华中科技大学

课 程 设 计 报 告

基于 API HOOK 的软件行为分析系统

课　　程　软件安全课程设计

院　　系　网络空间安全学院

专业班级　信安 1902

学　　号　U201910210

姓　　名　徐铭睿

指导教师　梅松

2021 年 10 月 04 日

目　录

1　课程设计任务书

1.1　课程设计目的

基于 API HOOK 的软件行为分析系统，通过对 detours 库的使用，实现 windows API 截获，加深对软件安全理论课程的理解。

1.2　课程设计要求

对于无源码情况下分析样本程序的行为，有多种方法。本次课程设计是利用 Detours 开源项目包提供的接口，完成基本的程序行为分析。具体课程设计任务见表 1。任务主要分为 API 调用截获及分析两大部分。任务可以自己选择 Windows（含 Win7/win8/win10）平台上实现,编程用语言为 C 或者 C++。其中使用的 Windows 平台使用微软的 Detours 开源库，可以在 VS2019 环境下编译后使用，后面会有详细介绍。

平台：Win7/Win8/Win10,VC++(VS2013/VS2015/VS2019)。

表 1 课程设计任务表

序号	任务	要求
1	实现基本的第三方进程 WindowsAPI 截获框架	框架包括 1.1 编译生成 Detours 库；1.2 完成挂钩框架 DLL，实现对 MessageBox 调用截获，能打印出调用的参数、进程名称以及进程 Exe 文件信息；1.3 自编或者利用已有恶意代码样例（包含弹出对话框动作）；1.4 完成注入动作开启和关闭的“注射器”控制程序
2	实现堆操作 API 截获	修改 1.2-1.4，实现堆操作（创建，释放）API

		进行截获，打印出所有参数信息。
3	实现文件操作 API 截获	实现对文件操作（创建，关闭，读写）API 进行截获，打印出所有参数信息。
4	注册表操作 API 截获	实现对注册表操作（创建，关闭，读写）API 进行截获，打印出所有参数信息。
5	堆操作异常行为分析	设计并完成算法，记录并给出提示： 1.检测堆申请与释放是否一致（正常）； 2.是否发生重复的多次释放（异常）
6	文件操作异常行为分析	设计并完成算法，记录并给出提示： 1. 判断操作范围是否有多个文件夹； 2. 是否存在自我复制的情况； 3. 是否修改了其它可执行代码包括 exe，dll，ocx 等； 4. 是否将文件内容读取后发送到网络（选做）；
7	注册表操作异常行为分析	设计并完成算法，记录并给出提示： 1. 判断是否新增注册表项并判断是否为自启动执行文件项； 2. 是否修改了注册表；

		3. 输出所有的注册表操作项；
8	提供系统界面	所设计实现的功能，有图形界面展示
9	行为检测样本库	提供 5 个待检测的可能存在恶意的 Exe 样本，覆盖被检测的行为；
10	网络通信操作异常行为分析（选做）	设计并完成算法，记录并给出提示： 1. 实现对网络传输 SOCKET 操作（连接、发送与接收）API 的截获； 2. 打印进程连接端口、协议类型、IP 信息 3. HTTP 连接协议的解析，判断传输的内容是否为明文
11	内存拷贝监测与关联分析（选做）	设计并完成算法，记录并给出提示： 能够输出内存拷贝信息，并分析拷贝的内容流向。

1.3 系统环境

操作系统环境:

版本 Windows 10 家庭中文版

版本号 20H2

安装日期 2021/8/21

操作系统内部版本 19042.1237

序列号 YX02S1WS

体验　Windows Feature Experience Pack 120.2212.3530.0

编译环境：VS2019

1.4　实验过程记录

按照课程设计实验指导书，首先进行了 Detours 库的编译，并完成了 dll 注射器框架的构建，之后对老师所给的 testapp 进行了 api 截获测试，并打印出了调用的参数及其他要求信息。

顺序实现了堆操作、文件操作、注册表操作的 api 分析，并使用了共享内存来解决自调用 api 截获的问题。

自行编写了五个待检测样本，提供五个可能存在恶意的 EXE 样本，并编写程序对堆操作、文件操作、注册表操作可能存在的异常行为进行分析。

最后使用 QT 编写了一个简单的图形界面展示。

2　绪言

本课程设计采用微软的开源工具库 Detours 实现 Windows API 截获（或称为绕道、挂钩）操作。Detours 是一个在 x86 平台上截获任意 Win32 函数调用的工具库。下面对 Detours 库原理进行简单介绍。

2.1　Detours 库原理

修改目标函数（一般为 Windows API 接口函数）：使用一个无条件转移指令来替换该目标函数的首部几条指令，将控制流直接转移到一个用户自己实现的截获函数（即 Detour 函数）。而原目标函数中被替换的指令被保存在一个被称为“Trampoline”函数中（译注：英文意为蹦床函数）。这些指令包括目标函数中被替换的代码以及一个重新跳转到目标函数正确位置的无条件分支。截获函数可以替换目标函数，或者通过执行“Trampoline”函数时，将目标函数作为子程序来调用的办法，在保留原来目标函数功能基础上，来完成扩展功能。

因为截获函数（Detour 函数）是执行时被插入到内存中目标函数的代码里，不是在硬盘上直接修改目标函数，所以，可以在一个很好的粒度上使得截获二进制函数的执行变得更容易。例如，一个应用程序执行时加载的 DLL 中的函数过程，可以被插入一段截获代码（detoured），与此同时，这个 DLL 还可以被其他应用程序按正常情况执行（译注：也就是按照不被截获的方式执行，因为 DLL 二进制文件没有被修改，所以发生截获时不会影响其他进程空间加载这个 DLL）。不同于 DLL 的重新链接或者静态重定向方式， Detours 库中使用的这种中断技术，确保不会影响到应用程序中的方法或者系统代码对目标函数的定位。

如果其他人为了调试或者在内部使用其他系统检测手段而试图修改二进制代码，Detours 将是一个可以普遍使用的开发包。Detours 是第一个可以在任意平台上将未修改的目标代码作为一个可以通过“trampoline”调用的子程序来保留的开发包。而以前的系统，在逻辑上预先将截获代码放到目标代码中，而不是将原始的目标代码作为一个普通的子程序来调用。独特的“trampoline”设计对于扩展现有的软件的二进制代码是至关重要的。

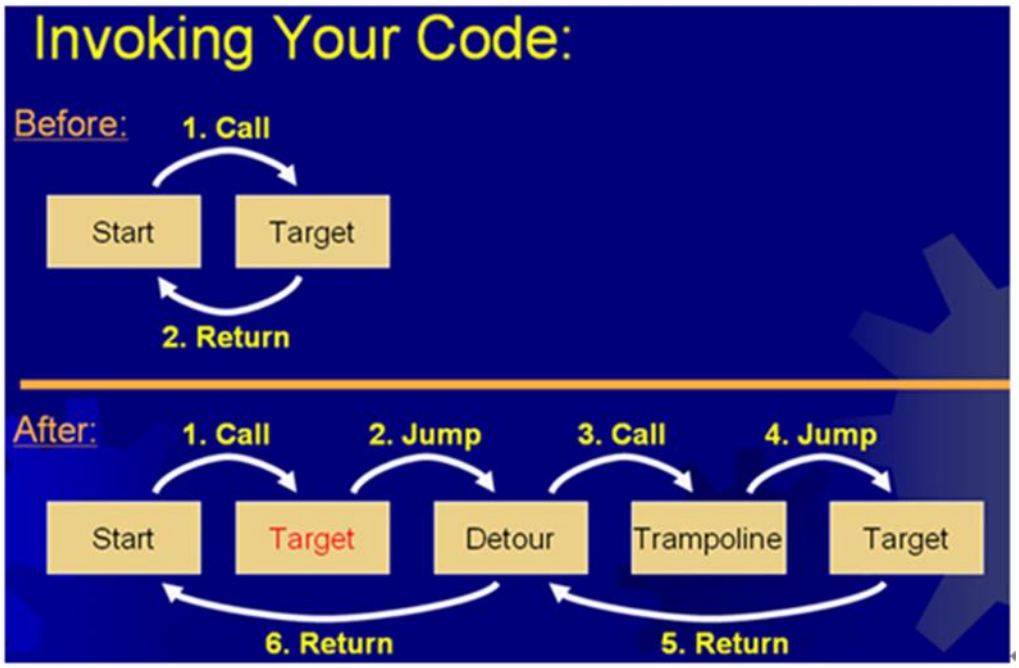
出于使用基本的函数截获功能的目的，Detours 同样提供了编辑任何 DLL 导入表的功能，

