Windows利用技巧：利用任意文件写进行本地提权

Edvison / 2018-08-30 14:06:01 / 浏览数 3345 技术文章 技术文章 顶(0) 踩(0)

本文是Windows Exploitation Tricks: Exploiting Arbitrary File Writes for Local Elevation of Privilege的翻译文章

前言

之前我提出了一种技术，可以在Windows上利用任意目录创建漏洞，从而对系统上的任意文件进行读取访问。 在即将发布的Spring Creators Update（RS4）中，我在前一篇文章中利用的挂载点链接到文件的bug已得到修复。
这就是一个如何利用漏洞获得长期安全的一个示例，让开发人员更有动力去找减少利用的方式。

我将在这篇文章中保持这种精神，介绍一种新技术，利用Windows 10上的任意文件写入。也许微软可能再一次强化操作系统，使其更难以利用这些类型的漏洞。我将通过详细描述Project Zero向Microsoft报告的最近修复的问题（问题1428）来证明此利用。

任意文件写入漏洞是用户可以在他们通常无法访问的位置创建或修改文件。
这是由于权限服务错误地清理了用户传递的信息，或者由于符号链接劫持攻击，用户可以将链接写入之后由权限服务使用的位置。
理想的漏洞是攻击者不仅控制正在写入的文件的位置，还控制整个内容。

利用任意文件写入的常用方法是执行DLL劫持。当Windows可执行文件开始执行NTDLL中的初始加载程序时，将尝试查找所有导入的DLL。
加载程序检查导入的DLL的位置比您预期的更复杂，但就我们的目的而言可以总结如下：

1. 检查已知DLL，它是操作系统已知的预缓存DLL列表。 如果找到，则DLL将从预加载的节对象映射到内存中。
2. 检查程序的目录，例如，如果导入TEST.DLL并且程序在C:\APP中，则它将检查C:\APP\TEST.DLL。
3. 检查系统位置，例如C:\WINDOWS\SYSTEM32和C:\WINDOWS。
4. 如果所有其他方法都失败，请搜索当前环境变量。

DLL劫持的目的是找到一个以高权限运行的可执行文件，它将通过漏洞允许我们在写入的位置加载DLL。 如果在先前检查的位置中尚未找到DLL，则仅劫持成功。

有两个问题导致DLL劫持很麻烦：

1. 您通常需要创建特权进程的新实例，因为在首次执行进程时会解析大多数DLL导入。
2. 将作为特权用户运行的大多数系统二进制文件，可执行文件和DLL将安装到SYSTEM32中。

第二个问题意味着在步骤2和3中，加载器将始终在SYSTEM32中查找DLL。假设覆盖DLL不太可能是一个选项（至少如果DLL已经加载，你不能写入文件），这使得找到合适

即使你发现一个合适的目标可执行文件DLL劫持，实现也可能非常难看。有时你需要为原始DLL实现存根导出，否则DLL的加载将失败。在其他情况下，运行代码的最佳位置
什么是好的是一个权限服务，它将为我们加载一个任意的DLL，没有劫持，不需要产生"正确的"特权进程。 问题是，这样的服务是否存在？

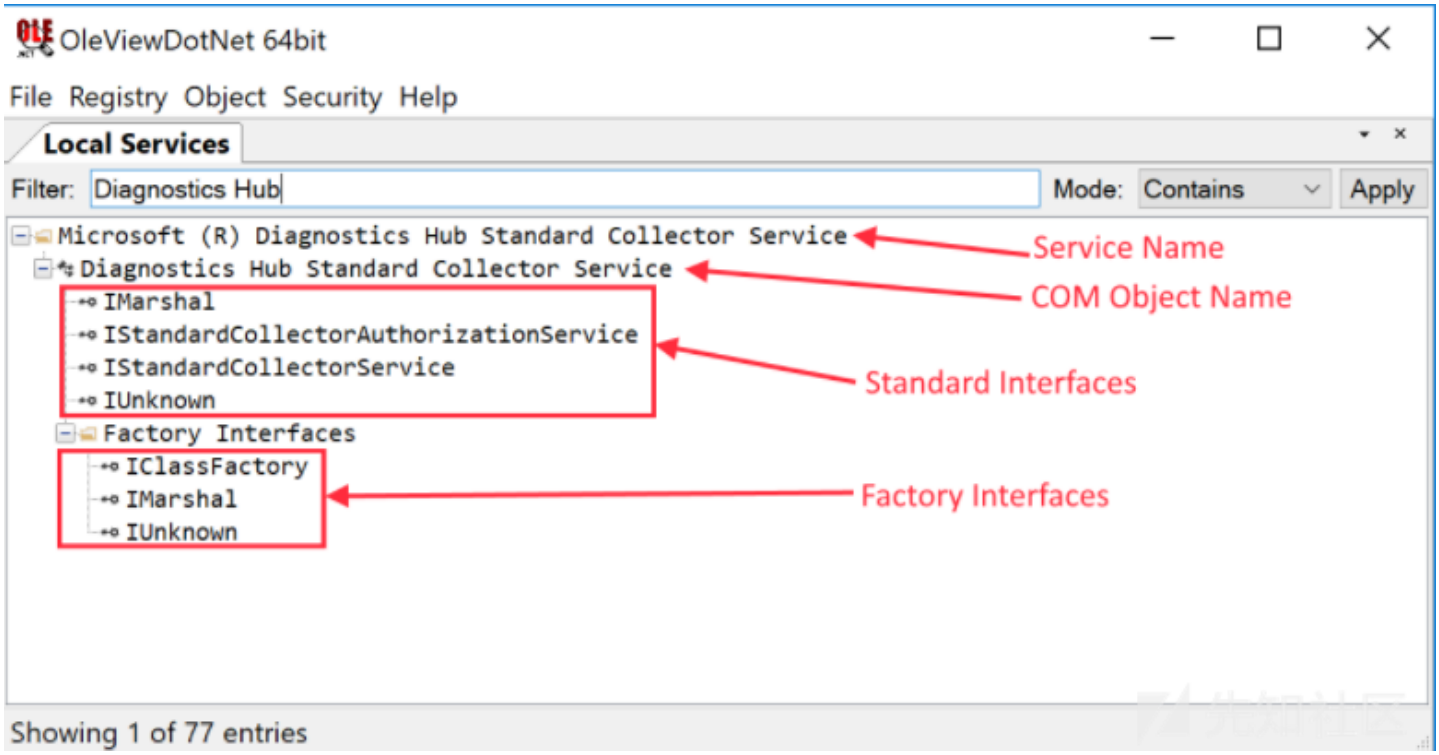
事实证明是的，并且服务本身之前至少被利用了两次，一次是Lokihardt用于沙箱逃逸，一次是由我user到EoP系统。
此服务名为"Microsoft（R）诊断中心标准收集器服务"，但我们将其简称为DiagHub。

DiagHub服务是在Windows 10中引入的，尽管有一项服务在Windows 7和8.1中执行类似的IE ETW Collector任务。
该服务的目的是代表沙盒应用程序（特别是Edge和Internet Explorer）使用Windows事件跟踪（ETW）收集诊断信息。它的一个有趣特性是它可以配置为从SYSTEM32目录加载任意DLL，这是Lokihardt和我获取提权的确切特性。该
此时你可以跳到最后，但如果你想了解我将如何实现DCOM对象，下一部分可能会引起关注。

