南開大學
Nankai University

网络空间安全学院

《恶意代码分析与防治技术》课程实验报告

实验：Rootkit77

姓名：王峥

学号：2211267

专业：信息安全

指导教师：王志、邓琮弋

2024 年 11 月 15 日

目录

1 实验目的

1. 理解 r77 Rootkit 的基本特征和工作原理，包括其在内存中持久存在的方式以及对系统文件、进程、服务、注册表项等的隐藏机制。学习 r77 的部署方法，特别是通过运行 Install.exe 实现对系统的注入和持久性。
2. 掌握文件系统、进程、注册表的隐藏技术，了解如何通过前缀、ID、名称等方式实现对这些实体的隐匿。了解 TCP 和 UDP 网络连接的隐藏原理，包括通过前缀、ID、名称以及特定端口配置的方式来实现连接的隐匿。
3. 运行 R77 程序，实现对指定的进程、文件、注册表、网络连接的隐藏。对实验结果进行截图，完成实验报告。

2 实验原理

2.1 Rootkit

Rootkit 是一种特殊的恶意软件，其主要功能是在安装目标上隐藏自身及指定的文件、进程和网络链接等信息。它通常被用于维持攻击者的长期访问权限，而不被用户或系统管理员察觉。以下是 Rootkit 的一些主要行为和特点：

1. **隐藏功能：** Rootkit 可以隐藏特定的文件或文件夹，使它们无法在文件系统中被正常访问或查看。同时，通过修改操作系统的进程列表，Rootkit 可以隐藏正在运行的恶意进程，从而避免被用户或安全软件发现。Rootkit 还可以隐藏恶意软件的网络通讯活动，如数据传输、远程控制等，以防止被网络监控工具检测到。
2. **权限提升：** Rootkits 通常试图提升其执行的权限，以绕过操作系统的安全层级。这可能涉及到提升到管理员或系统级别的权限，以执行更深层次的操纵。
3. **持久性：** Rootkits 致力于在系统中保持长期存在。它们常常会修改系统的启动项、注册表、或其他关键组件，以确保在系统重新启动后仍然存在。
4. **安装后门：** 通过隐藏的后门程序，攻击者可以在不被察觉的情况下远程访问和控制目标系统。
5. **内核级操作：** 一些 Rootkits 会操作在操作系统的内核级别，这使得它们更难被检测和清除，因为它们可以绕过用户空间的安全工具。

Rootkit 是一种具有隐蔽性、操纵性和数据收集能力的恶意软件技术，对计算机系统安全构成严重威胁。因此，用户应提高安全意识，采取必要的防御措施来保护自己的计算机系统免受 Rootkit 的攻击。

然而，rootkit 并不仅仅用于恶意目的。它们也被组织和执法机构用于监视员工，使他们能够调查机器并对抗可能的网络威胁。

2.2 Rootkit77

Rootkit77（通常称为 r77-Rootkit）是一款功能强大的无文件 Ring 3 Rootkit，了它具有如下的功能：

1. **隐藏功能：**能够在所有进程中隐藏文件、目录、连接、命名管道、计划任务、进程、CPU 用量、注册表键值、服务以及 TCP 和 UDP 连接等实体。并能通过前缀隐藏，即将**所有以“$77”为前缀命名的实体都将被隐藏**。动态配置系统允许通过 PID 或名称来隐藏进程，通过完整路径来隐藏文件系统，或通过指定端口隐藏 TCP 和 UDP 连接。

2. **动态配置：**r77 具有动态配置系统，可以通过 PID 和名称隐藏进程，通过完整路径隐藏文件系统项目，隐藏特定端口的 TCP 和 UDP 连接等。配置位于 HKEY_LOCAL_MACHINE\\SOFTWARE\\$77config，任何进程都可以写入，无需提升权限。此外，rootkit 会隐藏 $77config 键。R77 的部署只需要一个文件：Install.exe。执行后，R77 将在系统上持久存在，并注入所有正在运行的进程。Uninstall.exe 可以完全并优雅地从系统中移除 r77。Install.shellcode 是安装程序的 shellcode 等价物，这样，安装可以在不放置 Install.exe 的情况下集成。shellcode 可以简单地加载到内存中，转换为函数指针并执行。

所以，R77 是一个 64 位上操作系统上可以运行的 Ring3 Rootkit，出于其在只能在 64 位上运行的特殊性质，本次实验我选择在 Win10 操作系统上进行实验，避免不必要的情况发生。

2.3 Windows 的 Detours 机制

Detours 是由微软研究部门开发的一个开源库，它专门用于监视和拦截 Windows 平台上的函数调用。这个库在逆向工程、调试、系统监控和动态代码注入等领域有着广泛的应用。通过 Detours，开发者可以在 API 调用前后插入自己的代码，以监视、修改或替换原有的行为。

3 实验过程

3.1 实验环境搭建

首先在我们已有的 Win10 虚拟机上将我们从 github 仓库中下载的压缩文件复制，然后我们需要先关闭虚拟机的电脑的病毒威胁检测，不然压缩包一旦解压缩后，相关病毒文件会被自动删除：

图 3.1: 关闭病毒检测

接下来我们将文件夹解压，并进行快照，准备开始实验。

3.2 动态运行 R77

我们打开 R77 的文件夹进行查看：

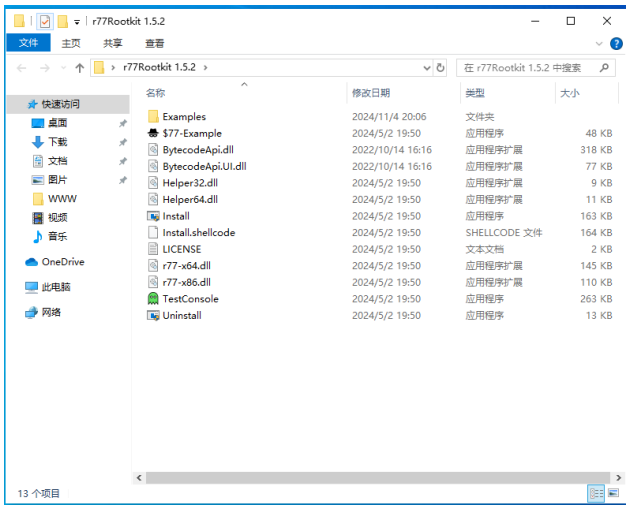

图 3.2: R77 文件中信息

然后双击 install.exe 运行，看到可以看到原本本文件夹下的 $77-Example.exe 也不见了。这验证了它会隐藏所有以 $77 为开头的文件，进程等的行为。

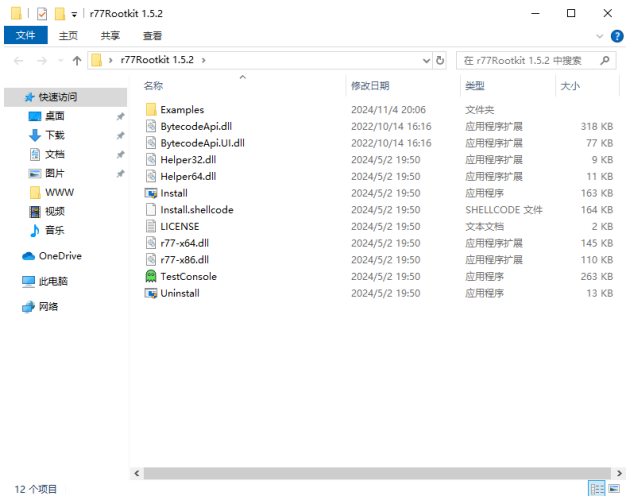

图 3.3: $77-Example.exe 消失

接下来我们依次对进程、文件、注册表和网络连接隐藏功能进行验证。

3.3 进程隐藏

以 $77 前缀开头的文件名对应的可执行文件的进程是被隐藏的。

3.3.1 任意文件的隐藏

首先查看其隐藏进程的行为，利用那个给定的以 $77 为开头的 $77-Example.exe，打开运行，为了能让他更好地体现出被隐藏但实际存在的效果，我将 CPU 占有率调为 100%

