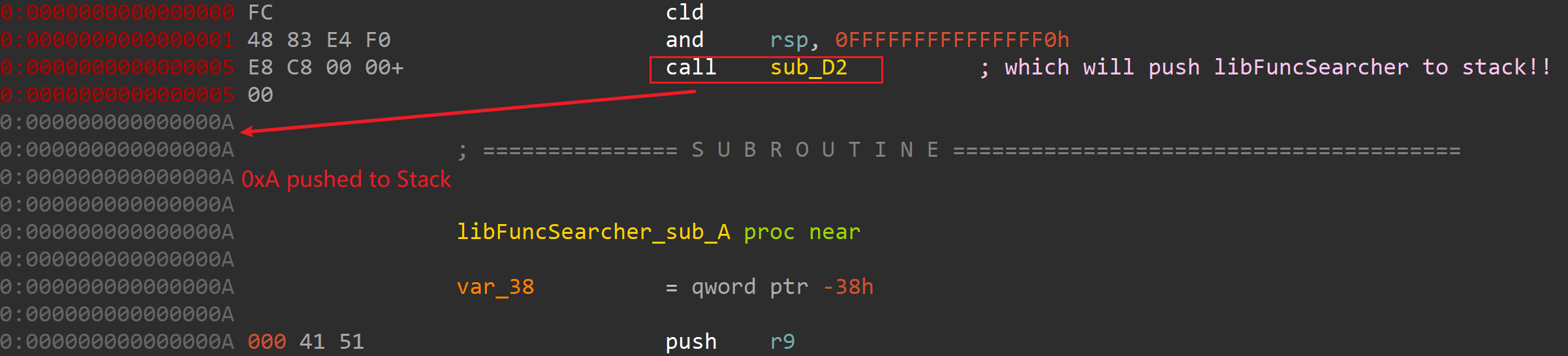


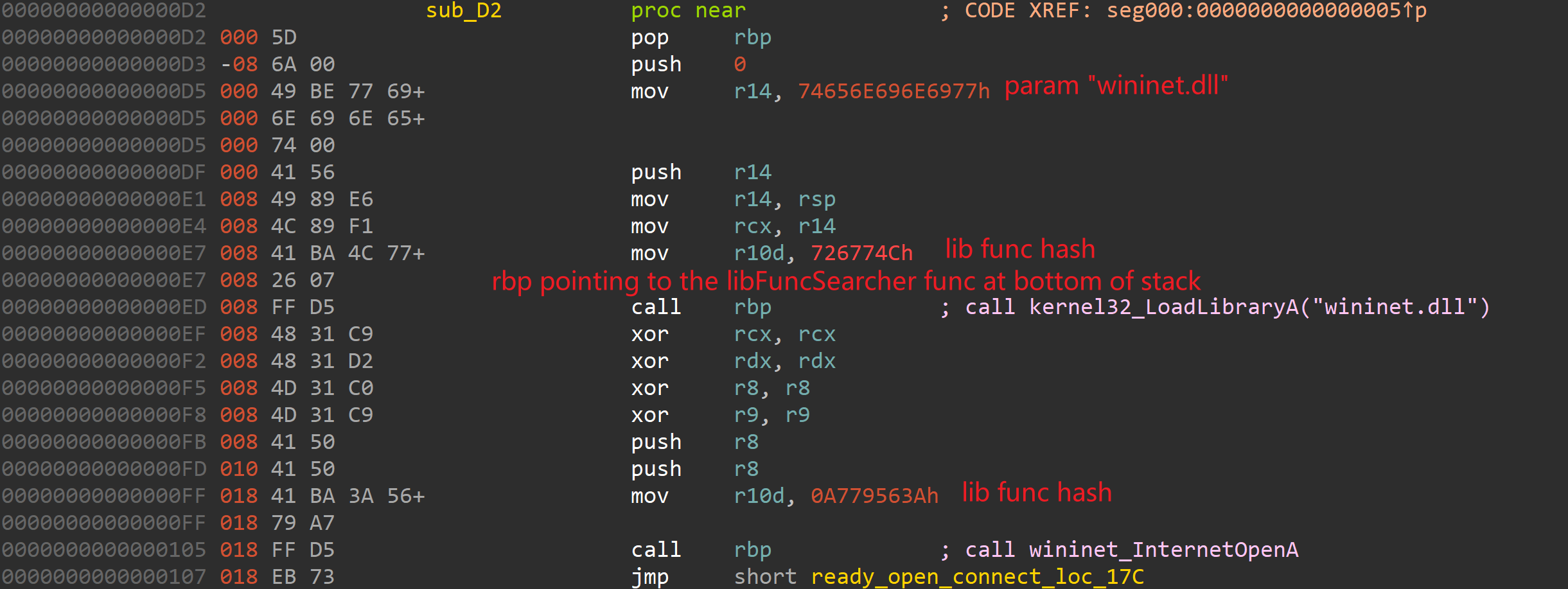
由于管道名称由符号Buffer控制, 故此搜索其交叉引用可寻至另一处函数内, 其函数逻辑将管道内容通过ReadFile接口全部读取, 并存放于函数的首个参数内, 基于其逻辑命名此函数为readFromNamedPipe. 而搜索此函数引用可知, 他处代码逻辑通过malloc分配了定长(0x379)内存, 并利用readFromNamedPipe函数将所有加密数据写入此内存中. 随后, 该内存首部指针被传入Shellcode解密函数进行处理.



解密函数逻辑很清晰, 首先通过VirtualAlloc分配内存空间, 再通过xor逐字节解密Shellcode并存入刚刚分配的内存中, 最终通过VirtualProtect修改内存属性为可执行, 随后执行Shellcode. 至此Beacon载荷加载逻辑分析完毕.

## Shellcode简单分析





shellcode头部会将另一个解密函数头部利用call命令压入栈内, 随后在需要使用到时弹出至rbp寄存器, 故此shellcode内大部分call rbp操作均为调用解密函数. 该解密函数的作用是通过入参的hash值, 在进程载入的Dll之中搜索并定位到指定目标函数入口, 随后jmp至函数内进行执行.

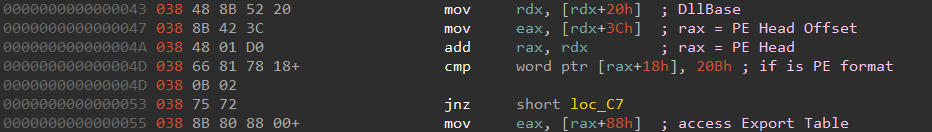
从该解密函数的分析过程中, 笔者学习并了解了进程可以通过gs段寄存器(x64情况, x86需要通过fs段寄存器)访问TEB以及PEB结构体, PEB结构体内又可以通过偏移访问\_PEB\_LDR\_DATA结构, 以此逐步遍历加载进进程的所有Dll. 同时学习到Windbg可以很方便地访问TEB等结构, 也可以通过从微软服务器拉取符号来观察对应的结构体(例如ntdll!\_TEB), 同时符号内显示的细节比MSDN给出的结构体细节要多, 因为MSDN会为了兼容性隐去部分细节

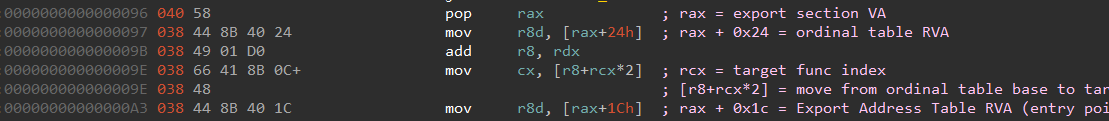
\_PEB\_LDR\_DATA结构

<https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data>

从此文章了解到LDR\_DATA\_TABLE\_ENTRY中的条目都是一一对应的, 而非直接的首尾相连 <https://blog.csdn.net/qq_59075481/article/details/135281393>

同时, 在解密函数通过TEB访问进程加载的DLL时, 解密函数还会依照PE结构对DLL基地址的各个偏移位置进行验证, 包括验证PE头部是否存在, 验证Export Table是否存在, 以及访问Export Table内所在的Section RVA等等位置以获取函数名称与DLL模块名称.





上图是当时调试时对关键位置的部分标记.

PE Format <https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#the-edata-section-image-only>

MSDN的PE Format章节非常详细地记录了一个标准PE文件应该包含的所有要素以及结构体. 例如上述链接描述了Export Table(.edata section)的具体格式. 其它格式可以借助010 Editor的PE解析模板辅助理解.

