**简易PE Hooker开发 进度记录与控制**

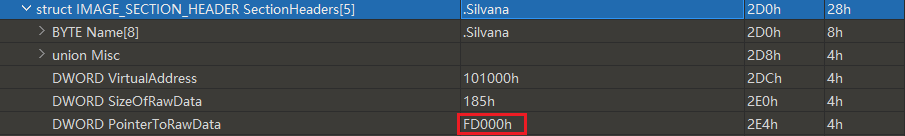
**TODO List**

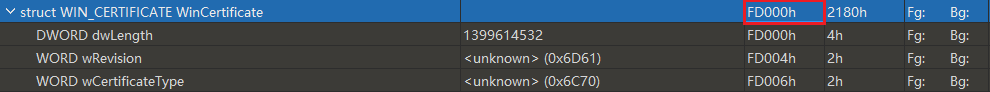
1. 预备Hook代码的机器指令
   1. MASM汇编指令开发
2. 劫持PE文件控制流到Hook代码
   1. 解析PE文件结构
   2. 添加新段并为指令适配空间

注脚: 下文中出现的所有术语”段”均指代Section而非Segment, 这种解释处于个人偏好而非误用.

**PE修改方案选择**

正常情况下, 如果需要劫持PE文件的控制流, 最简单的方法就是修改PE文件的导入表, 通过DLL加载需要的代码, 但为了项目后续的可拓展性, 本档选用另一种方案, 即首先预热Hook代码为机器指令, 随后将指令直接插入PE文件内, 修补IMAGE\_OPTIONAL\_HEADER.AddressOfEntryPoint以在较早的时机劫持控制流.





使用该方案时, 需要解决的问题也随之产生, 首先是如何存放Hook指令? 本档选择效仿工具LordPE的处理手法, 为PE文件添加新段, 并在段内存放所有必要数据, 但新问题随之而来, 在何处存放新段? LordPE选择将PE文件内的整个WIN\_CERTIFICATE结构作为新段以存储数据, 这么做的好处首先是, 覆盖该结构除了会让文件丢失数字签名, 大概率不会影响原PE文件的代码运行逻辑, 其次, 数字签名的体积足够大, 可以轻易存储大部分所需数据与Hook机器指令.

但考虑到PEHooker项目需要尽可能地处理大部分PE文件, 同时, 并非所有PE文件均携带数字签名, 且覆盖原文件仍有可能导致不可预料的错误, 因此, 考虑到兼容性因素, 本档选择将新段添加在PE文件的末尾, 以此保证原文件的大部分内容不发生改变, 且如此做的好处是, 相比于覆盖WIN\_CERTIFICATE结构, 在末尾插入数据并不受体积因素限制, 这反而增加了此方案的可拓展性. 插入新段的具体操作参考了此文: <https://www.sunshine2k.de/reversing/tuts/tut_addsec.htm>

**PEHandler结构**

需要提前记录的信息

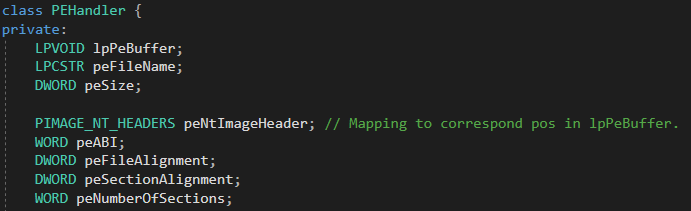
Magic PE魔数, 方便后续区分X86与X64的处理方案

Section/FileAlignment 插入新段时进行内存对齐

PEBuffer PE文件内容

PESize PE文件大小

更多细节详见代码片段, 不再赘述.



**MASM汇编指令开发**

原方案是希望在CPP层使用内联汇编, 或者直接编译完整方法为指令后, 动态地进行提取, 但考虑到编译器可能优化掉方法的某些指令细节, 且无法对指令进行精细控制, 这不利于为PE文件定制事实可行的注入指令, 因此为了兼顾后续的可拓展性与稳定性, 最终选定使用MASM手动编写指令. 以下是开发过程中的部分需求.