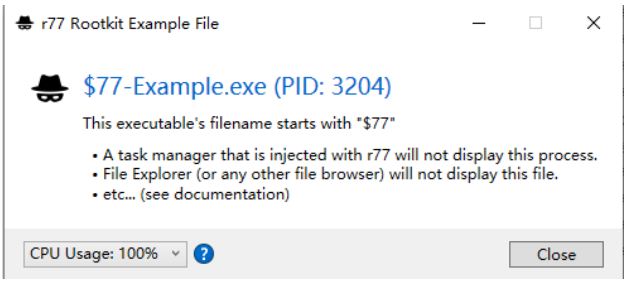

图 3.4: CPU 占有率

这是我们运行 install 前：

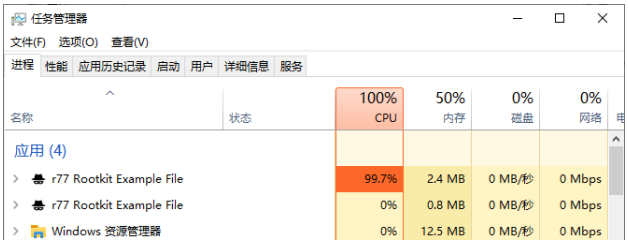

图 3.5: 运行 install 前

我们看到 $77-Example.exe 这个可执行文件是完全可见的。
接下来我们运行 install.exe 文件，再查看任务管理器：

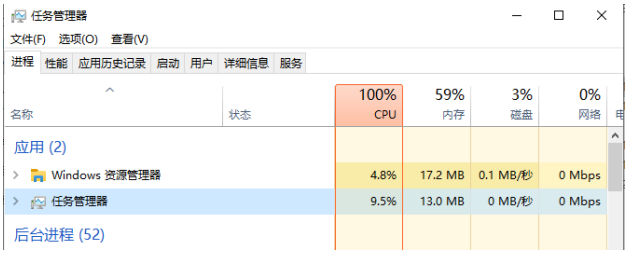

图 3.6: 运行 install 后

这次我们看到任务管理器中的可执行性文件 Example 已经不见了，但是由于其被我设置为了 CPU100%，我们还是能看到 CPU 此时的极高运用率，但其他三个进程都没干什么事几乎是 0，明显就是被隐藏了。而且我们再刷新文件目录，就像我们前分析的一样 $77-Example.exe 找不到了，说明也是被隐藏了。接下来我们尝试用工具查看，首先是 Process Explorer：

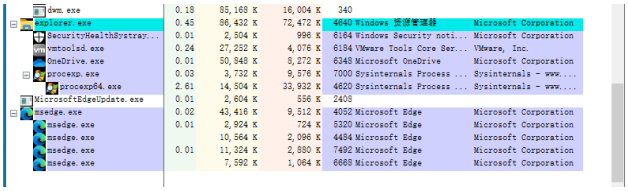

图 3.7: Process Explorer

看到此时显示的进程目录树中明显没有这个进程，确实证明其被隐藏。那接下来我们再尝试用 Procmon 来看：

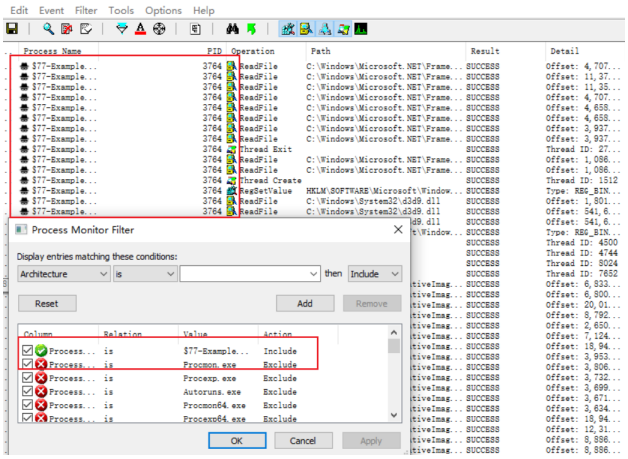

图 3.8: Procmon

这里我们惊奇的发现，之前隐藏的可执行文件被检测了，其中包括读取文件等行文。

3.3.2 指定进程检测

除了上面说的隐藏任意 $77 开头的文件外，还可以通过 TestConsole.exe 实现指定进程的隐藏：

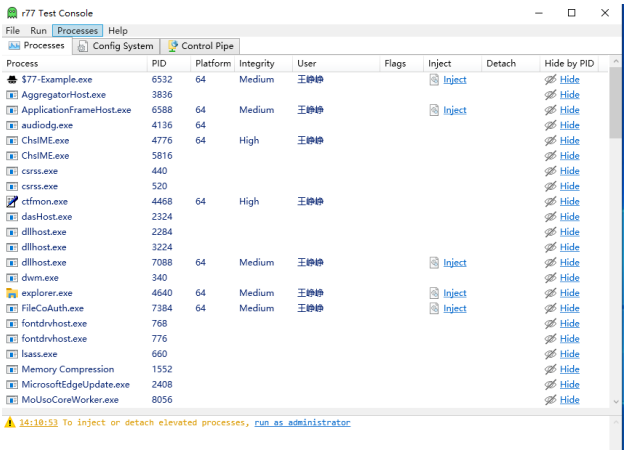

图 3.9: 打开 TestConsole

可以看到所有进程列表，不管是被隐藏的 Example 还是别的，右边还有 Hide 选项。我这里为了实现指定进程隐藏，我们将 TestConsole.exe 隐藏，我进行对照分析：

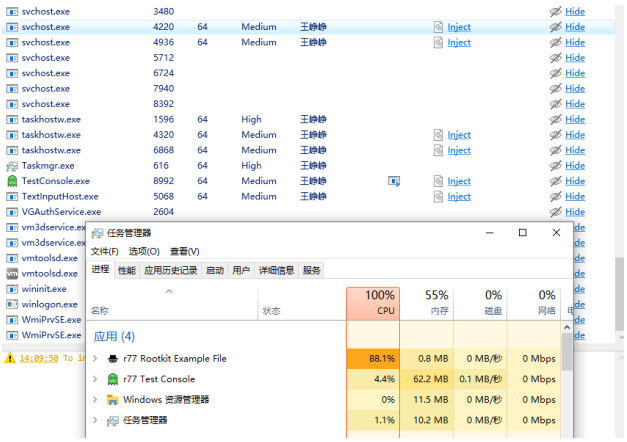

图 3.10: 隐藏前

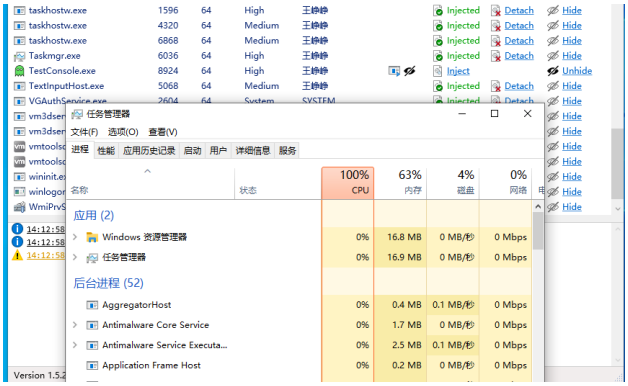

图 3.11: 隐藏后

我们看到点击 Hide 后，变成了 Unhide，但一开始我点击并没有成功，它提示我权限不够，我们获取用户权限后再运行，发现任务管理器已经找不到 TestConsole 进程了，隐藏成功。到此就实现了全部的任意或者指定进程的隐藏。

3.3.3 进程隐藏原理分析

根据我阅读手册，了解到 R77 实现进程隐藏具有如下的一些方式和原理：

1. 名称驱动的隐藏策略

   - 操作原理：R77 通过检查进程的文件名来决定是否隐藏该进程。如果一个进程的文件名以特定前缀（如 $77）开头，R77 将自动将该进程隐藏。
   - 技术手段：该策略通常通过劫持和修改系统 API 调用来实现，特别是那些列举进程的 API（如 EnumProcesses、NtQuerySystemInformation 等）。当这些 API 被触发时，R77 会干预并修改它们的输出，去掉带有 $77 前缀的进程。这样，用户在任务管理器等工具中无法看到这些进程。

2. **基于进程 ID 的隐蔽技术：**

   R77 支持通过进程 ID（PID）来隐藏进程。用户可以通过配置系统输入要隐藏的进程 ID，这样即使进程的名称没有被修改，它仍然可以通过其 ID 被隐藏。这种方法尤其适用于无法修改文件名的进程，或者那些由恶意软件在内存中动态创建的进程。