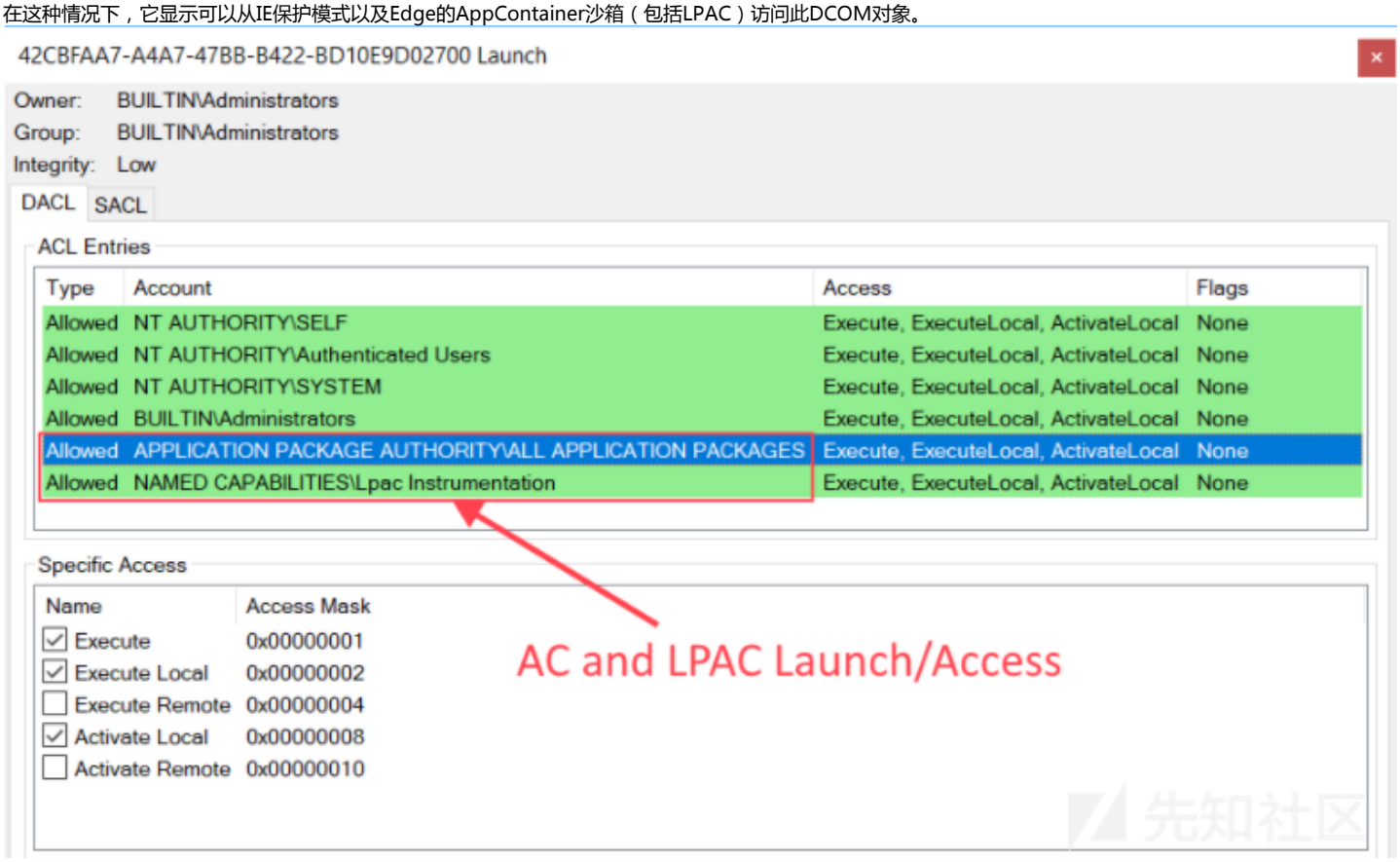
逆向DCOM对象

让我们来看看我将尝试找到未知DCOM对象支持的接口并找到实现的步骤，以便我们可以对它们进行逆向。我通常会采用两种方法，直接在IDA Pro中使用RE或类似方法，或者首先进行一些系统检查以缩小我们需要调查的范围。在这里，我们将采用第二种方法，因为它能提供更多信息。我不能说Lokihardt是如何找
我选择魔法。

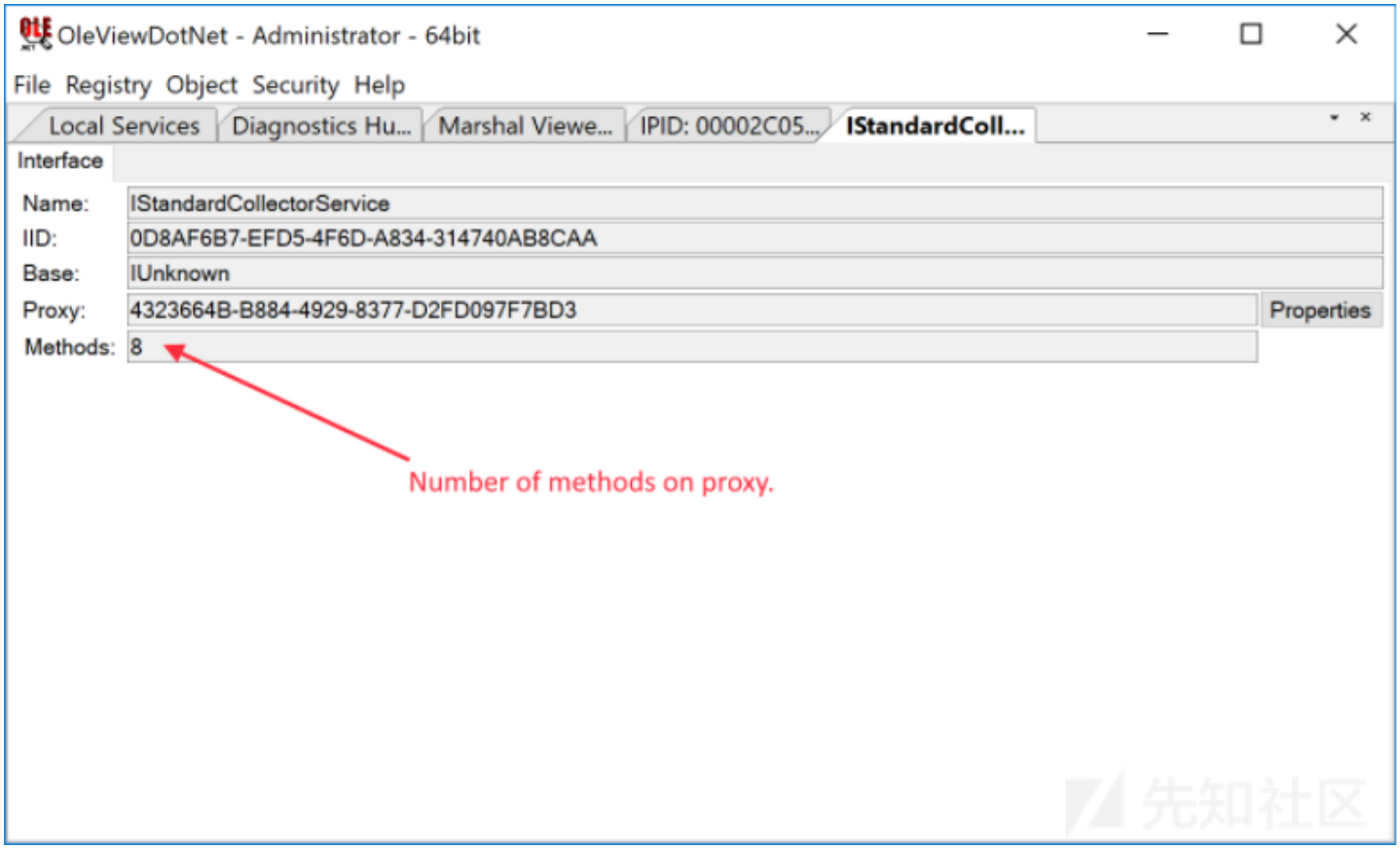
使用这种方法，我们需要一些工具，特别是来自github的OleViewDotNet v1.4 +（OVDN）工具以及SDK中的WinDBG安装。第一步是找到DCOM对象的注册信息，并发现可访问的接口。我们知道DCOM对象托管在一个服务中，所以一旦你加载了OV
如果你找到了"Microsoft（R）诊断中心标准收集器服务"服务（此处应用过滤器很有帮助），你应该能在列表中找到该条目。如果打开服务树节点，会看到一个子节点"诊断
下面的截图中展示了这点：