# 调试Proxifier3.4.2

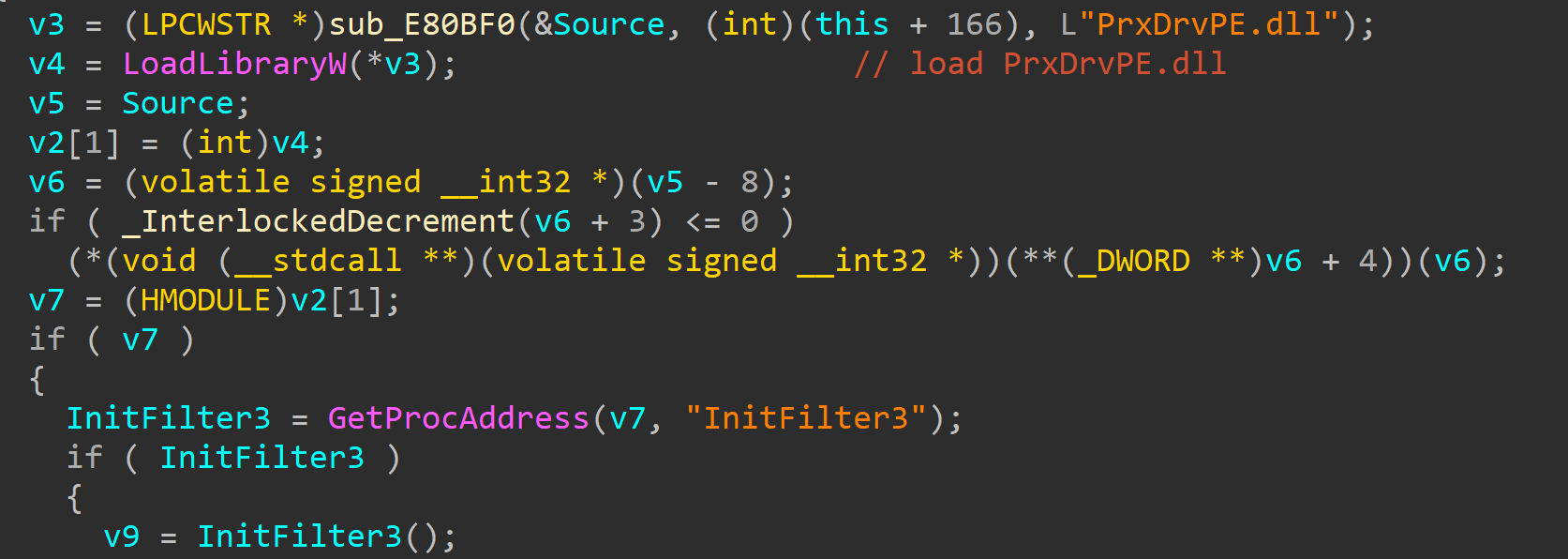
## 注册码破解 – 粗略记录

由于此工作进行时, 本档未被创建, 大多数细节此刻不进行记录, 故此仅记录思路. Enter Registration按钮内会要求输入对应注册人与注册码, 错误注册码会弹出对应错误提示, 例如注册码格式应该为XXX, 注册码过期等等, 在IDA中可以搜索到对应引用, 并修改对应判断逻辑以绕开校验逻辑.

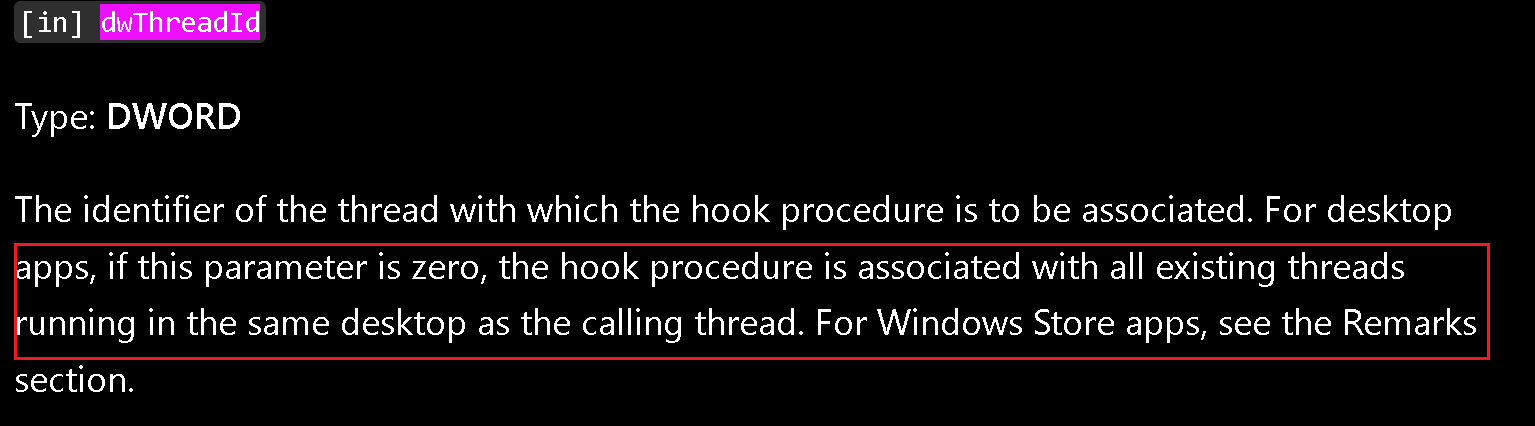
另外需要注意的是, 软件字符串使用了宽字符模式(wchar\_t), 所有字符后均跟随一个00字节, 搜索时应注意转换.

## 全局消息挂钩与Dll注入

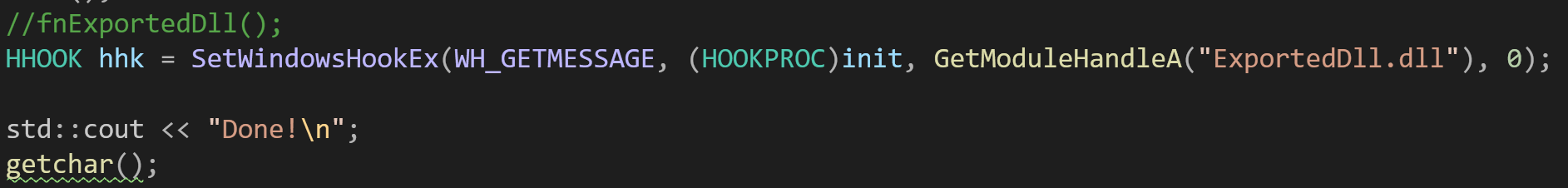
软件目录下存在PrxDrvPE.dll文件, 全局搜索之, 可以发现PrxDrvPE\_Hook\_sub\_412F30内存在引用.



函数在调用LoadLibraryW加载此dll后找到其InitFilter3符号处地址, 并随之执行. 分析可知此地址下的指令调用SetWindowsHookExW将WH\_GETMESSAGE设置了全局钩子, 且threadId指定为0, 这意味着所有会触发WH\_GETMESSAGE的进程都会被注入此DLL. 详细可见此接口MSDN文档.

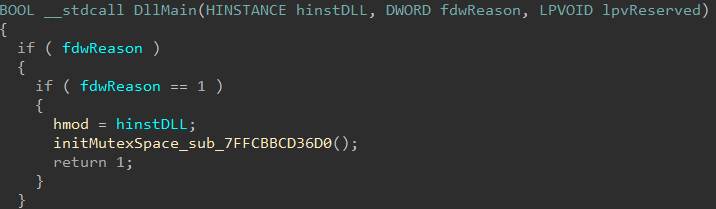


<https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexw>

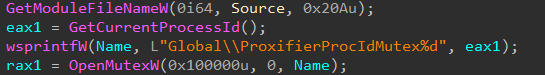


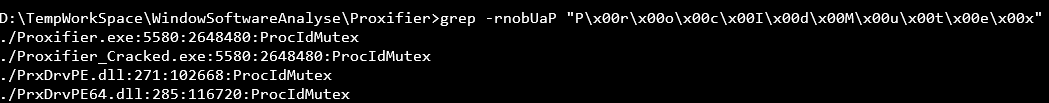
编写一个程序进行实验, 以上代码可以向所有触发了WH\_GETMESSAGE的进程注入DLL. 换言之, Proxifier试图向所有使用了消息机制的进程注入PrxDrvPE.dll, 而至于为什么选用此挂钩回调, 猜测应该只是为了选择一个合适的注入时机.