达到向已存在的二进制代码中添加任意数据节表的目的，向一个新进程或者一个已经运行着的进程中注入一个 DLL。一旦向一个进程注入了 DLL，这个动态库就可以截获任何 Win32 函数，不论它是在应用程序中或者在系统库中。

3　系统方案设计

3.1　利用 detours 库实现 Windows API 截获

Detours 在 Target 函数的开头加入 jmp Address_of_ Detour_ Function 指令（共 5 个字节），把对 Target 函数的调用引导到自己的 Detour 函数， 把 Target 函数的开头的 5 个字节(push ebp……push esi)以及 jmp Address_of_ Target _ Function+5(共 10 个字节)作为 Trampoline 函数的内容。



（图 2:　Detour 函数的调用过程）

Detours 提供的 API 接口可以作为一个共享 DLL 给外部程序调用，也可以作为一个静态 Lib 链接到您的程序内部。

Trampoline 函数可以动态或者静态的创建，如果目标函数本身是一个链接符号，使用静态的 trampoline 函数将非常简单。如果目标函数不能在链接时可见，那么可以使用动态 trampoline 函数。

要使用静态的 trampoline 函数来截获目标函数，应用程序生成 trampoline 的时候必须使用 DETOUR_TRAMPOLINE 宏。DETOUR_TRAMPOLINE 有两个输入参数：trampoline 的原型和目标函数的名字。

注意，对于正确的截获模型，包括目标函数，trampoline 函数，以及截获函数都必须是完全一致的调用形式，包括参数格式和调用约定。当通过 trampoline 函数调用目标函数的时

候拷贝正确参数是截获函数的责任。由于目标函数仅仅是截获函数的一个可调用分支（截获函数可以调用 trampoline 函数也可以不调用），这种责任是一种强制性要求。

使用相同的调用约定可以确保寄存器中的值被正确的保存，并且保证调用堆栈在截获函数调用目标函数的时候能正确的建立和销毁。

可以使用 DetourFunctionWithTrampoline 函数来截获目标函数。这个函数有两个参数：trampoline 函数以及截获函数的指针。因为目标函数已经被加到 trampoline 函数中，所有不需要在参数中特别指定。

我们可以使用 DetourFunction 函数来创建一个动态的 trampoline 函数，它包括两个参数：一个指向目标函数的指针和一个截获函数的指针。DetourFunction 分配一个新的 trampoline 函数并将适当的截获代码插入到目标函数中去。

当目标函数不是很容易使用的时候，DetourFindFunction 函数可以找到那个函数，不管它是 DLL 中导出的函数，或者是可以通过二进制目标函数的调试符号找到。

DetourFindFunction 接受两个参数：库的名字和函数的名字。如果 DetourFindFunction 函数找到了指定的函数，返回该函数的指针，否则将返回一个 NULL 指针。 DetourFindFunction 会首先使用 Win32 函数 LoadLibrary 和 GetProcAddress 来定位函数，如果函数没有在 DLL 的导出表中找到，DetourFindFunction 将使用 ImageHlp 库来搜索有效的调试符号（译注：这里的调试符号是指 Windows 本身提供的调试符号，需要单独安装，具体信息请参考 Windows 的用户诊断支持信息）。DetourFindFunction 返回的函数指针可以用来传递给 DetourFunction 以生成一个动态的 trampoline 函数。

3.2 编写 dll 实现 detours 库的调用

如果要截获第三方进程调用 API，需要将 Dll 代码注入到第三方进程空间，往往采用全局消息 Hook 或者远程线程创建的方式进行。

静态方法：

建立一个 Dll 工程，名称为 ApiHook，这里以 VS 开发环境，以截获 ASCII 版本的 MessageBoxA 函数来说明。在 Dll 的工程加入：

```
DETOUR_TRAMPOLINE(int WINAPI Real_Messagebox(
HWND hWnd ,
LPCSTR lpText,
LPCSTR lpCaption,UINT uType), ::MessageBoxA);
```

生成一个静态的 MessageBoxA 的 Trampoline 函数，在 Dll 工程中加入目标函数的 Detour

函数：

```
int WINAPI MessageBox_Mine(
HWND hWnd ,
LPCSTR lpText,
LPCSTR lpCaption,
UINT uType)
{
   CString tmp= lpText;
   tmp+=” 被 Detour 截获”;
   return Real_Messagebox(hWnd,tmp,lpCaption,uType);
//   return ::MessageBoxA(hWnd,tmp,lpCaption,uType);   //Error
}
```

在 Dll 入口函数中的加载 Dll 事件中加入：

```
DetourFunctionWithTrampoline((PBYTE)Real_Messagebox, (PBYTE)MessageBox_Mine);
```

在 Dll 入口函数中的卸载 Dll 事件中加入：

```
DetourRemove((PBYTE)Real_Messagebox, (PBYTE)MessageBox_Mine);
```

动态方法：

建立一个 Dll 工程，名称为 ApiHook，这里以 Visual C++6.0 开发环境，以截获 ASCII 版本的 MessageBoxA 函数来说明。在 Dll 的工程加入：

```
//声明 MessageBoxA 一样的函数原型
typedef int   (WINAPI * MessageBoxSys)(
HWND hWnd ,
LPCSTR lpText,
LPCSTR lpCaption,
UINT uType);
//目标函数指针
MessageBoxSys SystemMessageBox=NULL;
//Trampoline 函数指针
MessageBoxSys Real_MessageBox=NULL;
```

在 Dll 工程中加入目标函数的 Detour 函数：

```
int WINAPI MessageBox_Mine( HWND hWnd ,
```

```
        LPCSTR lpText,
        LPCSTR lpCaption,
        UINT uType)
{
    CString tmp= lpText;
    tmp+=” 被 Detour 截获”;
    return Real_Messagebox(hWnd,tmp,lpCaption,uType);
//    return ::MessageBoxA(hWnd,tmp,lpCaption,uType);    //Error
}
```

在 Dll 入口函数中的加载 Dll 事件中加入：

```
SystemMessageBox=(MessageBoxSys)DetourFindFunction("user32.dll","MessageBoxA");
if(SystemMessageBox==NULL)
{
    return FASLE;
}
Real_MessageBox=(MessageBoxSys)DetourFunction((PBYTE)SystemMessageBox,
(PBYTE)MessageBox_Mine);
```

在 Dll 入口函数中的卸载 Dll 事件中加入：

```
DetourRemove((PBYTE)Real_Messagebox, (PBYTE)MessageBox_Mine);
```

4　系统实现

4.1　Dll 框架的搭建及注射器程序的编写

首先根据实验指导书内容以及老师提供的样例，进行 dll 框架的搭建。如图 4.1-1 所示，进行 MessageBoxA 的蹦床函数编写，在 api 截获成功后，将所需要的参数打印出来。

```
extern "C" __declspec(dllexport)int WINAPI NewMesssageBoxA(_In_opt_ HWND hWnd, _In_opt_ LPCSTR lpText, _In_opt_ LPCSTR lpCaption, _In_ UINT uType)
{
    printf("\n\n*****************************************\n");
    printf("MessageBoxA Hooked\n");
    GetLocalTime(&st);
    printf("DLL日志输出：%d-%d-%d %02d: %02d: %02d: %03d\n", st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond, st.wMilliseconds);
    printf("*****************************************\n\n");
    return OldMesssageBoxA(NULL, "new MessageBoxA", "Hooked", MB_OK);

}
```

