

[software_security](#)

Doc

摘要

1 课程设计

- 1.1 设计目的
- 1.2 设计要求
- 1.3 系统环境

2 系统方案设计

- 2.1 总体设计
- 2.2 UI设计
- 2.3 DLL设计
- 2.4 后端(注射器)设计

3 系统实现

- 3.1 UI实现
- 3.2 DLL实现
 - 3.2.1 截获WINAPI
 - 3.2.2 获取参数信息
 - 3.2.3 发送信息至后端
- 3.3 后端实现
 - 3.3.1 启动dll
 - 3.3.2 接收信息分析恶意行为
 - 3.3.3 发送信息至UI程序

4 系统测试

- 4.1 测试样例

software_security

HUST Software Security Curriculum Design

华中科技大学软件安全课程设计

Doc

摘要

本项目为主要用于监测代码的行为，观察其调用了哪些基本WINAPI函数，分为前后端两个部分，前端负责与用户交互、启动后端，后端监测数据返回给前端显示在Interface上。

后端设计上使用Microsoft提供的Detours OpenSource项目，使用其WINAPI挂钩和线程注入技术，记录接收到的WINAPI信息，并联合多个信息来判断此操作的危险性。对于如何接收到被Hook的程序的信息，我们使用的是WINAPI的管道（Pipe）技术，其基本上就是多开一个文件流，使得两个进程之间能够相互通信。

前端设计上使用Google OpenSource的语言Dart和其跨平台UI框架Flutter，Dart特点为Debug时为解释型语言，Release时为编译型语言，这就保证了Debug的便利性和Release后的高性能；而Flutter也是高性能的框架，他使用Skia的图形引擎自绘而不是链接到Native的UI库，保证了它在跨平台同时所拥有的高性能。对于前端线程和后端线程的管理，由于后端要监听来自Hook的程序的监控数据，所以必然是阻塞的，为了不影响到UI线程影响到用户与Interface的交互，两个线程必须分开，我这里使用的是Flutter里的Isolate技术。正如其名，两个线程完全分

开，甚至内存都不能共享，失去了便利性但是保证了多线程的安全性，接着使用Port进行Isolate间通信，前端通过用户交互启动后端Isolate开始监测。监测File、Net、Reg等基本操作和判断危险操作，最终生成恶意样本监控报告。

关键词：WINAPI Hook；Detours；恶意代码监测；Flutter；Dart；Skia；Isolate；Pipe；危险行为检测；

1 课程设计

1.1 设计目的

设计用于监测代码的行为，观察其调用了哪些基本WINAPI函数，完成对目标程序的行为监控，分析目标程序对于注册表、文件等等的可能恶意行为，判断是否为恶意程序以及有哪些恶意操作，并展示到前端与用户交互。

1.2 设计要求

本次课程设计主要是利用Detours提供的接口，首先完成对指定API的截获，其次是基于上部分完成对软件行为的分析。在API截获部分，实现基本的第三方进程WindowsAPI 截获框架、堆操作 API截获、文件操作API截获、注册表操作API截获、堆操作异常行为分析、文件操作异常行为分析、注册表操作异常行为分析、提供系统界面、行为检测样本库、网络通信操作异常行为分析、内存拷贝监测与关联分析。具体任务书如下表所示。