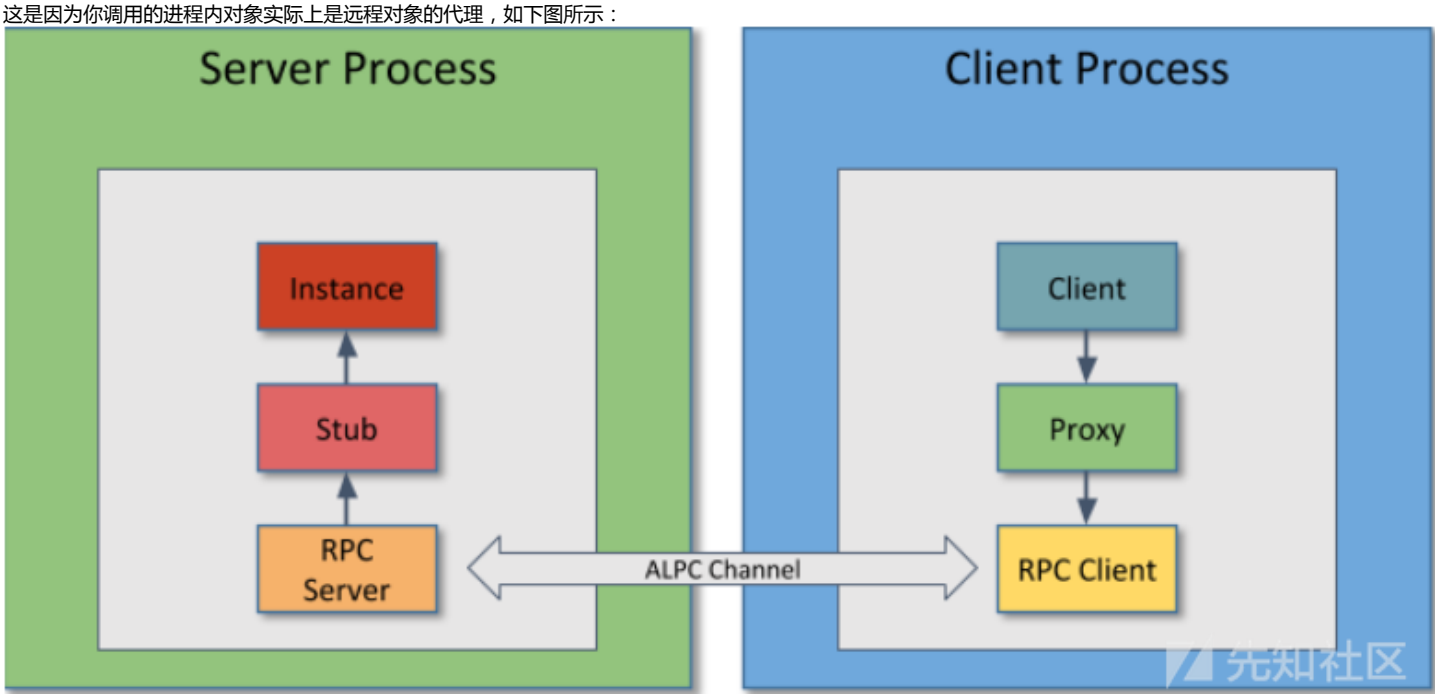
虽然我们在这里检查访问DCOM对象所需的安全性是有用的。 如果右键单击treenode类，则可以选择"View Access Permissions（查看访问权限）"或"View Launch Permissions（查看启动权限）"，然后您将看到一个显示权限的窗口。
在这种情况下，它显示可以从IE保护模式以及Edge的AppContainer沙箱（包括LPAC）访问此DCOM对象。



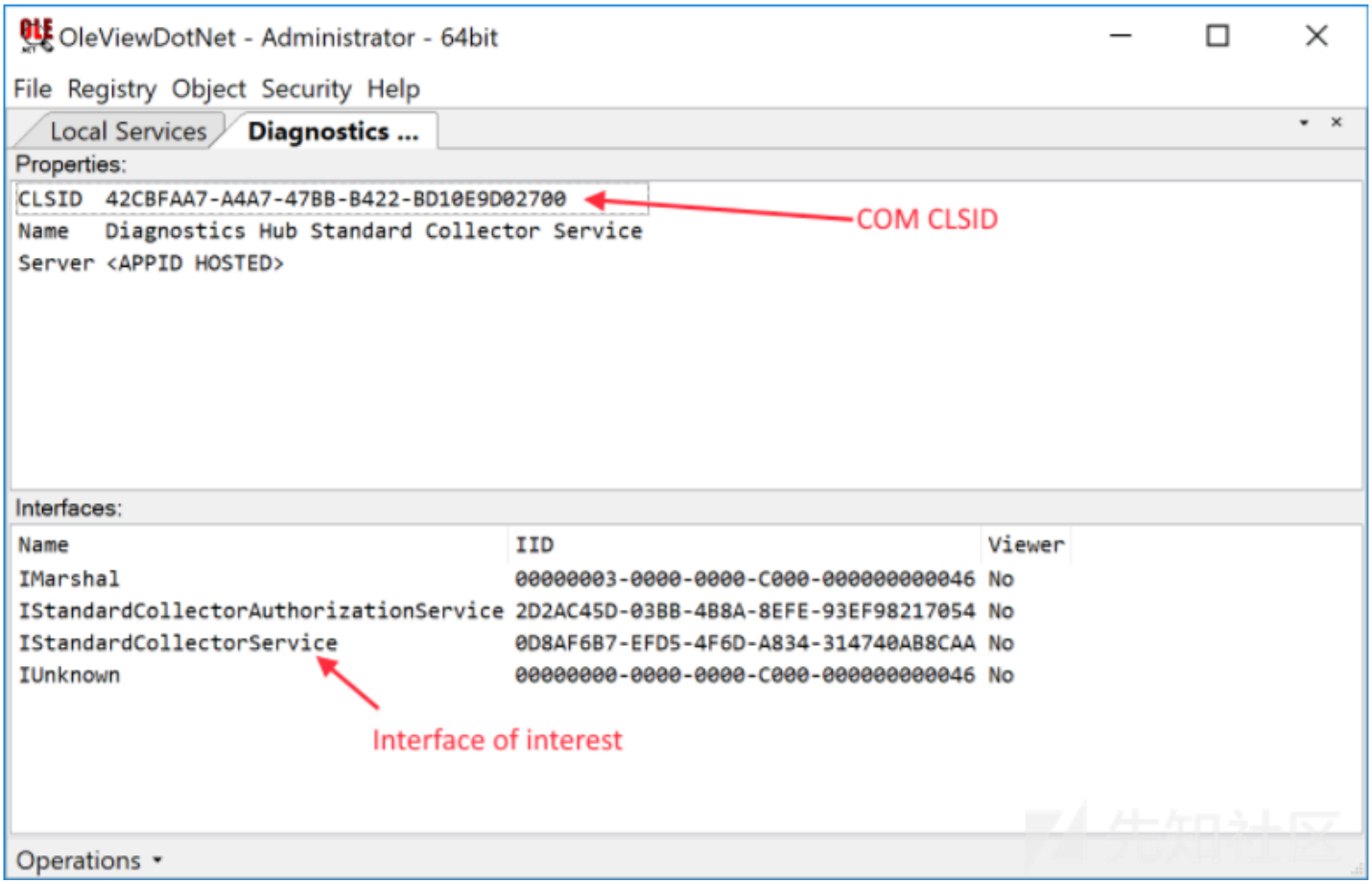
在显示的接口列表中，我们只关心标准接口。 有时会有一些有趣的界面，但在这种情况下却没有。
在这些标准接口中，有两个我们关心的，`IStandardCollectorAuthorizationService`和`IStandardCollectorService`。
我已经知道它是我们感兴趣的`IStandardCollectorService`服务，但是由于以下过程对于每个接口都是相同的，所以首先选择哪一个并不重要。
如果右键单击界面treenode并选择"Properties"，则可以看到有关已注册界面的一些信息。