图 4.1-1

之后进行 DLL_PROCESS_ATTACH 与 DLL_PROCESS_DETACH 的编写，将所需的 detours 库函数进行装载使用及卸载退出。

```
case DLL_PROCESS_ATTACH:
{
    DisableThreadLibraryCalls(hModule);
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    DetourAttach(&(PVOID&)OldMesssageBoxW, NewMesssageBoxW);
    DetourAttach(&(PVOID&)OldMesssageBoxA, NewMesssageBoxA);
    DetourAttach(&(PVOID&)OldCreateFile, NewCreateFile);
    DetourAttach(&(PVOID&)OldCloseFILE, NewCloseFILE);
    DetourAttach(&(PVOID&)OldWriteFile, NewWriteFile);
    //DetourAttach(&(PVOID&)OldReadFile, NewReadFile);
    DetourAttach(&(PVOID&)OldHeapCreate, NewHeapCreate);
    //DetourAttach(&(PVOID&)OldHeapAlloc, NewHeapAlloc);
    //DetourAttach(&(PVOID&)OldHeapFree, NewHeapFree);
    DetourAttach(&(PVOID&)OldHeapDestroy, NewHeapDestroy);
    DetourAttach(&(PVOID&)OldRegCreateKeyEx, NewRegCreateKeyEx);
    DetourAttach(&(PVOID&)OldRegOpenKeyEx, NewRegOpenKeyEx);
    DetourAttach(&(PVOID&)OldRegSetValueEx, NewRegSetValueEx);
    DetourAttach(&(PVOID&)OldRegCloseKey, NewRegCloseKey);
    DetourTransactionCommit();//拦截生效
}
```

图 4.1-2

```
case DLL_THREAD_DETACH:
case DLL_PROCESS_DETACH:
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    DetourDetach(&(PVOID&)OldMesssageBoxW, NewMesssageBoxW);
    DetourDetach(&(PVOID&)OldMesssageBoxA, NewMesssageBoxA);
    DetourDetach(&(PVOID&)OldCreateFile, NewCreateFile);
    DetourDetach(&(PVOID&)OldCloseFILE, NewCloseFILE);
    //DetourDetach(&(PVOID&)OldReadFile, NewReadFile);
    DetourDetach(&(PVOID&)OldWriteFile, NewWriteFile);
    DetourDetach(&(PVOID&)OldHeapCreate, NewHeapCreate);
    //DetourDetach(&(PVOID&)OldHeapAlloc, NewHeapAlloc);
    //DetourDetach(&(PVOID&)OldHeapFree, NewHeapFree);
    DetourDetach(&(PVOID&)OldHeapDestroy, NewHeapDestroy);
    DetourDetach(&(PVOID&)OldRegCreateKeyEx, NewRegCreateKeyEx);
    DetourDetach(&(PVOID&)OldRegCloseKey, NewRegCloseKey);
    DetourDetach(&(PVOID&)OldRegOpenKeyEx, NewRegOpenKeyEx);
    DetourDetach(&(PVOID&)OldRegSetValueEx, NewRegSetValueEx);
    DetourTransactionCommit();//拦截生效
    break;
```

图 4.1-3

Dll 框架编译成功后，对注射器模板进行完善，把编译通过的.dll 文件及需要测试的 testapp.exe 路径输入注射器中，将 dll 文件装载到 testapp 中进行测试。

```
#include<detours.h>
#pragma comment(lib,"detours.lib")

int main()
{

    STARTUPINFO si;//数据结构，用于指定新进程的主窗口特性
    PROCESS_INFORMATION pi;//数据结构，返回有关新进程及其主线程的信息
    ZeroMemory(&si, sizeof(STARTUPINFO));
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));//初始化结构体，用0填充一段内存
    si.cb = sizeof(STARTUPINFO);
    WCHAR DirPath[MAX_PATH + 1];
    //errno_t wcscpy_s( wchar_t *restrict dest, rsize_t destsz,const wchar_t* restrict src );
    //dest -   指向复制目标的宽字符数组的指针;src - 指向复制来源的空终止宽字符串的指针;destsz - 要写入的最大字符数，典型地为目标缓冲区的大小
    //MAX_PATH = 260;
    wcscpy_s(DirPath, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\InjectDll\\Debug");
    char DLLPath[MAX_PATH + 1] = "C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\InjectDll\\Debug\\InnjectDll.dll";
    WCHAR EXE[MAX_PATH + 1] = { 0 };
    //选择测试程序
    int i = 0;
    scanf("%d", &i);
    getchar();
    if (i == 1)
        wcscpy_s(EXE, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\TestFunc\\Debug\\TestFunc.exe");
    else if (i == 2)
```

图 4.1-4

如下图所示，弹窗 messboxa 的信息被截获并篡改为“hooked”，“new MessageBoxA”，测试成功。至此，整个系统框架基本构建完成。

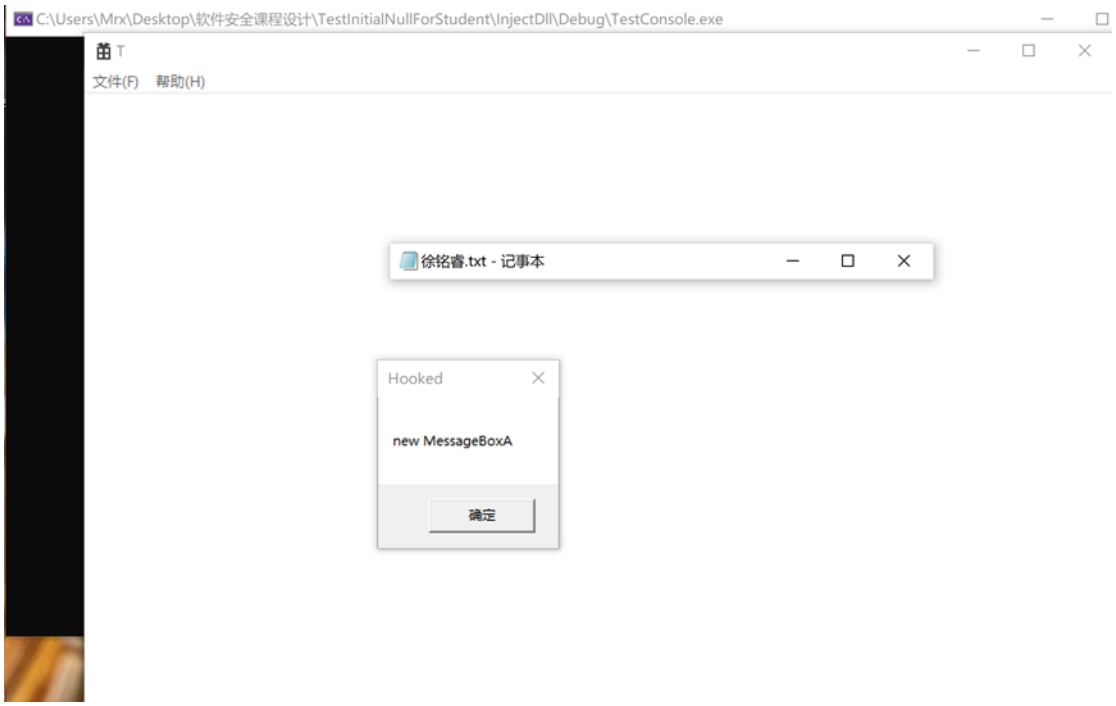


图 4.1-5

4.2 Api 截获实现

堆操作、文件操作、注册表操作 api 截获的实现过程大同小异，这里以堆操作 api 截获的实现为例。

首先对要截获的目标函数进行声明（图 4.2-1），然后在 NewHeapCreate 中进行蹦床函数的编写，来获取所需参数信息等（图 4.2-2），最后进行 detours 函数的装载与卸载（图 4.2-3）。

```
static HANDLE(WINAPI* OldHeapCreate)(DWORD fIOoptions, SIZE_T dwInitialSize, SIZE_T dwMaximunSize) = HeapCreate;

extern "C" __declspec(dllexport)HANDLE WINAPI NewHeapCreate(DWORD fIOoptions, SIZE_T dwInitialSize, SIZE_T dwMaximunSize){ ... }
```

