映像加载程序

加载程序负责的一部分主要任务如下：

1. 为应用程序初始化用户模式状态，例如创建初始堆、设置线程本地存储 (TLS)和纤程本地存储(FLS)槽。
2. 解析应用程序的导入地址表 (Import Address Table, IAT)，查找所需的全部DLL(随后递归地解析每个DLL的IAT)，接着解析 DLL的导出表，以确保需要调用的外部函数函数确实存在。[可通过特殊的转发器项(Forwarder entry)将导出表重定向至另一个DLL。]
3. 在运行时或者需要时加载和卸载 DLL，维护包含所有已加载模块的列表(模块数据库)。
4. 处理Windows并行(Side-by-Side，SxS)支持所需的清单文件，以及多语言用户界面(Multiple Language User，MUI)文件和资源。
5. 读取应用程序兼容性数据库中的填充码，并在需要时加载填充码引擎DLL。

填充码引擎 DLL（Padding Engine DLL）通常是一种反破解技术，它可以在二进制文件中插入大量的无用代码，使逆向分析者在分析代码时难以识别有用的代码。通常情况下，填充码引擎 DLL 会先对可执行文件进行加密，然后在程序运行时通过填充码引擎 DLL 进行动态解密，最终将可执行文件恢复到原始状态。

具体实现方式可以分为以下几个步骤：

1. 加密可执行文件：使用加密算法对可执行文件进行加密，可以使用对称加密算法如 AES、RC4、DES 等等。
2. 插入填充代码：插入大量无用的代码来干扰分析者，填充代码可以通过编写一些无用的函数、跳转指令和无用的指令来实现。
3. 插入反调试代码：为了防止调试，可以在填充代码中插入反调试的代码，比如检查 PEB 结构体中的 BeingDebugged 字段是否为 1，或者通过特定的指令检查当前代码是否在调试器中运行。
4. 动态解密：在程序运行时，填充码引擎 DLL 可以解密可执行文件，将其恢复到原始状态，然后执行原始代码。

填充码引擎 DLL 可以防止静态分析和反编译，并提高对抗反调试的能力，常被用于保护程序的知识产权和安全性。但是，填充码引擎 DLL 也有一些缺点，如可能会降低程序的性能，并且可能被一些高级的恶意软件绕过。

1. 启用对API集和API重定向的支持，这也是One Core 功能的核心部件，可用创建通用Windows平台(Universal Windows Platform，UWP)应用程序。
2. 通过 SwitchBack 机制启用动态运行时兼容性缓解措施，并与填充码引擎和应用程序验证器机制交互

如果不执行这些任务，从调用外部函数到使用堆，所有工作都会立即失败。当进程成功创建后，加载程序会调用NtContinuespecial这个原生API，像异常句柄那样根据栈中的异常帧(exception frame)继续执行。这个异常帧由内核创建，并且如前所述，其中包含了应用程序真正的入口点。

进程初始化的早期工作

由于加载程序位于 Ntdll.dll 中，而Ntdll.dll是一个不属于任何特定子系统的原生 DLL，因此所有进程都遵循相同的加载程序行为(但也有些细微差别)。

当进程启动时，加载程序将执行下列步骤。

1. 检查 LdrpProcessInitialized 是否已设置为1，或TEB 中是否已设置了 SkipLoaderInit标志。对于后一种情况，会跳过所有初始化任务，并等待 3s，由其他东西调用 LdrpProcessnitializationComplete。这种做法常见于 Windows 错误报告使用进程反射(process reflection)时，或其他进程企图创建分叉 (fork)进而不需要初始化加载程序时。
2. 将 LdrInitState 设置为 0，这意味着加载程序尚未初始化。此外会将 PEB 的rocessInitializing 标志设置为1，并将TEB的 RanProcessInit 设置为1。
3. 初始化动态函数表，JIT代码可使用该表获得展开(unwind)/异常(exception)支持。

JIT（Just-In-Time）代码是一种在程序运行时动态编译的代码。与预编译代码不同，JIT代码是在程序运行时根据需要动态生成的。JIT编译器在运行时将代码编译成本地机器代码，然后再执行该代码。