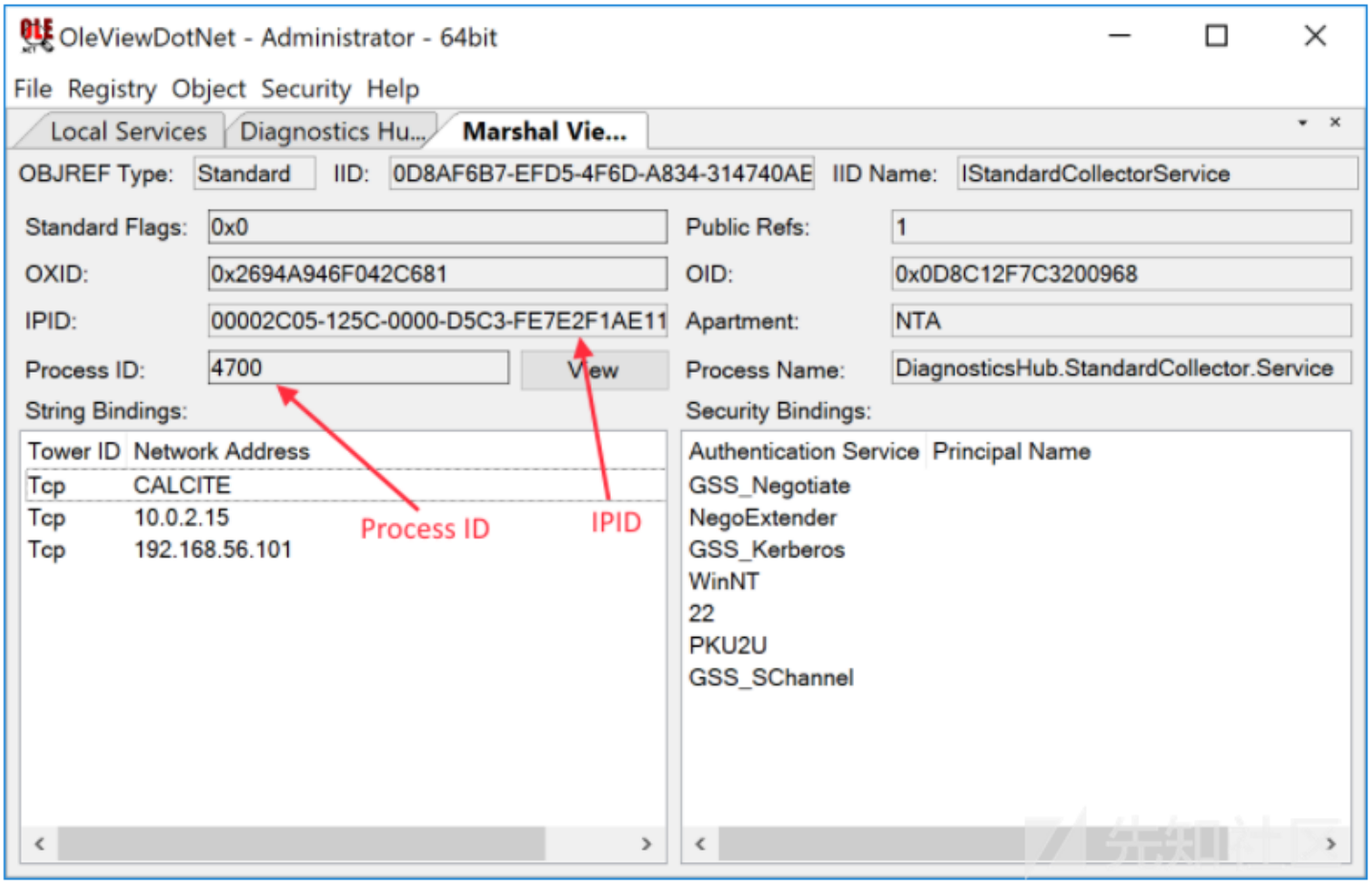
除了我们可以看到此界面上有8种方法之外，没有更多的信息可以帮助我们。
与许多COM注册信息一样，此值可能会丢失或错误，但在这种情况下，我们假设它是正确的。
要了解这些方法，我们需要在COM服务器中跟踪`IStandardCollectorService`的实现。
这些知识使我们能够将RE工作的目标定位于正确的二进制值和正确的方法。
为进程内COM对象执行此操作相对简单，因为我们可以通过取消引用几个指针直接查询对象的VTable指针。 但是，进程外的情况要更复杂。
这是因为你调用的进程内对象实际上是远程对象的代理，如下图所示：



然而，什么都没有丢失; 我们仍然可以通过提取存储在服务器进程中的对象的信息来找到OOP对象的VTable。首先右键单击“Diagnostics Hub Standard Collector Service”对象树节点，然后选择 `Create Instance`。 创建COM对象的新实例，如下所示：



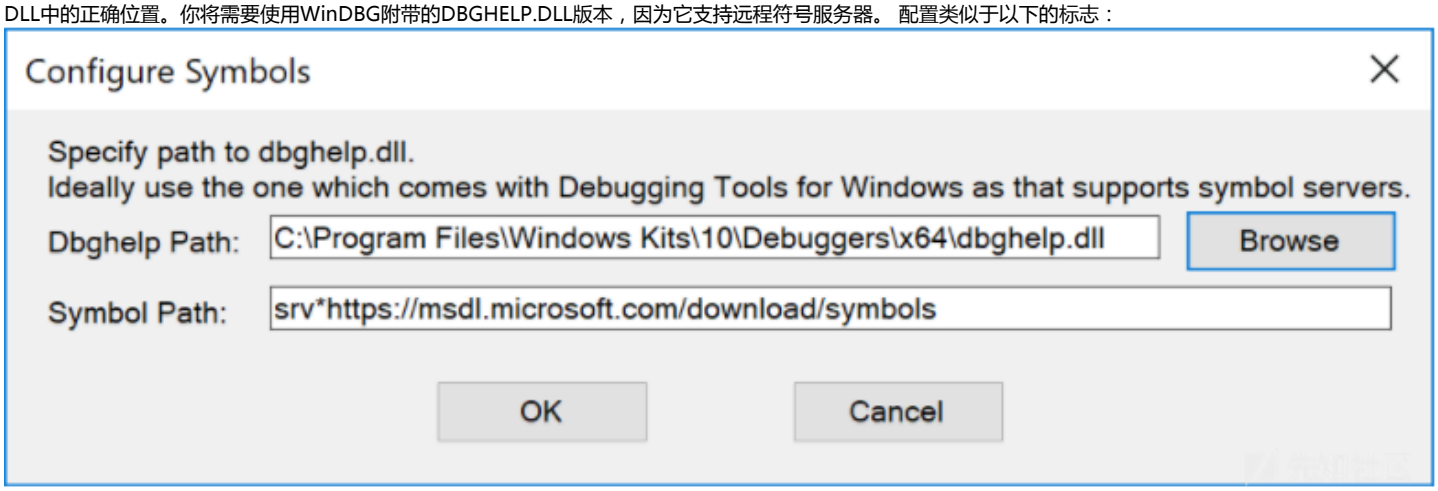
该实例为你提供基本信息，例如我们稍后需要的对象的CLSID（在本例中为{42CBFAA7-A4A7-47BB-B422-BD10E9D02700}）以及支持的接口列表。
现在我们需要确保连接到我们感兴趣的接口。为此，选择下方列表中的`IStandardCollectorService`接口，然后在底部的`Operations`菜单中选择`Marshal⇒ViewPro`
如果成功，你将看到以下界面：



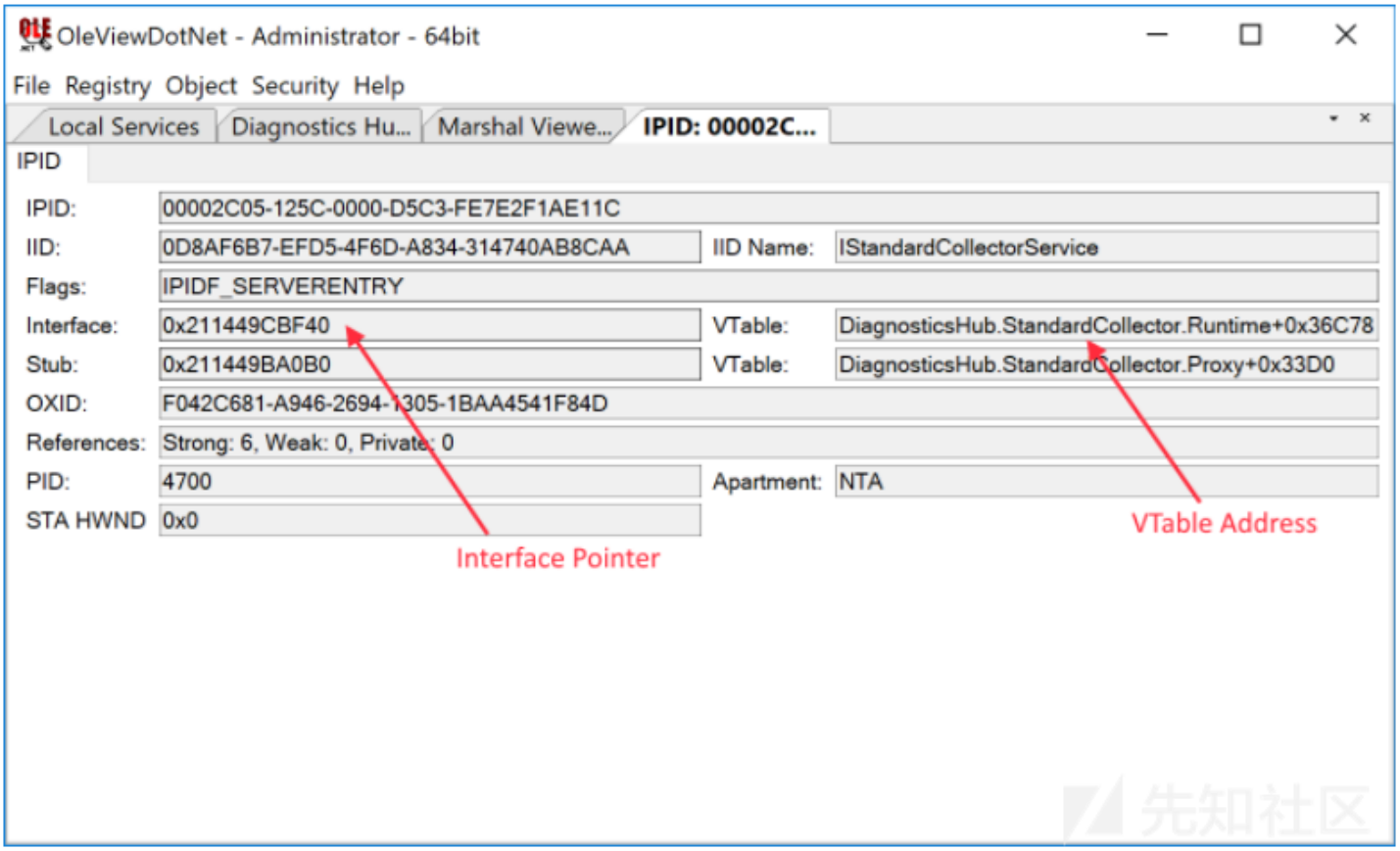
此图中有很多信息，但我们最感兴趣的两个部分是托管服务的进程ID和接口指针标识符（IPID）。在这种情况下，当服务在其自己的进程中运行时，进程ID应该是显而易见的
- 有时当你创建COM对象时，并不知道哪个进程实际托管COM服务器，因此这些信息非常宝贵。
IPID是DCOM对象的服务器端的托管过程中的唯一标识符; 我们可以结合使用进程ID和IPID来查找此服务器，并从中找出实现COM方法的实际VTable的位置。