## WS2\_32.DLL关键函数HOOK



在通过钩子使得目标程序载入Dll时, **DLL\_PROCESS\_ATTACH标记会在DllMain函数被执行时传入(详细看MSDN相关文档), 此时执行到分支内的函数, 暂且将其命名为initMutexSpace.**



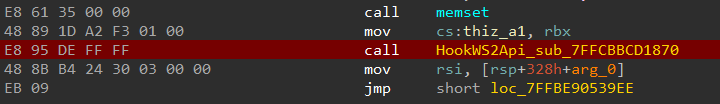


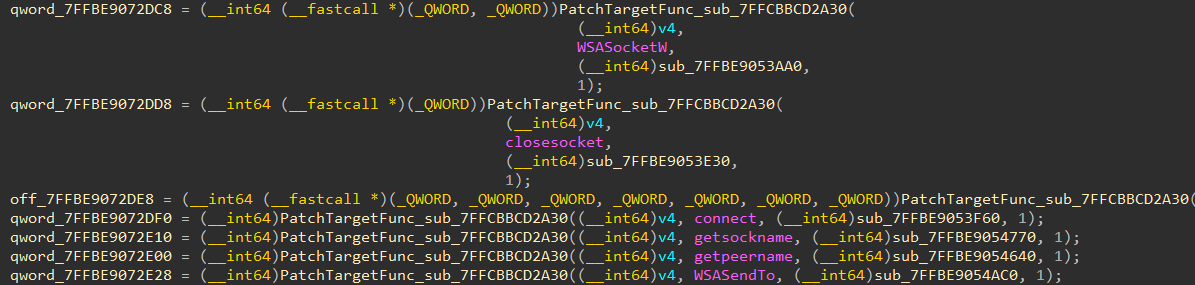
函数在开头通过特定的标记获取Mutex对象句柄, 该句柄由主程序代码创建, 创建时机与相关逻辑暂未分析, 不过猜测主要目的是为了协调/同步被Hook进程的注入后逻辑. 根据其代码逻辑, 故此命名之**initMutexSpace.**

有关MutexObject的规范以及用途(主要用于进程同步)可见MSDN文档:

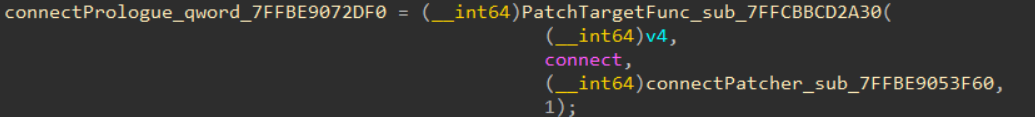
Mutex Objects: <https://learn.microsoft.com/en-us/windows/win32/sync/mutex-objects>

Wait Functions: <https://learn.microsoft.com/en-us/windows/win32/sync/wait-functions>

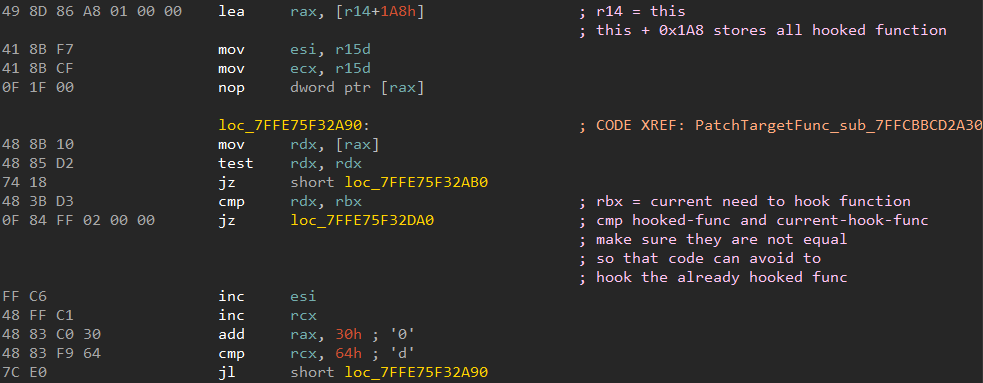




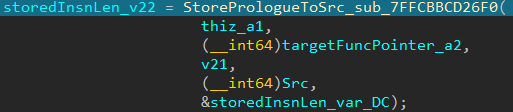
在获取互斥对象后, 代码会执行进入另一处函数以对进程的WS2进行Hook, 顾名思义, 命名之为HookWS2Api, 其内部逻辑对WS2的多个主要函数进行了Patch, 例如WSASocketW, closesocket, connect等等, 修补逻辑函数命名为PatchTargetFunc, 此时暂且忽略HookWS2Api内的其它逻辑, 仅跟进最核心的修补逻辑.



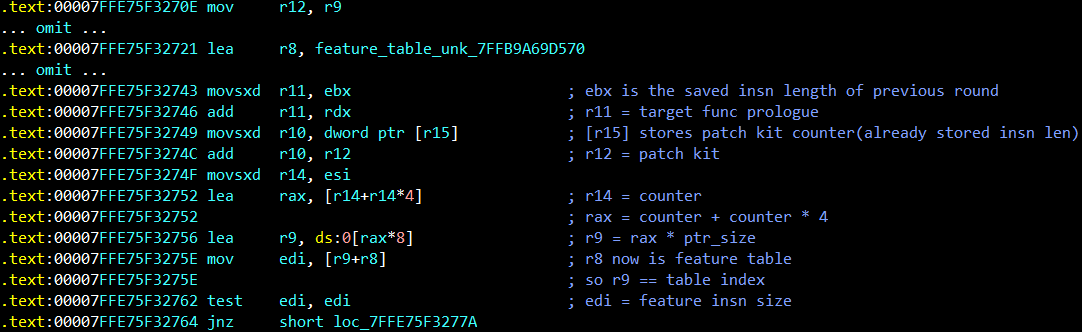
首先观察传入PatchTargetFunc函数的各项参数, 此函数需要4个参数, 分别是this指针, 指针指向的结构体存储了许多关键内容与标记, 在后续的分析中, 本档不会对结构体内的所有细节进行分析, 仅仅记录在结构体内使用到的关键偏移与内容; 其次是需要被Hook的WS2\_32目标函数指针与接管目标的Patcher函数指针; 最后一位参数通常为1, 暂不知晓其作用.



PatchTargetFunc函数的指令序言中, rcx(this指针)与rdx寄存器(被Hook的目标函数指针)分别被转储至r14与rbx寄存器内, 随后指令将rax寄存器内, 即[r14+0x1A8]偏移处4字节(指针长度)的具体内容取出, 并判断其是否为0, 若不为0则将其与目标函数指针进行对比, 一旦内容一致则立刻jmp至函数尾声并结束执行, 反之, 则在此偏移处继续偏移0x30, 随后重新进入循环. 若偏移处内容为0, 则直接跳出循环以执行PatchTargetFunc函数的其它逻辑. 经过后续的调试, 可以得知this指针偏移0x1A8处实际上存储已经被Hook过的目标函数指针列表, 列表内的每一项长度为0x30, 那么这部分指令的逻辑便不难理解了, 代码尝试遍历列表内的函数指针, 将其与目前需要Hook的目标函数指针进行对比, 并避免对已Hook后的函数进行重新处理导致的程序出错. 同时, 这段循环还存在一个函数总量上限, 被设定为0x64(即为100), 这意味着被Hook函数不能超过100个.



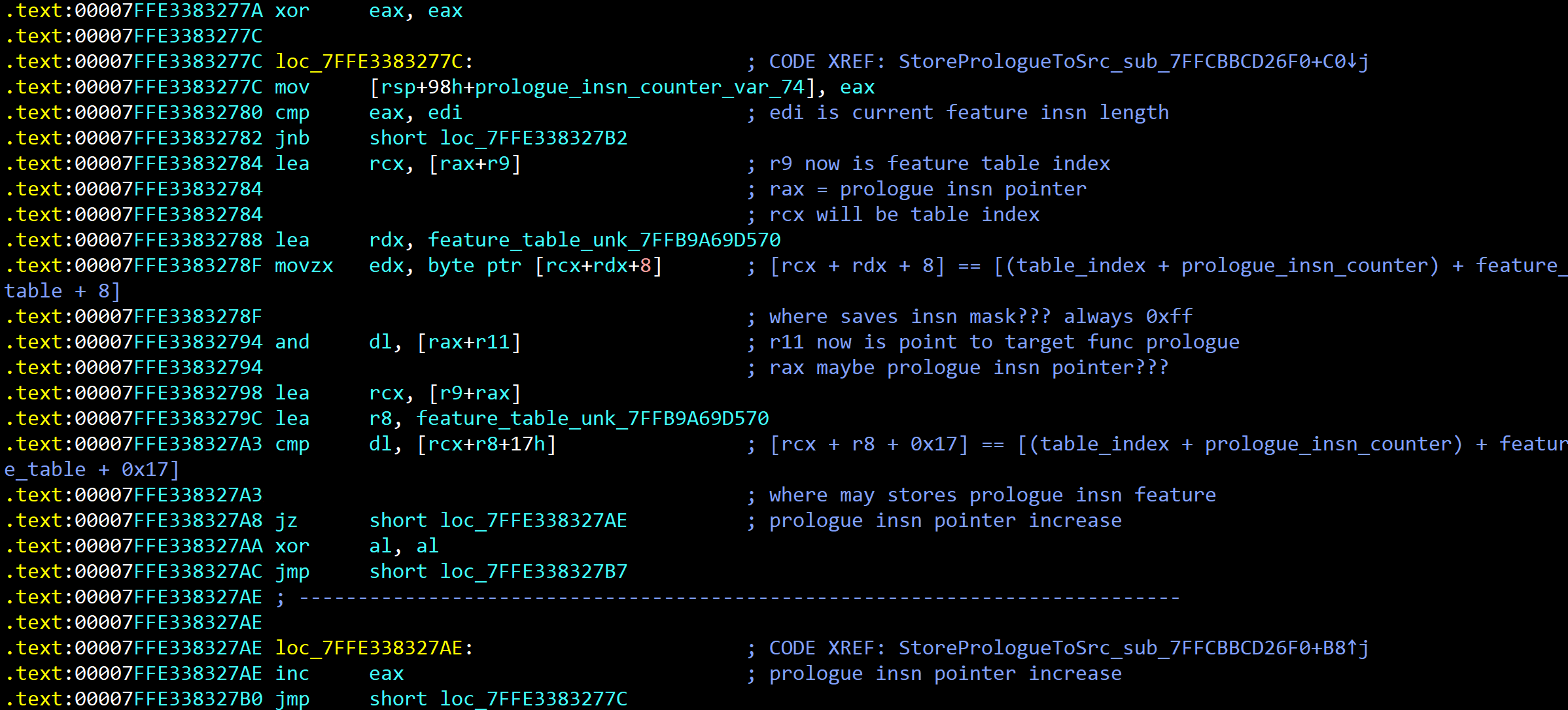
在检查完毕需要Hook的目标函数没有出错后, 忽略无关代码, 执行将进入另一个函数. 经过后续调试, 该函数用以将目标函数序言内指定长度的指令转储至第四个参数所指的地址位置, 并返回一共存储了多长的指令, 故此命名此函数为StorePrologueToSrc. 同样先分析传入的5个参数, 首两个参数分别是this指针与目标函数指针, 第三个参数指定了至少需要从目标序言内转储的指令总量, 该值通常为0xE, 第五个参数作用较为复杂, 后续分析中会提及.