1.3 系统环境

系统环境：macOS Monterey 12.6, Windows 10（编译DLL所需Windows和Detours库）

IDE：Clion & Android Studio & Visual Studio Code

编译器、框架版本：Flutter 3.3.0、Dart 2.18.0、MSVC、Detours

Release executable架构：PE32+ executable (GUI) x86-64, for MS Windows

DLL 架构：PE32+ executable (DLL) (console) x86-64, for MS Windows

2 系统方案设计

2.1 总体设计

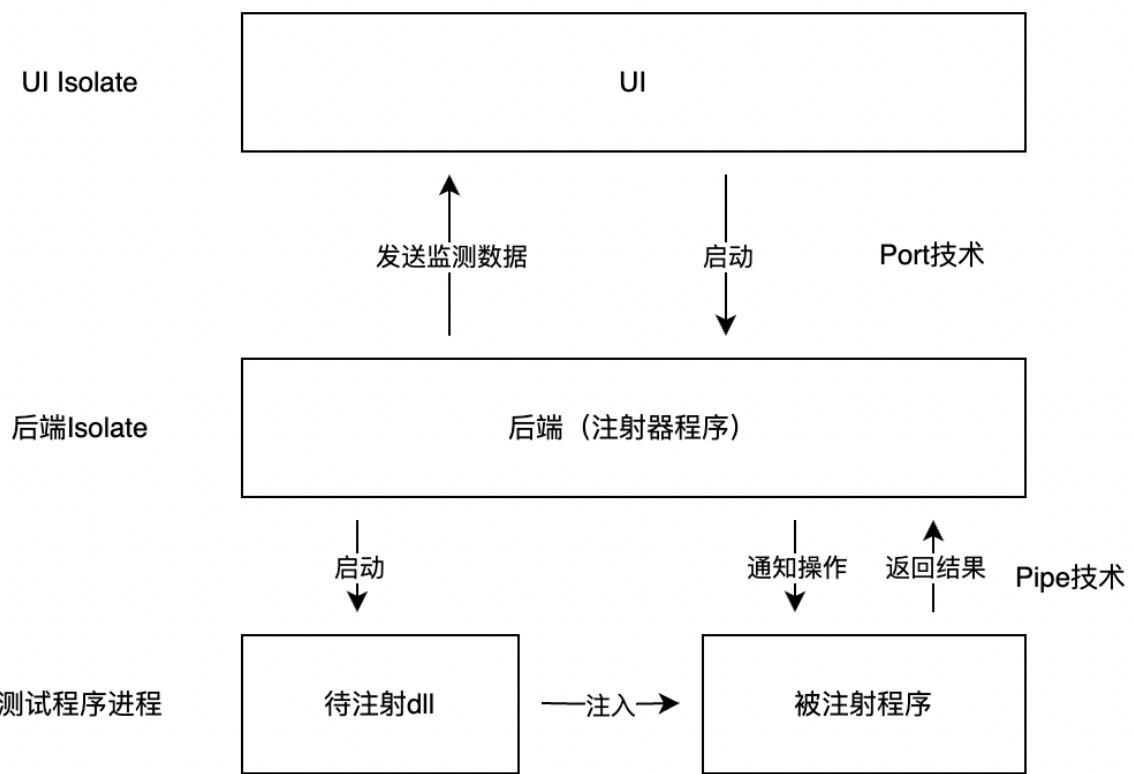
系统总体分为三个模块，

1. UI界面程序，用于与用户交互，接受并展示监测到的数据，启动后端。
2. 注射器程序（后端），负责启动 dll 注入目标程序，接收截获之后的数据以及对数据的分析。
3. 待注射的 dll 程序，其中包含对WINAPI 的截获功能，以及对截获到的函数参数信息的处理，通过Pipe发送到后端。

功能表：

模块	Function1	Function2	Function3
UI界面程序	启动后端	接受并展示监测到的数据	
注射器程序（后端）	启动 dll 注入目标程序	接收截获之后的信息	对数据的分析
待注射的 dll 程序	对 WINAPI 的截获	截获到的函数参数信息的处理	发送数据到后端

