



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 12 月 1 日

# 目录

<b>1 实验目的</b>	<b>2</b>
<b>2 实验原理</b>	<b>2</b>
2.1 Rootkit . . . . .	2
2.2 Rootkit77 . . . . .	2
2.3 Windows 的 Detours 机制 . . . . .	3
<b>3 实验过程</b>	<b>3</b>
3.1 Install 分析 . . . . .	3
3.2 资源节功能分析 . . . . .	9
3.3 Helper32.dll 分析 . . . . .	9
3.4 分析总结 . . . . .	11
3.5 Detours 与 R77 结合 . . . . .	11
<b>4 实验结论及心得体会</b>	<b>12</b>
4.1 实验结论 . . . . .	12
4.2 心得体会 . . . . .	12

## 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，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。

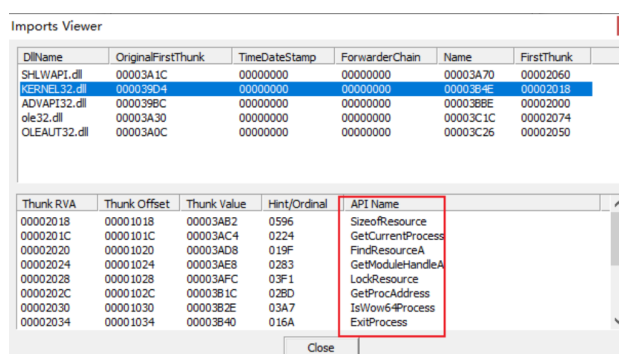
## 2.3 Windows 的 Detours 机制

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 Install 分析

我们先简单进行静态分析:



DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
SHLWAPI.dll	00003A1C	00000000	00000000	00003A70	00002060
KERNEL32.dll	00002804	00000000	00000000	00003B4E	00002018
ADVAPI32.dll	000039BC	00000000	00000000	00003B8E	00002000
ole32.dll	00003A30	00000000	00000000	00003C1C	00002074
OLEAUT32.dll	00003A0C	00000000	00000000	00003C26	00002050

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00002018	00001018	00003AB2	0596	SizeofResource
0000201C	0000101C	00003AC4	0224	GetCurrentProcess
00002020	00001020	00003AD8	019F	FindResourceA
00002024	00001024	00003AE8	0283	GetModuleHandleA
00002028	00001028	00003AFC	03F1	LockResource
0000202C	0000102C	00003B1C	028D	GetProcAddress
00002030	00001030	00003B2E	03A7	IsWow64Process
00002034	00001034	00003B40	016A	ExitProcess

图 3.1: 查看导入表

我们发现了许多寻找和加载资源节的函数:

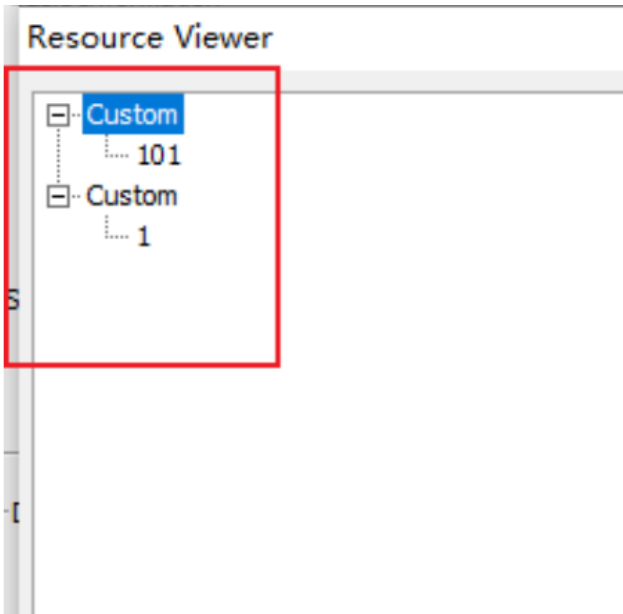


图 3.2: 查看资源节

接下来我们用 IDA Pro 来分析：

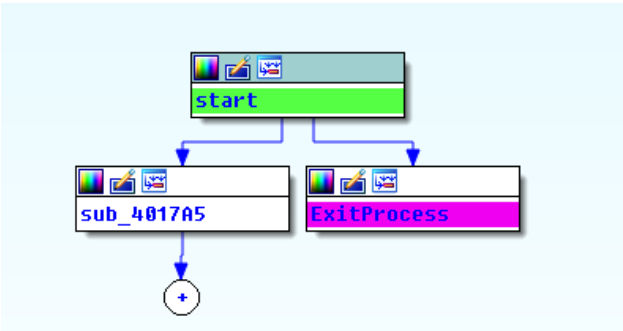


图 3.3: IDA 分析

我们接下来着重去分析 sub\_4017A5 函数:

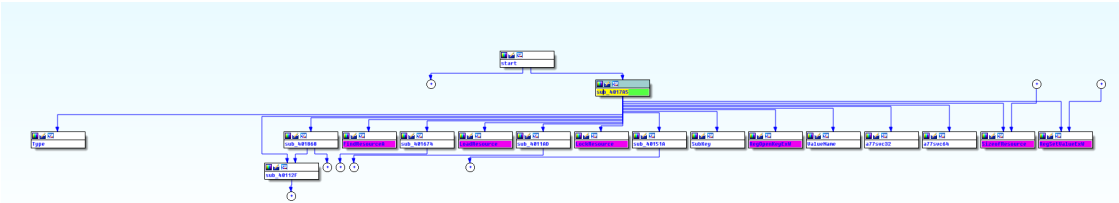


图 3.4: 函数调用关系

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:004017B8      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 3.5: 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+701p
.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 aFunctionLocalG ; "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.Harsl
.text:004018B3      jmp      short loc_4018B8

```

图 3.6: 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。这样做可以提供是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+8B4p
00401674 ; sub_4017A5+974p
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 ; pReserved
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 3.7: 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_401705+9C↓p
0040112F ; sub_401868+3D↓p
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+10f↓
00401154 ; sub_40112F+22f↓
00401154 mov eax, esi
00401156 pop esi
00401157 leave
00401158 retn
00401158 sub_40112F endp
00401158
00401159

```

图 3.8: 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
xt:004011C2      call     esi ; SysAllocString
xt:004011C4      push     offset 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]
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
xt:004011F7      push     ebx
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
xt:0040120A      push     ebx

```

图 3.9: 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\SYSTEM" 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
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:0040153B      push     ebx
xt:0040153D      push     ebx
xt:0040153F      mov      [ebp+bstrString], eax
xt:00401545      call     ds:CoInitializeEx
xt:00401547      test     eax, eax
xt:0040154F      js       loc_40165F
xt:00401550      push     ebx
xt:00401551      push     ebx
xt:00401552      push     ebx
xt:00401553      push     3
xt:00401554      push     6
xt:00401555      push     ebx
xt:00401556      push     0FFFFFFFh
xt:00401558      push     ebx
xt:00401559      call     ds:CoInitializeSecurity
xt:0040155F      test     eax, eax
xt:00401561      jns      short loc_40156E
xt:00401563      cmp      eax, 80010119h
xt:00401568      inc      eax