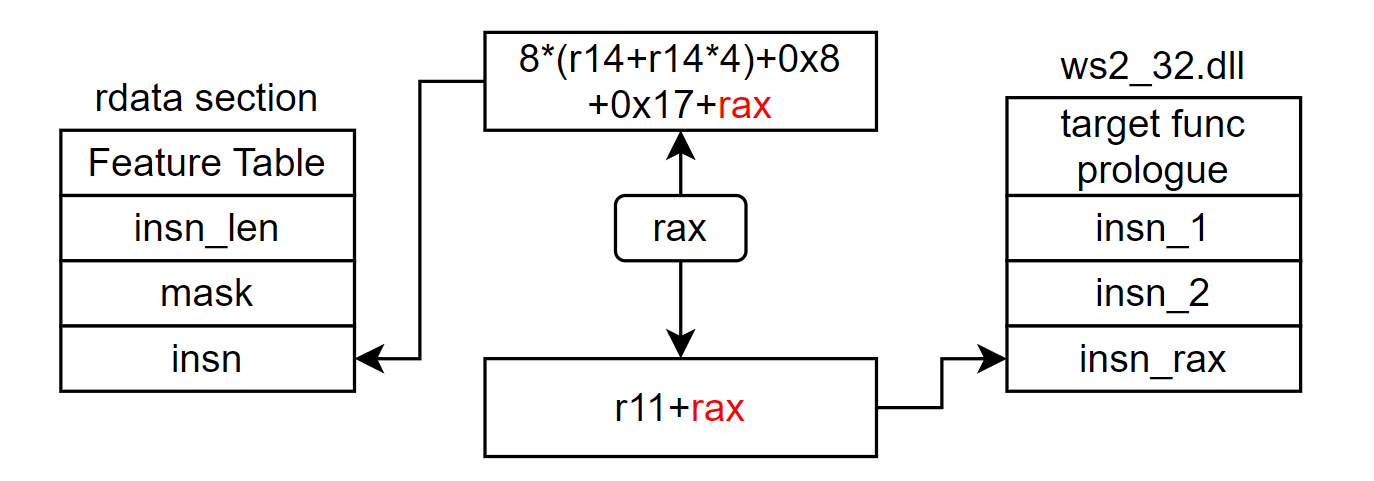
为了辅助理解与方便记录, 本档为部分指令添加了注释, 并预设当前视角的调试者已经知晓部分调试后期才能得知的讯息. 这是函数序言的部分指令, 首先其将r9寄存器(第四参数)转储至r12寄存器, 随后将rdata段内的某段地址转储至r8寄存器, 注意, 该地址内存储了一个重要的结构体, 且此结构体大量参与了后续指令中的序言转储工作. 接着, 指令使r11寄存器等于rdx + ebx, 其中ebx寄存器记录了StorePrologueToSrc函数在处理当前目标函数时已经转储的指令长度(由于此时函数还未处理目标序言, 因此ebx此刻为0), 换言之, r11在此时应始终指向目标序言内还未处理的指令首部. 继续向下执行, [r15]也存储了已转储的序言指令长度, 随之令r10等于r12 + [r15], 结合前文内容可知r10与r11是同步更新的, 拥有转储空间与目标序言空间内的映射关系.

继续执行, r14实际上是r8寄存器所指结构体内的”计数器”, 函数的转储工作实际上是在一个大的指令循环体内完成的, 一旦有序言指令成功转储, 则ebx, [r15]与r14寄存器均会同步更新, 不同的是每次循环时, 前者更新的是已转储指令长度, r14计数器则仅进行一次递增. 接着, 指令利用计数器将edi寄存器指向r8结构体内的指定偏移位置8\*(r14+r14\*4), 若内容非空则进入下一分支.

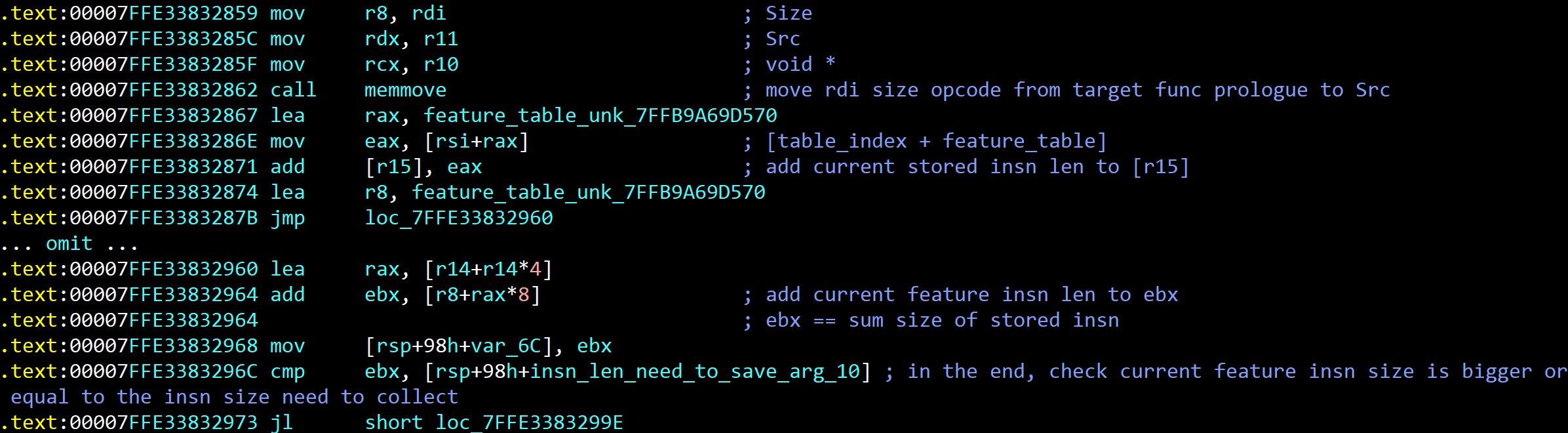
此处, 本档开始适当地解析r8寄存器所指结构体的内部结构与整体作用, 以便辅助理解上述指令以及后续指令中的代码意图, 此后, 本档将继续沿用此记录模式, 即先阐述指令行为, 后分析部分指令意图与关键结构, 最终在函数指令整体执行完毕后, 对关键结构与函数整体进行行为总结. r8寄存器所指结构体实际上是一个特征指令列表, 因此上文中本档将rdata段内的此处符号命名为feature\_table, 该列表的每项存储了不同的特征, 以及与特征相关联的元素或数据, 前文的指令意在将r14寄存器作为列表的索引值, 而列表的相对偏移位置8\*(r14+r14\*4)即为每项首部, 首部存储了此项内特征指令的总长度.