值得注意的是，IPID的最大进程ID大小为16位;
但是，现代版本的Windows可以拥有更大的PID，因此你可能需要手动查找过程或多次重启服务，直到获得合适的PID。

现在我们将使用OVDN的一个功能，它允许我们进入服务器进程的内存并找到IPID信息。 你可以通过`main menu Object ⇒ Processes`访问有关所有进程的信息，但我们知道我们感兴趣的进程只需单击编组视图中进程ID旁边的`View`按钮。你需要以管理员身份运行OVDN，否则将无法打开服务流
DLL中的正确位置。你将需要使用WinDBG附带的DBGHELP.DLL版本，因为它支持远程符号服务器。 配置类似于以下的标志：



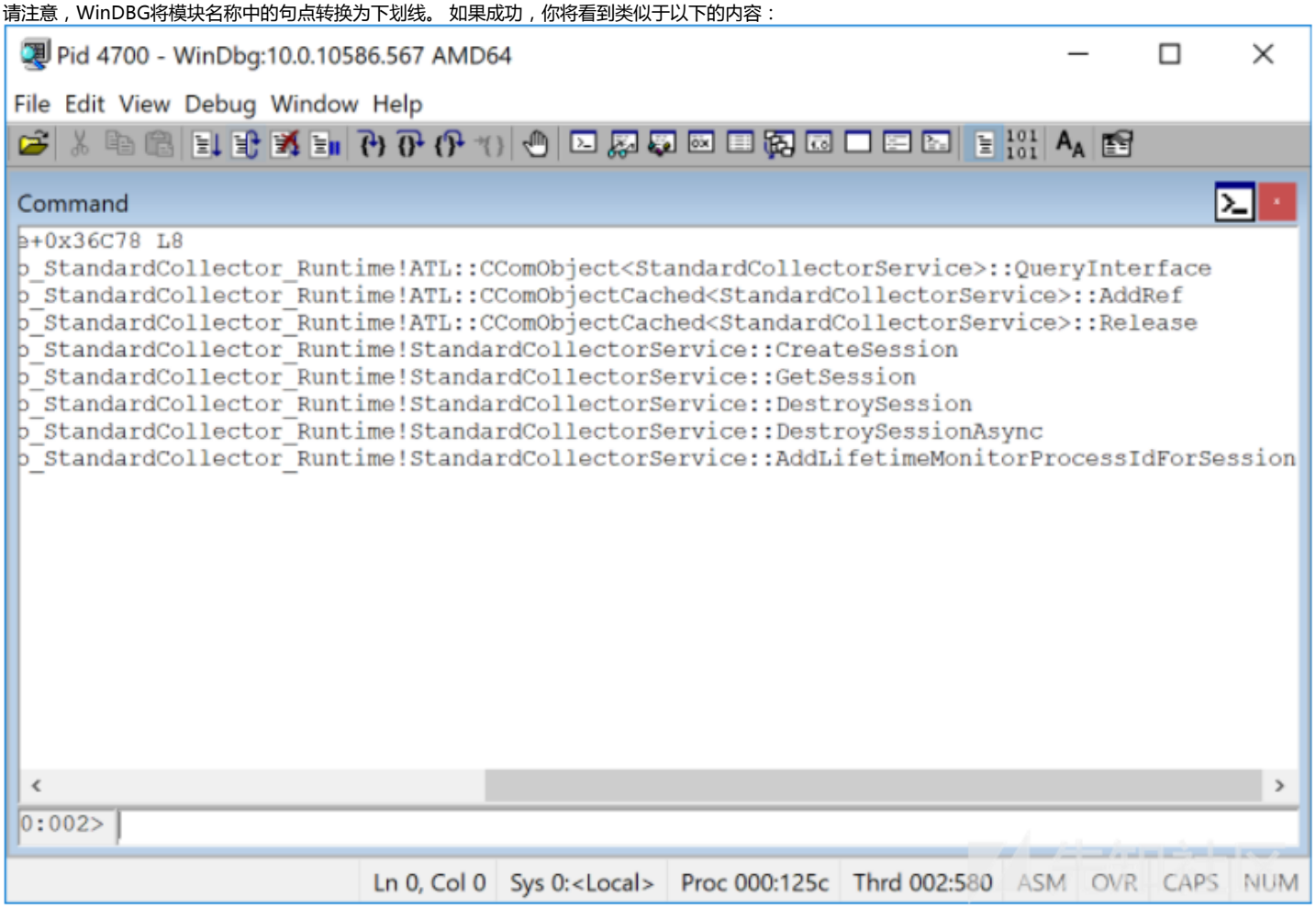
如果一切都配置正确并且您是管理员，您现在应该看到有关IPID的更多详细信息，如下所示：



这里最有用的两条信息是接口指针，它是堆分配对象的位置（如果要检查其状态），以及接口的VTable指针。VTable地址为我们提供了COM服务器实现所在位置的信息。
正如我们在这里看到的那样，VTable位于主可执行文件（`DiagnosticsHub.StandardCollector.Server`）的不同模块（`DiagnosticsHub.StandardCollector.`
我们可以通过使用WinDBG附加到服务进程并将符号转储到VTable地址来验证VTable地址是否正确。
我们之前也知道我们期待8种方法，因此我们可以使用以下命令将其考虑在内：

```
dqs DiagnosticsHub_StandardCollector_Runtime+0x36C78 L8
```