3. **替代的名称基准隐藏**

   R77 不仅支持使用前缀来隐藏进程，还允许用户通过配置文件指定具体的进程名称进行隐藏。这种方式对那些文件名不具有特定前缀的进程同样有效。

4. 枚举与直接操作的区别

   - **隐藏的含义**：在 R77 中，隐藏并不意味着完全销毁或终止进程。隐藏只是将进程从系统的进程枚举列表中移除。这意味着，在任务管理器等工具中看不到该进程，但如果知道其 ID 或名称，用户仍然可以直接访问或操作该进程。
   - **关键提示**：R77 并没有拦截那些打开进程或文件的 API，因此如果用户通过进程 ID 或进程名称直接访问，被隐藏的进程不会返回“未找到”的错误。

5. **反射式 DLL 注入技术**

   R77 使用反射式 DLL 注入技术将 DLL 注入到目标进程中。这种方法的特点是，注入的 DLL 不需要写入硬盘，而是通过内存注入。R77 有两个版本：一个是用于 32 位系统的 r77-x86.dll，另一个是用于 64 位系统的 r77-x64.dll。

   注入流程如下：

   - 内存预留：首先，R77 通过调用 VirtualAllocEx 函数在目标进程的虚拟地址空间中分配一块内存区域。
   - 写入 DLL：使用 WriteProcessMemory 函数将 DLL 的内容（即 r77-x86.dll 或 r77-x64.dll）写入到目标进程中分配的内存区域。
   - 定位启动函数：R77 使用 GetReflectiveLoaderOffset 函数来找到 DLL 中 ReflectiveLoader 函数的偏移地址。ReflectiveLoader 是一个特别的启动函数，它能够加载 DLL 并初始化其内容。
   - 线程生成与执行：在目标进程中创建一个新线程，通过调用 ReflectiveLoader 来加载并执行注入的 DLL。
   - 执行监控：使用 WaitForSingleObject 函数监控新创建的线程，确保它能够顺利运行并完成 DLL 的初始化过程。

   反射式 DLL 注入技术的关键特点在于它通过内存注入的方式，不需要将 DLL 写入磁盘，从而避开了传统的磁盘扫描检测。此外，DLL 的加载和执行是在目标进程内的单独线程中完成的，而不是通过标准的加载机制，这使得该注入方式更难被传统的反病毒软件或系统监控工具检测。

6. **进程创建时的拦截**

   - 启动机制：一旦 R77 服务启动，它会被注入到所有正在运行的进程中。R77 通过劫持系统调用 NtResumeThread 来监控新进程的创建。
   - 注入时刻：这种方法确保 R77 能够在任何新进程开始执行前就注入并初始化自己。这对于一些应用程序特别重要，例如注册表编辑器（RegEdit）等程序，它们会在启动后快速进行初始化并列举系统中的项。通过注入，R77 能够隐藏自己或其他指定进程，避免这些进程在进程列表中显现。

通过上述分析，我们发现 R77 通过一系列的技术手段实现了进程的隐蔽性，特别是在反射式 DLL 注入、进程 ID 和文件名隐藏方面，展现了恶意软件高度的隐蔽性和攻击能力。其通过劫持和修改系统 API、内存注入和线程控制等手段，使得其难以被常规的安全监测工具发现。

3.4 文件隐藏

我们初步了解到文件隐藏的功能实现与 API Hooking 技术有关，具体而言，它通过拦截和修改系统界别的文件系统调用实现。这样让特定文件或目录在操作系统的标准文件或者浏览工具比如 Windows 资源管理器中不可见。因此，文件隐藏的实现需要文件必须以 $77 为文件名开头。

3.4.1 文件隐藏实验

接下来我们进行实验验证：我们先将整个文件复制并命名为 $77，放在 C 盘下：

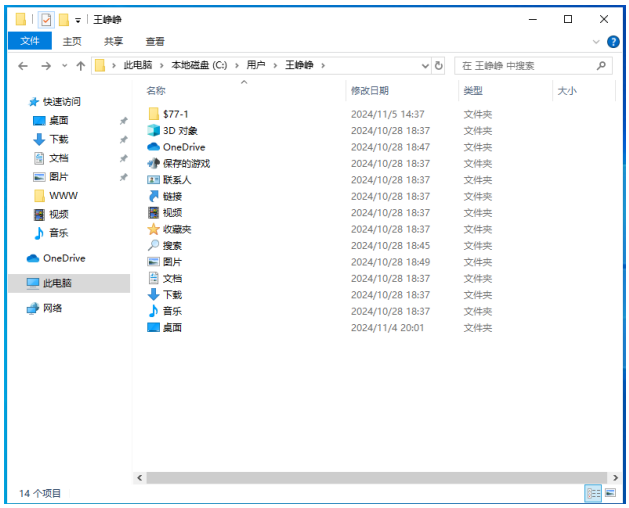

图 3.12: 文件隐藏前

接下来我们点击 install，刷新文件夹我们看到我们刚才看的的文件消失了！

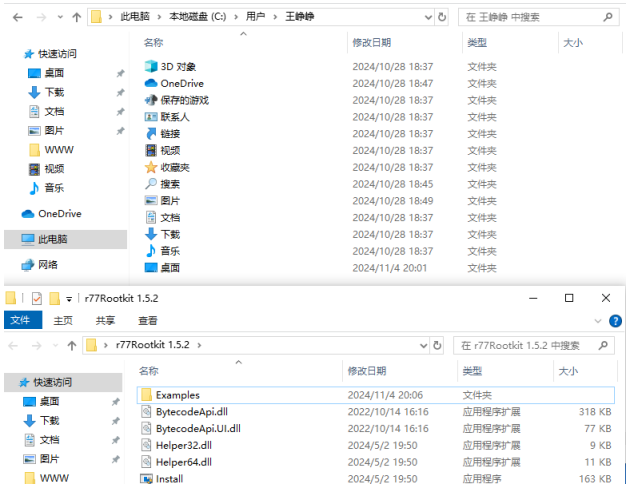

图 3.13: 文件隐藏后

我还把原本在桌面的文件夹也改名，然后提示我文件目录不存在了，我想点 uninstall，他也提示我目录名无效，还好之前有进行快照

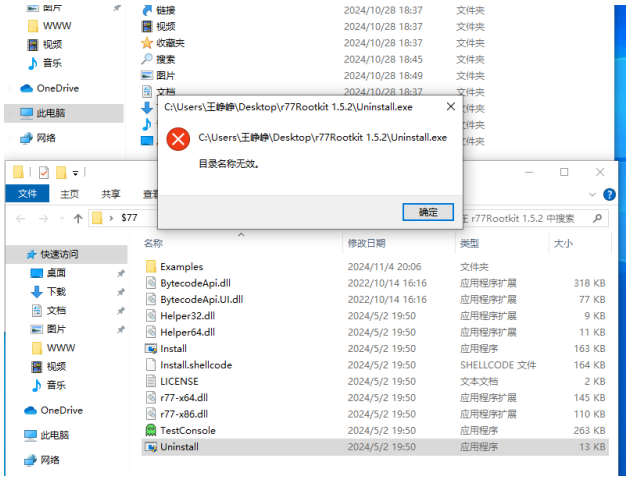

图 3.14: 桌面文件隐藏后

3.4.2 文件隐藏原理

我们继续了解我一些 R77 文件隐藏机制有一些的实现原理：

1. **劫持文件系统核心函数**

   R77 通过劫持 Windows 的核心文件系统函数来实现文件和目录的隐藏。最常见的劫持目标是 NtQueryDirectoryFile 和 NtQueryDirectoryFileEx，这两个函数负责列举目录中的文件、子目录、符号链接和命名管道。其中 NtQueryDirectoryFile 是 Windows 系统中关键的 API 函数，用于获取指定目录中的文件信息。它返回的是目录内容的枚举结果，包括文件名、目录、符号链接等。R77 通过劫持这些函数，修改它们的行为，使得它们在枚举目录时不返回那些被隐藏的文件和目录的名字。