1. 在PEB中初始化加载程序锁。
2. 初始化可变只读堆区域 (Mutable Read Only Heap Section，MRDATA)，该区域可用于存储与安全性有关、不应被漏洞利用所修改的全局变量。
3. 在PEB中初始化加载程序数据库。
4. 为进程初始化国家语言支持表 (National Language Support, NLS)表 (借此实现际化支持)。
5. 为应用程序构建映像路径名称。
6. 从.pdata 区域获取SHE异常句并构建内部异常表。
7. 为5个关键的加载程函数获取系统调用形式转换 NtCreateSection、NtOpenFile。NtOueryAttributesFile、NtOpenSection和NtMapViewOfSection
8. 读取应用程序的缓解选项(由内核通过 LdrSystemDllnitBlock 这个导出变量传入)
9. 查询应用程序的映像文件执行选项(IFEO)注册表键。其中包含了诸如全局标志(在储于GlobalFlags 中)、堆调试选项 (DisableHeapLookaside、 ShutdownFlags以及FrontEndHeapDebugOptions)、加载程序设置 (UnloadEventTraceDepth、 MaxLoaderThreads、UselmpersonatedDeviceMap)、ETW 设置(TracingFlags)等选项。其他选项则包括 MinimumStackCommitInBytes和 MaxDeadActivationContexts。在这部分工作中，应用程序验证器包和相关的验证器 DLL将被初始化，并从CFGOptions 读取控制流防护(Control Flow Guard, CFG)选项。
10. 在可执行文件的文件头中检查这是否是 .NET 应用程序(取决于是否存在与 .NET 相关的映像目录)，以及是否是 32 位映像。此外，还会查询内核以验证这是否为 Wow64进程。如果有必要，将处理一个32位的仅 IL 映像，而不需要 Wow64。
11. 加载可执行文件的映像加载配置目录中指定的任何配置选项。开发者可以在编译应用程序时定义这些选项,编译器和链接器也可以使用这些选项实现某些安全和缓解功能，例如CFG，借此控制可执行文件的行为。
12. 对FLS和TLS进行最低限度的初始化。
13. 为关键区域设置调试选项，如果启用相应的全局标志，还将创建用户模式栈跟踪数据库，并从映像文件执行选项中查询StrackTraceDatabaseSizeInMb。
14. 为进程初始化堆管理器，并创建首个进程堆。这一过程会使用多种加载配置、映像文件执行、全局标志以及可执行文件头选项来设置必要的参数。
15. 如果启用，则会在堆损坏缓解措施中启用终止进程。
16. 如果启用相应的全局标志，则会初始化异常分发日志。
17. 初始化支持线程池API 所需的线程池包。为此需要查询并处理 NUMA 信息
18. 初始化并转换环境块和参数块，需要着重强调的是这是支持 Wow64 进程所必需的。
19. 打开\KnownDlls 对象目录并构建已知 DLL的路径。对于 Wow64 进程，则会用\KnownDlls32。
20. 对于商店应用，将读取应用程序模型策略选项，这些信息已经被编码至令牌WIN://PKG和WP://SKUID声明中。
21. 判断进程的当前目录、系统路及默认加路径(用于加映像和打开文件)以及有关默认DLL 搜索顺序的规则。此时还需要为通用 Windows 平台(UWP)、Desktop Bridge(Centennial)以及 Silverlight (Windows Phone 8)读取打包应用程序(或服务)的当前策略设置。
22. 为Ntdll.dll构建第一个加载程序数据表项，并将其插入模块数据库。
23. 构建展开 (unwind)历史表。
24. 初始化并行加载程序，它将使用线程池和并发线程加载所有(不具备交叉依赖的)依赖项。
25. 为主可执行文件构建下一个加载程序数据表项，并将其插入模块数据库
26. 如果需要，将重新定位主可执行文件映像。
27. 如果启用，将初始化应用程序验证器。
28. 对于 Wow64进程，将初始化 Wow64 引擎。此时64位加载程序将完成自己的初始化工作，并由 32位加载程序接手控制，重新启动从此处向前提到的大部分操作。
29. 对于.NET映像，将进行验证并加载Mscoree.dll (.NET 运行时填充码)，同时会获取主可执行文件入口点(\_CorExeMain)，覆盖异常记录，将它设置为入口点并取代常规的主函数。
30. 为进程初始化 TLS 槽。
31. 对于 Windows 子系统应用程序，无论进程的实际导入是什么，都将手动加载Kernel32.dIl和Kernelbase.dll。如果需要，还将使用这些库初始化SRP/Safer (软件限制策略)机制，并捕获 Windows 子系统线程初始化形式转换 (thunk)函数。最后，将解析这两个库中明确存在的任何 API集依赖项。
32. 初始化填充码引擎并解析填充码数据库。
33. 只要核心加载程序函数之前的扫描未发现任何系统调用挂钩或附加了“便道”将根据通过策略和映像文件执行选项配置的加载程序线程数量启用并行映像加载程序
34. 将 LdrInitState 变量设置为1，意味着“正在进行导入加载”。

