软件安全课程设计
基于 API HOOK 的软件行为分析系统

一、课程设计题目

基于 API HOOK 的软件行为分析系统

二、课程编码：130681

课程性质：必修

三、学时学分

学时：2 周

学分：1 学分

四、先修课程

C 语言、汇编语言、数据结构、软件安全

五、课程设计任务与要求

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，完成简单的程序行为分析。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。

六、关键技术原理介绍

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。中断代码可以在运行时动态加载，也可以在静态执行文件处理。

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。

(1) WIN32 进程的内存管理

WINDOWS 实现了虚拟存储器，每一 WIN32 进程拥有 4GB 的虚存空间， 关于 WIN32 进程的虚存结构及其操作的具体细节请参阅 WIN32 API 手册， 以下仅指出与 Detours 相关的几点：

1) 进程要执行的指令也放在虚存空间中；

2) 可以使用 QueryProtectEx 函数把存放指令的页面的权限更改为可读、可写、可执行，再改写其内容，从而修改正在运行的程序

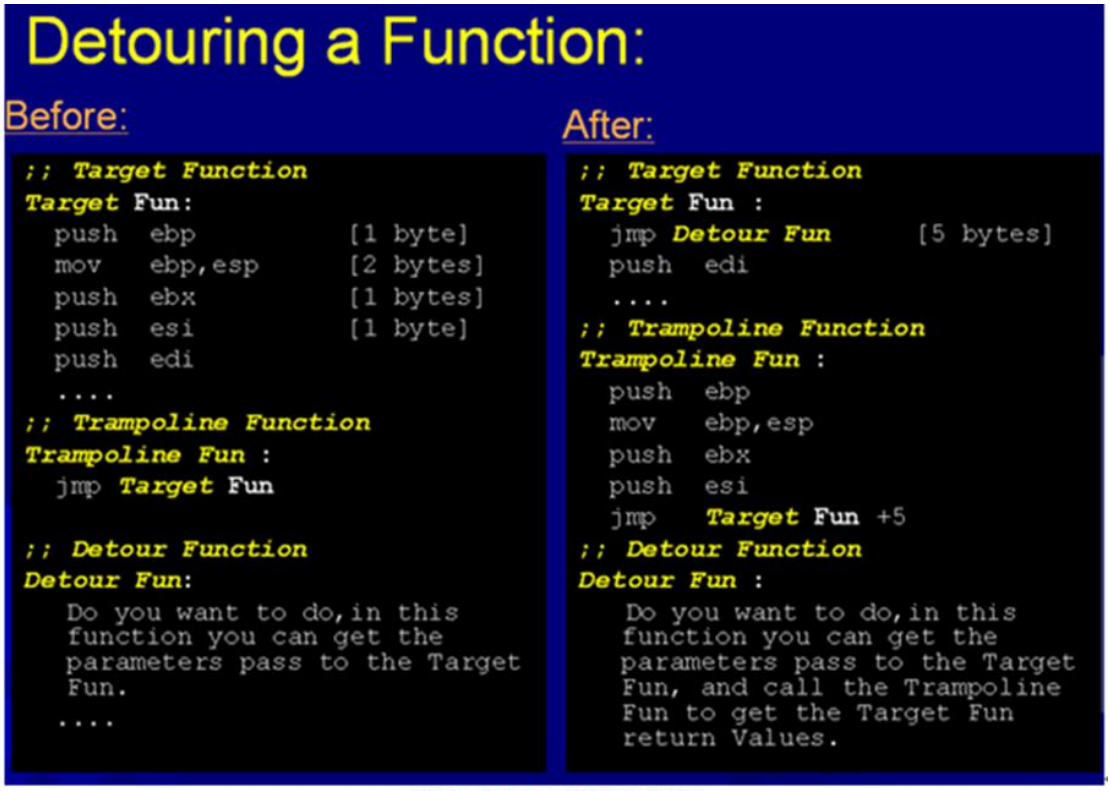
3) 可以使用 VirtualAllocEx 从一个进程为另一正运行的进程分配虚存，再使用 QueryProtectEx 函数把页面的权限更改为可读可写可执行，并把要执行的指令以二进制机器码的形式写入，从而为一个正在运行的进程注入任意的代码 。