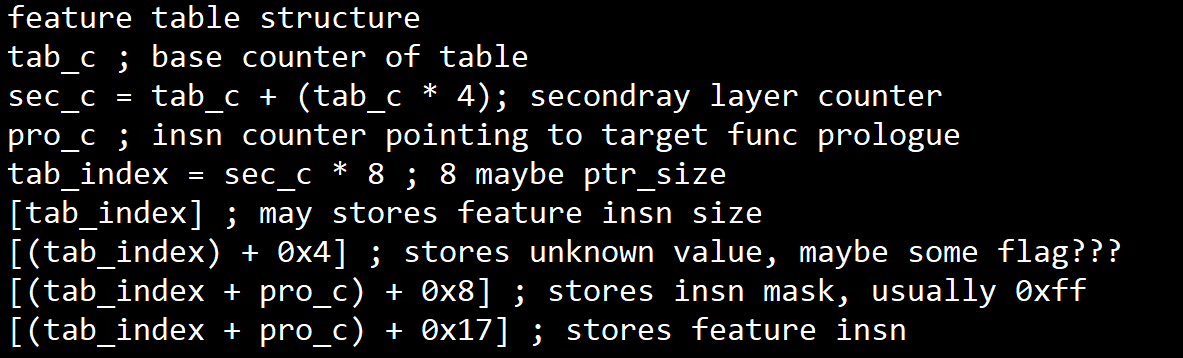
继续执行, 指令首先利用xor将eax寄存器置零, 接着进入一个循环块. 循环块首部指令会判断eax是否大于edi(特征指令总长), 是则跳出循环块执行其它逻辑, 反之继续执行. 接着, 指令使edx指向特征列表的相对偏移rax+(8\*(r14+r14\*4))+8处, 该偏移存储的是指令掩码, 大多数情况下均为0xff, 随后程序使用and指令将掩码与[r11+rax]进行一次与操作, 并将操作结果与特征列表掩码处再偏移0x17处的值进行比较, 若结果不同则置零rax寄存器, 并跳跃到远处的指令, 这最终将使r14寄存器递增并回到处于函数序言附近, 这些操作意味着函数正在索引至特征列表的下一项. 反之, 若结果相同则使rax递增, 并回到循环块开头, 重复前文操作.



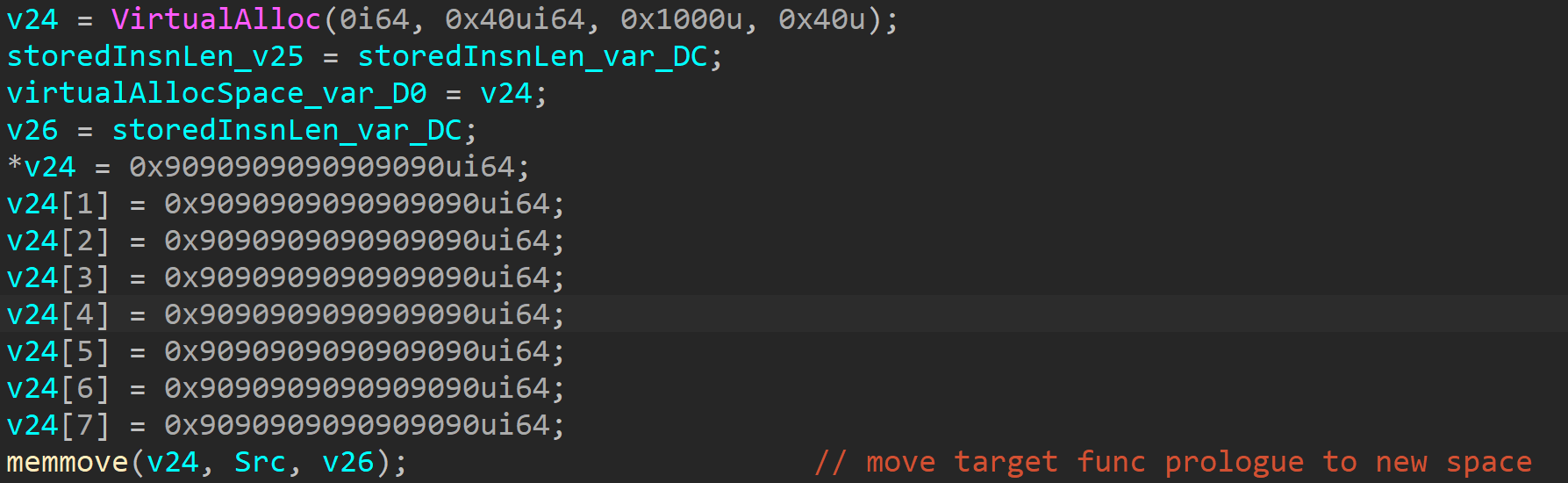
上述操作中, r11指向目标函数序言首部, 那么rax显然是一个在特征列表与目标序言内相互同步的指针, 每次循环块进入下一次循环时, rax均会递增, 指令随即会利用rax索引存储在特征列表内的指令, 并同时索引至目标序言内的指令相互进行比对, 直至rax大于edi内的特征指令总长并结束循环.



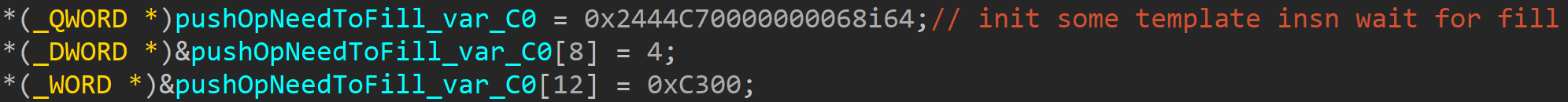
在跳出循环块后, 指令调用memmove将目标序言内rdi长度的指令拷贝至r10所指地址, 根据前文提及的StorePrologueToSrc函数序言逻辑可知, r10指向第四参数再偏移已转储指令长度[r15]的位置. 拷贝完毕后, 指令随即更新[r15]内的已转储指令长度, 随后判断目前已转储的指令长度是否大于需要转储的最少指令总量(第三参数), 若大于则进入函数尾声, 结束函数, 反之则回到函数序言继续进行转储.

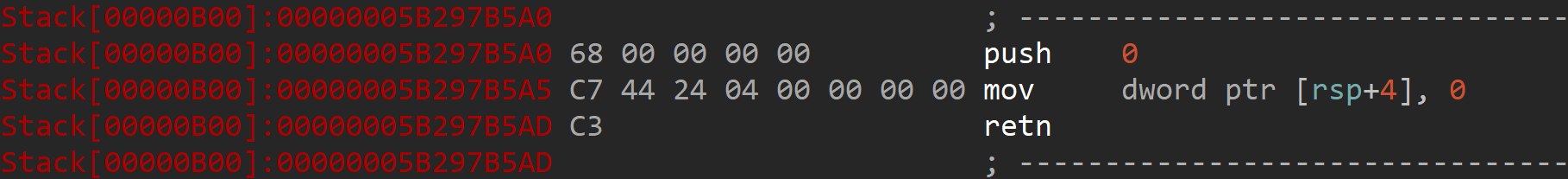


上图为本档在当时进行调试时, 对特征指令列表结构的分析结果.