至此，映像加载程序已经准备好开始解析应用程序可执行文件的导入表，并开始加载应用程序编译过程中动态链接的任何 DLL。无论是 .NET 映像(通过调用 .NET 运行时来处理自己的导入)还是普通映像，都将进行该操作。导入的每个 DLL 可能还有自己的导入表，因此该操作会持续递归，直到所有 DLL 均已成功导入并且所有要导入的函数均已找到。随着加载每个 DLL，加载程序会维持其状态信息并构建模块数据库。

在较新版本的Windows中，加载程序会事先构建依赖项映射图，并通过具体的节点来描述每一个 DLL 及其依赖性关系图，构建可并行加载的单独节点。如果在任何时候需要进行序列化，线程池工作队列将被“排空”进而充当同步点。其中有一个同步点会先于调用所有静态导入的所有 DLL初始化例程存在，这也是加载程序需要执行的最后一个步骤。该步骤完成后，将调用所有静态 TLS的初始化。最终，对于Windows应用程序，在这两个步骤之间还要在最开始调用Kernel32线程初始化形式转换函数(BaseThreadInitThunk)并在最终结束前调用 Kernel32 进程初始化后的例程。

DLL名称解析和重定向

名称解析的处理过程为: 当调用方未指定或无法指定唯一文件标识时，系统会将 PE格式二进制文件的名称转换为物理文件。这样做是因为多个目录(如应用程序目录、系统目录等)的位置无法在链接时以硬编码的方式确定，此外这一过程还包括解析所有二进制依赖项，以及在调用方未指定完整路径时执行的 LoadLibrary 操作。

在解析二进制依赖项时，最基本的windows应用程序模型会通过搜索路径查找文件。如果将使用相同基本名称的恶意二进制文件放置在用程序的当前目录中，会使系统二进制文件的加载操作被覆盖，这种技术通常也被叫作二进制植入 (binary planting)。为了避免这种行为可能导致的安全风险，路径搜索计算过中加入了一个名为安全DLL 搜索模式的功能，该功能默认会对所有进程启用。在安全搜索模式下，当前目录会被移动到3个系统目录之后，并产生如下的路径顺序:

1. 启动应用程序时的来源目录;
2. 原生Windows系统目录(如C:\Windows\System32);
3. 16位Windows 系统目录(如C:\Windows\System)
4. Windows目录(如C:\Windows);
5. 应用程序启动时的当前目录;
6. %PATH%环境变量指定的任何目录

后续的每个 DLL 加载操作都会重新计算 DLL 搜索路径。计算该路径所用的算法与计算默认搜索路径所用的算法相同，但应用程序可使用 SetEnvironmentVariable API更改%PATH%变量，进而改变特定的路径元素。使用SetCurrentDirectory API更改当前目录或使用 SetDlIDirectory API为进程指定 DLL目录。如果指定了DLL目录，指定的这个录将取代搜索路径中的当前目录，加载程序会忽略进程的安全 DLL 搜索模式设置。

如果应用程序是打包的应用程序、未打包的服务，或遗留的 Silverlight 8.0 Windows Phone应用程序，搜索路径顺序可能会受到影响。对于这些情况，DLL 搜索顺序将不再使用传统的机制或API，而是限制为基于软件包进行图(graph)搜索。如果使用LoadPackagedLibrary API而非普通的 LoadLibraryEx 函数，也会出现类似情况。基于软件包的图是根据UWP应程序清单文件中<Dependencies>下的<PackageDependency>项计算而来的，这样可以保证不会有任何 DLL 被无意载入软件包中。

然而尽管有安全搜索模式和面向遗留应用程序的默认路径搜索算法，通常还是会首先包含应用程序本身的目录，二进制文件依然有可能从自己的常规位置复制到用户可访问的位置(例如从c:\windows\system32\notepad.exe 复制到c:\temp\notepad.exe，这一操作并不需要管理员权限)。这种情况下，攻击者即可将一个特别准备的DLL放入应用程序所在目录，随后由于前文提到的顺序问题，让这个DLL先于系统DLL 被处理。借此即可常驻或影响应用程序，而这个应用程序可能还具备一定的特权(在用户不知道这一变化的情况下,甚至可能通过 UAC 提升自己的权限)。为了防范这种问题，进程或管理员可以使用一个名为Prefer System32 Images (优先使用 System32 映像)的进程缓解策略。顾名思义，该策略可以逆转前文所述的第一个和第二个位置的顺序。