(2)拦截 WIN32 API 的原理

Detours 定义了三个概念：

1) Target 函数（Target Fun）：要拦截的函数，通常为 Windows 的 API；

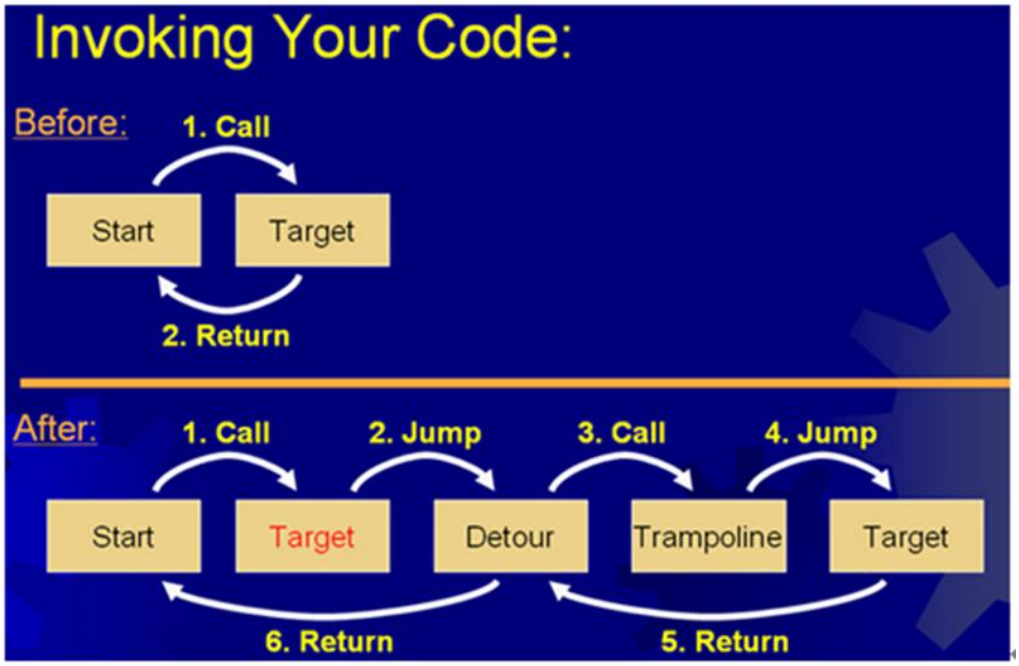
2) Trampoline 函数(Trampoline Fun)：Target 函数的部分复制品。因为 Detours 将会改写 Target 函数，所以先把 Target 函数的前 5 个字节复制保存好，一方面仍然保存 Target 函数的过程调用语义，另一方面便于以后的恢复。



(图 1：Detour函数的过程)

3) Detour 函数(Detour Fun)：用来替代 Target 函数的函数。

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



(图 2： Detour函数的调用过程)

jmp Address_of_ Target _ Function+5(共10个字节)作为Trampoline函数的内容。请参考图1和图2。

如果未看懂以上图示，可以再看一下对三个函数的说明：

Target 函数：原来的被调用目标函数的函数体（二进制）需要至少有5个字节以上，才能被改写为Target函数。理由如下：改写后的Target函数，第一条是一个跳转到Detour函数的语句，需要覆盖5个字节，之后才是原始的被调用函数剩余的未被覆盖的内容（例如图1中的push edi是第6个字节的指令，将从Trampoline函数跳回该处）。同时，微软的说明文档中，Trampoline函数的函数体为:从原来被调用函数拷贝被覆盖的这前5个字节,再加一个无条件跳转指令，就完成了蹦床函数。因此，前5个字节必须是完整指令（当第5个字节和第6个字节不能是一条不可分割的指令时，会导致Trampoline函数执行错误，复制前5个字节后，一条完整的指令被硬性分割开来，造成程序崩溃）。

如果第5字节和第6个字节是不可分割指令，需要调整拷贝到Trampoline函数的字节个数（具体拷贝多少，可以查看目标函数的汇编代码得到）。此函数是目标函数的修改版本，不能在Detour函数中直接调用，需要通过对Trampoline函数的调用来达到间接调用。