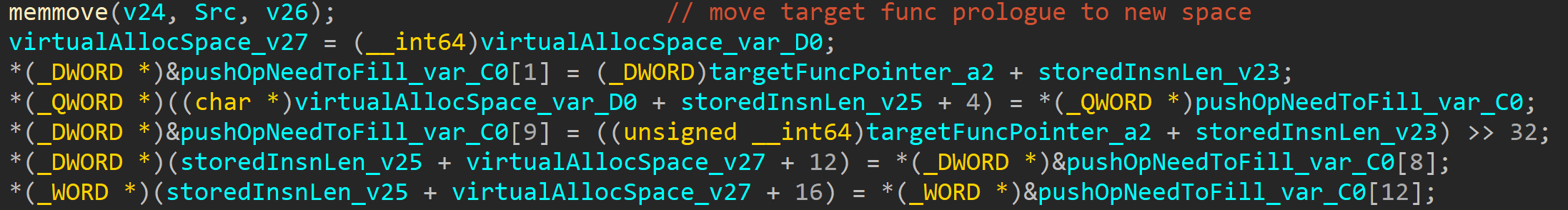


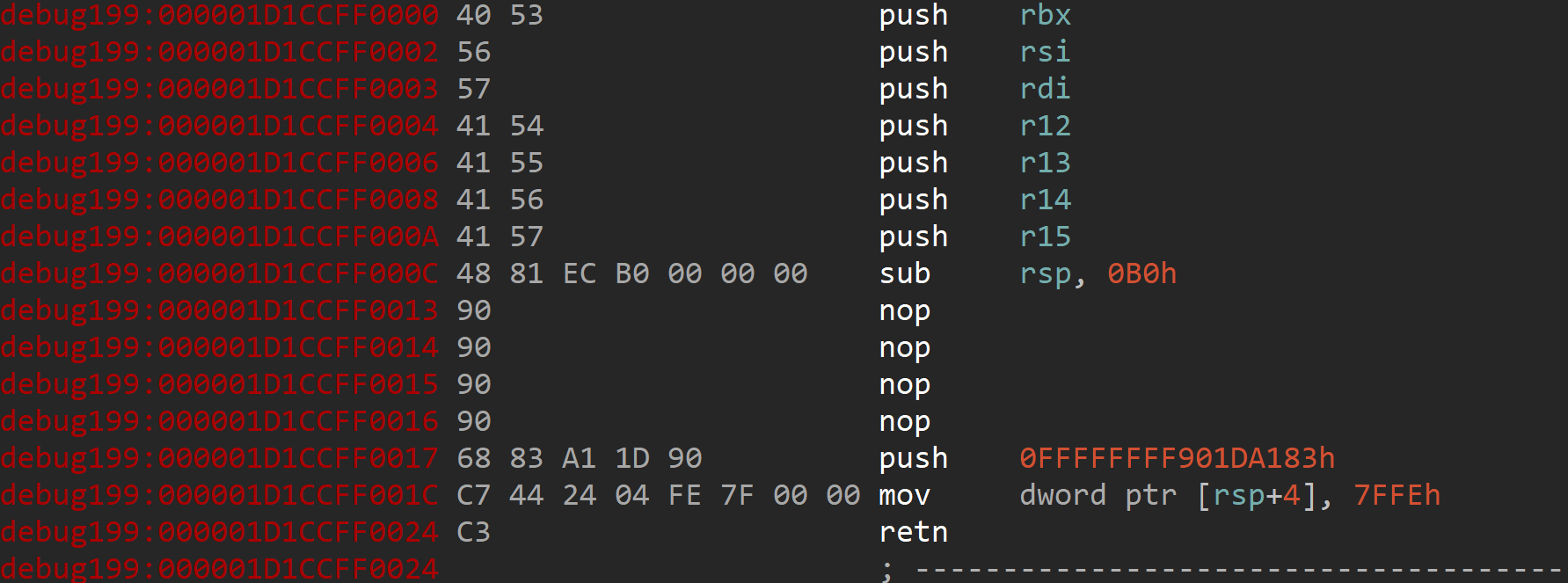
结束序言转储逻辑后回到PatchTargetFunc函数, 代码通过VirtualAlloc分配了0x40大小的空间, 并在此空间内填充了足够多的nop指令(0x90), 随后将前文通过StorePrologueToSrc函数转储的目标序言指令串再次填充到这个新空间的首部. 该空间由调试器IDA自行命名为”debug199”.

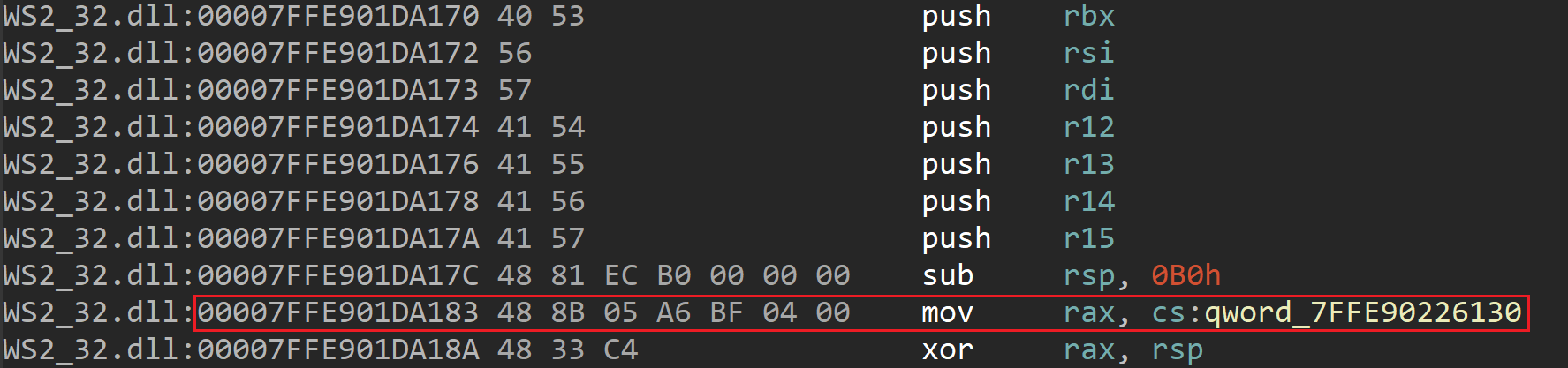




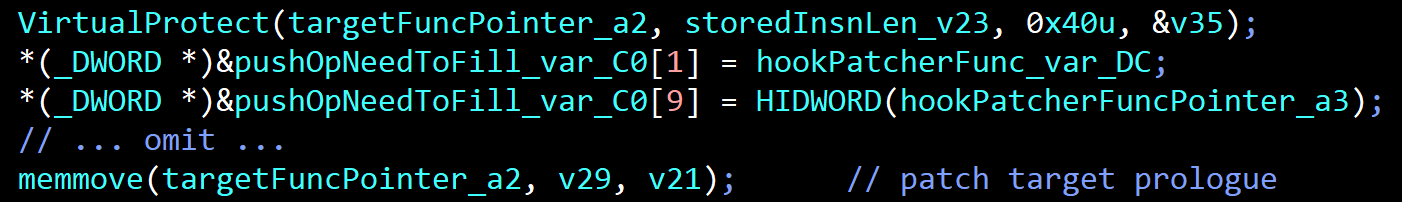
随后, 代码在栈内填充了一些指令模板, 进入对应的栈空间查看填充的内容, 可见填充的指令显然是在尝试通过push与mov指令控制栈内返回地址, 并利用retn指令劫持控制流至程序自行定义的地址内, 此时该模板还未指定需要劫持进入的目标地址, 因此指令操作数均为0, 且由于模板以指令push开头, 命名栈上变量为pushOpNeedToFill.

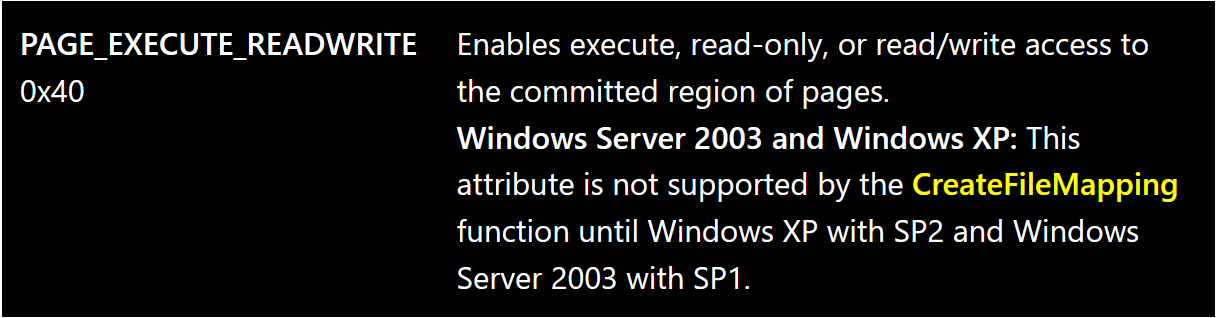






在转移序言指令到debug199段后, 代码开始填充控制流劫持指令模板, 随后将填充结果也一并填充至该空间内序言指令之后. 以WSASocketW函数作为需要Hook的目标函数为例, 进入debug199段首部, 可见在其拷贝出的序言指令之后, 控制流劫持指令将返回地址劫持至0x7ffe901da183, 进入此地址后可知其实际上是原WSASocketW函数内未被拷贝的其余序言指令首部. 而经过后续调试可知, 当接管WSASocketW函数的Patcher执行结束后, Patcher会主动进入到该段以使控制流回到原函数逻辑.



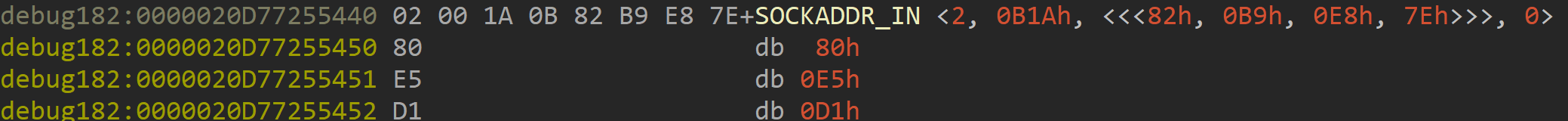


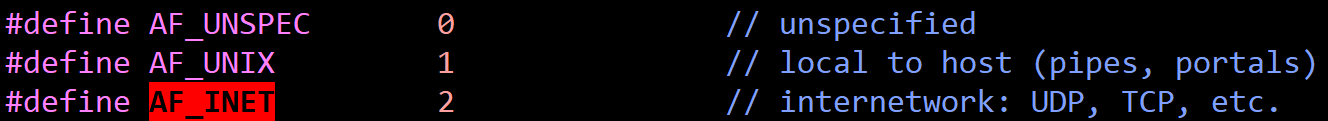
在处理完debug199段后, 代码通过VirtualProtect接口使目标函数序言内已拷贝指令长度的内存保护属性修改为RWX, 接着再次填充劫持指令模板, 不同的是, 此次的填充目标是Patcher函数, 随后, 代码通过memmove将劫持指令填充至目标函数首部. 如此一来, 当原程式尝试执行WSASocketW时, 控制流将先由对应的Patcher函数接管, 随后才会回到原函数逻辑.