总体结构如图所示：

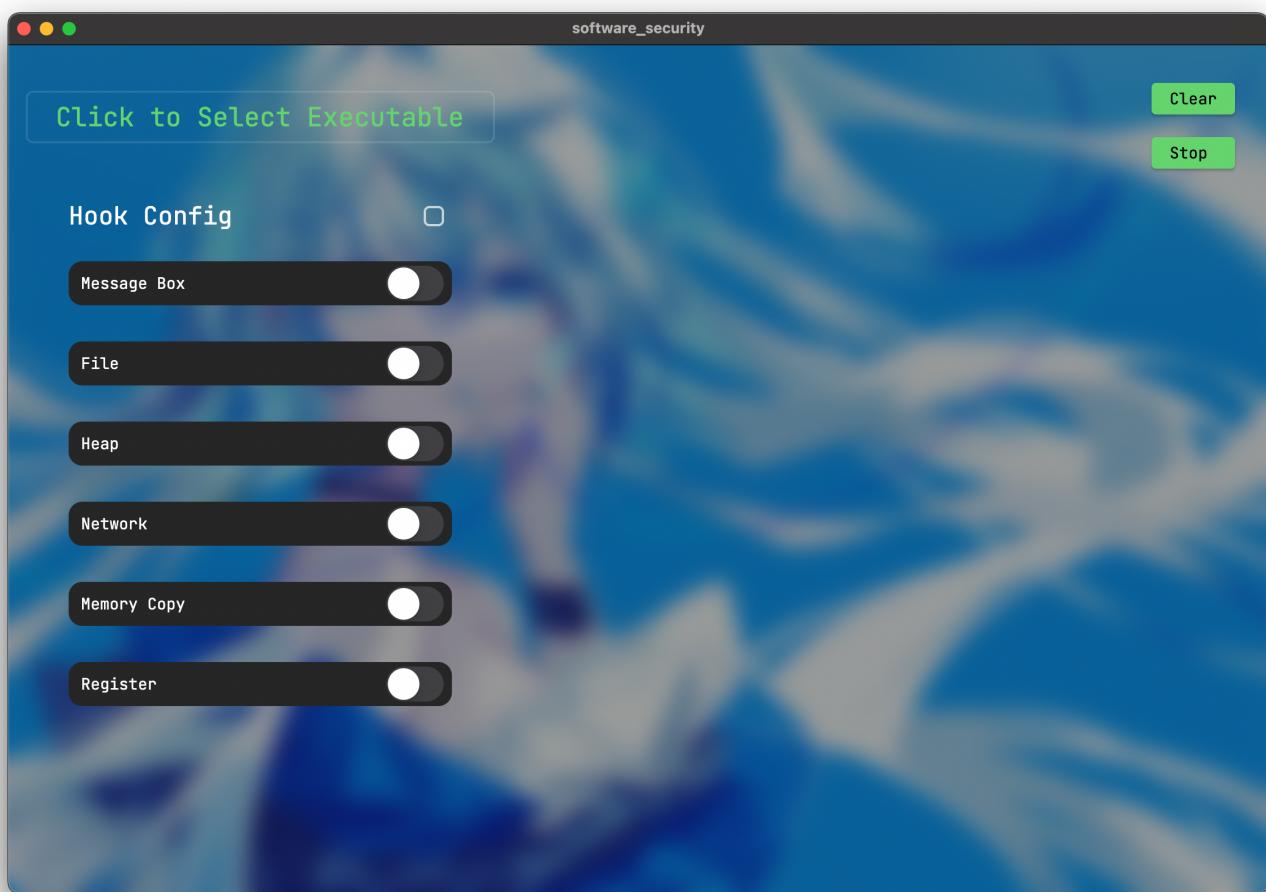


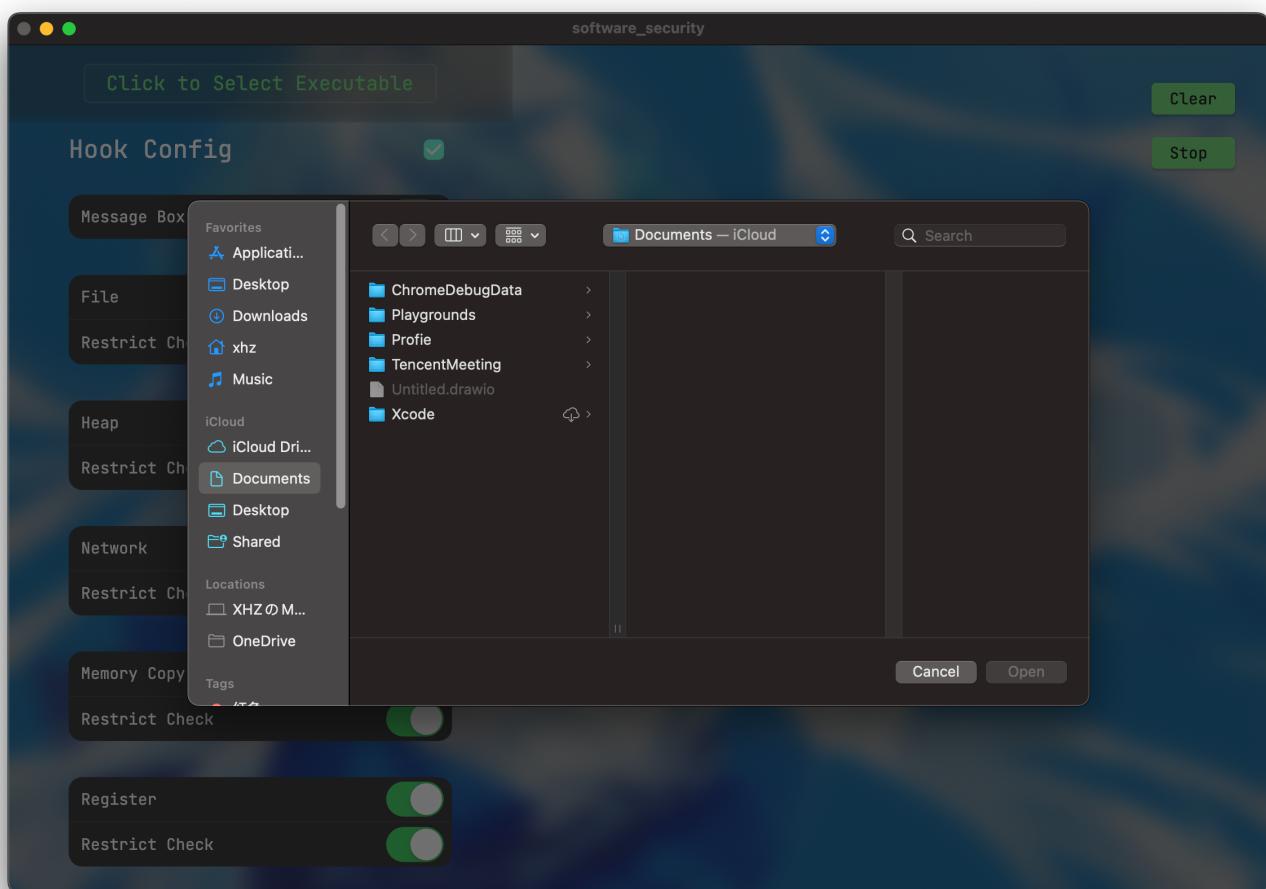
2.2 UI设计

该节截图是在macOS运行下的demo结果

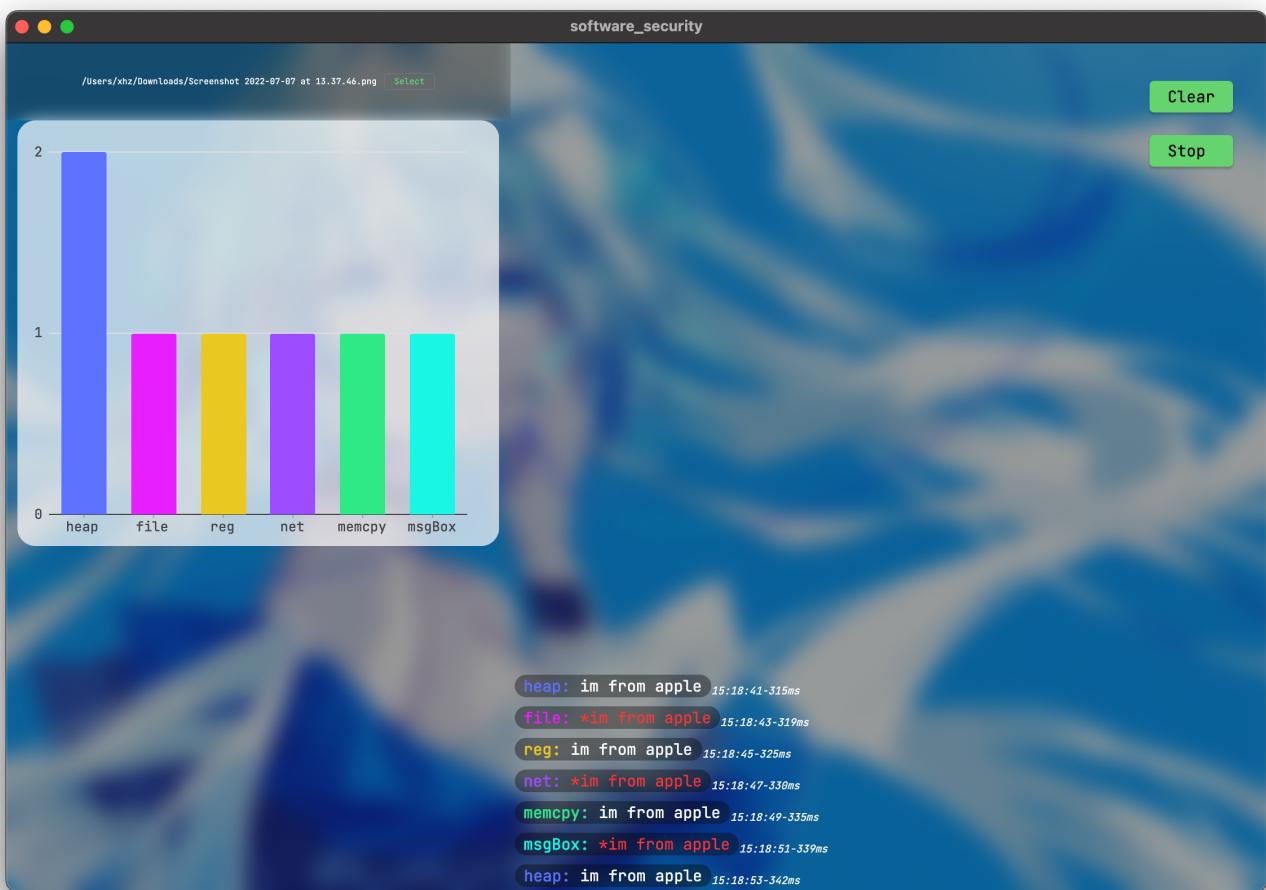
整个界面设计以简洁明了为主，进入主要就是选择hook执行程序和配置hook哪些行为，可以选择的Hook参数有

1. Message Box 此时会对消息弹出框进行截获并打印参数到UI上
2. File 此时会对文件操作相关的WINAPI（比如CopyFile）进行截获并打印参数到UI上
3. Heap此时会对堆操作相关的WINAPI（比如HeapCreate）进行截获并打印参数到UI上