```

图 3.10: 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。**

### 3.2 资源节功能分析

通过浏览 R77 的文档并根据上述我们在 IDA pro 和使用其他工具的分析，我们最终了解了部分资源节功能：

- **DLL 挂钩移除：**Stager 的一项重要功能是移除 DLL 挂钩。这通常是指还原安全软件对关键系统 DLL（如 NTDLL.dll 或 KERNEL32.dll）的修改，这些修改通常用于监控或截获系统调用。此操作旨在避免恶意行为被安全工具检测到。
- **调整系统权限：**Stager 会修改系统权限，例如启用 SeDebugPrivilege 权限。这种权限是 Windows 系统中的高级特权，允许进程访问或操作其他进程的内存空间，通常用于执行诸如进程注入的复杂恶意操作。
- **服务模块的解密与解压缩：**Stager 会对加载的服务模块进行解密和解压操作，这些模块可能包含实施恶意活动的核心代码。
- **服务模块注入：**在完成解密和解压后，Stager 会将服务模块注入到指定目标进程中，从而使恶意代码能够在受害系统上以隐秘方式执行。
- **进程空洞化技术：**Stager 使用了“进程空洞化”的手段。该方法通过创建一个合法的系统进程，将其内存空间清空后填入恶意代码，从而实现利用合法进程伪装以执行恶意行为。
- **父进程 ID 欺骗：**为了进一步掩盖其活动，Stager 可能使用父进程 ID（PPID）欺骗技术，将自己伪装成合法进程的子进程来启动。这种方式有助于规避基于行为的安全监控。

综合来看，Stager 的核心任务包括移除 DLL 挂钩、调整 SeDebugPrivilege 权限、解密并解压服务模块，同时将这些模块注入到目标进程（32 位或 64 位）中，并通过多种隐蔽技术规避检测。

### 3.3 Helper32.dll 分析

首先我们使用 PEiD 来查看导出表的重要函数：

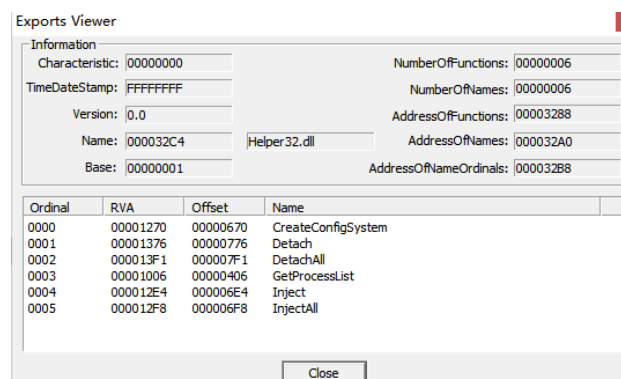


图 3.11: Helper.dll 导出表

通过观察导出表内容，我们观察到一些特定的导出函数，CreateConfgSystem 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

```

.text:000000001800013E0
.text:000000001800013E7
.text:000000001800013EC
.text:000000001800013EF
.text:000000001800013F7
.text:000000001800013FA
.text:000000001800013FF
.text:00000000180001406
.text:0000000018000140C
.text:0000000018000140E
.text:00000000180001410
.text:00000000180001416
.text:0000000018000141B
.text:0000000018000141F
.text:00000000180001424
.text:00000000180001427
.text:0000000018000142E
.text:00000000180001434
.text:00000000180001436
.text:00000000180001438
.text:0000000018000143D
.text:00000000180001442
.text:00000000180001447
.text:0000000018000144D
.text:00000000180001452
.text:00000000180001458
.text:00000000180001458 loc_180001458:
.text:00000000180001458
.text:0000000018000145D
.text:00000000180001463
.text:00000000180001468
.....
mov     rcx, aSoftware77conf ; "SOFTWARE\\$77conf"
lea     rdx, aSoftware77conf
and     dword ptr [r11-28h], 0
xor     r9d, r9d
mov     [rsp+58h+var_30], 0F013Fh
xor     r8d, r8d
and     [rsp+58h+var_38], 0
mov     rcx, 0FFFFFFF8000002h
call    cs:RegCreateKeyExW
test    eax, eax
jnz     short loc_18000146A
and     [rsp+58h+arg_8], 0
lea     r9, [rsp+58h+arg_0]
and     [rsp+58h+arg_0], eax
lea     r8, [rsp+58h+arg_8]
lea     edx, [rax+1]
lea     rcx, aD0iciGaAu0ic ; "D:(A;OICI;GA;;;AU)(A;OIC
call    cs:ConvertStringSecurityDescriptorToSecurityDesc
test    eax, eax
jz      short loc_180001458
mov     r8, [rsp+58h+arg_8]
mov     edx, 0
mov     rcx, [rsp+58h+arg_10]
call    cs:RegSetKeySecurity
mov     rcx, [rsp+58h+arg_8]
call    cs:LocalFree
; CODE XREF: CreateConfigSystem+
mov     rcx, [rsp+58h+arg_10]
call    cs:RegCloseKey
mov     eax, 1
jmp     short loc_18000146C