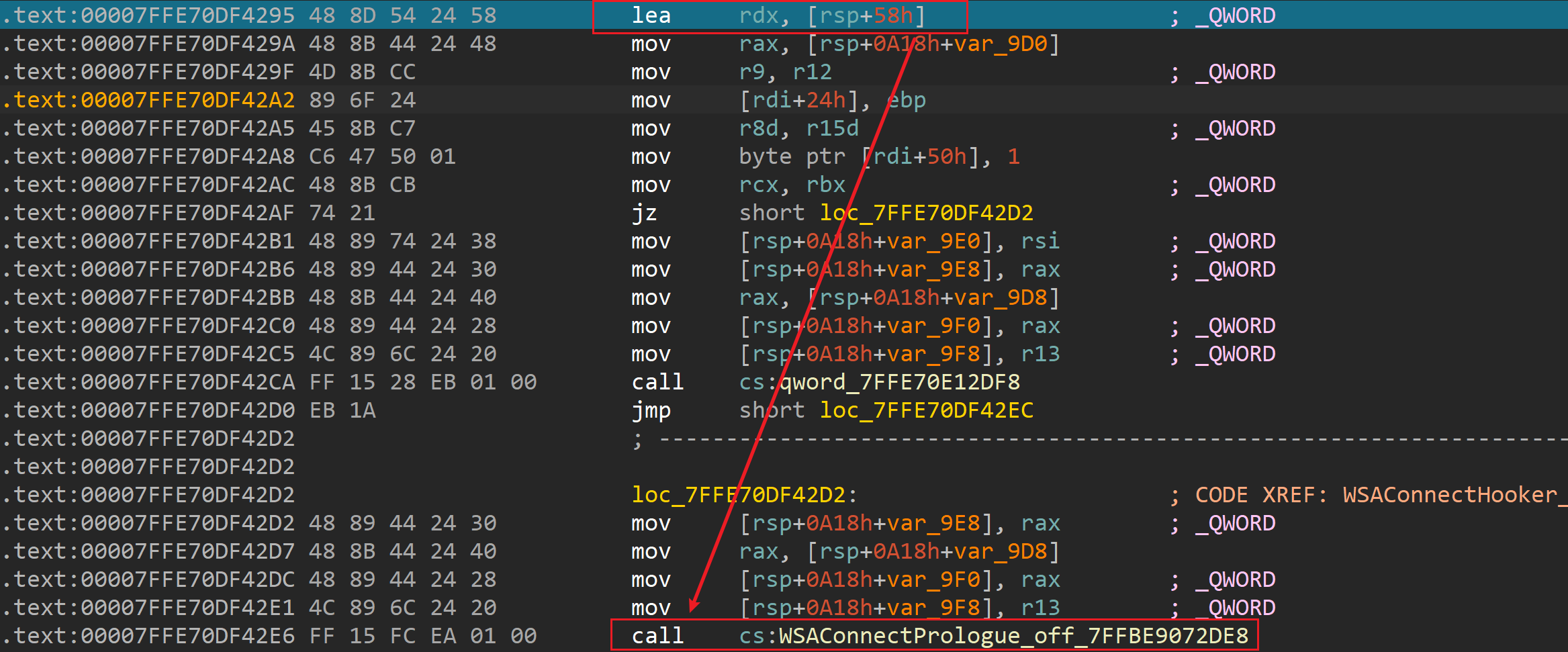
可见目标函数已被劫持. 至此, WS2\_32的完整Hook逻辑结束.

## Patcher接管控制细节

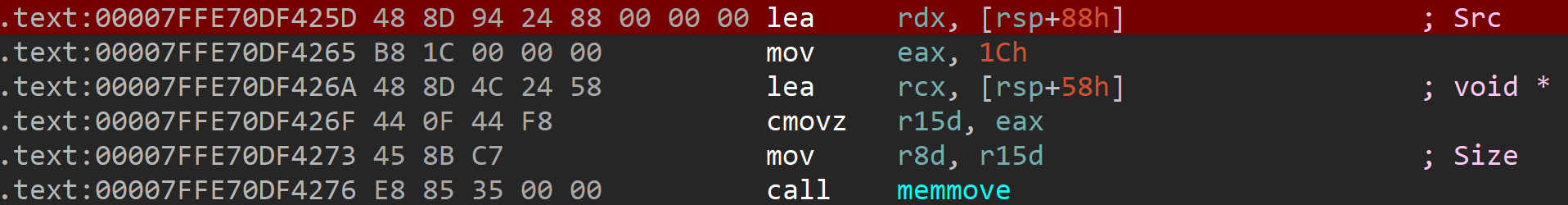


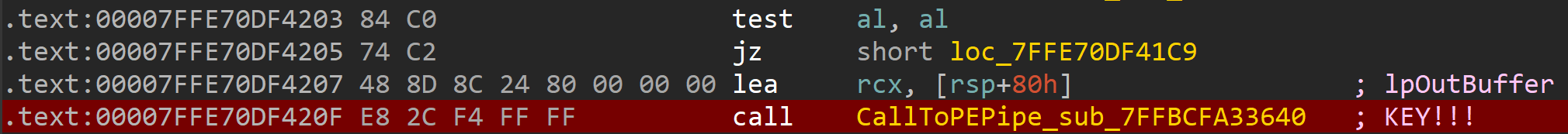


以WS2\_32.dll!connect函数为例, 在执行至其对应的Patcher函数后挂起调试器, 可见作为第二参数的rdx实际上是SOCKADDR\_IN结构, 跳跃到rdx所指内存并转换其内存结构, 其sin\_family被指定为AF\_INET, 其后跟随了软件需要连接的目的IP与端口. 当前SOCKADDR\_IN内存放的IP是”130.185.232.126”.

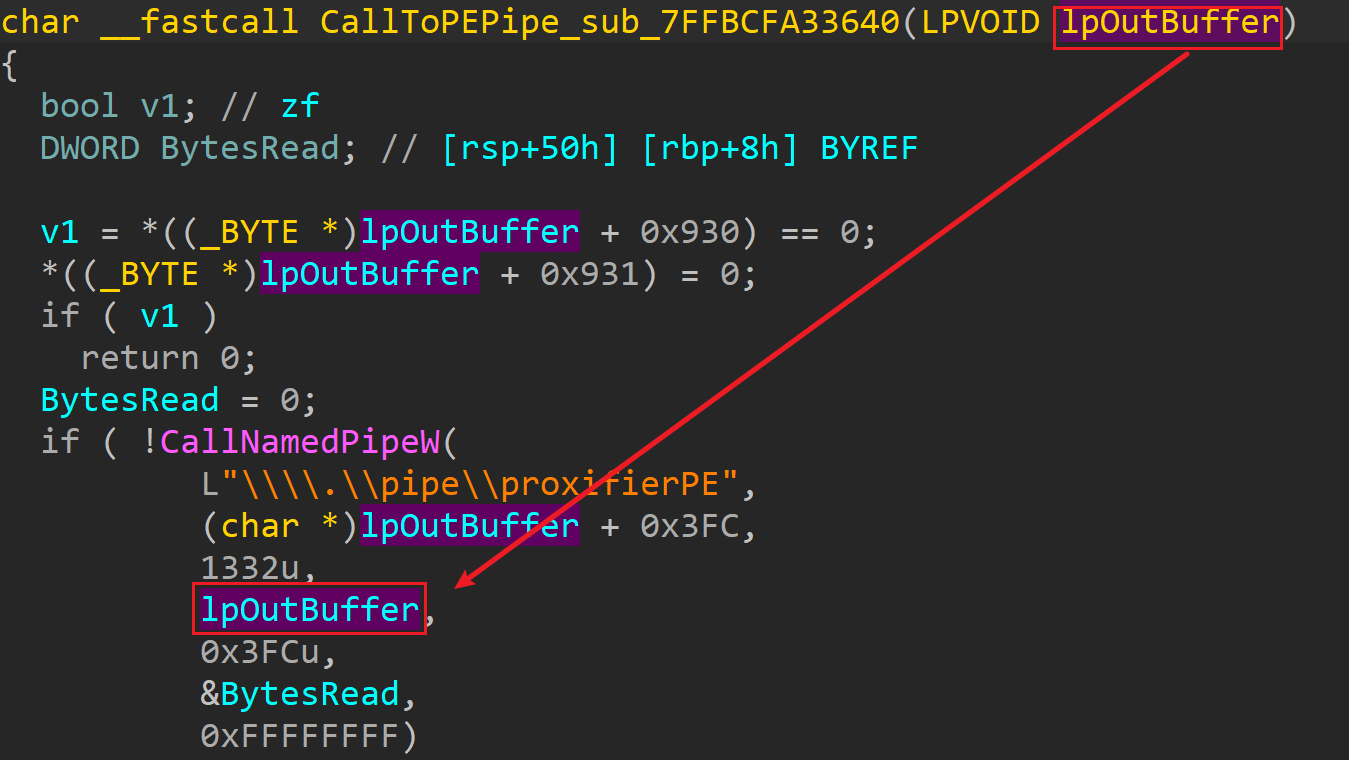


由于Proxifier的主要功能是流量中转, 那么在Patcher最终将控制流交还给WS2\_32前, 一定会控制/修改最终SOCKADDR\_IN结构的内容. 前文在分析WS2的Hook细节时提到, 函数StorePrologueToSrc会将欲Hook的目标函数序言首部的部分指令进行转储, 随后再由PatchTargetFunc函数向转储的序言指令后插入控制流劫持模板, 以让控制流回到WS2对应函数的其余指令, 那么显然, Patcher函数一定会在所有处理工作结束后, 设法进入转储函数对应序言指令的空间内.