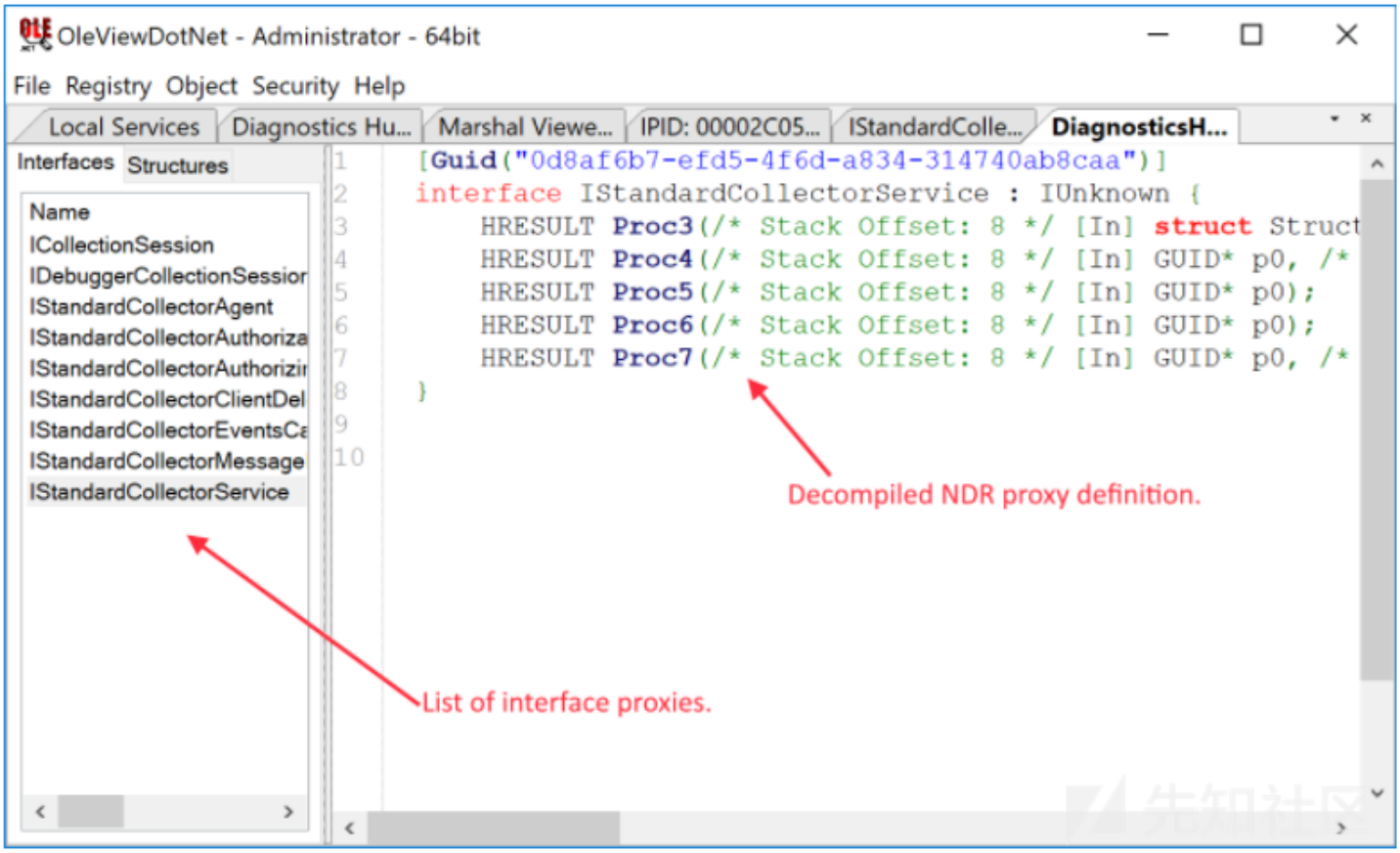
请注意，WinDBG将模块名称中的句点转换为下划线。 如果成功，你将看到类似于以下的内容：



提取出这些信息，得到方法的名称（如下所示）以及二进制文件中的地址。 我们可以设置断点并查看在正常操作期间调用的内容，或者获取此信息并启动RE过程。

```
ATL::CComObject<StandardCollectorService>::QueryInterface
ATL::CComObjectCached<StandardCollectorService>::AddRef
ATL::CComObjectCached<StandardCollectorService>::Release
StandardCollectorService::CreateSession
StandardCollectorService::GetSession
StandardCollectorService::DestroySession
StandardCollectorService::DestroySessionAsync
StandardCollectorService::AddLifetimeMonitorProcessIdForSession
```

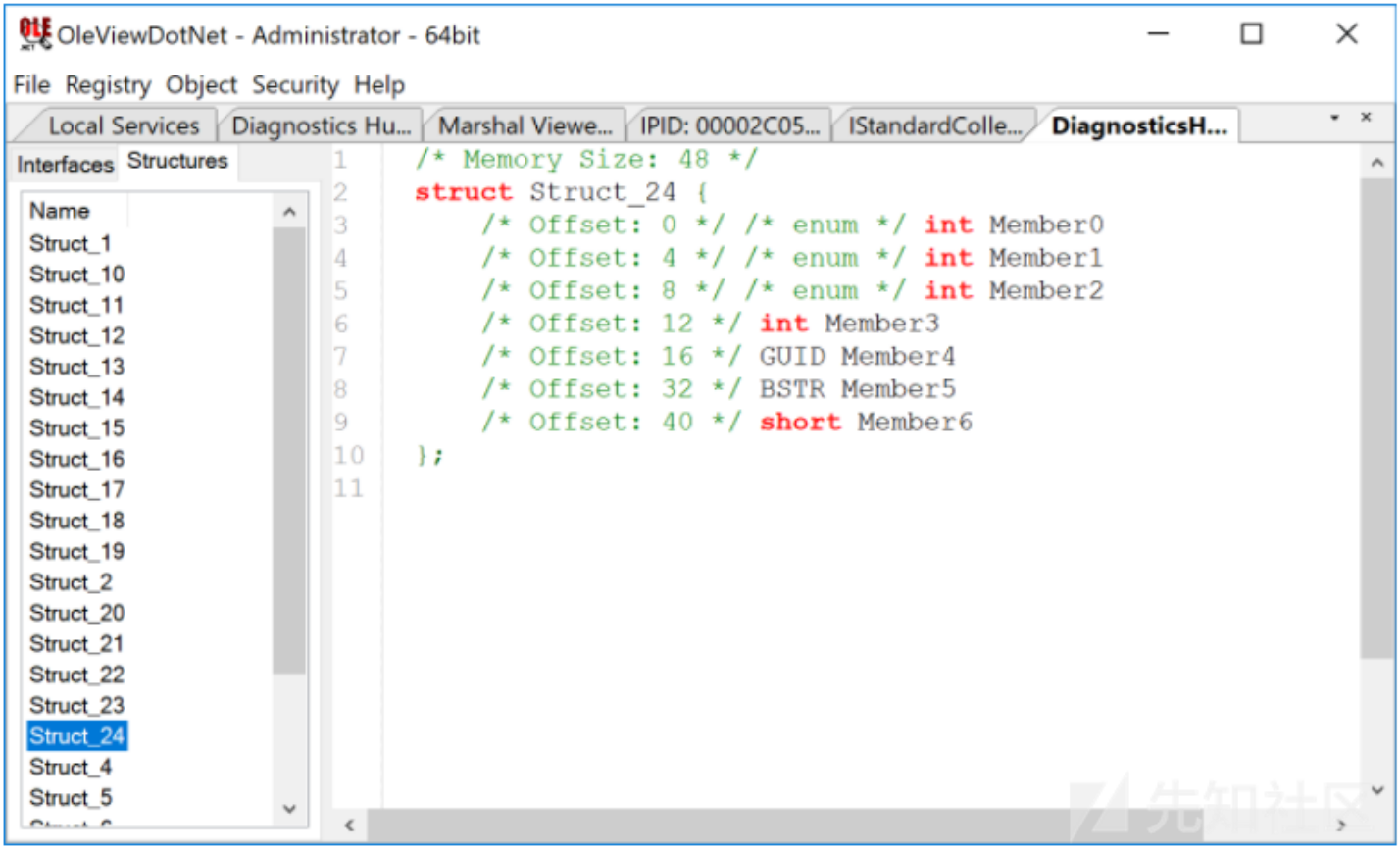
方法列表看起来是正确的：它们从COM对象的3种标准方法开始，在这种情况下由ATL库实现。 以下这些方法由`StandardCollectorService`类实现。
作为公共标志，这并不告诉我们期望传递给COM服务器的参数。 由于包含某些类型信息的C++名称，IDA
Pro也许能为你提取该信息，但并不一定会告诉你可能传递给该函数的任何结构的格式。
幸运的是，由于如何使用网络数据表示（NDR）解释器实现COM代理来执行编组，因此可以将NDR字节码反转回我们可以理解的格式。
在这种情况下，请返回原始服务信息，右键单击`IStandardCollectorService treenode`并选择`View Proxy Definition`。
这会使OVDN解析NDR代理信息并显示新视图，如下所示。



查看代理定义还将解析该代理库实现的任何其他接口。 也许对进一步的逆向工作有用。
反编译的代理定义以类似C#的伪代码显示，但是根据需要应该很容易转换为能工作的C#或C++。 请注意，代理定义不包含方法的名称，但我们已经提取出来了。
因此，应用一些cleanup和方法名称，我们得到一个如下所示的定义：

```
[uuid("0d8af6b7-efd5-4f6d-a834-314740ab8caa")]
struct IStandardCollectorService : IUnknown {
  HRESULT CreateSession(_In_ struct Struct_24* p0,
                        _In_ IStandardCollectorClientDelegate* p1,
                        _Out_ ICollectionSession** p2);
  HRESULT GetSession(_In_ GUID* p0, _Out_ ICollectionSession** p1);
  HRESULT DestroySession(_In_ GUID* p0);
  HRESULT DestroySessionAsync(_In_ GUID* p0);
  HRESULT AddLifetimeMonitorProcessIdForSession(_In_ GUID* p0, [In] int p1);
}
```