4. Network此时会对网络操作相关的WINAPI（比如connect(SOCKET, const sockaddr *, int)）进行截获并打印参数到UI上
5. Memory Copy此时会对内存相关的WINAPI进行截获并打印参数到UI上
6. Register此时会对注册表相关的WINAPI（比如RegOpenKey）进行截获并打印参数到UI上





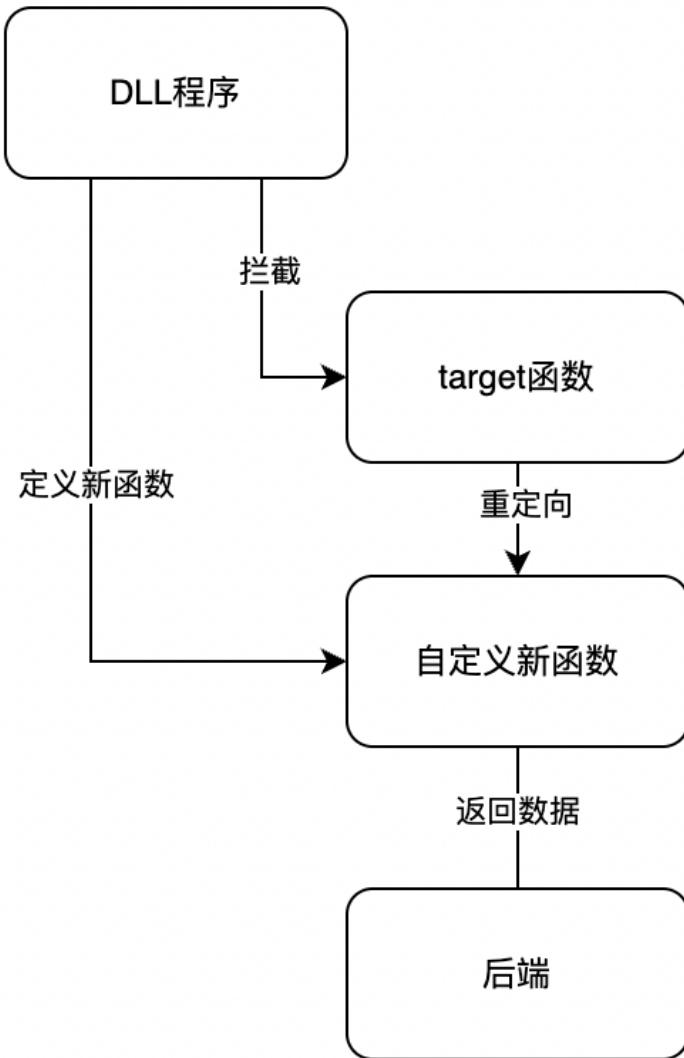
选择完executable后会立即进入注入-运行阶段，开始监听特定WINAPI函数的调用，右侧变为输出参数面板，左侧变为动画图表来统计当前已输出的数据的各个种类的分布，clear按钮可以清空输出，stop按钮可以停止hook回到初始页面