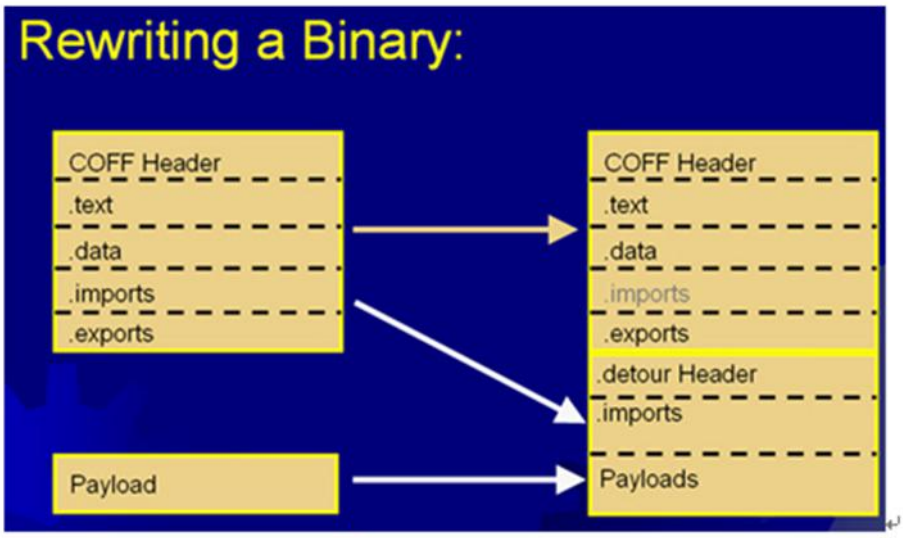
Trampoline 函数：此函数默认分配了32个字节，其函数的内容，就是拷贝的目标函数的前5个字节，加上一个JMP Address_of_ Target _ Function+5指令，共10个字节。

此函数仅供Detour函数调用，执行完前5个字节的指令后，再绝对跳转到修改后的目标函数（Target 函数）的第6个字节，继续执行原目标函数功能。

Detour 函数：（Detour，英文中指绕道，类似于2号食堂二楼对着明德楼的门坏了需要维修，就在此处设置一个detour的指示牌，指引大家从食堂外的楼梯绕道，从食堂一楼正门进去，之后，再从该楼梯返回到二楼）此函数是用户需要的截获API的一个模拟版本，调用方式，参数个数必须和目标函数相一致。如目标函数是__stdcall，则Detour函数声明也必须是 __stdcall，参数个数和类型也必须相同，否则会造成程序崩溃。此函数在程序调用目标函数的第一条指令的

时候就会被调用（无条件跳转过来的），如果在此函数中想继续调用原来的目标函数，必须调用 Trampoline 函数（Trampoline 函数在执行完目标函数的前 5 个字节的指令后会无条件跳转到目标函数的 5 个字节后继续执行），不能再直接调用目标函数，否则将进入无穷递归（目标函数跳转到 Detour 函数，Detour 函数又跳转到目标函数的递归，因为目标函数在内存中的前 5 个字节已经被修改成绝对跳转）。通过对 Trampoline 函数的调用后，可以获取目标函数的执行结果，此特性对分析目标函数非常有用，而且可以将目标函数的输出结果进行修改后再传回给应用程序。

Detour 提供了向运行中的应用程序注入 Detour 函数和在二进制文件基础上注入 Detour 函数两种方式。本章主要讨论第二种工作方式。通过 Detours 提供的开发包可以在二进制 EXE 文件中添加一个名称为 Detour 的节表，如下图 3 所示，主要目的是实现 PE 加载器加载应用程序的时候会自动加载您编写的 Detours DLL，在 Detours DLL 中的 DLLMain 中完成对目标函数的 Detour。



（图 3）

(3) Detours 提供的截获 API 的相关接口

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。

我们可以调用 DetourRemoveTrampoline 来去掉对一个目标函数的截获。

注意，因为 Detours 中的函数会修改应用程序的地址空间，请确保当加入截获函数或者去掉截获函数的时候没有其他线程在进程空间中执行，这是程序员的责任。一个简单的方法保证这个时候是单线程执行就是在加载 Detours 库的时候在 DllMain 中呼叫函数。

(4)使用 Detours 实现对 API 的截获的两种方法

1）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);