**需求列表**

**完全的同段操作**

由于最终编译的机器指令需要注入至PE文件的单个新段内, 那么在预期情况下, 指令的所有功能必须全部在同一个段内实现, 不允许出现跨段的访问, 这是因为跨段操作可能会导致部分指令的语义发生变动, 例如编译器在跨段情况下可能将JMP/CALL NEAR更替为JMP/CALL FAR, 或者使某些PC相对寻址操作更替为绝对寻址, 一旦在其它PE文件内执行这些指令, 就有可能因为这些变动而导致程式产生未定义行为.

**借助PEB结构访问DLL**

PE内的指令访问或调用外部DLL的函数, 一般是通过Linker在编译期间将IAT进行填充并对相关的指令进行关联来实现的, 但注入的新指令无论是编译过程还是其对外部函数的访问过程均与被注入PE文件无关, 本需求需要在被注入PE内自由地执行Hook代码, 根据拟定的方案, 就要借助外部DLL来实现. 因此, 注入的指令要绕开IAT, 借助PEB来寻找需要进入的外部函数所在地址.

**字符串对比**

基础需求, 预计用于PEB索引目标DLL

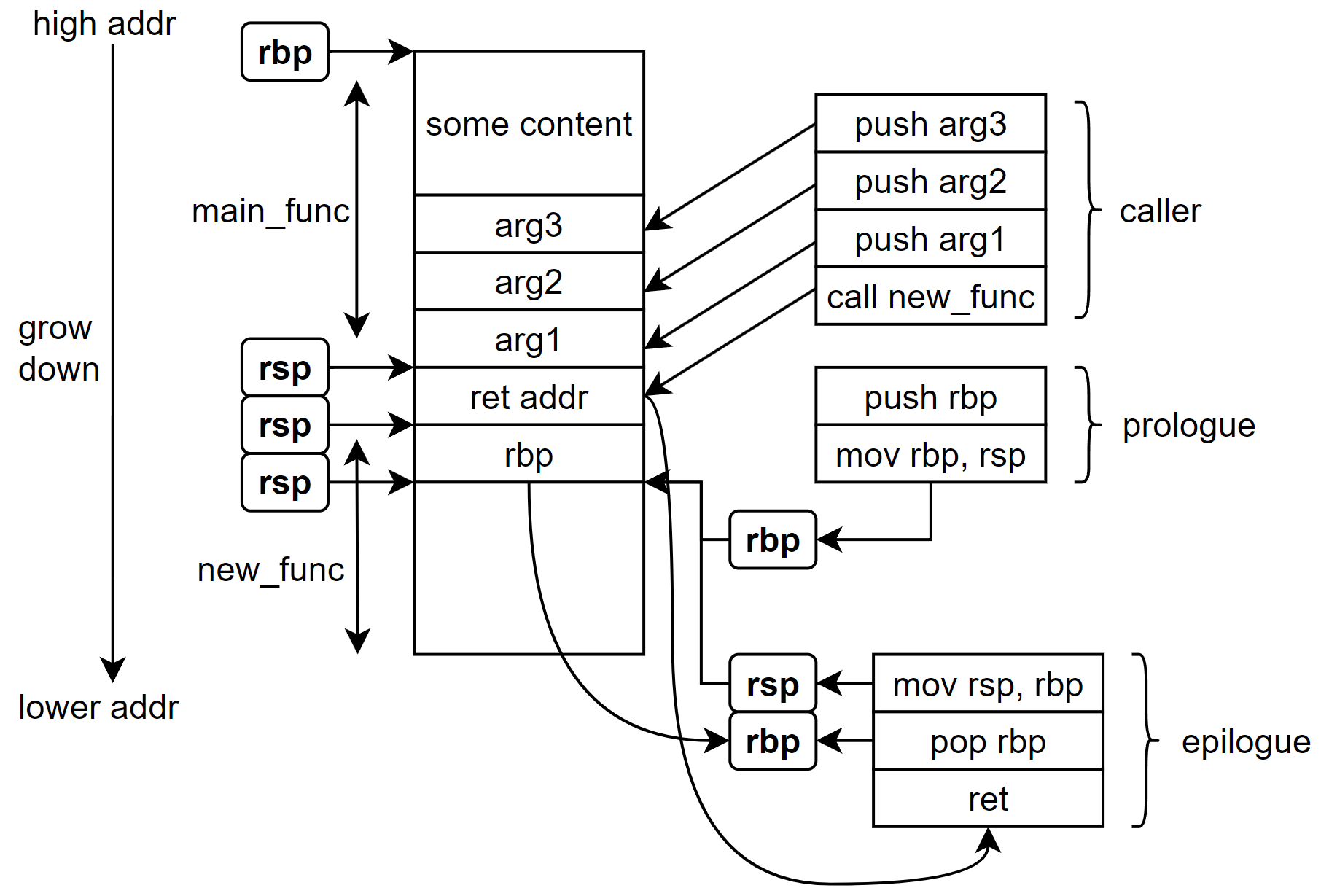
**MSVC加速开发**

微软构建工具链中, cl.exe用于编译C/C++源码, 可借助/FA参数生成汇编代码, 携带/c参数取消链接过程. 其它命令参数详见: <https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options-listed-by-category?view=msvc-170>

**需求实现**

**采取STDCALL调用约定**

所有参数从右向左推入栈内, 整个调用流程的栈结构如下所示.



**字符串对比流程**

函数名strcmp, 入参str1, str2, len

1. 对比i与len, 如果CF=0则跳至eplio
2. 对比[str1 + i]与[str2 + i], ZF=0则跳至not\_eq.
3. 自增i, 跳回序言附近对比处.

补充信息, EFLAGS: <https://www.cs.uaf.edu/2010/fall/cs301/lecture/09_22_controlflow.html>

x86版本由于寄存器较少, 需要额外使用ecx与ebx寄存器, 可能破坏原有控制流状态, 因此对栈空间做了一些额外处理

**PEB结构访问**

函数名walkPEB, 入参dllname, len