2. **修改枚举结果**

   R77 会在拦截并成功调用原始的 NtQueryDirectoryFile 函数之后，审查每一个被枚举的文件或目录名称。

   - 隐藏条件：如果文件或目录的名称符合特定条件（如以 $77 前缀命名），R77 会将这些文件或目录从返回结果中剔除。
   - 过滤机制：R77 的核心机制是过滤文件系统枚举的结果，因此，即使这些文件或目录在文件系统中实际存在，它们也不会在通过 NtQueryDirectoryFile 等函数列举的目录中出现。换句话说，这些文件在 Windows 资源管理器、命令行界面（如 dir 命令）中将完全不可见。

3. **特定路径的隐蔽支持**

   精细控制：除了通过特定前缀（如 $77）来隐藏文件或目录，R77 还允许用户在其配置系统中指定精确的路径。用户可以指定某个文件或目录的完整路径，然后将其加入 R77 的隐藏列表中，这样就可以对特定的文件或目录进行个性化的隐藏。通过 R77 的配置系统，用户可以有选择性地隐藏任何路径下的文件或目录，无论其名称是否符合某种预设的规则。

4. **隐藏效果的实现**

- 不可见性：R77 的文件隐藏机制使得被隐藏的文件和目录在使用常规工具（如 Windows 资源管理器或命令行工具）浏览文件系统时完全不可见。这种方法在视觉上隐藏了文件，但并不会修改文件的实际内容或物理存储位置。
- 直接访问：需要注意的是，R77 的文件隐藏只影响文件的枚举操作。也就是说，如果用户知道被隐藏文件或目录的确切路径，仍然可以直接访问它们（例如，通过命令行输入完整路径）。这意味着 R77 没有完全删除文件，而是通过修改文件系统的枚举行为来让它们“消失”。

5. **安全性与隐蔽性的考虑**

- 隐蔽性：通过劫持和修改文件系统函数，R77 提供了极高的隐蔽性。因为它直接影响的是操作系统核心层面的文件访问行为，使得普通的安全工具和用户很难检测到隐藏的文件或目录。
- 安全隐患：然而，这种隐藏机制也带来了潜在的安全风险。恶意软件可以利用这种技术隐藏其存在，使得检测和移除变得更加困难。特别是当恶意软件能够以特定前缀或路径隐藏其文件时，传统的反病毒软件和文件监控工具往往无法有效检测到这些文件的存在。

6. **NtQueryDirectoryFile 的实现步骤**

R77 在拦截 NtQueryDirectoryFile 时，一般会按照以下步骤进行操作：

- 原函数调用：R77 首先调用原始的 NtQueryDirectoryFile 函数，正常获取目录内容的枚举结果。
- 状态检查：在返回的结果中，R77 会检查函数的返回状态，确保有文件或目录被列出。
- 内容过滤：在过滤过程中，R77 会检查每个列出的文件或目录名称，如果名称符合预设条件（如以 $77 为前缀），则会将其从返回的目录枚举结果中删除。
- 返回修改后的结果：最后，R77 将过滤后的结果返回给调用方（如 Windows 资源管理器或其他 API 调用），从而实现文件的隐藏。

3.5 注册表隐藏

3.5.1 隐藏注册表实验验证

接下来验证 Rootkit77 的注册表隐藏功能，其配置信息存储在 HKEY_LOCA_MACHINE\SOFTWARE$77confi 中，并且可以在未提权状态下由任何进程写入。这个键的 DACL 被设置为可以给任意用户授予完整访问权 1。“$77config” 键在注册表编辑器被注入了 Rootkit 之后会自动隐藏。同时还能实现对任意以“$77” 开头作为注册表键进行隐藏。我们看到，我们打开注册表编辑器，创建了一个以 $77 开头的表项，我们接下来用 Install 来刷新注册表：

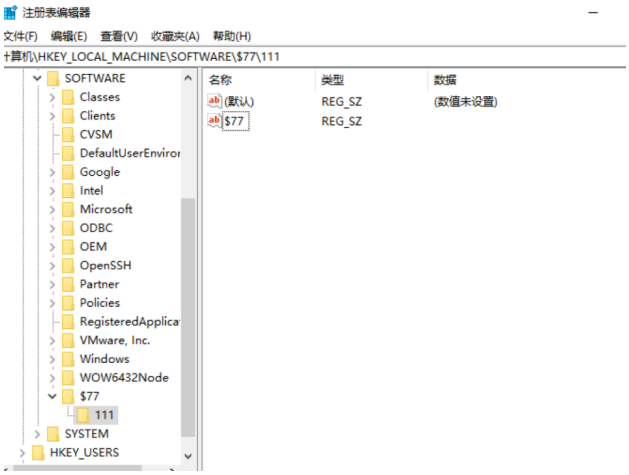

图 3.15: 创造以 $77 为开头的注册表键

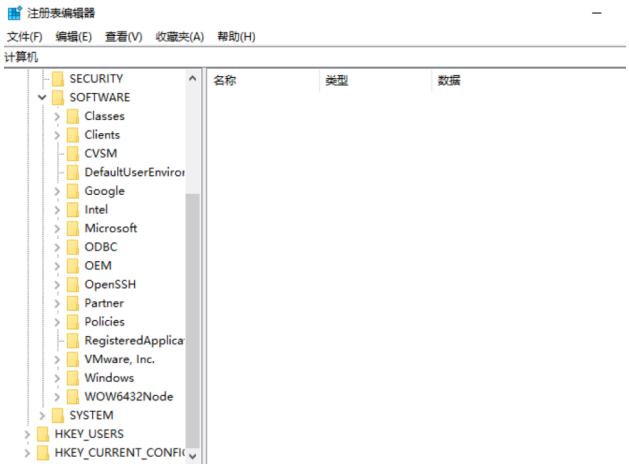

图 3.16: 注册表项被隐藏

我们看到已经找不到对应的键值了，说明 $77 被隐藏成功了，验证成功。

3.5.2 隐藏注册表原理

R77 的注册表隐藏机制与其文件隐藏机制类似，通过劫持和修改核心注册表访问函数来实现对特定注册表项和值的隐蔽。

1. **关键函数**

   R77 劫持了两个内核级注册表函数：NtEnumerateKey 和 NtEnumerateValueKey。这些函数在 Windows 系统中负责遍历注册表键和值，是许多系统和应用程序访问注册表的核心.

2. **枚举过程的改造**

   - 标准枚举流程：当应用程序或系统调用 NtEnumerateKey 和 NtEnumerateValueKey 时，R77 首先执行标准的枚举操作。这意味着，它首先允许注册表项和注册表值的正常扫描，不改变原始的操作流程。

- 筛选隐藏项：在正常的枚举流程完成后，R77 会审查枚举到的每一个注册表项和注册表值，检查它们是否符合隐藏条件。R77 通常会检查项名和键值名是否符合特定的命名规则（例如以 $77 为前缀）。如果发现项或值符合隐藏条件，R77 将把它们从枚举结果中剔除。
- 重构枚举索引：为了确保这些被隐藏的项不会出现在返回的结果中，R77 会重新计算枚举索引。即使某些注册表项或注册表值实际存在，它们也不会出现在 API 的返回结果列表中。通过这种方式，R77 能够在不直接修改注册表的内容的情况下，隐藏它们在枚举时的可见性。

3. **NtEnumerateKey 的具体实现**

对于 NtEnumerateKey 函数，R77 的操作一般会包括以下几个步骤：

- 执行原函数：R77 首先执行原始的 NtEnumerateKey 函数，列举目标注册表项下的所有子项。这个过程保持和标准操作一致，不做任何修改。
- 结果验证：R77 检查函数调用的返回状态，确认是否成功列出了注册表项。如果枚举成功，返回值应包括目标注册表项下的所有子项。
- 注册表键过滤：在 FilterRegistryKeys 函数中，R77 根据预设的隐藏条件（如项名前缀）检查每个返回的注册表项。对于符合条件的项（例如名称以 $77 开头的项），R77 会将它们从返回的枚举结果中删除。
- 重新计算索引：为了避免留下空洞或不一致的结果，R77 会根据已删除项的数量重新计算剩余项的枚举索引。这确保了在后续的注册表遍历中，剩余项的枚举是连续和一致的。
- 返回修改后的结果：最终，R77 将过滤后的结果返回给调用方。由于隐藏的注册表项或值已经被去除，调用者将无法看到这些项，即使它们在实际的注册表中存在。

3.6 网络连接隐藏

3.6.1 网络隐藏实验

最后我们 Uninstall 前面的 install，然后验证一下 R77 对网络连接例如 TCP 和 UDP 的隐藏。我们使用 Edge 来查看，接下来进行试验：