```

图 3.12: 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

```

.text:000000001800014EA
.text:000000001800014EA loc_1800014EA:
.text:000000001800014EA
.text:000000001800014F0
.text:000000001800014F3
.text:000000001800014F8
.text:000000001800014FB
.text:000000001800014FE
.text:00000000180001501
.text:00000000180001506
.text:00000000180001508
.text:0000000018000150C
.text:0000000018000150E
.text:0000000018000150E loc_18000150E:
.text:0000000018000150E
.text:00000000180001513
.text:00000000180001513 loc_180001513:
.text:00000000180001519
mov     ecx, [rsi+rdi*4]
mov     r8d, r15d
mov     rdx, r12
call    Inject_0
mov     ecx, [rsi+rdi*4]
mov     r8d, ebp
mov     rdx, r14
call    Inject_0
inc     edi
cmp     edi, [rsp+48h+var_28]
jb      short loc_1800014EA
; CODE XREF: InjectAll+904j
mov     edi, 1
; CODE XREF: InjectAll+6Cfj
; CODE XREF: InjectAll+50fj
call    cs:GetProcessHeap
mov     r8, rsi

```

图 3.13: InjectAll

从上图我们清楚地看到两个回调：

- 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。
- 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. 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。

### 3.4 分析总结

R77 是一个开源的 Ring 3 级别 rootkit，主要功能是隐藏系统中的文件、目录、进程、CPU 使用率、注册表键值、服务、网络连接、命名管道和计划任务等元素。其核心技术是通过钩子修改 Windows API 的行为，实现对特定系统元素的隐藏。

我们看到，R77 的结构由四个主要模块组成：安装器、阶段模块、服务模块和核心模块。

1. **安装器模块**：主要任务是将阶段模块的 PE 文件存储在注册表中，这是一种常见的持久化技术。随后，它生成一个 PowerShell 命令以从注册表加载并执行阶段模块。安装器还通过创建计划任务来确保该命令定期执行。
2. **阶段模块**：负责将服务模块和核心模块注入到系统中。
3. **服务模块**：以 Windows 服务的形式运行，其任务是将核心模块注入所有活动进程。
4. **核心模块**：作为 rootkit 的核心，通过钩子技术修改 Windows API，实现对系统元素的全面隐藏。

此外，R77 还提供动态配置功能，允许通过进程 ID 或名称隐藏进程，基于完整路径隐藏文件或目录，或屏蔽特定端口的网络连接。配置数据存储在注册表路径 `HKEY_LOCAL_MACHINE\SOFTWARE\$77config` 下，并且无需权限提升即可修改。

同时，R77 使用了一些规避安全工具的技术。例如：

- **AMSI 绕过**：通过修改 `amsi.dll!AmsiScanBuffer` 函数，使其始终返回干净的结果。
- **DLL 反钩子**：重新加载 `ntdll.dll` 的未修改版本以恢复原始 API 行为。

综上，R77 通过利用 Windows API 钩子和多种高级技术，实现对系统元素的隐藏，并提高恶意软件在系统中的生存能力，同时规避常见的安全防护措施。

### 3.5 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

本次实验我们主要在上一次实验的基础上, 对 R77 的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, 通过简单分析, 我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, 相比上一次简单的使用了 R77, 这次我们在实验中主要是详尽的分析了 R77 背后的原理和功能, 以及其实现的逻辑, 这都让我对这些而一代吗有了更深层次的理解。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。