2)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","MessageB
oxA");
 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);
```

(5)关于截获第三方应用的 API

前面所述办法只能截获进程自身的 API，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。

七、Win10 环境中 Detours API 截获参考例子

(1)准备工作：使用 VS2019 编译 Detours

在 Detours-github 上下载 Detours 的源码，解压得到文件夹 Detours-master，

然后，在开始菜单中找到 x64 Native Tools Command Prompt for VS 2019 和 x86 Native Tools Command Prompt for VS 2019，这两个可以分别用来编译 64 位和 32 位的 Detours。如图 4 所示。

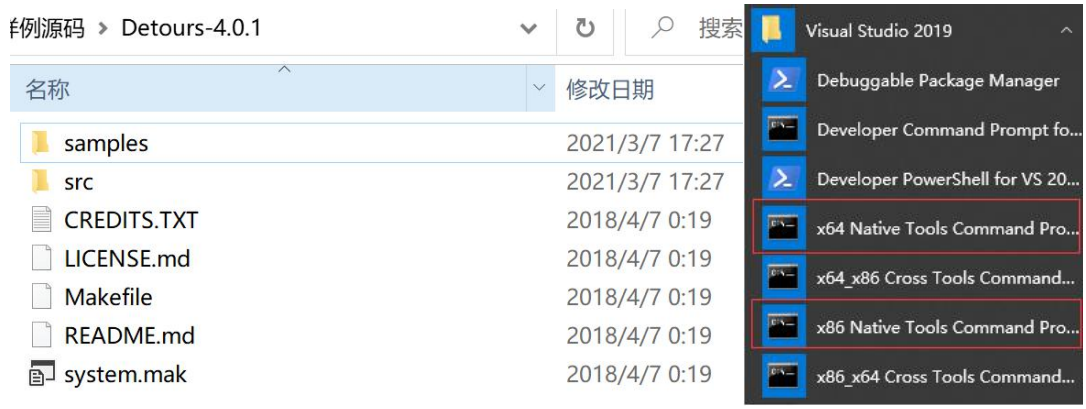


图 4 源码及编译 Detours 的工具

定位路径到解压的文件夹的 src 目录下，然后编译：

cd Detours-master/src

nmake /f Makefile