图 4.2-1

```
extern "C" __declspec(dllexport)HANDLE WINAPI NewHeapCreate(DWORD fIOoptions, SIZE_T dwInitialSize, SIZE_T dwMaximunSize)
{
    HANDLE hHeap = OldHeapCreate(fIOoptions, dwInitialSize, dwMaximunSize);
    printf("\n\n*****************************************\n");
    printf("HeapCreate Hooked\n");
    GetLocalTime(&st);
    printf("DLL日志输出：%d-%d-%d %02d: %02d: %02d: %03d\n", st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond, st.wMilliseconds);
    printf("fIOoptions :  0x%x\n", fIOoptions);
    printf("dwInitialSize :  0x%x\n", dwInitialSize);
    printf("dwMaximunSize :  0x%x\n", dwMaximunSize);
    printf("hHeap初始地址 :  0x%x\n", hHeap);
    printf("*****************************************\n\n");
    HeapC.insert(hHeap);
    return hHeap;
}
```

图 4.2-2

```
//DetourAttach(&(PVOID&)OldReadFile, NewReadFile);
DetourAttach(&(PVOID&)OldHeapCreate, NewHeapCreate);

//DetourDetach(&(PVOID&)OldReadFile, NewReadFile);
DetourDetach(&(PVOID&)OldHeapCreate, NewHeapCreate);
```

图 4.2-3

4.3 数据共享段避免自调用 api 截获问题

DLL 进程间共享数据段是指，在多个进程间共享数据，windows 提供了这种方法，就是创建自己的共享数据节，并将需要共享的变量放入该内存中。如果是在相同程序的多个实例间共享数据，只要在 exe 文件创建共享节即可，否则就需要在 DLL 中创建共享节，其它进程加载该 DLL 来共享数据。

在 Win32 环境下要想在多个进程中共享数据，就必须进行必要的设置。在访问同一个 Dll 的各进程之间共享存储器是通过存储器映射文件技术实现的。也可以把这些需要共享的数据分离出来，放置在一个独立的数据段里，并把该段的属性设置为共享。并且必须给这些变量赋初值，否则编译器会把没有赋初始值的变量放在一个叫未被初始化的数据段中。

下面是参考老师的提示，进行共享数据段相关代码段。

```
#pragma data_seg ("MySeg")
char seg[1000][256] = {};//共享数据段 //必须在定义的同时进行初始化!!!!
int count1 = 0;//共享数据段
volatile int HeapAllocNum = 0;//共享数据段
volatile int buffer[1024] = {};
#pragma data_seg ()
#pragma comment (linker,"/section:MySeg,RWS")
//[注意：数据节的名称is case sensitive]
//那么这个数据节中的数据可以在所有DLL的实例之间共享。所有对这些数据的操作都针对同一个实例的，
//而不是在每个进程的地址空间中都有一份。
std::mutex mtx;
```

图 4.3-1

并在操作前后添加了原子锁，以避免 HeapAlloc 的自调用。

```
    mtx.lock();
    count1++;
    if (count1 >= 1000) {
        count1--;
    }
    else {
        strcpy_s(seg[count1], str);//将后放到前
    }
    mtx.unlock();
}

extern "C"  __declspec(dllexport)void GetSeg(char* str, int i) {   //在主程序中调用获取共享信息
    mtx.lock();
    strcpy_s(str, 256, seg[i]);
    mtx.unlock();
}

extern "C" _declspec(dllexport)void GetNum(int& num) {  //获得多少个共享数据
    mtx.lock();
    num = count1;
    mtx.unlock();
}

extern "C" __declspec(dllexport)void GetHeapAllocNum(int& num) {
    num = HeapAllocNum;
}
```

项目: InnjectDll (Ctrl+F2)

使用下拉菜单可查看和切换到此文件(dllw.cpp

图 4.3-2

4.4 异常行为分析

堆异常分析部分：重复释放检测（是否正常），判断 HeapDestroy 次数是否大于一次，若大于，提示堆发生重复释放。

```
extern "C" __declspec(dllexport)BOOL WINAPI NewHeapDestroy(HANDLE hHeap)
{
    printf("\n\n*****************************************\n");
    printf("HeapDestroy Hooked\n");
    GetLocalTime(&st);
    printf("DLL日志输出：%d-%d-%d %02d: %02d: %02d: %03d\n", st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute,
    printf("hHeap :  0x%x\n", hHeap);
    printf("*****************************************\n\n");
    HeapD = hHeap;
    std::set<HANDLE>::iterator it = HeapC.find(HeapD);
    if (it == std::end(HeapC)) {
        printf("!!!!!0x%x地址的堆重复释放!!!!!\n", hHeap);
        return 0;
    }
    else {
        HeapC.erase(it);
        return OldHeapDestroy(hHeap);
    }
}
```

图 4.4-1

文件异常行为分析：

1. 自我复制

利用_splitpath_s 函数获得文件对应名称，通过文件名是否相同来判断是否进行了自我复制。

```
ChangeSeg(pchar);
char szDrive[256];   //磁盘名
char szDir[256];    //路径名
char szFname[256];  //文件名
char szExt[256];   //后缀名
_splitpath_s(pchar, szDrive, szDir, szFname, szExt);
if (!strcmp(FileCopy, szFname)) {
    printf("!!!!正在进行自我复制!!!!\n");
}
```

图 4.4-2

2. 修改了其他可执行代码

同理，利用_splitpath_s 函数获得文件对应后缀名，来判断是否修改了其他可执行代码。

```
_splitpath_s(pchar, szDrive, szDir, szFname, szExt);
if (!strcmp(FileCopy, szFname)) {
    printf("!!!!正在进行自我复制!!!!\n");
}
//printf("%s",szFname);
//else printf("没有发生自我复制!\n");
if (!strcmp(szExt, EXE)) {
    printf("修改了扩展名为exe的可执行文件\n");
}
else if (!strcmp(szExt,".dll")) {
    printf("修改了扩展名为dll的可执行文件\n");
}
else if (!strcmp(szExt, ".ocx")) {
    printf("修改了扩展名为ocx的可执行文件\n");
}
else if (!strcmp(szExt, ".bat")) {
    printf("修改了扩展名为bat的可执行文件\n");
}
```

图 4.4-3

注册表异常行为分析：

判断是否新增注册表项并判断是否为自启动执行文件项；是否修改了注册表；输出所有的注册表操作项；

```
CString str2(_T("Software\\Microsoft\\Windows\\CurrentVersion\\Run"));
if (0 == str1.CompareNoCase(str2))
{
    printf("该日志文件为开机自启项\n");
}
else
{
    printf("该日志文件不是开机自启项\n");
}
```

图 4.4-4

```
LONG a = OldRegSetValueEx(hKey, lpValueName, Reserved, dwType, lpData, cbData);
if (a == ERROR_SUCCESS) printf("注册表修改成功\n");
```