DLL 名称重定向

在尝试将 DLL 名称字符串解析为文件之前，加载程序会试着应用 DLL 名称重定向规则。这些重定向规则可用于扩展或覆盖 DLL 命名空间(通常相当于 Win32 文件系统命名空间)的部分区域，借此对 Windows 应用程序模型进行扩展。按照应用的先后顺序，这些规则如下。

1. MinWin API集重定向。API集这机制的设计目标是让不同版本的 Windows 能够用对应用程序来说透明的方式，通过合约(contract) 这一概念更改可以导出特定系统API的二进制文件。
2. .LOCAL 重定向。.LOCAL 重定向机制可以让应用程序忽略所指定的完整路径,对某一特定 DLL 基名称的所有加载操作重定向至应用程序目录下的 DLL本地副本，具体方法则有两种: 使用相同基本名称后跟“LOCAL”创建DLL副本(如MyLibrary.dll.local);在应用程序目录下创建名为“.LOCAL”的文件夹，随后将 DLL 的本地副本保存在该文件夹中(如 C:\\MyApp\.LOCAL\MyLibrary.dll)。通过.LOCAL 机制重定向的 DLL，其处理方式与 SxS 重定向的 DLL 相同。但只有在可执行文件不包含对应的(嵌入式或外部)清单的情况下，加载程序才会为 DLL 使用.LOCAL重定向。该功能默认并未启用。为了全局启用该功能，可在基本IFEO注册表(HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Image File Execution Options)键下创建名为 DevOverrideEnable 的 DWORD值，并将其设置为1。
3. Fusion(SxS)重定向。Fusion (也叫作并行，即 SxS)是对 Windows 应用程序模型的扩展，可以让组件嵌入名为清单的二进制资源，借此表达更详细的二进制依赖信息(通常为版本信息)。Fusion 机制最初的用途在于让应用程序能够加载正确版本的 Windows公共控件包 (comctl32.dll)，因为这个二进制文件可以被拆分为不同版本，并且多版本共存。此后，其他二进制文件也开始通过类似的方式进行版本管理。到了 Visual Studio 2005版本，使用微软链接器构建的应用程序可以通过 Fusion 机制定位正确版本的C运行时库而 VisualStudio 2015 和后续版本可以使用API集重定向机制实现通用CRT的理念。
4. 已知DLL重定向。已知DLL机制意在将特定 DLL的基本名称映射至系统目录中的文件，从而避免 DLL 被其他位置的不同版本所替换。

DLL路径搜索算法中的一种极端情况是这样的:针对64位Wow64应用程序执行DLL版本。如果找到了基本名称相符的 DLL，但随后发现该文件是针对另一种体系结构的计算机编译的，例如 32 位应用程序中找到了 64 位映像，加载程序会忽略这些错误并恢复路径搜索操作，从导致找到错误文件的路径元素开始继续向下查找。按照设计，这种行为是为了让应用程序能够在全局%PATH%环境变量中同时指定 64 位和 32位项

已加载模块数据库

加载程序会维持进程已加载全部模块(DLL 以及主可执行文件)的列表。这些信息存储在PEB中，主要位于由Ldr 标的一个名为PEB\_LDR\_DATA的子结构中。在该结构中，加载程序维护了3个双向链表，其中均包含了相同信息，但顺序有所差异(按照加载顺序、内存位置或初始化顺序)。这些列表中包含的结构也叫作加载程序数据表项(LDR\_DATA\_TABLE\_ENTRY)，存储了有关每个模块的信息。

另外，由于对链表进行查找在算法上需要巨大开销(以线性时间的方式完成)，加载程序还维持了两个红黑树(red-black tree)，这是一种在二进制查找方面更高效的树。第一个树按照基址排序，第二个树按照模块名的散列排序。通过这些树，搜索算法即可以“对数时间”的方式进行，这种措施大幅提升了 Windows 8 和后续版本的进程创建效率和速度。此外，作为一种安全措施，与链表不同，这两个树的根不能在 PEB 中访问。这确保了它们更难被 Shell代码定位，因为 Shell代码通常是在启用了地址空间布局随机化 (Address Space Layout Randomization，ASLR)的环境中运行的。