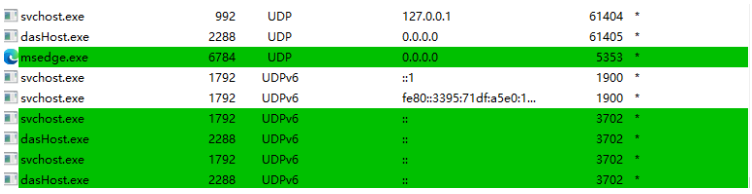

图 3.17: TcpView 查看

我们打开 TCPView，查看现在有一个正在运行在 PID=6784 的 Edge 进程，我们再看 TestConsole.exe 中也有同样的运行：

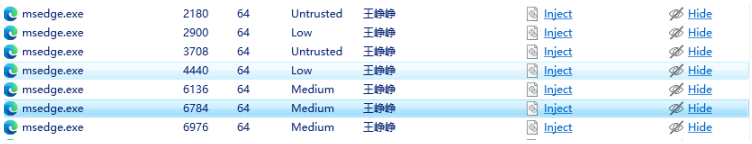

图 3.18: TestConsole.exe 查看

那我们点击 Hide 进行查看：

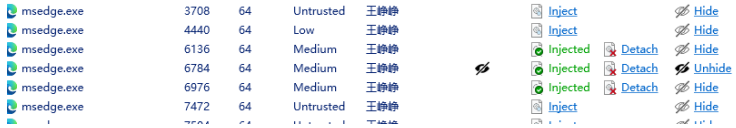

图 3.19: 使用 TestConsole.exe 进行隐藏

我们再看 TcpView 查看结果，没有发现 PID=6784 的 Edge 进程，证明隐藏成功！

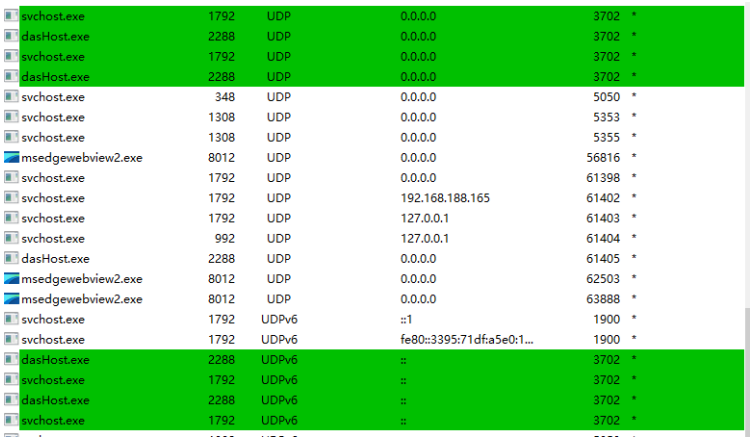

图 3.20: 隐藏后 TcpView 查看

相似的是，我们再进行 TCP 的隐藏：我们看到有一个 PID=1528 的 TCP 进程：

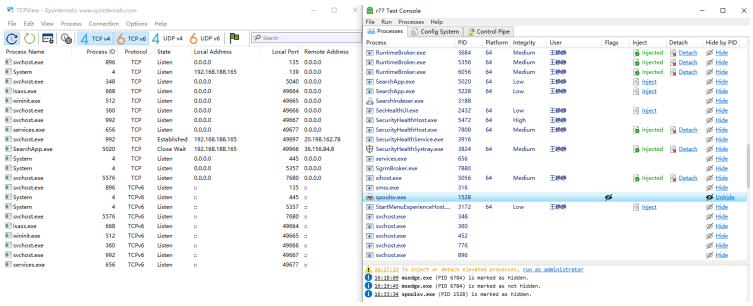

图 3.21: TCP 隐藏前

接下来我们还是进行隐藏：

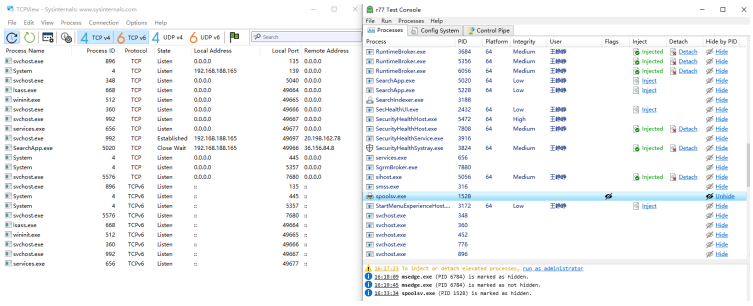

图 3.22: TCP 隐藏后

我们看到这个 spoolsv.exe 也被隐藏了，验证成功。

3.6.2 网络隐藏原理

R77 的网络连接隐藏技术是通过截取和修改网络 I/O 控制函数来实现的，特别是通过拦截 NtDeviceIoControlFile 函数来隐藏 TCP 和 UDP 连接。这个技术使得在常规的网络监控工具中，某些 TCP/UDP 连接可以“消失”，从而增加恶意软件的隐蔽性。

1. **网络 I/O 控制函数的截取**

   R77 通过截取 NtDeviceIoControlFile 函数来控制网络 I/O 请求。这个函数是 Windows 操作系统中处理设备 I/O 控制请求的核心函数，通常用于与硬件驱动程序进行通信。对于 TCP/UDP 连接的隐藏，特别是当应用程序或系统请求网络连接数据时，R77 会拦截该函数。

   在处理 NtDeviceIoControlFile 函数时, R77 检查传递的 I/O 控制代码, 特别是针对 TCP 和 UDP 连接数据的查询请求（如查询
   Device
   Nsi 驱动的连接信息）。这使得 R77 能够识别出哪些请求是关于网络连接的，并且有机会修改这些请求的返回数据。

2. **状态和 I/O 控制代码的检查**

   当 NtDeviceIoControlFile 函数被调用时，R77 首先执行原始操作，确保正常处理 I/O 请求。然后，它会检查返回的状态码，确认操作是否成功执行，尤其是与 TCP/UDP 连接列表相关的请求。

   R77 在判断是否是查询 TCP/UDP 连接数据时，分析 I/O 控制代码。如果检测到是网络连接数据查询请求，R77 才会继续干预，避免在其他 I/O 请求下造成不必要的影响。

3. **处理连接数据**

   如果请求的是 TCP 或 UDP 连接数据，R77 会调用 FilterNetworkConnections 函数，对这些连接数据进行处理。

   - 遍历和判断：FilterNetworkConnections 函数会遍历所有当前的网络连接信息，逐个检查每个连接的特征（如源地址、目标地址、端口号、协议类型等）。根据预定的规则（如 IP、端口号、连接状态等），R77 决定是否将某个连接标记为需要隐藏。
   - 隐藏判断：通常，R77 会选择隐藏特定的连接，例如那些涉及恶意活动的连接或需要避免被监控的连接。通过这种方法，R77 能够有效控制哪些连接出现在网络查询工具（如任务管理器、netstat 等）中。

4. **隐藏特定连接**

   一旦 R77 识别出需要隐藏的连接，它会采取措施确保这些连接不出现在返回的连接数据列表中。为了实现这一点，R77 会重新排列剩余连接的数据。

   - 移动连接数据：对于每个被隐藏的连接，R77 会将后续的连接数据向前移动，覆盖掉被隐藏的连接。这意味着隐藏的连接将从数据列表中完全消失。
   - 更新连接计数：为了保持数据的一致性，R77 会调整连接数据的计数。隐藏连接后，返回给调用者的连接数将会被相应地减少，以确保没有不一致的结果。