图 4.4-5

4.5 检测样本

5 个待检测的可能存在恶意的 Exe 样本在源码中，这里不在赘述。

4.6 系统界面

使用 qt 进行了一个十分简单的 ui 编写，将注射器代码放入 qt 中，在 ui 中进行注射器的启动以及功能的选择。


MainWindow
api截获
堆多次释放
文件自我复制
dll文件
注册表操作
exe文件
开始测试
徐铭睿.txt - 记事本

图 4.6-1

```
int i =1;

void MainWindow::on_pushButton_pressed()
{
    STARTUPINFO si;//数据结构，用于指定新进程的主窗口特性
        PROCESS_INFORMATION pi;//数据结构，返回有关新进程及其主线程的信息
        ZeroMemory(&si, sizeof(STARTUPINFO));
        ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));//初始化结构体，用0填充一段内存
        si.cb = sizeof(STARTUPINFO);
        WCHAR DirPath[MAX_PATH + 1];
        //errno_t wcscpy_s( wchar_t *restrict dest, rsize_t destsz,const wchar_t* restrict src );
        //dest  -   指向复制目标的宽字符数组的指针;src - 指向复制来源的空终止宽字符串的指针;destsz - 要写入的最大字符数，典型地为目标缓冲区的大小
        //MAX_PATH = 260;
        wcscpy_s(DirPath, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\InjectDll\\Debug");
        char DLLPath[MAX_PATH + 1] = "C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\InjectDll\\Debug\\InnjectDll.dll";
        WCHAR EXE[MAX_PATH + 1] = { 0 };
        //选择测试程序
        //int i = 1;
        //scanf("%d", &i);
        //getchar();
        if (i == 1)
            wcscpy_s(EXE, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\TestFunc\\Debug\\TestFunc.exe");
        else if (i == 2)
            wcscpy_s(EXE, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\Sample\\Heap repeatedly release.exe");
        else if(i==3)
            wcscpy_s(EXE, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\Sample\\selfcopy.exe");
        else if(i==4)
            wcscpy_s(EXE, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\Sample\\executable file.exe");
        else if(i==5)
            wcscpy_s(EXE, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\Sample\\regedit.exe");
        else if(i==6)
            wcscpy_s(EXE, MAX_PATH, L"C:\\Users\\Mrx\\Desktop\\software-security-design\\TestInitialNullForStudent\\Sample\\executable(exe).exe");
        else printf("error!");
        if (DetourCreateProcessWithDllEx(EXE, NULL, NULL, NULL, TRUE, CREATE_DEFAULT_ERROR_MODE | CREATE_SUSPENDED, NULL, DirPath, &si, &pi, DLLPath, NULL))
        {
            ResumeThread(pi.hThread);//回复线程运行
            WaitForSingleObject(pi.hProcess, INFINITE);//你可以使用WaitForSingleObject函数来等待一个内核对象变为已通知状态：
            //DWORD WaitForSingleObject(HANDLE hObject, //指明一个内核对象的句柄;DWORD dwMilliseconds); //等待时间
        }
        else
        {
            char error[100];
            sprintf_s(error, "%d", GetLastError());//sprintf将数据格式化输出到字符串;GetLastError()返回上一个函数调用设置的线程的32位错误代码
        }
}

void MainWindow::on_pushButton_2_clicked()
```