地址空间布局随机化（Address Space Layout Randomization，ASLR）是一种安全技术，其目的是防止攻击者预测程序的内存地址空间，从而利用缓冲区溢出漏洞。它通过随机地分配程序的内存地址空间来使攻击者无法预测可利用的漏洞位置。ASLR通过随机地放置程序的堆、栈和其他内存区域，以使攻击者无法预测其位置。这种随机化的方法需要操作系统支持，并且必须在编译时启用。一旦启用，每次程序启动时，其内存地址空间都将被随机分配。

虽然本节介绍了 Ntdll.dll 中用户模式的加载程序，但别忘了，内核也会通过自己的加载程序来处理驱动程序和依赖的 DLL，并使用了一个类似的加载程序项结构，名为KLDR\_DATA\_TABLE\_ENTRY。类似地，内核模式的加载程序也会用自己的数据库保存每个项，该数据库可通过 PsLoadedModuleList 全局数据变量直接访问。

导入解析

在介绍完导入程序对进程中所有已加载模块进行跟踪的方式后，我们还可以继续分析加载程序所执行的启动初始化任务。在这一阶段，加载程序将执行下列工作。

1. 加载进程可执行映像导入表中引用的每个 DLL。
2. 查询模块数据库检查该 DLL 是否已加载。如果在列表中未找到，加载程序会打开该DLL 并将其映射至内存。
3. 在映射操作中，加载程序首先会在多个路径下尝试找到该 DLL，并确定该 DLL是否为已知DLL(已知DLL 是指系统启动时已经加载过的DLL，此时可通过全局内存映射文件访问它)。有些时候的做法可能会与标准查找算法有所差异，例如使用.local 文件(可以迫使加载程序使用本地路径下的 DLL)，或使用清单文件指定重定向的 DLL，借此确保使用特定的版本。
4. 找到磁盘上的 DLL 并成功映射后，加载程序会检查内核是否已将其载入其他地方(即重定位)。如果加载程序检测到重定位，将解析 DLL 中的重定位信息并执行所需操作。如果不存在重定位信息，DLL 会加载失败。
5. 加载程序会为该 DLL 创建加载程序数据表项，并将其插入数据库。
6. DLL 成功映射后，针对该 DLL 重复这个过程，以便解析其导入表和所有依赖项
7. 每个 DLL 均载入后，加载程序将解析 IAT 以查找已经导入的函数。通常这是通过名称来进行的，但也可以按照序号 (索引编号) 进行。对于每个名称，加载程序会解析已导入 DLL 的导出表，并尝试着找到匹配的结果。如果未找到匹配结果，操作将终止。
8. 映像的导入表可能已经被绑定。这意味着在链接时，开发者已经通过分配静态地址指向了外部 DLL 中的导入函数。此时就无须对每个名称进行查找，但这样做的前提是应用程序将要用到的 DLL 始终位于相同地址下。由于 Windows 使用了地址空间随机化，系统应用程序和库通常无法这样做。
9. 已导入 DLL 的导出表还可以使用转发项，这意味着实际函数是在另一个DLL中实现的。但另一个函数必须被视作导入项或依赖项，因此解析完导出表后，转发项引用的每个DLL 也需要加载，所以需要回到第 (1) 步再次执行。

当所有导入的 DLL(及其依赖项或导入项)均已成功加载、所有需要的函数已经查找过并已找到，并且所有转发项也已加载并处理后，导入解析操作就完成了，即应用程序及其各种 DLL 在编译时所定义的依赖性均已满足。在执行过程中，延迟依赖项(也叫作延迟加载)以及运行时操作(例如调用 LoadLibrary)可以调用至加载序中，而本质上它们所做的工作是相同的。不过需要注意，上述过程如果是在应用程序启动进程的过程中进行的，那么所遇到的失败操作会导致应用程序启动错误。

导入过程初始化的后处理

所需依赖项加载完毕后，必须执行一系列初始化任务，最终完成应用程序的启动操作。在这一阶段，加载程序将执行下列工作.