最后一项丢失了; 我们不知道Struct_24结构的定义。 可以从RE过程中提取它，但幸运的是在这种情况下我们不必这样做。
NDR字节码必须知道如何编组这种结构，因此OVDN只是自动为我们提取结构定义：选择`Structures`选项卡并找到Struct_24。



当你实践RE过程时，可以根据需要重复此过程，直到你了解一切如何运作。 现在让我们开始实际利用DiagHub服务，并展示它在真实环境中的应用。

示例exp

经过逆向分析的努力，我们发现，为了从SYSTEM32加载DLL，需要执行以下步骤：

1. 使用IStandardCollectorService :: CreateSession创建新的诊断会话。
   在新会话上调用ICollectionSession :: AddAgent方法，传递要加载的DLL的名称（没有任何路径信息）。
   ICollectionSession :: AddAgent的简化加载代码如下：

```
void EtwCollectionSession::AddAgent(LPWCSTR dll_path,
                                    REFGUID guid) {
WCHAR valid_path[MAX_PATH];
if ( !GetValidAgentPath(dll_path, valid_path)) {
return E_INVALID_AGENT_PATH;
HMODULE mod = LoadLibraryExW(valid_path,
   nullptr, LOAD_WITH_ALTERED_SEARCH_PATH);
dll_get_class_obj = GetProcAddress(hModule, "DllGetClassObject");
return dll_get_class_obj(guid);
}
```

我们可以看到它检查代理路径是否有效并返回完整路径（这是以前的EoP错误存在的位置，检查不足）。
使用LoadLibraryEx加载此路径，然后查询DLL以获取导出的方法DllGetClassObject，再调用该方法。
为了使代码更容易执行，我们需要实现该方法并将文件放入SYSTEM32。 已实现的DllGetClassObject将在加载程序锁之外调用，以便我们执行任何操作。
以下代码（删除错误处理）可以加载名为dummy.dll的DLL。

```
IStandardCollectorService* service;
CoCreateInstance(CLSID_CollectorService, nullptr, CLSCTX_LOCAL_SERVER, IID_PPV_ARGS(&service));

SessionConfiguration config = {};
config.version = 1;
config.monitor_pid = ::GetCurrentProcessId();
CoCreateGuid(&config.guid);
config.path = ::SysAllocString(L"C:\Dummy");
ICollectionSession* session;
service->CreateSession(&config, nullptr, &session);

GUID agent_guid;
CoCreateGuid(&agent_guid);
```

```
session->AddAgent(L"dummy.dll", agent_guid);
```

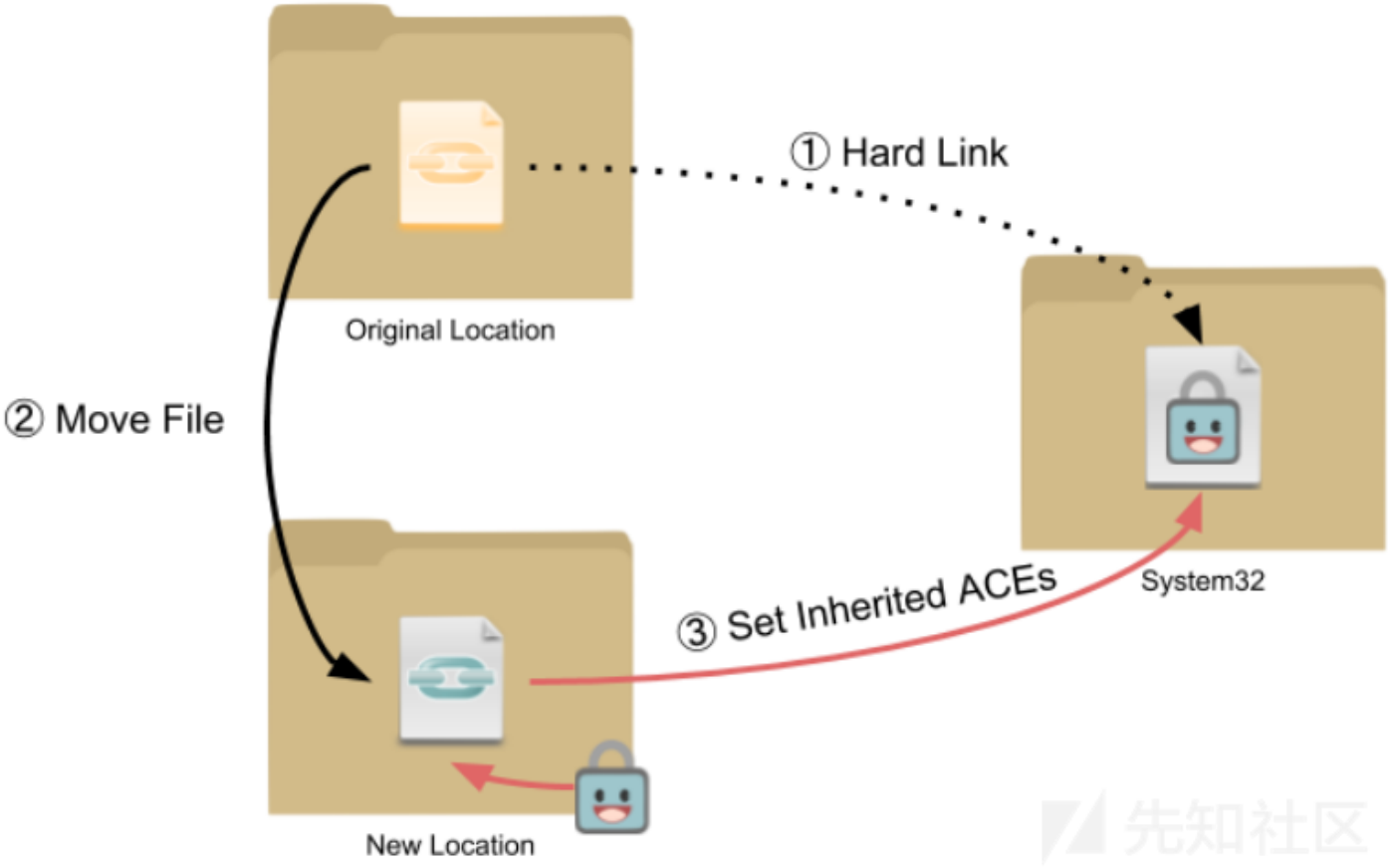
现在我们需要的是任意文件写入，以便将DLL放入SYSTEM32，加载并提升我们的权限。 为此，我将演示我在Storage Service系统中SvcMoveFileInheritSecurity RPC方法里发现的漏洞。 这个功能引起了我的注意，因为它用于探索由ClémentRouault和Thomas Imbert在PACSEC 2017中发现和呈现的ALPC中的漏洞。虽然这种方法只是漏洞的有效原函数，但我意识到，潜伏在其中的实际上是两个漏洞（至少来自普通用户权限）。
SvcMoveFileInheritSecurity的任何修复之前的代码如下所示：

```
void SvcMoveFileInheritSecurity(LPCWSTR lpExistingFileName,
                                LPCWSTR lpNewFileName,
                                DWORD dwFlags) {
PACL pAcl;
if (!RpcImpersonateClient()) {
  // Move file while impersonating.
  if (MoveFileEx(lpExistingFileName, lpNewFileName, dwFlags)) {
    RpcRevertToSelf();
    // Copy inherited DACL while not.
    InitializeAcl(&pAcl, 8, ACL_REVISION);
    DWORD status = SetNamedSecurityInfo(lpNewFileName, SE_FILE_OBJECT,
        UNPROTECTED_DACL_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION,
        nullptr, nullptr, &pAcl, nullptr);
      if (status != ERROR_SUCCESS)
        MoveFileEx(lpNewFileName, lpExistingFileName, dwFlags);
  }
  else {
    // Copy file instead...
    RpcRevertToSelf();
  }
}
}
```

这个方法的目的似乎是移动文件，然后将任何继承的ACE从新目录位置应用于DACL。
这是必要的，因为当文件在同一卷上移动时，旧文件名被取消链接并且文件链接到新位置。 但是，新文件将保持从其原始位置分配的安全性。
只有在目录中创建新文件时才能应用继承的ACE，或者在这种情况下，通过调用SetNamedSecurityInfo等函数显式应用ACE。