图 4.6-2

5 系统测试

5.1 Api 截获测试

1. MessageBox 截获测试

图 5.1-1

2. 堆操作 api 截获测试

图 5.1-2

3. 文件操作 api 截获测试

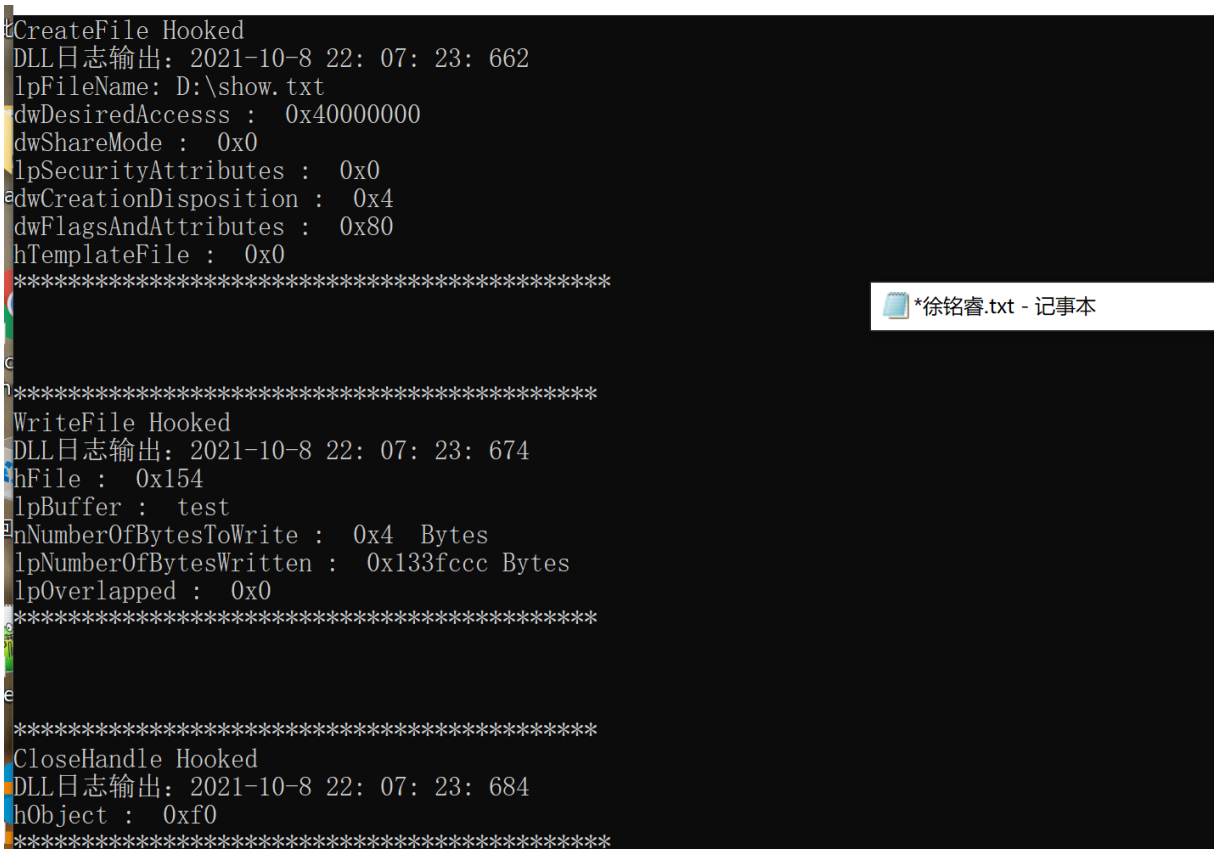


图 5.1-3

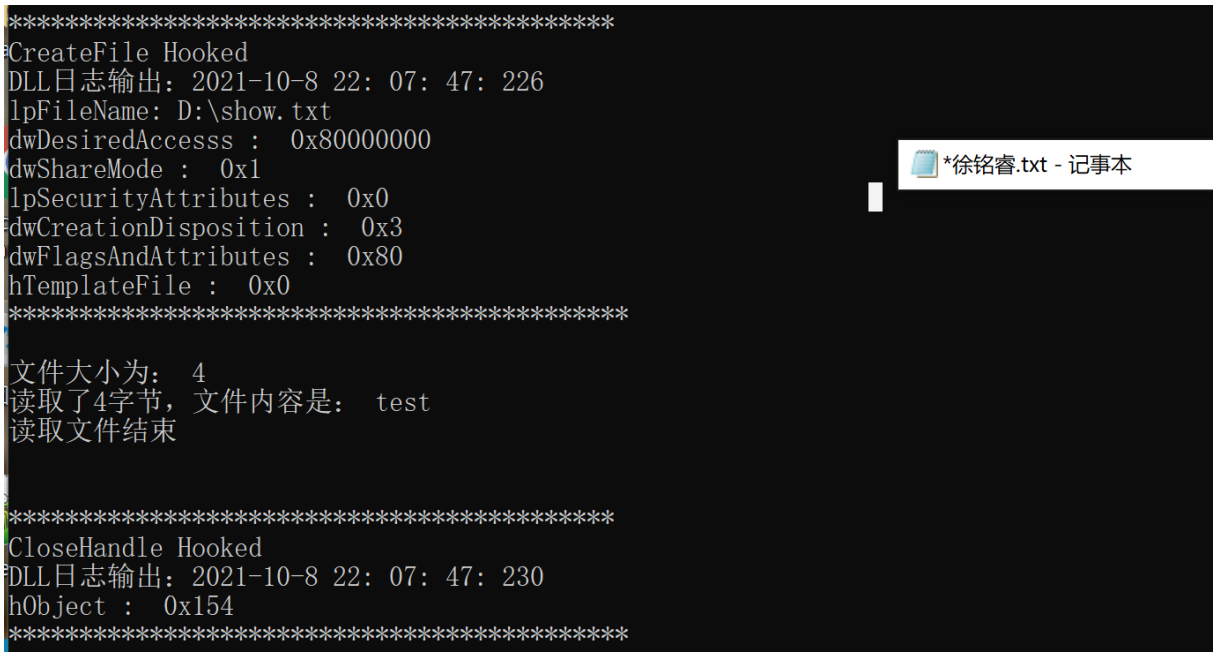


图 5.1-4

4. 注册表 api 截获测试

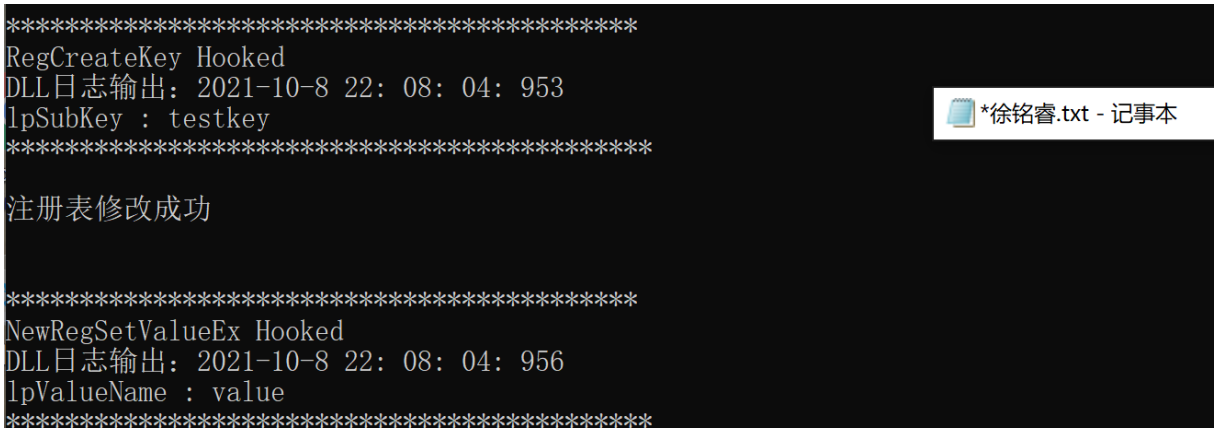


图 5.1-5

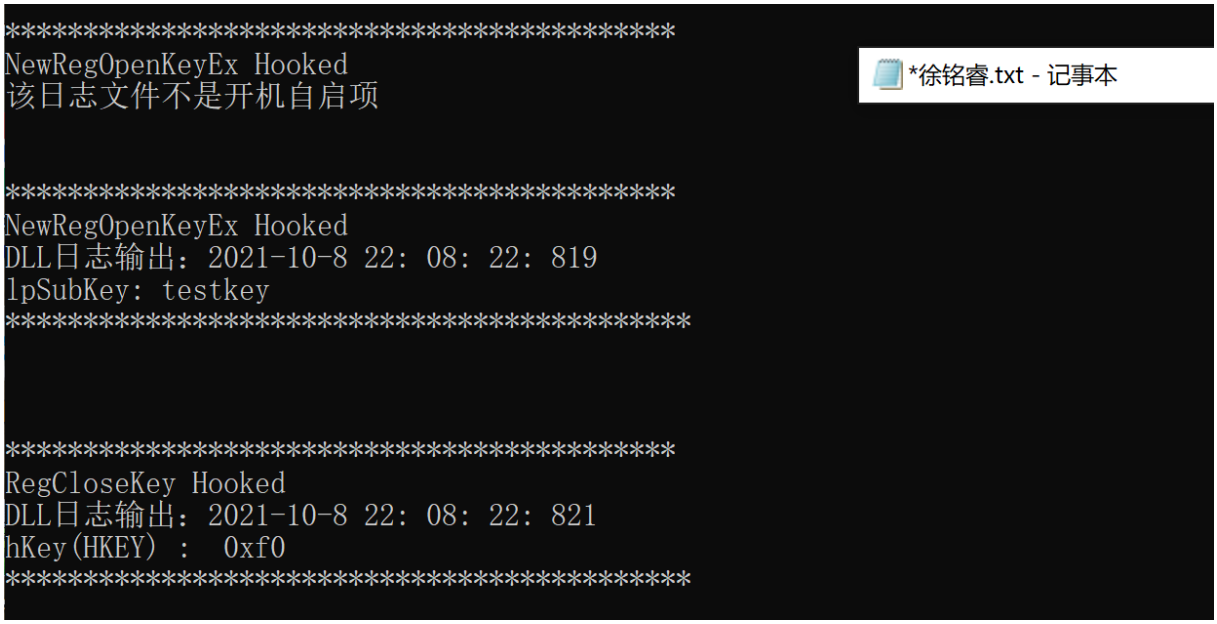


图 5.1-6

5.2 异常行为分析测试

1. 堆操作异常行为分析测试

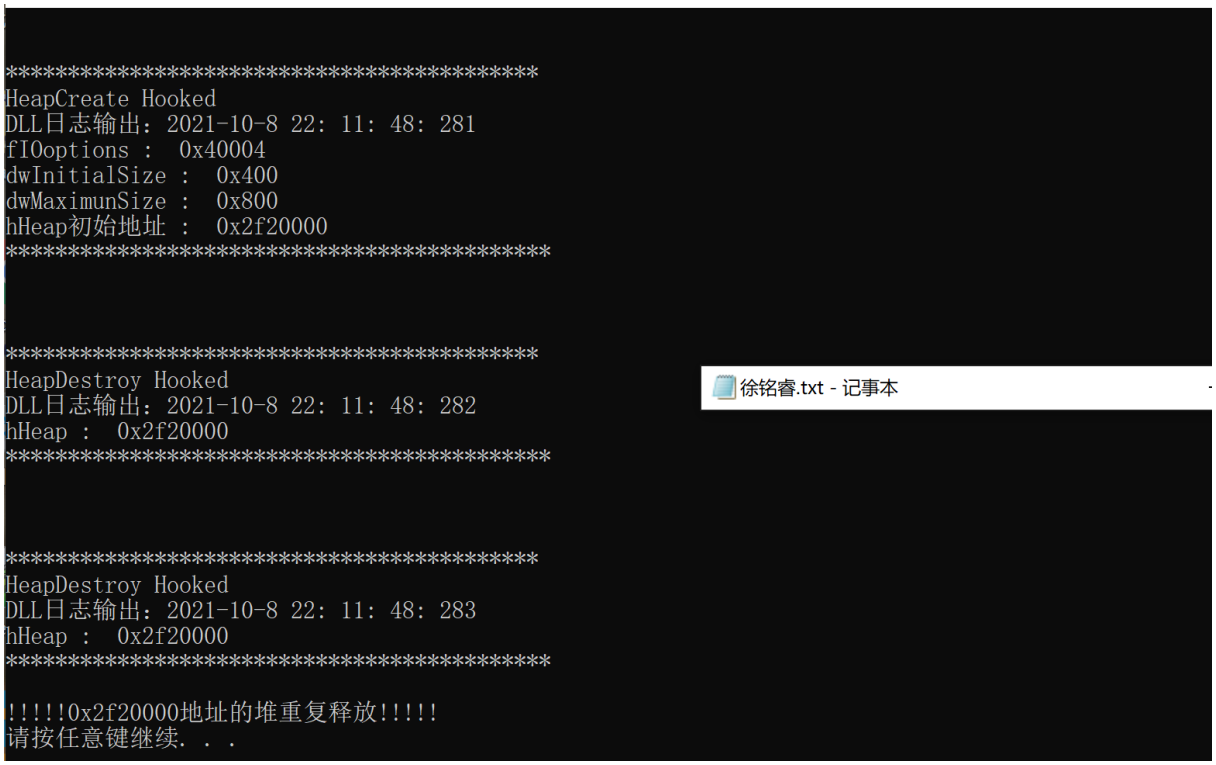


图 5.2-1

2. 文件操作异常行为分析测试

   检测到文件自我复制，并将自我复制内容打印出来。

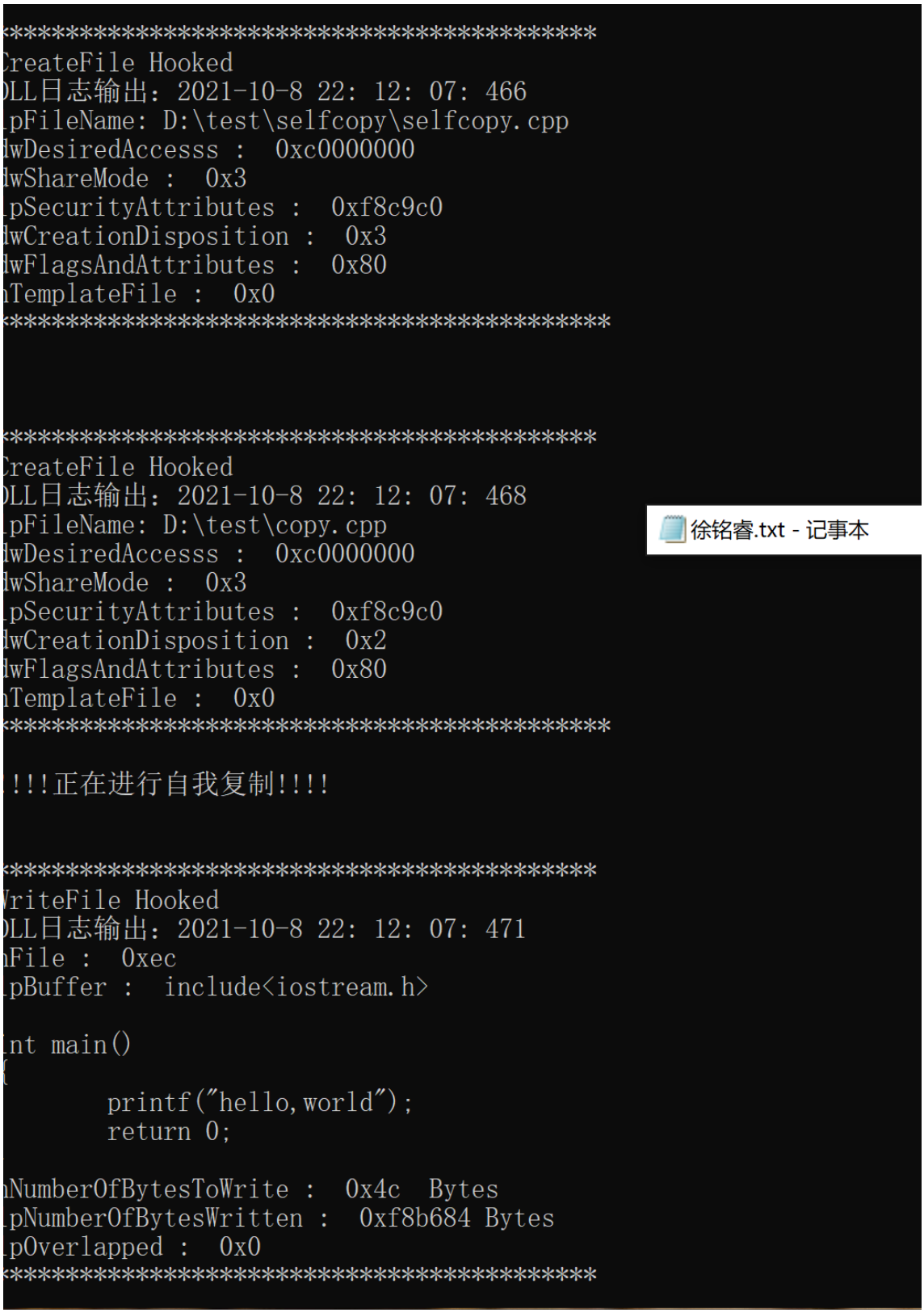


图 5.2-2

检测到文件修改了扩展名为 dll 的可执行文件，并把文件路径进行打印。

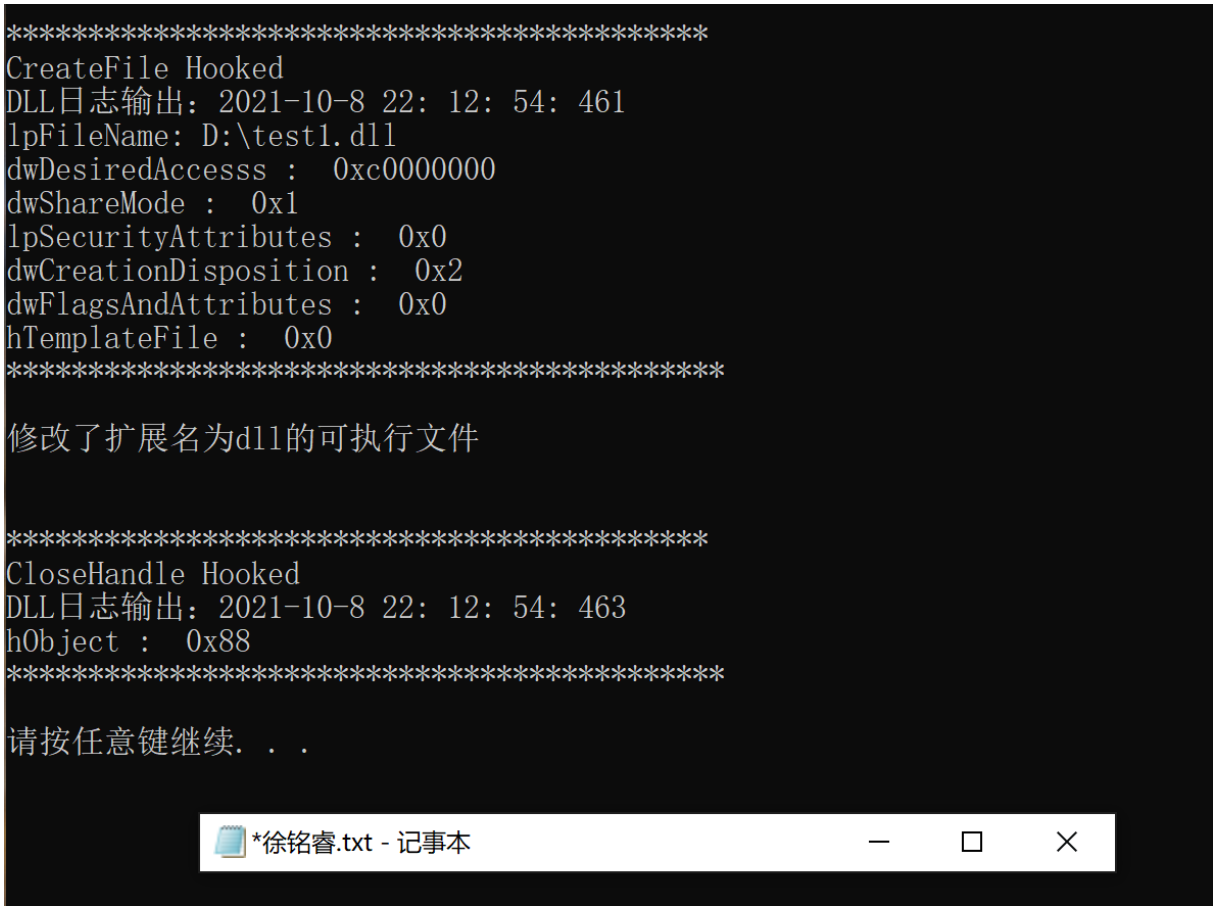


图 5.2-3

3. 注册表异常行为分析测试

   判断是否新增注册表项并判断是否为自启动执行文件项；是否修改了注册表；输出所有的注册表操作项；

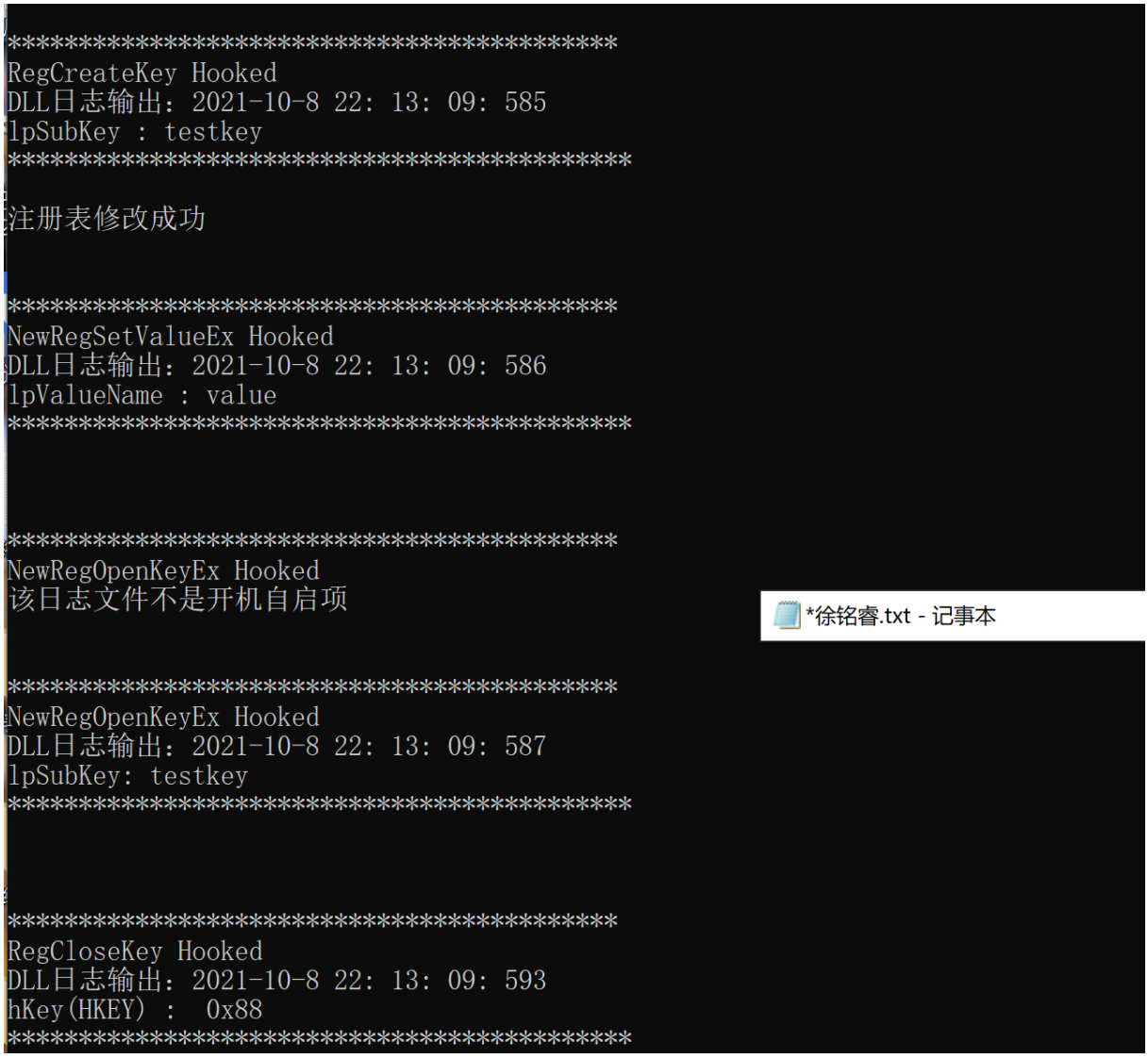

****
RegCreateKey Hooked
DLL日志输出：2021-10-8 22：13：09：585
lpSubKey ： testkey