编译之后，可以在根目录找到 bin.X64、lib.X64、include 这三个文件夹，同理如果使用 x86 Native Tools Command Prompt for VS 2019 的话，生成 bin.X86、lib.X86、include 这三个文件夹。如图 5 所示。

至此，编译结束。

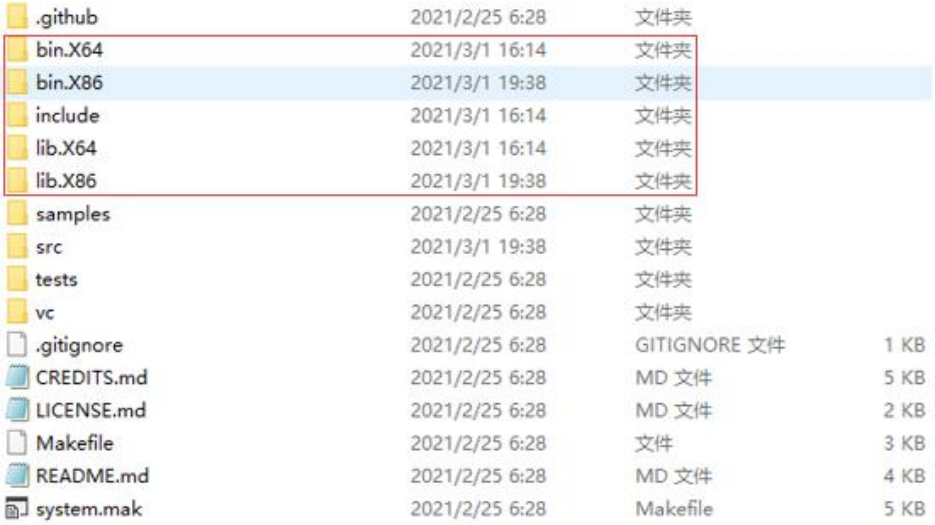


图 5 编译后生成的目录结构

(2) 准备工作：在 VS 上配置 Detours 库（使用 Detours，采用 dll 注入的方式进行 Hook）

在 VS 创建 DLL 项目，调试---调试属性。如图 6 所示。

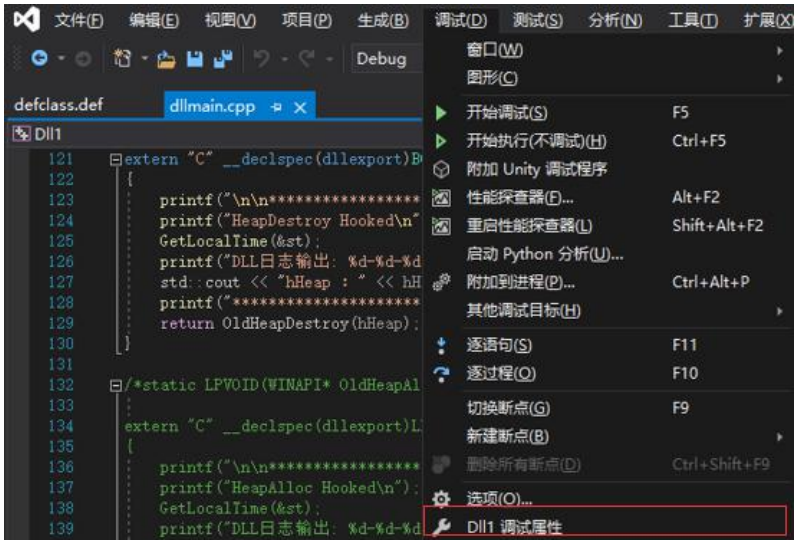


图 6 VS 2019 中配置包含目录及库目录-1

配置属性中选择 VC++目录，在**包含目录**中加上刚才编译出的 **include 文件夹路径**，在**库目录**上加上 **lib.X86**（64 位系统则对应 lib.X64）。如图 7 所示。

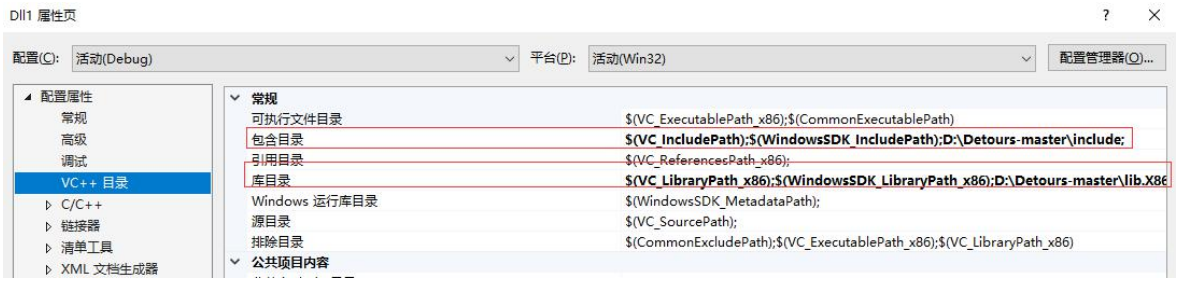


图 7 VS 2019 中配置包含目录及库目录-2

(3)准备 DLL

新建 DLL 项目（若需要 Hook 的 API 为 VOID WINDOWSAPI(VOID p1,VOID p2,VOID p3);

1）定义和引入需要挂钩（Hook）的函数和替换的函数