当找到Patcher调用序言的指令后, 可以发现rdx由栈空间[rsp+0x58]控制, 而该处又通过memmove接口由[rsp+0x88]控制. 虽然未发现0x88处的直接引用, 但指令上文中的某处, 将相对于0x88处的低位地址[rsp+0x80]作为首个参数传入了另一个函数, 由于此函数与命名管道有关, 命名之为CallToPEPipe.



观察CallToPEPipe函数, 代码通过CallNamedPipeW向特定管道发送数据, 并同时接收由管道回传的数据. 经验证, 名为proxifierPE的管道则由Proxifier程式本身创建, 本档暂时不关心被Hook的程式向管道发送了何种数据(其大概率是一些配置信息, 这对了解控制细节不太重要), 但重要的是, 前文中作为第一参数的栈内指针rsp+0x80则被用于接收管道的回传数据, 同时已知栈内数据[rsp+0x88]用以控制最终的SOCKADDR\_IN结构, 那么回传数据完全有可能覆盖/控制rsp+0x88处.



通过IONinja对管道进行监控, 可以看见proxifierPE互传的数据中, 第8个字节开始的数据恰好对应SOCKADDR\_IN的数据结构, 其目的IP被修改为”127.0.0.1”, 且正好可以覆盖至rsp+0x88的所在空间, 这符合预期, 且说明Proxifier正在将软件的真正通讯地址引导至本地代理中.

至于本档为何可以使用商业软件IONinja, 这便是另外一个需求, 在其它章节内, 本档将记录如何破解5.4版本的IONinja, 并顺带分析其工作原理.

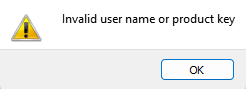
# IONinja原理分析

本章的需求源于前文对Proxifier创建的命名管道的监控与分析需求, 经过调研, 发现闭源商业软件IONinja恰巧可以承担此任务, 其PipeMonitor模块可以实时监控系统创建的管道, 并捕获管道内的通讯内容. 故此, 本章的任务是分析IONinja的功能原理.

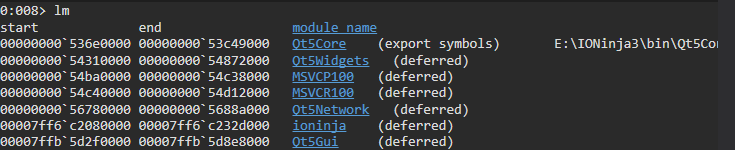
不过需要注意的是, 开源项目cyberark/PipeViewer同样可用于监控命名管道, 然而, 考虑到其功能相较于IONinja更为有限, 且无法捕获管道内的通讯内容, 因此本档仅对其简单记录, 而不进行深入分析. 20250124

## 3.14 AMD程式破解

破解思路仅供参考, 部分信息可能有错误, 请自行判断.



若要使用ioninja的正常功能, 其中一种方案就是通过注册码获得使用授权, 正常情况下错误注册码会提示”Invalid user name or product key”. 换言之, 负责使用/访问这串字符串的指令正处于注册失败的逻辑内.



Windbg运行ioninja.exe, 待其将页面完全加载后, 在调试器内挂起进程. 使用lm命令列出所有已加载的模块, 可以看见ioninja加载的内存范围是00007ff6`c2080000 ~ 00007ff6`c232d000, 需要在此范围内搜索目标字符串.



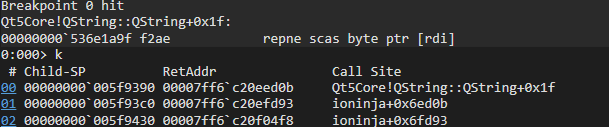
s -a 00007ff6`c2080000 00007ff6`c232d000 "Invalid user name or product key"

通过s -a命令搜索指定范围内的字符串, 可以发现内存中有两处地址存放了目标字符串, 分别是00007ff6`c2216728与00007ff6`c221ac92. 随后, 通过ba指令分别在两处地址设置读写硬件断点, 具体指令如下:

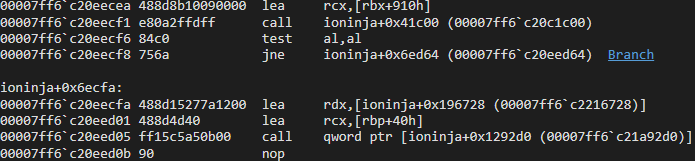
ba r4 00007ff6`c2216728

ba r4 00007ff6`c221ac92

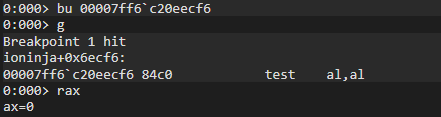
此命令可以控制Windbg在目标地址处设置4个字节的读写硬件断点. 随后使用g命令继续运行.



最终, c2216728处的字符串触发了断点, 并于00000000`536e1a9f处挂起调试器, 通过k命令查看调用栈可以轻易得知, 此操作是QT5的GUI库正在封装目标字符串, 而在其返回地址00007ff6`c20eed0b处正是IONinja本身的代码逻辑.



uf命令观察c20eed0b处所在函数的全部指令, 可见控制流进入该返回地址处是由于c20eecf8处的jne指令判断ZF标志位为1导致的, 而其上一条指令test则正是用于控制ZF标志位的, 且指令操作数为rax, 那么显然可以得知, 00007ff6`c20c1c00地址处的指令用以判断注册码是否合法, 但此处无需关系这部分逻辑.



通过bu 00007ff6`c20eecf6命令在test指令处设置断点, 随后再次在IONinja内尝试点按注册, 命中断点的同时查看rax寄存器, 可见其值为0, 由于指令执行结果为0, 这将导致ZF标志位被设置为1, 随后进入注册失败的分支. 故此, 使用r rax=1命令修改rax寄存器为1, 接着使用g命令放行.



可见绕过注册失败的逻辑后, IONinja直接显示注册完成, 说明思路正确. 随后在IDA内修改此处的JNE指令为JMP, 直接对判断进行绕过, 破解完毕.

## 5.4 x86程式破解

详情待补充 20250411

# TODO List

[完全分析CS4.1 Beacon](file:///D:\TempWorkSpace\WindowSoftwareAnalyse\WinDbgNote.docx#CS41Beacon)

* Beacon加载与初始化流程 √ 20241210
* Shellcode解密与运行流程 √ 20241210
* Shellcode详细通讯流程 TODO
  + 流量中转需求 -> 分析Proxifier √
* CS4.1 后渗透子模块 TODO

[完全分析Proxifier](file:///D:\TempWorkSpace\WindowSoftwareAnalyse\WinDbgNote.docx#Proxifier342)

* 程式注册破解 √ 20241213
* 控制流劫持/Hook WS2细节分析 √ 20241220
* Patcher接管/SOCKADDR劫持细节分析 √ 20241228
  + 命名管道监控需求 -> 分析IONinja

IONinja 原理分析

* 3.14 程式注册破解 √ 20241230
* 5.4 程式注册破解 √ 20250111
  + 信号槽逆向 -> 学习Qt5框架 √ 20250115
  + 编写KeyGen -> 使用Qt5开发 √ 20250122
* 分析Jancy解释器结构 √
* IONinja 驱动逆向 WIP 20250201
  + 简易 PE Hook/Patcher 框架开发 √ 20250211
  + WDK驱动开发 TODO

编译原理

* GCC0.9分析 TODO 20250228
* 简易C语言编译器开发 TODO

Windows编程 (长期进行)

* Qt5 GUI开发
* MASM 汇编开发 WIP 20250212
  + x86 Shellcode开发√ 20250411
  + x64 Shellcode开发√ 20250411

分析反病毒程式 TODO

分析扫描类程式 TODO

// Note: 由于三月份一整月在AOSP安全研究上投入时间, 并且在准备Blackhat议题材料, 因此三月是空档期.