****
注册表修改成功
****
NewRegSetValueEx Hooked
DLL日志输出：2021-10-8 22：13：09：586
lpValueName ： value
****
****
NewRegOpenKeyEx Hooked
该日志文件不是开机自启项
*徐铭睿.txt - 记事本
****
NewRegOpenKeyEx Hooked
DLL日志输出：2021-10-8 22：13：09：587
lpSubKey: testkey
****
****
RegCloseKey Hooked
DLL日志输出：2021-10-8 22：13：09：593
hKey(HKEY) ： 0x88
****

图 5.2-4

6　总结与展望

在开始阶段，本次课程设计给我的第一印象是无从下手，看到 windows api 截获以及 detours 库的时候感到很陌生，感觉在之前大一大二的学习过程中对这方面的接触不多，好在软件安全理论课程对 windows api 有一定的讲解。

之后根据实验指导书以及老师的提示，完成了整个课设基本框架的构建，dll 使用 detours 库截获 windows api；注射器将 dll 注入测试程序等等。经过框架的构建，整个课设的方向逐渐清晰起来了。并参考对 MessageBox 的截获，完成了其他 api 截获的函数编写。

本实验的一个难点在于，在截获 api 并打印参数时，会对自身的 api 进行截获，这样重复截获，陷入无线循环，导致程序崩溃。这里参考老师的共享缓冲区，并经过资料查阅借鉴，利用了进程线程的原子锁来解决这一问题。

最后使用 QT 编写了一个十分简单的用户界面，但比起之前的命令行形式控制程序，对用户还是友好了很多。