```
static VOID (WINAPI* OLD_WINDOWSAPI)(VOID p1,VOID p2,VOID p3) =
WINDOWSAPI;
extern "C" __declspec(dllexport) VOID NEW_WINDOWSAPI(VOID p1,VOID
p2,VOID p3)
{
 //TODO:
    return OLD_WINDOWSAPI(p1,p2,p3);
}
```

2）DLL 主体实现参考代码

参考代码如图，阅读可以发现哪些函数被挂钩，请结合课程设计的要求，挂钩对应的函数。

```cpp
// dllmain.cpp : 定义 DLL 应用程序的入口点。
#include "pch.h"
#include "framework.h"
#include "detours.h"
#include "stdio.h"
#include "stdarg.h"
#include "windows.h"
#include <iostream>

#pragma comment(lib,"detours.lib")

SYSTEMTIME st;

// 定义和引入需要Hook的函数，和替换的函数
static int (WINAPI* OldMesssageBoxW)(_In_opt_ HWND hWnd, _In_opt_ LPCWSTR lpText, _In_opt_ LPCWSTR lpCaption, _In_ UINT uType) = MessageBoxW;
static int (WINAPI* OldMesssageBoxA)(_In_opt_ HWND hWnd, _In_opt_ LPCSTR lpText, _In_opt_ LPCSTR lpCaption, _In_ UINT uType) = MessageBoxA;


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);
}
```

```cpp
extern "C" __declspec(dllexport)int WINAPI NewMesssageBoxW(_In_opt_ HWND hWnd, _In_opt_ LPCWSTR lpText, _In_opt_ LPCWSTR lpCaption, _In_ UINT uType)
{
    printf("\n\n*****************************************\n");
    printf("MessageBoxW 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 OldMesssageBoxW(NULL, L"new MessageBoxW", L"Hooked", MB_OK);
}

// 文件操作 OpenFile  CreateFile

static HANDLE(WINAPI* OldCreateFile)(
    LPCTSTR lpFileName,                         // 文件名
    DWORD dwDesiredAccess,                      // 访问模式
    DWORD dwShareMode,                          // 共享模式
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // 安全属性(也即销毁方式)
    DWORD dwCreationDisposition,                // how to create
    DWORD dwFlagsAndAttributes,                 // 文件属性
    HANDLE hTemplateFile                        // 模板文件句柄
    ) = CreateFile;
```

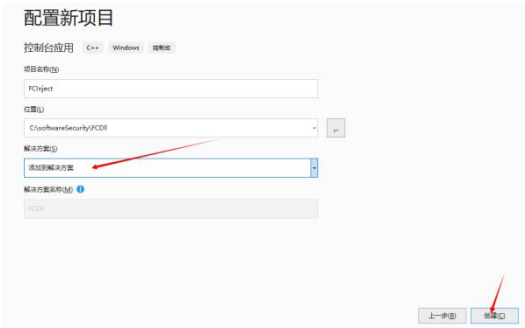
```cpp
extern "C" __declspec(dllexport)HANDLE WINAPI NewCreateFile(
    LPCTSTR lpFileName,                         // 文件名
    DWORD dwDesiredAccess,                      // 访问模式
    DWORD dwShareMode,                          // 共享模式
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // 安全属性(也即销毁方式)
    DWORD dwCreationDisposition,                // how to create
    DWORD dwFlagsAndAttributes,                 // 文件属性
    HANDLE hTemplateFile                        // 模板文件句柄
)
{
    printf("\n\n*****************************************\n");
    printf("CreateFile 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);
    int num = WideCharToMultiByte(CP_OEMCP, NULL, lpFileName, -1, NULL, 0, NULL, FALSE);
    char* pchar = new char[num];
    WideCharToMultiByte(CP_OEMCP, NULL, lpFileName, -1, pchar, num, NULL, FALSE);

    std::cout << "lpFileName : " << pchar << std::endl;
    std::cout << "dwDesiredAccess :  0x" << std::hex << dwDesiredAccess << std::endl;
    std::cout << "dwShareMode :  0x" << std::hex << dwShareMode << std::endl;
    std::cout << "lpSecurityAttributes :  0x" << std::hex << lpSecurityAttributes << std::endl;
    std::cout << "dwCreationDisposition :  0x" << std::hex << dwCreationDisposition << std::endl;
    std::cout << "dwFlagsAndAttributes :  0x" << std::hex << dwFlagsAndAttributes << std::endl;
    std::cout << "hTemplateFile :  0x" << std::hex << hTemplateFile << std::endl;

    printf("*****************************************\n\n");
    return OldCreateFile(lpFileName, dwDesiredAccess, dwShareMode, lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes, hTemplateFile);
}
```