2.3 DLL设计

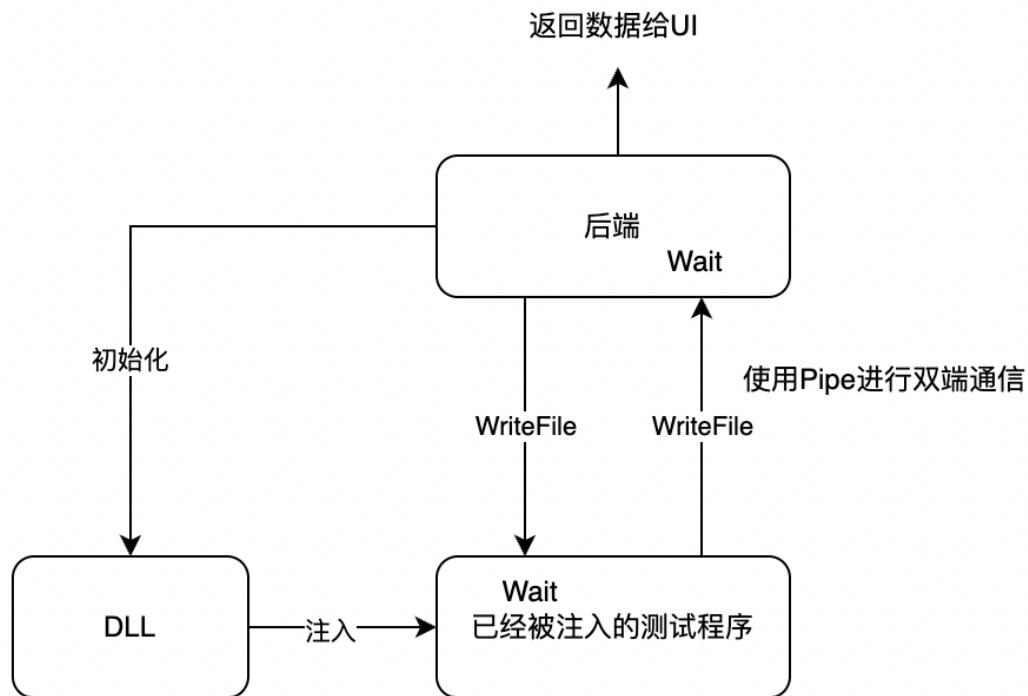
dll程序主要负责的部分是对目标程序具有的WINAPI进行截获，关于截获方面detours库将会定义三个函数，分别是Target函数、Trampoline函数以及Detour 函数，其中detours会在函数头部加入JMP至自己定义的detour函数的指令，将target的调用引导至detour。

在拦截函数之后，可以获取到函数的参数内容，同时考虑到参数类型的不同会做一定处理后发送至注射器程序进行分析。dll程序整体设计思路如下图所示



2.4 后端(注射器)设计

后端主要负责启动dll注入目标程序同时启动目标程序，并在dll启动后接收dll发送过来的参数信息进行恶意行为分析，并将数据汇总发送至UI程序。设计思路如下图所示



3 系统实现

整体使用git进行代码储存和多人协作 ([Github repo](#))

项目文件夹结构：

ci是后端和dll源代码位置， lib下是UI源代码位置， test下是测试executable位置， 其他是各个平台与Flutter的Native链接

ci	修改判断条件及dll
fonts	Add JB font, stop action
images	Improve UI, Add config
lib	Add charts ui and Update test exe
macos	[Add, Make]: change c interface, improve ui
test	Add charts ui and Update test exe
web	init repo
windows	Rename to .cpp, Add setdll code

3.1 UI实现

使用dart_ffi实现对C interface的function的直接调用以及C直接回调Dart函数

首先在后端的头文件中定义要export的函数和结构体

```
typedef long long ci_time_t;

/// struct_attach_ data type
/// configurations on every bit
#define msg_box_t          0b1
#define heap_basic_t        0b10
#define file_basic_t        0b100
#define reg_basic_t         0b1000
#define net_basic_t         0b10000
#define memcpy_basic_t      0b100000
#define heap_restrict_t     0b1000000
#define file_restrict_t     0b10000000
#define reg_restrict_t      0b100000000
#define net_restrict_t       0b1000000000
#define memcpy_restrict_t   0b10000000000
#define restrict_t          0b100000000000

typedef unsigned int u32_t;

typedef struct struct_send_ {
    const u32_t type;
    const char *str;
} *send_data_t;

typedef void send_fn_t(send_data_t);

typedef struct struct_attach_ {
    send_fn_t *send_fn;
    const ci_time_t time;
    const u32_t type;// configurations on every bit
    const char *executable_path;
} *attach_data_t;

extern "C" void ci_init(attach_data_t);
```