要确保此方法不允许任何人在作为服务的用户（在本例中为本地系统）运行时移动任意文件，需要模拟RPC调用者。
在第一次调用MoveFileEx后立即启动故障，模拟被还原并调用SetNamedSecurityInfo。
如果该调用失败，则代码再次调用MoveFileEx以尝试恢复原始移动操作。 这是第一个漏洞; 原始文件名位置现在可能指向其他位置，例如滥用符号链接。
这很容易导致SetNamedSecurityInfo失败，只需将本地系统的拒绝ACL添加到文件的WRITE_DAC的ACE中，它会返回一个错误，导致恢复并获得任意文件创建。
这被报告为问题1427。

事实上，这并不是我们将要利用的漏洞，因为这太简单了。
相反，我们将在同一代码中利用第二个漏洞：在本地系统运行时获取服务以在我们喜欢的任何文件上调用SetNamedSecurityInfo。
这可以通过在执行初始MoveFileEx时滥用模拟设备映射来重定向本地驱动器号（例如C
:)来实现，然后导致lpNewFileName指向任意位置，或者更有趣地滥用硬链接。 这被报告为问题1428。我们可以使用硬链接来利用它，如下所示：



在SYSTEM32中创建一个我们要覆盖的目标文件的硬链接。 因为你不需要对文件具有写权限来创建到它的硬链接，至少在沙箱之外。

创建一个新目录位置，该目录位置具有可为每个人或经过身份验证的用户的组的可继承ACE，以允许修改任何新文件。 甚至不需要明确地执行此操作;
例如，在C盘根目录中创建的任何新目录都有一个用于Authenticated Users的继承ACE。 然后，请求RPC服务将硬链接文件移动到新目录位置。
只要我们有`FILE_DELETE_CHILD`访问新位置的`FILE_DELETE_CHILD`和我们可以编辑的`FILE_ADD_FILE`，该移动就会在模拟下成功。

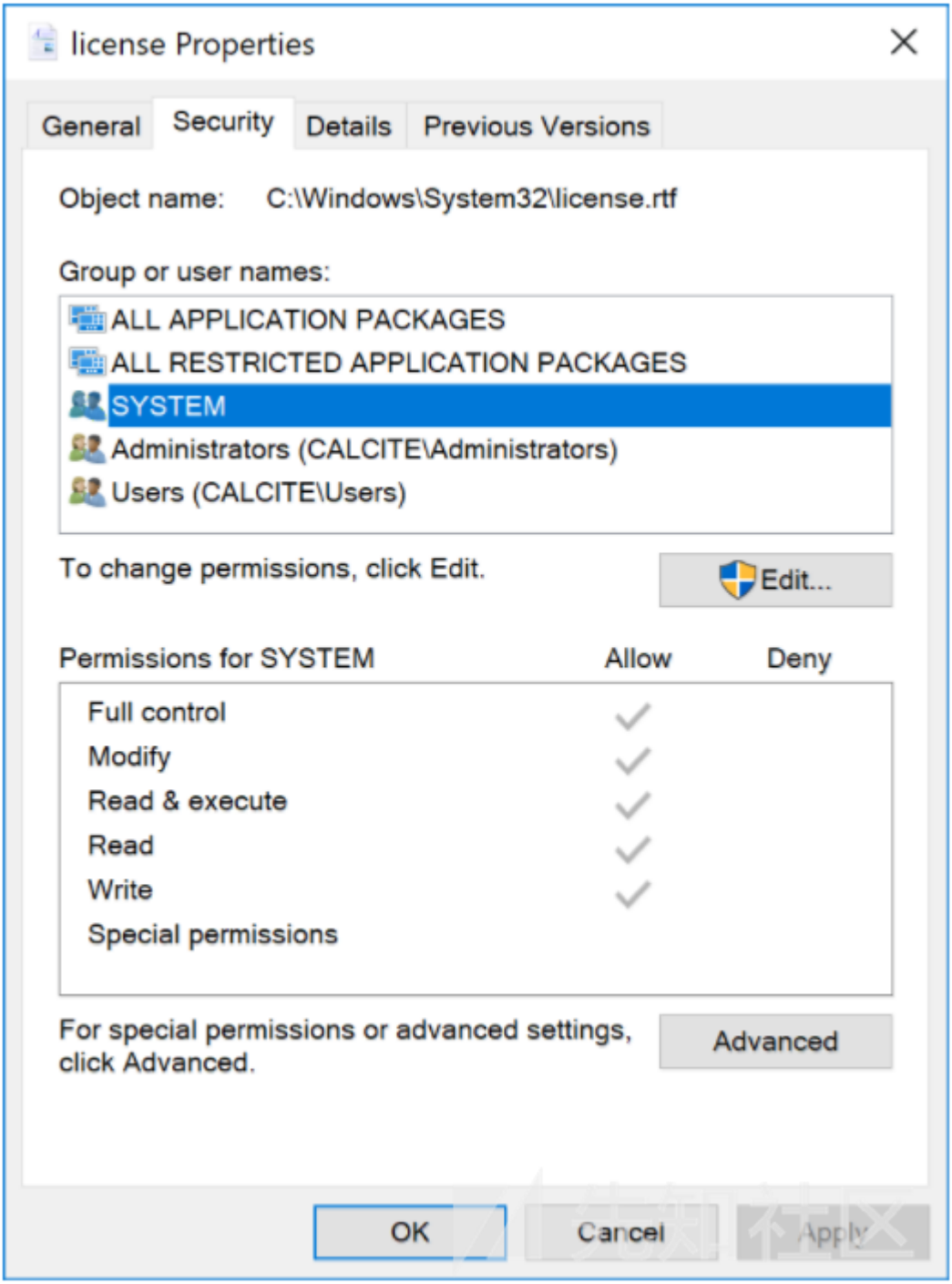
该服务现在将在移动的硬链接文件上调用`SetNamedSecurityInfo`。`SetNamedSecurityInfo`将从新目录位置获取继承的ACE，并将它们应用于硬链接文件。
将ACE应用于硬链接文件的原因是从`SetNamedSecurityInfo`的角度看，硬链接文件位于新位置，即使我们链接到的原始目标文件位于SYSTEM32中。

利用这一点，我们可以修改本地系统以访问WRITE_DAC访问的任何文件的安全系统。 然后我们修改SYSTEM32中的文件，再使用DiagHub服务加载它。
但是，有一个小问题。 SYSTEM32中的大多数文件实际上由TrustedInstaller组拥有，即使是本地系统也无法修改。
因此，我们需要找到一个可以写入的文件，该文件不归TrustedInstaller所有。 此外，我还想选择一个不会导致操作系统安装损坏的文件。
我们不关心文件的扩展名，因为`AddAgent`仅检查文件是否存在并使用`LoadLibraryEx`加载它。
可以通过多种方式找到合适的文件，例如使用SysInternals AccessChk实用程序，但要100%确定存储服务的令牌可以修改文件，然后使用我的NtObjectManager PowerShell模块（特别是其`Get-AccessibleFile cmdlet`，它接受从中进行访问检查的进程）。
虽然该模块设计用于检查沙箱中的可访问文件，但它也可用于检查特权服务可访问的文件。
如果以管理员身份运行以下脚本并安装了模块，则`$files`变量将包含Storage Service具有WRITE_DAC访问权限的文件列表。

```
Import-Module NtObjectManager

Start-Service -Name "StorSvc"
Set-NtTokenPrivilege SeDebugPrivilege | Out-Null
$files = Use-NtObject($p = Get-NtProcess -ServiceName "StorSvc") {
  Get-AccessibleFile -Win32Path C:\Windows\system32 -Recurse `
   -MaxDepth 1 -FormatWin32Path -AccessRights WriteDac -CheckMode FilesOnly
}
```

查看文件列表，我决定选择文件license.rtf，其中包含Windows的简短许可证声明。
这个文件的优点是它很可能对系统的操作不是很关键，因此覆盖它应该不会导致安装损坏。



把它们放在一起：

1. 使用Storage Service漏洞更改SYSTEM32中`license.rtf`文件的安全性。
2. 复制DLL，它通过`license.rtf`文件实现`DllGetClassObject`。
3. 使用DiagHub服务将修改后的许可证文件作为DLL加载，将代码执行作为本地系统并执行我们想要的任何操作。

如果你有兴趣看到一个完整的示例，我已经在tracker上上传了原始问题的完整漏洞。

总结

在这篇博客文章中，我描述了一个适用于Windows 10的有用漏洞原函数，你甚至可以从Edge LPAC等沙盒环境中使用它。
找到这些类型的原函数使得利用更简单，更不容易出错。 此外，我已经让你了解如何在类似的DCOM实现中找到自己的错误。

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板