```cpp
// 堆操作 HeapCreate HeapDestroy HeapAlloc HeapFree
static HANDLE(WINAPI* OldHeapCreate)(DWORD fIOoptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize) = HeapCreate;

extern "C" __declspec(dllexport)HANDLE WINAPI NewHeapCreate(DWORD fIOoptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize)
{
    HANDLE hHeap = OldHeapCreate(fIOoptions, dwInitialSize, dwMaximumSize);
    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);
    std::cout << "fIOoptions : " << fIOoptions << std::endl;
    std::cout << "dwInitialSize :  " << dwInitialSize << std::endl;
    std::cout << "dwMaximumSize :  " << dwMaximumSize << std::endl;
    std::cout << "hHeap :  " << hHeap << std::endl;
    printf("*****************************************\n\n");
    return hHeap;
}

static BOOL(WINAPI* OldHeapDestroy)(HANDLE) = HeapDestroy;

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, st.wSecond, st.wMilliseconds);
    std::cout << "hHeap : " << hHeap << std::endl;
    printf("*****************************************\n\n");
    return OldHeapDestroy(hHeap);
}
```

```cpp
BOOL WINAPI DllMain(HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    {
        DisableThreadLibraryCalls(hModule);
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourAttach(&(PVOID&)OldMesssageBoxW, NewMesssageBoxW);
        DetourAttach(&(PVOID&)OldMesssageBoxA, NewMesssageBoxA);
        DetourAttach(&(PVOID&)OldCreateFile, NewCreateFile);
        DetourAttach(&(PVOID&)OldHeapCreate, NewHeapCreate);
        DetourAttach(&(PVOID&)OldHeapDestroy, NewHeapDestroy);
        DetourTransactionCommit();
    }
    break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourDetach(&(PVOID&)OldMesssageBoxW, NewMesssageBoxW);
        DetourDetach(&(PVOID&)OldMesssageBoxA, NewMesssageBoxA);
        DetourAttach(&(PVOID&)OldCreateFile, NewCreateFile);
        DetourAttach(&(PVOID&)OldHeapCreate, NewHeapCreate);
        DetourAttach(&(PVOID&)OldHeapDestroy, NewHeapDestroy);
        DetourTransactionCommit();
        break;
    }
    return TRUE;
}
```

(4)创建待注入的客户端程序

为了测试前面的挂钩是否生效，可以创建一个简单的 WPF 程序，程序名为 app.exe（该程序名会在后面的“注射器“中使用），并增加一个按钮，弹出一个对话框（这样就调用了前面 DLL 中挂钩的函数 MessageBox）。代码略去。



（5）完成开启和关闭注入的工具（注射器）

新建一个控制台程序，使用 DetourCreateProcessWithDllEx 将 DLL 注入目标客户端程序，代码如下。样例代码只会检测给定的目标程序，如果需要监测所有应用，则可以采用全局钩子的形式 SetWindowsHookEx。本指导书给出截获基本的 API 实例代码，其它 API 可以参照样例，由同学们自己实现。

主文件实现：

```
#include <iostream>
#include <iostream>
#include <cstdio>
#include <windows.h>
#include <detours.h>
#pragma comment(lib, "detours.lib")
int main()
{
    //std::cout << "Hello World!\n";
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(STARTUPINFO));
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    si.cb = sizeof(STARTUPINFO);
    WCHAR DirPath[MAX_PATH + 1];
    wcscpy_s(DirPath, MAX_PATH, L"C:\\softwareSecurity\\FCDll\\Debug"); //Dll的文件夹
    char DLLPath[MAX_PATH + 1] = "C:\\softwareSecurity\\FCDll\\Debug\\FCDll.dll";//Dll的地址
    WCHAR EXE[MAX_PATH + 1] = { 0 };
    wcscpy_s(EXE, MAX_PATH, L"C:\\softwareSecurity\\APP\\Debug\\APP.exe");//需要注入程序的地址
```

```
    if (DetourCreateProcessWithDllEx(EXE, NULL, NULL, NULL, TRUE,
        CREATE_DEFAULT_ERROR_MODE | CREATE_SUSPENDED, NULL, DirPath, &si, &pi,
        DLLPath, NULL))
    {   //MessageBoxA(NULL, "INJECT", "INJECT", NULL);
        ResumeThread(pi.hThread);
        WaitForSingleObject(pi.hProcess, INFINITE);
    }
    else
    {   char error[100];
        sprintf_s(error, "%d", GetLastError());//MessageBoxA(NULL, error, NULL, NULL);
    }
    return 0;
}
```