然后generate在dart语言下对应的函数和struct

```
import 'dart:ffi' as ffi;

/// ffi lib
class ffilib {
    /// Holds the symbol lookup function.
    final ffi.Pointer<T> Function<T extends ffi.NativeType>(String symbolName)
```

```

    _lookup;

    /// The symbols are looked up in [dynamicLibrary].
    ffilib(ffi.DynamicLibrary dynamicLibrary) : _lookup = dynamicLibrary.lookup;

    /// The symbols are looked up with [lookup].
    ffilib.fromLookup(
        ffi.Pointer<T> Function<T extends ffi.NativeType>(String symbolName)
            lookup)
        : _lookup = lookup;

void ci_init(
    attach_data_t arg0,
) {
    return _ci_init(
        arg0,
    );
}

late final _ci_initPtr =
    _lookup<ffi.NativeFunction<ffi.Void Function(attach_data_t)>>('ci_init');
late final _ci_init = _ci_initPtr.asFunction<void Function(attach_data_t)>();
}

class struct_send_ extends ffi.Struct {
    @u32_t()
    external int type;

    external ffi.Pointer<ffi.Char> str;
}

typedef u32_t = ffi.UnsignedInt;

class struct_attach_ extends ffi.Struct {
    external ffi.Pointer<send_fn_t> send_fn;

    @ci_time_t()
    external int time;

    @u32_t()
    external int type;

    external ffi.Pointer<ffi.Char> executable_path;
}

typedef send_fn_t = ffi.NativeFunction<ffi.Void Function(send_data_t)>;
typedef send_data_t = ffi.Pointer<struct_send_>;
typedef ci_time_t = ffi.LongLong;
typedef attach_data_t = ffi.Pointer<struct_attach_>;

```

UI的细节就不讲了0.0

讲下后端Isolate和UI main Isolate的通信

首先init时会传进来自目标程序路径和config，接着spawn一个Isolate，传过去一个sendport，在当前Isolate监听那边send过来的消息。

等那边也传过来一个sendPort，就实现了双线链接，这边可以listen也可以send，那边也如此。

```
Isolate? _iso;
ReceivePort? _receivePort;
SendPort? _rSendPort;
SendPort? _sendPort;
ReceivePort? _rRcvPort;

void initLib(String path, int config) async {
  _receivePort = ReceivePort();
  _iso = await Isolate.spawn(_newIsolate, _receivePort!.sendPort); //传sendPort
  _receivePort!.listen((message) {
    if (message is SendPort) { // 传过来了SendPort
      _rSendPort = message; //ready
      _rSendPort!.send(LocalizedSentData(config, LIB_START_SIG, path));
    } else {
      final data = message as LocalizedSentData;
      if (data.type == send_data_to_header) {
        ffi_channel_str.value = data.str;
      } else {
        ffi_channel_list.add(data);
        channelStreamController.sink.add(LIST_INCREASE);
      }
    }
  });
}

void _newIsolate(SendPort sendPort) {
  _sendPort = sendPort;
  _rRcvPort = ReceivePort();
  sendPort.send(_rRcvPort!.sendPort); //send过去port
  _rRcvPort!.listen(iso_listen); //listen当前Isolate
}

void iso_listen(message) async {
  final msg = message as LocalizedSentData;
  if (msg.time == LIB_START_SIG) {
    ffi.Pointer<send_fn_t> fn = ffi.Pointer.fromFunction(_callback);
    final data = calloc.allocate<struct_attach_>(ffi.sizeOf<struct_attach_>());
    data.ref.executable_path = msg.str.toNativeUtf8().cast();
    data.ref.time = DateTime.now().millisecondsSinceEpoch;
    data.ref.type = msg.type;
    data.ref.send_fn = fn;
    _lib.ci_init(data);
  }
}
```

```
    calloc.free(data);
}
}
```

3.2 DLL实现

3.2.1 截获WINAPI

使用detours对api的截获有静态与动态两种方法，此处使用静态方法进行截获。以MessageBoxW为例，首先声明MessageBoxW的静态Trampoline函数：

```
static int (WINAPI* OldMessageBoxW)(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType) = MessageBoxW;
```

然后加入目标函数的detour函数：

```
extern "C" __declspec(dllexport) int WINAPI NewMessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType){
    return OldMessageBoxW(hWnd, lpText, lpCaption, uType);
}
```

最后在入口函数中调用detours库中的DetourAttach函数将目标函数进行截获和替换，以及调用DetourDetach解除hook：

```
DetourAttach(&(PVOID&)OldMessageBoxW, NewMessageBoxW);
DetourDetach(&(PVOID&)OldMessageBoxW, NewMessageBoxW);
```