3.7 Install 分析

我们先简单进行静态分析：

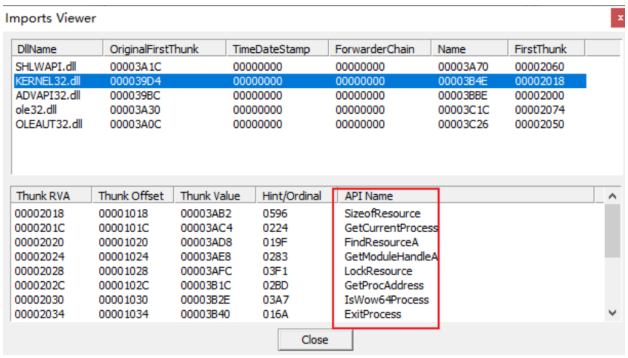

图 3.23: 查看导入表

我们发现了许多寻找和加载资源节的函数，接下来我们用 IDA Pro 来分析：

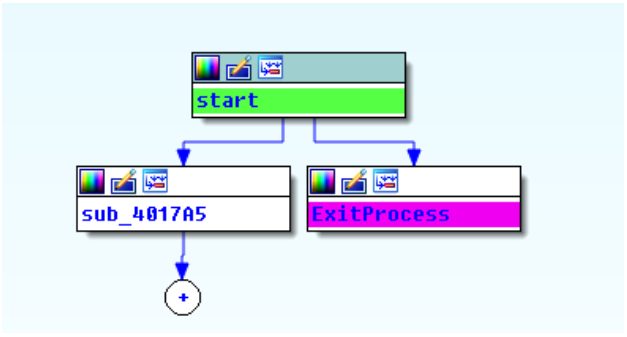

图 3.24: IDA 分析

我们接下来着重去分析 sub_4017A5 函数:

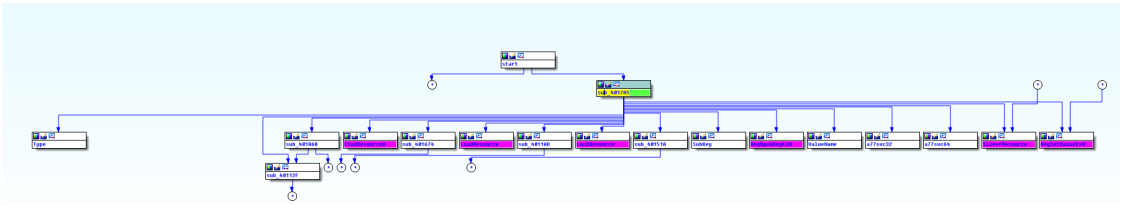

图 3.25: 函数调用关系

我们来分析具体内容：

```
.text:004017AA                push    esi
.text:004017AB                push    offset Type     ; "EXE"
.text:004017B0                push    65h             ; lpName
.text:004017B2                xor     ebx, ebx
.text:004017B4                push    ebx             ; hModule
.text:004017B5                call    ds:FindResourceA
.text:004017BB                mov     esi, eax
.text:004017BD                test    esi, esi
.text:004017BF                jz      loc_401862
.text:004017C5                push    edi
.text:004017C6                push    esi             ; hResInfo
.text:004017C7                push    ebx             ; hModule
.text:004017C8                call    ds:SizeofResource
.text:004017CE                mov     edi, eax
.text:004017D0                test    edi, edi
.text:004017D2                jz      loc_401861
.text:004017D8                push    esi             ; hResInfo
.text:004017D9                push    ebx             ; hModule
.text:004017DA                call    ds:LoadResource
.text:004017E0                test    eax, eax
.text:004017E2                jz      short loc_401861
.text:004017E4                push    eax             ; hResData
.text:004017E5                call    ds:LockResource
.text:004017EB                mov     esi, eax
.text:004017ED                lea     eax, [ebp+phkResult]
.text:004017F0                push    eax             ; phkResult
.text:004017F1                push    0F013Fh         ; samDesired
.text:004017F6                push    ebx             ; ulOptions
.text:004017F7                push    offset SubKey   ; "SOFTWARE"
.text:004017FC                push    80000002h       ; hKey
.text:00401801                call    ds:RegOpenKeyExW
.text:00401807                test    eax, eax
```

图 3.26: sub_4017A5 函数

我们发现很多对其资源节进行加载的函数,还有用 RegOpenKeyExW 打开注册表键 HKEY\LOCAL\MACHINE, 然后调用了 sub_401868 函数，其中包括很多 powerShell 相关的内容，接下来我们重点分析以下调用的函数:

- sub_401868

```
.text:00401868 sub_401868     proc near               ; CODE XREF: sub_4017A5+7D↑p
.text:00401868                push    ebx
.text:00401869                push    esi
.text:0040186A                push    edi
.text:0040186B                push    8000h           ; dwBytes
.text:00401870                push    0               ; dwFlags
.text:00401872                call    ds:GetProcessHeap
.text:00401878                push    eax             ; hHeap
.text:00401879                call    ds:HeapAlloc
.text:0040187F                mov     ebx, offset asc_40222C ; "\""
.text:00401884                mov     esi, eax
.text:00401886                push    ebx
.text:00401887                push    esi
.text:00401888                call    ds:StrCpyW
.text:0040188E                call    sub_401159
.text:00401893                mov     edi, ds:StrCatW
.text:00401899                test    eax, eax
.text:0040189B                jz      short loc_4018C5
.text:0040189D                push    offset aFunctionLocal1G ; "Function Local:Get-Delegate{P
.text:004018A2                push    esi
.text:004018A3                call    edi ; StrCatW
.text:004018A5                call    sub_40112F
.text:004018AA                test    eax, eax
.text:004018AC                jz      short loc_4018B5
.text:004018AE                push    offset aRuntime_intero ; "[Runtime.InteropServices.Mars
.text:004018B3                jmp     short loc_4018BA
```

图 3.27: sub_401868

1. **堆内存分配和字符串复制**

   GetProcessHeap 和 HeapAlloc 被用来从当前进程的堆空间中分配一块内存。这样做可以提供一个动态的内存区域，用于存放 PowerShell 命令。StrCpyW 函数被调用，将起始的字符串复制到这块新分配的内存中，作为命令的初始部分。

2. **构建 PowerShell 命令**

   – 使用 [Reflection.Assembly]::Load 加载命令：install.exe 的最终目的是构造一个 PowerShell 命令，通过反射加载.NET stager 到内存中。这一过程无需写入磁盘，从而减少了被静态检测发现的可能性。

   – 反射加载：.NET stager 通常用于加载后续恶意组件，例如恶意的.NET 程序集，直接在

内存中执行，而不经过文件系统。这种方法借助 PowerShell 的反射功能，可以直接执行恶意代码。

3. **AMSI 规避**

   Microsoft 的反恶意软件扫描接口（AMSI）是 Windows 的一项安全特性，它在加载和执行脚本之前扫描潜在的恶意内容。AMSI 的目标是拦截恶意脚本或宏攻击。实验中，install.exe 通过修改 AmsiScanBuffer API 的返回值，使其始终返回 AMSI_RESULT_CLEAN（表示内容安全）。这种方法确保即使 PowerShell 脚本包含恶意内容，AMSI 也不会触发任何警报，因为它会假装扫描未发现威胁。

4. **PowerShell 命令的混淆**

   - 子函数调用（如 sub_401986）：install.exe 多次调用 sub_401986 函数以实现对命令的混淆。
   - 混淆实现：通过将命令中的变量名称替换为随机字符串，install.exe 可以增加命令的复杂性，使其更难以解析。混淆的过程可以掩盖命令的真正意图，并规避基于特征的检测机制。
   - 隐蔽性增强：随机化变量名称和整体结构，使得即使命令被截获，静态分析工具也更难识别出该命令的作用。

- sub_401674

```
00401674 ; int __thiscall sub_401674(OLECHAR *psz)
00401674 sub_401674      proc near               ; CODE XREF: sub_4017A5+88↓p
00401674                                         ; sub_4017A5+97↓p
00401674
00401674 pvarg           = VARIANTARG ptr -20h
00401674 var_10          = dword ptr -10h
00401674 var_C           = dword ptr -0Ch
00401674 ppv             = dword ptr -8
00401674 bstrString      = dword ptr -4
00401674
00401674                 push    ebp
00401675                 mov     ebp, esp
00401677                 sub     esp, 20h
0040167A                 push    ebx
0040167B                 push    esi
0040167C                 mov     esi, ds:SysAllocString
00401682                 xor     ebx, ebx
00401684                 push    edi
00401685                 push    ecx             ; psz
00401686                 call    esi ; SysAllocString
00401688                 push    offset asc_40218C ; "\\"
0040168D                 mov     [ebp+bstrString], eax
00401690                 call    esi ; SysAllocString
00401692                 push    ebx             ; dwCoInit
00401693                 mov     edi, eax
00401695                 push    ebx             ; pvReserved
00401696                 mov     [ebp+var_10], edi
00401699                 call    ds:CoInitializeEx
0040169F                 test    eax, eax
004016A1                 js      loc_401782
004016A7                 push    ebx             ; pReserved3
004016A8                 push    ebx             ; dwCapabilities
004016A9                 push    ebx             ; pAuthList
004016AA                 push    3               ; dwImpLevel
004016AC                 push    6               ; dwAuthnLevel
```