通过段寄存器访问TEB->PEB

WikiPedia对Windows下的段寄存器访问内容进行了详细的描述, 详见:

<https://en.wikipedia.org/wiki/Win32_Thread_Information_Block>

以x86为例, 通过段偏移fs:[0x18]访问TEB结构, 通过以下指令访问结构体\_LDR\_DATA\_TABLE\_ENTRY

; access TEB, x86 access by fs

mov eax, fs:[18h] ; \_TEB

mov eax, [eax + 30h] ; \_PEB

mov eax, [eax + 0ch] ; \_PEB\_LDR\_DATA

mov ebx, [eax + 0ch] ; InLoadOrderModuleList

为了访问指定DLL, 需要在结构体内访问BaseDllName(+0x02c), 字符串以结构UNICODE\_STRING进行存储, 需要对结构内的Buffer进行额外解析, 解析过程可能不会被其它函数复用, 直接内联在walkPEB函数内.

**EAT结构访问**

函数名walkEAT, 入参dllbase, funcname

返回funcaddress

实现步骤

1. IMAGE\_DOS\_HEADER->e\_lfanew 偏移0x3c获取IMAGE\_NT\_HEADERS的RVA, x86与x64偏移一致
2. IMAGE\_NT\_HEADERS->OptionalHeader偏移0x18直接访问IMAGE\_OPTIONAL\_HEADER, x86与x64偏移一致
3. IMAGE\_OPTIONAL\_HEADER->DataDirectory
   1. x86偏移0x60直接访问IMAGE\_DATA\_DIRECTORY[]
   2. x64偏移0x70直接访问IMAGE\_DATA\_DIRECTORY[]
   3. IMAGE\_DATA\_DIRECTORY单项体积8字节
4. IMAGE\_DATA\_DIRECTORY->VirtualAddress获取RVA, 此处RVA一般需要转化为FOA.
5. EAT访问细节详见调试记录 https://github.com/MG1937/InformalNotes/blob/main/DebugTrick/WinDBG/export\_table.md