* 1. 将 LdrInitState 变量设置为2，意味着导入项已加载。
  2. 如果使用 WinDbg 等调试器，将抵达初始调试器断点。前文实验中必须输入 g以继续执行，就是因为到达了这个阶段。
  3. 检查应用程序是否为 Windows 子系统应用程序。如果是，那么早期的进程初始化步骤中应该捕获到 BaseThreadInitThunk 函数。此时将调用该函数并检查是否成功。类似的还有TermsrvGetWindowsDirectoryW函数，早期应该已经捕获了该函数(前提是系统支持终端服务)，接着将调用该函数以重置系统和 Windows 目录路径。
  4. 使用分布式关系图递归所有依赖项并对映像的所有静态导入进行初始化。这一步需要调用每个DLL的DIlMain 例程(借此让每个DLL执行自己的初始化工作，甚至在运行时载入新的 DLL)并处理每个 DLL 的TLS初始化。此时也是应用程序加载可能失败的最后一个环节。如果在 DllMain 例程运行完毕后，加载的所有 DLL 没能返回成功返回代码，加载程序会终止应用程序的启动。
  5. 如果映像使用了任何 TLS 槽，则会进行 TLS 的初始化。
  6. 如果模块通过填充码实现兼容性，则运行初始化后填充码引擎回调

1. 运行 PEB 中注册的相关子系统 DLL 后处理初始化例程。例如对于 Windows应用程序，此时还将进行与终端服务相关的检查。
2. 写入一条ETW事件，代表进程已成功加载
3. 如果存在最低限度的栈提交，则强制线程栈进行已提交页面的页面换入操作
4. 将LdrInitState 设置为3，意味着初始化已完成。将 PEB的ProcessInitializing字段重新设置为0，随后更新 LdrpProcessInitialized 变量。

随着每个新版Windows 陆续修复现有 API函数中诸如竞争条件和错误的参数验证检查等 bug，或大或小的变化都可能导致应用程序出现兼容性问题。Windows在加载程序中实现了一种名为 SwitchBack 的技术，该技术可以帮助软件开发者在可执行文件相关联的清单中嵌入目标版本Windows 对应的GUID。

API集

在某些特定的应用程序兼容性场景中，SwitchBack 用到了 API 重定向，而实际上Windows还会为所有应用程序使用一种更普遍的、名为“API集”(API set)的重定向机制。API集的目的在于通过更细化的分类，将 Windows API 分为多种子 DLL，而无须跨越近千种目前以及未来 Windows 系统可能并不完全需要的API使用大量多用途DLL。该技术的开发原本主要是对 Windows 架构的最底层进行重构，将其与高层分开，进而将Kernel32.dll和Advapi32.dll (以及其他 DLL)拆分为多个虚拟的 DLL 文件。

首先，这样做使得以后的应用程序只需链接至提供了自己所需功能的 API 库。其次，如果微软打算创建一个不支持某些功能的Windows 版本，例如不支持本地化(例如一种并非面向用户的、仅支持英文的嵌入式系统)，只需简单地移除相应的子 DLL 并修改 API 集的架构即可实现。这样可以获得更小的 Kernel32 库，而任何不需要本地化的应用程序就可以继续运行。

借助这样的技术，我们就定义出了一种名为MinWin的“基准”Windows 系统，并(在源代码层面可以)构建出最小化的服务集，其中可以只包含内核、核心驱动程序(包括文件系统、诸如 CSRSS 等基本系统进程、服务控制管理器，以及少量 Windows 服务)。Windows Embedded 及其 Platform Builder 就提供了类似这样的技术。借助 System Builder，我们可以移除不需要的“Windows 组件”，例如Shell或网络 。然而从 Windows中移除某些组件会导致棘手的依赖性问题，某些代码可能由于所依赖的组件被移除而执行失败，不过MinWin的依赖性是完全自包含的。

进程管理器初始化后会调用 PspInitializeApiSetMap 函数，该函数负责用%SystemRoot%\System32\ApiSetSchema.dll 中存储的API集重定向表创建节对象。该DLL 不包含可执行代码，但其中有一个名为.apiset 的节中包含的API集映射数据可将虚拟API集DLL映射至实现该API所需的逻辑 DLL。当新进程启动后，进程管理器便会将该节对象映射至进程的地址空间，并设置进程PEB的ApiSetMap 字段，使其指向节对象所映射到的基址。

随后，如果(以动态或静态方式)加载的新导入库的名称以“API-”开头，如前所述,通常负责处理local和SxS/Fusion 清单重定向的加载器 LdrpApplyFileNameRedirection函数还会检查API集重定向数据。API集表会按照库进行整理，每个入口均描述了可以在哪个逻辑 DLL 中找到相关函数，随后会加载这些 DLL。虽然架构数据是二进制格式，但我们可以使用Sysinternals Strings 工具转储其字符串并查看目前定义的 DLL。