图 3.28: sub_401674 函数

1. **双重系统兼容性调用**

   - 双重调用策略：sub_401674 先进行 32 位系统的相关调用，再对 64 位系统执行相同的操作。这样设计的目的是保证无论在 32 位还是 64 位系统中，该函数都能正常运行。
   - 系统架构检测：函数通常会判断系统架构，然后分别调用相应的代码路径，以适配不同的系统环境。

2. **COM 对象操作的核心**

- COM 环境初始化：函数在主体操作之前执行 COM 环境的初始化。这通常包括调用 CoInitialize 或 CoInitializeEx 函数，以确保 COM 库已启动，并可以为后续的 COM 对象操作提供基础。
- ppv+n 操作：在操作中涉及到对 ppv+n 的处理，ppv 是一个指向 COM 接口的指针，n 代表偏移量。通过调整偏移量，函数可以访问和调用 COM 对象的特定方法或属性，以完成某些操作。

- **sub_40112F**

```
0040112F sub_40112F      proc near               ; CODE XREF: sub_4017A5+9C↓p
0040112F                                         ; sub_401868+3D↓p
0040112F
0040112F Wow64Process    = dword ptr -4
0040112F
0040112F                 push    ebp
00401130                 mov     ebp, esp
00401132                 push    ecx
00401133                 push    esi
00401134                 lea     eax, [ebp+Wow64Process]
00401137                 xor     esi, esi
00401139                 push    eax             ; Wow64Process
0040113A                 mov     [ebp+Wow64Process], esi
0040113D                 call    ds:GetCurrentProcess
00401143                 push    eax             ; hProcess
00401144                 call    ds:IsWow64Process
0040114A                 test    eax, eax
0040114C                 jz      short loc_401154
0040114E                 cmp     [ebp+Wow64Process], esi
00401151                 jz      short loc_401154
00401153                 inc     esi
00401154
00401154 loc_401154:                             ; CODE XREF: sub_40112F+1D↑j
00401154                                         ; sub_40112F+22↑j
00401154                 mov     eax, esi
00401156                 pop     esi
00401157                 leave
00401158                 retn
00401158 sub_40112F      endp
00401158
00401159
```

图 3.29: sub_40112F 函数

1. 起始获取句柄：在 sub_40112F 的起始阶段，函数调用 GetCurrentProcess() 以获得当前进程的特殊句柄。这个句柄不是普通的句柄，而是一个代表执行该函数的进程的伪句柄。
2. 系统架构检测：然后函数利用 IsWow64Process() 来判断当前进程是否在 WoW64（Windows32 位在 Windows64 位上的子系统）环境下运行。这个检测过程需要两个参数：一是进程的句柄，二是一个指向布尔类型变量的指针，用于存储检测结果。
3. 架构相关决策: 根据 IsWow64Process() 的返回结果，函数能够判断出操作系统是 32 位还是 64 位。这个判断对于后续操作至关重要，因为它决定了下一个函数调用时传递的参数是”$77svc32” 还是”$77svc64”，这两个参数分别对应 32 位和 64 位服务。

- **sub_4011AD**

```
xt:004011AD             push    ebp
xt:004011AE             mov     ebp, esp
xt:004011B0             sub     esp, 84h
xt:004011B6             push    ebx
xt:004011B7             push    esi
xt:004011B8             mov     esi, ds:SysAllocString
xt:004011BE             xor     ebx, ebx
xt:004011C0             push    edi
xt:004011C1             push    ecx             ; psz
xt:004011C2             call    esi ; SysAllocString
xt:004011C4             push    offset psz      ; psz
xt:004011C9             mov     [ebp+bstrString], eax
xt:004011CC             call    esi ; SysAllocString
xt:004011CE             mov     edi, eax
xt:004011D0             push    offset aPowershell ; "powershell"
xt:004011D5             mov     [ebp+var_34], edi
xt:004011D8             call    esi ; SysAllocString
xt:004011DA             push    [ebp+psz]       ; psz
xt:004011DD             mov     [ebp+var_38], eax
xt:004011E0             call    esi ; SysAllocString
xt:004011E2             push    offset asc_40218C ; "\\"
xt:004011E7             mov     [ebp+var_3C], eax
xt:004011EA             call    esi ; SysAllocString
xt:004011EC             push    offset aSystem  ; "SYSTEM"
xt:004011F1             mov     [ebp+var_40], eax
xt:004011F4             call    esi ; SysAllocString
xt:004011F6             push    ebx             ; dwCoInit
xt:004011F7             push    ebx             ; pvReserved
xt:004011F8             mov     [ebp+var_44], eax
xt:004011FB             call    ds:CoInitializeEx
xt:00401201             test    eax, eax
xt:00401203             js      loc_4014F0
xt:00401209             push    ebx             ; pReserved3
xt:0040120A             push    ebx             ; dwCapabilities
```

图 3.30: sub_4011AD 函数

1. 参数分析：在深入函数内部之前，首先关注其关键参数。我们开到调用前传入了三个参数，分别是与操作系统的位数（32 位或 64 位）直接相关，一个整型数值，最后一个则是之前经过混淆处理的 PowerShell 命令字符串。
2. COM 对象的应用: 函数核心部分涉及到 COM（组件对象模型）对象的使用。它主要通过调用 ppv + 40 等位置的函数来实现其功能。这些 COM 对象的调用是函数执行的关键部分。
3. PowerShell 命令执行: 根据传入的参数和函数开头构造的”PowerShell”字符串，可以推测该函数的主要作用是通过 COM 对象来执行 PowerShell 指令。这种执行方式可能与系统的位数和特定的 PowerShell 命令有关。

- **sub_40151A**

```
xt:0040151A             push    ebp
xt:0040151B             mov     ebp, esp
xt:0040151D             sub     esp, 38h
xt:00401520             push    ebx
xt:00401521             push    esi
xt:00401522             mov     esi, ds:SysAllocString
xt:00401528             xor     ebx, ebx
xt:0040152A             push    edi
xt:0040152B             push    ecx             ; psz
xt:0040152C             call    esi ; SysAllocString
xt:0040152E             mov     edi, eax
xt:00401530             push    offset asc_40218C ; "\\"
xt:00401535             mov     [ebp+var_14], edi
xt:00401538             call    esi ; SysAllocString
xt:0040153A             push    ebx             ; dwCoInit
xt:0040153B             push    ebx             ; pvReserved
xt:0040153C             mov     [ebp+bstrString], eax
xt:0040153F             call    ds:CoInitializeEx
xt:00401545             test    eax, eax
xt:00401547             js      loc_40165F
xt:0040154D             push    ebx             ; pReserved3
xt:0040154E             push    ebx             ; dwCapabilities
xt:0040154F             push    ebx             ; pAuthList
xt:00401550             push    3               ; dwImpLevel
xt:00401552             push    6               ; dwAuthnLevel
xt:00401554             push    ebx             ; pReserved1
xt:00401555             push    ebx             ; asAuthSvc
xt:00401556             push    0FFFFFFFFh      ; cAuthSvc
xt:00401558             push    ebx             ; pSecDesc
xt:00401559             call    ds:CoInitializeSecurity
xt:0040155F             test    eax, eax
xt:00401561             jns     short loc_40156E
xt:00401563             cmp     eax, 80010119h
xt:00401568             jnz     loc_401659
```