对于每一种不同类型的操作，分别截获了若干不同数量的函数，如表3-1所示

表3-1 截获的函数

操作类型	函数名称	函数功能
消息框	MessageBoxW	消息框弹出（宽字符）
	MessageBoxA	消息框弹出（窄字符）
堆操作	HeapCreate	创建堆
	HeapDestroy	销毁堆
	HeapAlloc	申请堆
文件操作	HeapFree	释放堆
	CreateFile	创建或打开文件
	ReadFile	读文件
文件操作	WriteFile	写文件
	CopyFile	复制文件
	RegOpenKeyEx	打开注册表键
注册表操作	RegCreateKeyEx	创建注册表键
	RegQueryValueEx	读取注册表键
	RegSetValueEx	设置注册表键值
	RegCloseKey	关闭注册表项句柄
	RegDeleteKeyEx	删除注册表键
	RegDeleteValue	删除注册表键值
	RegDeleteTree	删除键（除了settings）
	RegSetKeyValue	设置注册表项与子项
	connect	连接服务器
网络操作	bind	绑定地址
	recv	接收数据
	send	发送数据

3.2.2 获取参数信息

当截获到目标函数之后，函数的参数信息可以直接获取，但考虑到部分参数类型为句柄或者宏定义，如果直接输出的话为十六进制地址值或乱码，所以需进行本地化处理。

例如ReadFile中的hFile参数，类型为文件句柄，函数通过此句柄对相应文件进行操作，但如果直接输出hFile则无法知道是何处的文件，所以需要通过句柄获取文件所在路径，输出路径信息而不是输出句柄地址

通过调用fileapi.h中的函数获取文件路径:

```
TCHAR file_path[size];
GetFinalPathNameByHandle(hFile,file_path, size,FILE_NAME_NORMALIZED);
```

其次是项目使用UNICODE字符集，在字符串不同类型之间会产生冲突或错误，需要在不同类型之间进行转化。

例如注册表函数RegOpenKeyEx中，参数lpsubkey为LPCWSTR类型，即为宽字节类型（wchar），需要转换为多字节（char）型才能正常输出

通过WideCharToMultiByte函数转换宽字节类型:

```
LPSTR pchar = new char[num];
WideCharToMultiByte(CP_OEMCP, NULL, lpSubKey, -1, pchar, num, NULL, FALSE);
```

3.2.3 发送信息至后端

在截获函数获取到参数信息后，需要将信息发送至注射器程序，考虑到是在两个进程之间进行通信，采用了命名管道实现。

在注射器创建管道之后，首先调用WaitNamedPipe连接管道，若连接失败会返回false

```
WaitNamedPipe ( pipeName, NMPWAIT_WAIT_FOREVER )
```

然后打开管道并获得管道句柄，打开失败则返回INVALID_HANDLE_VALUE

```
hpipe = CreateFile (pipeName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
```

在发送之前定义了一个用于保存数据的结构体，发送与接收时都会按照结构体大小字节数进行，保证信息的完整性

```
struct argument{
    u32_t type;
    int argNum; //参数数量
    char function_name[20] = { 0 }; //函数名称
    char arg_name[10][30] = { 0 }; //参数名称
    char value[10][150] = { 0 }; //参数内容
};
```

在截获函数之后，将函数的参数信息处理并保存在结构体中，然后将结构体内容通过字节流发送至管道内：

```
memcpy(SendBuffer, &arg, sizeof(argument));
WriteFile(hPipe, SendBuffer, (DWORD)sizeof(argument), &dwWriteLen, NULL);
```

值得注意的是部分函数会在程序运行过程中调用很多次，导致显示界面被迅速占满，所以需要对其进行筛选过滤，只显示有意义的部分

例如HeapAlloc函数，经测试在运行过程中会被大量调用，所以这里创建了一个unordered_set无序容器，用于保存HeapCreate中创建的堆句柄，之后截获HeapAlloc时，只输出那些是申请容器中堆的HeapAlloc信息。

创建一个容器保存堆句柄：

```
unordered_set<HANDLE> hHeaps;
```

每次创建堆时，将创建的堆的句柄hHeap添加至容器：

```
hHeaps.emplace(hHeap);
```

当每次HeapAlloc申请的堆不是上述创建的堆时，即 `hHeaps.find(hHeap) == hHeaps.end()`，则直接返回而不输出信息，达到筛选过滤的功能。

最后在hook结束时关闭管道 `CloseHandle(hPipe)`

3.3 后端实现

3.3.1 启动dll

主要通过detours库中的DetourCreateProcessWithDllEx函数启动dll注入到目标程序，
DetourCreateProcessWithDllEx函数的参数定义如下所示

表3-1 DetourCreateProcessWithDllEx函数

参数名称	参数类型	参数含义
lpApplicationName	LPCTSTR	CreateProcess API定义的应用程序名称
lpCommandLine	LPTSTR	CreateProcess API定义的命令行
lpProcessAttributes	LPSECURITY_ATTRIBUTES	为CreateProcess API定义的流程属性
lpThreadAttributes	LPSECURITY_ATTRIBUTES	为CreateProcess API定义的线程属性
bInheritHandles	BOOL	继承CreateProcess API定义的句柄标志
dwCreationFlags	DWORD	为CreateProcess API定义的创建标志
lpEnvironment	LPVOID	为CreateProcess API定义的进程环境变量
lpCurrentDirectory	LPCTSTR	为CreateProcess API定义的进程当前目录
lpStartupInfo	LPSTARTUPINFO	为CreateProcess API定义的进程启动信息
lpProcessInformation	LPPROCESS_INFORMATION	为CreateProcess API定义的进程句柄信息
lpDllName	LPCSTR	要插入到新进程的DLL的路径名
pfCreateProcessW	PDETOUR_CREATE_PROCESS_ROUTINEW	指向特定程序的CreateProcess API替换的指针