通过这次实验，我对 Detours 库的基本使用方式，如何对 windows api 进行截获有了一定的了解，也对 DLL 的作业与编写有了更深入的认识。同时，这是实验的用户界面设计，也让我有了新的收获，对 QT 也有了一个基本的了解。并且，在这次实验中，我对许多 windows api 的调用方式方法，以及一些系统函数，都有了更加深入的了解，自己学会使用了一些 windows api。比如文件、堆和注册表操作的 api。

总之，这次实验收获良多。

7　课程建议与意见

个人感觉老师在第一次实验讲解时，对于 detours 库的编译，dll 框架构建等等这种实验指导书上写的很详细的内容可以不必要一步一步进行演示，而可以把讲解重点侧重在整个课程设计的整体思路上，比如 api 截获部分可以参照样例程序中 Messagebox 的 api hook 进行编写，需要写出实现 api 截获的测试函数，行为检测样本库具体指什么等等。

可以适当减少 api 截获部分及异常分析部分内容，有一定重复性，实验掌握相关部分的编写方法即可。

8 参考文献

[1] (美)理查德 著,黄陇,李虎 译. Windows 核心编程:原书第 4 版.机械工业出版社

[2] https://blog.csdn.net/allenq/article/details/8363363?utm_source

[3] http://blog.sina.com.cn/s/blog_7f23cf2e0102vzh8.htm

[4] https://www.cnblogs.com/shiyangxt/archive/2009/02/27/1399852.html

[5] https://www.cnblogs.com/coolcpp/p/regoperator.html