图 3.31: sub_40151A 函数

我们看到这个函数主要是调用 COM 对象的相关函数。

所以，通过简单查看我们得出结论：Install.exe 通过使用资源节的.Net stager 文件，加载后绕过系统监测。最终目的是让其注册服务并加载进内存。

3.8 Helpers.dll 分析

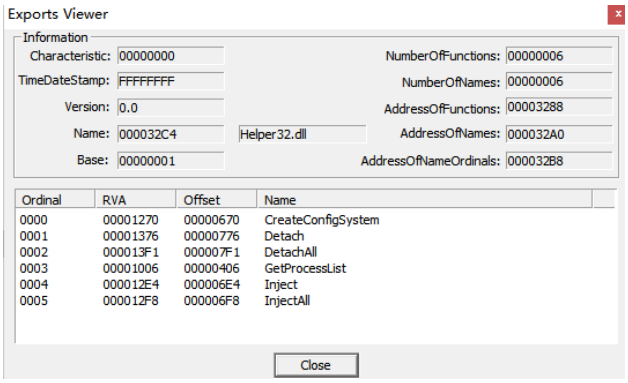

图 3.32: Helper.dll 导出表

我们看到了 CreateConffgSystem 来配置必要环境以及 DetachAll 与 InjectAll 用于卸载和注入恶意行为。

1. **配置系统的建立**

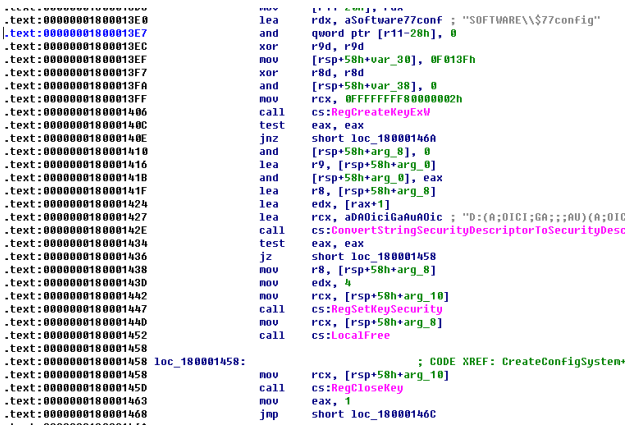

图 3.33: CreateConfgSystem

看到服务模块首先在注册表中建立配置系统。它在 HKEY_LOCAL_MACHINE\SOFTWARE\$77confg 下创建一个键值，这个键值可以被计算机上的任何用户修改。接着，服务模块将当前运行的进程 ID 作为值存储在 HKEY_LOCAL_MACHINE\SOFTWARE\$77confg\pid 下的 svc32 或 svc64 注册表项中，这取决于系统的架构类型。

2. **核心 Rootkit 的注入**

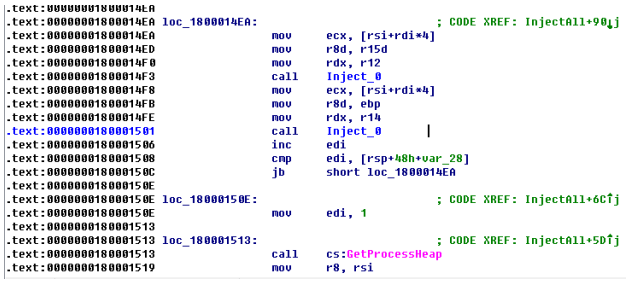

图 3.34: InjectAll

从上图我们清楚地看到两个回调：

(a) 第一个回调负责向所有运行中的进程注入 Rootkit 核心，通过每 100 毫秒枚举所有进程来实现。

(b) 第二个回调负责向新创建的、已被感染父进程的子进程注入。服务模块利用进程间通信捕获子进程的创建，并在条件允许时进行注入。

3.9 总结 Rootkit 核心的作用

根据上述分析，以及我查找资料，最终发现核心 Rootkit 的主要作用是在关键的 WindowsAPI 上安装钩子，并根据 Rootkit 的配置过滤这些 API 的输出，它们包括 NtQuerySystemInformation、NtResumeThread、NtQueryDirectoryFile 等 API 函数，它们通常用于获取系统信息。通过在这些 API 上设置挂钩，核心模块能够有选择性地对系统用户和安全工具隐藏特定的文件、进程或注册表项。

3.10 Detours 与 R77 结合

在与 R77 结合时，Detours 机制可以用来实现一系列攻击功能，通过修改目标程序的执行流来进行恶意行为。

1. **在目标程序中植入特定的恶意代码，改变程序的执行逻辑**

   - **拦截关键 API 函数**：R77 可以选择拦截目标程序的特定函数或系统调用，插入自定义的恶意代码。例如，R77 可以拦截 CreateFile 和 WriteFile 函数，修改文件的读取和写入操作，从而获取或篡改文件内容。
   - **跳转到恶意代码**：通过 Detours，在目标程序的函数入口处插入一个跳转指令，将执行流重定向到自定义的恶意代码。例如，攻击者可以植入恶意代码来窃取敏感信息、下载更多恶意软件，或者修改程序的内部状态。
   - **绕过检测机制**：通过插入的恶意代码，R77 可以对目标程序的反病毒和检测机制进行规避。例如，拦截对文件系统的操作，使恶意文件不会被发现，或者通过注入恶意代码来执行隐藏操作（如注入自身到内存中）。

2. **通过拦截和修改系统调用，获取敏感数据或者提升权限**

   - **拦截敏感系统调用**：R77 可以利用 Detours 拦截如 ReadFile、WriteFile、RegOpenKey、CreateProcess 等系统调用，获取访问的文件、注册表或其他敏感资源。通过修改这些系统调用，R77 可以收集目标程序的敏感数据，或者阻止程序正常执行某些操作。

- **权限提升**：通过拦截与权限控制相关的调用，如 OpenProcess 或 AdjustTokenPrivileges，R77 可以修改目标程序的权限或注入恶意代码进入其他更高权限的进程，从而实现权限提升。

3. **将自身隐藏在目标程序中，避免检测和分析**

- **隐藏自身进程**：R77 可以通过拦截 EnumProcesses 或 CreateToolhelp32Snapshot 等系统调用，过滤掉其自身的进程信息，使其在任务管理器中不可见。通过修改系统调用返回的数据，R77 可以让自己完全隐形，从而避免被管理员发现。
- **防止调试**：通过修改目标程序的调试相关函数，如 IsDebuggerPresent、CheckRemoteDebuggerPresent 等，R77 可以让目标程序认为自己没有在调试环境中运行，从而避免被调试工具（如 OllyDbg、x64dbg 等）检测到。
- **内存隐藏**：R77 可以将恶意代码注入到目标程序的内存中，并通过 Detours 隐藏其在内存中的存在。例如，R77 可以拦截 VirtualAlloc 和 VirtualProtect 系统调用，控制内存分配和保护属性，将其代码隐藏在受保护的内存区域中，或者使代码不可执行，从而避免被检测到。
- **绕过沙箱分析**：通过拦截沙箱检测机制的 API 函数（如 GetSystemInfo、GetTickCount 等），R77 可以避免被动态沙箱环境（如 Cuckoo Sandbox）识别。通过操控这些系统调用，R77 可以伪装成正常的系统环境，隐藏恶意行为的存在。

4 实验结论及心得体会

4.1 实验结论

本次实验我们不仅完成了对恶意代码 Rootkit 的功能进行了验证全部实现了隐藏进程、文件、注册表和网络连接的功能。同时我们也对其组成的文件进行了简单的逆向分析，对该恶意代码有了更深的了解。

r77 的无文件设计使其更难被检测，而持久性机制确保它能够在系统重新启动后继续运行，增加了对抗安全防御的难度。

通过运行 Install.exe 实现 r77 的注入和持久性，且只需一个单独的可执行文件，使得部署相对简便。

r77 采用了多种隐藏技术，包括文件系统、进程、注册表和网络连接的隐藏，通过前缀、ID、名称等方式实现了对多个系统实体的隐匿。

r77 的设计考虑了对抗检测的因素，通过隐藏文件和进程，以及修改网络连接的可见性，增加了检测和清除的难度。

总的来说，R77 rootkit 是一个高度复杂和灵活的工具，它利用了 Windows API 的钩子技术和其他一些高级技术来隐藏系统元素，从而使恶意软件能够在系统中持久存在并避免被检测。

4.2 心得体会

本次实验，我认真的了解了 R77 rootkit 的一些列的功能，感觉真的很难被发现，在一开始的进程隐藏就可以看到有些工具根本无法检测到，确实是很惊讶。后续通过对文件、协议等内容的隐藏，我更是感受到它丰富的功能，其实 R77 的核心是反注射 dll 注入。一旦注入到进程中，对应进程就不会显示被隐藏相关信息。文件被写入远程进程内存，并调用 ReflectiveDllMain 导出以最终加载 DLL 并调用 DllMain。因此，DLL 不会在 PEB 中列出。通过我们的简单分析，我也能感受到这种复杂的恶

意代码背后的原理，他和我们之前做的实验相比更加的复杂。未来我也将继续积累经验，能多将理论知识应用到实战中来。