使用DetourCreateProcessWithDllEx函数启动dll注入目标程序：

```
DetourCreateProcessWithDllEx(EXE, NULL, NULL, NULL, TRUE, CREATE_DEFAULT_ERROR_MODE |  
CREATE_SUSPENDED, NULL, DirPath, &si, &pi, DllPath, NULL);
```

其中DirPath为目标程序路径， DllPath为dll所在路径， 函数创建了目标进程并将dll注入。

3.3.2 接收信息分析恶意行为

在注射器与dll间通信采用了命名管道的方式，在启动dll之前，首先初始化管道相关数据，通过CreateNamedPipe创建管道，其中管道传输类型为异步读写。函数创建成功则会获得管道句柄：

```
CreateNamedPipe(pipeName, PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED, 0, 1, size, size, 0, NULL);
```

然后创建事件对象：

```
CreateEvent(NULL, TRUE, FALSE, NULL);
```

最后创建与管道的连接准备进行信息的发送和接收：

```
ConnectNamedPipe(hPipe, &ovlap);
```

管道创建完毕后则启动dll注入目标程序，在dll中进行另一方进程与管道的连接与发送信息，这时在注射器进程可以执行信息的接收，这里使用ReadFile函数从管道句柄中接收信息

每次读取结构体大小的字节数，保证信息完整

```
ReadFile(hPipe, Buffer, sizeof(argument), &dwReadLen, NULL);
```

在收到参数信息以后便进行异常行为的分析，包含对文件操作、堆操作、注册表操作等可能的异常进行监测。

1) 文件操作异常

1. 判断操作范围是否有多个文件夹

考虑到常见对文件的操作几乎都会执行CreateFile函数，如fopen等，所以对CreateFile进行监测。创建了一个无序容器，每次通过CreateFile的lpFileName参数即操作对象的文件路径，获取其所在的文件夹路径，并保存至容器中；因为容器中的元素互不相同，所以判断容器中元素的数量，若大于等于2则代表操作范围有多个文件夹。

2. 是否存在自我复制的情况

通过监测两个函数判断是否有自我复制，其一为CopyFile函数，若程序调用了CopyFile函数，且CopyFile的lpExistingFileName参数为程序本身，则可认为有自我复制的可能；其二为CreateFile函数，若dwDesiredAccess为读(GENERIC_READ)或读写(GENERIC_READ | GENERIC_WRITE)，且新创建的文件名与原程序相同，则也可认为有自我复制的可能。

3. 是否修改了其它可执行代码包括exe, dll, ocx等

同样监听CreateFile函数，判断lpFileName参数即文件路径后缀名是否为exe、dll、ocx等，若有则认为可能修改可执行代码。

4. 是否将文件内容读取后发送到网络

监测CreateFile与Send函数，当CreateFile中dwDesiredAccess参数含有可读(GENERIC_READ)，且紧跟有Send函数执行，则可认为可能将文件内容发送至网络。

2) 堆操作异常

1. 创建与销毁是否一致

在dll中创建了容器hHeaps，每次执行HeapCreate创建堆时，会向容器中添加创建的堆的句柄，在后续HeapDestroy销毁堆时会在容器中删除对应的堆句柄，在每次删除时首先判断待删除的堆句柄是否在容器里，若不在则可认为创建与销毁不一致，并设置一个ERROR标志，若在注射器接收到了ERROR，则可认为创建与销毁可能不一致。

2. 申请与释放是否一致

与创建、销毁类似，定义一个Hallocs无序容器，在每次申请堆时将申请到的有效的内存指针添加至容器里，在后续HeapFree释放时判断待释放的内存指针是否在容器，若不在则可认为申请与释放不一致，有重复释放的可能，同时设置一个ERROR标志，发送至注射器。

3) 注册表异常

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

通过监测RegCreateKeyEx函数判断是否有新增项创建，同时在dll中当RegCreateKeyEx创建失败时会发送ERROR标志，用以判断创建成功；并且通过RegCreateKeyEx函数中hKey句柄可以得到注册表项的完整路径，判断其是否在自启动文件项中（SOFTWARE\Microsoft\Windows\CurrentVersion\Run）。

2. 是否修改了注册表

修改注册表可以是写入或删除，所以监测的函数包含了RegSetValueEx、RegDeleteTree、RegDeleteKey、RegSetValue，若函数执行成功，则可认为对注册表进行了修改，并且同样通过其函数参数hKey即注册表句柄获得了修改目标的完整路径，判断是否为自启动项。

3. 输出所有的注册表操作项

在每次截获注册表相关函数时，都通过hKey参数即待操作的注册表项句柄获取到目标项的完整路径并输出。

3.3.3 发送信息至UI程序

通过UI程序定义的send函数，每次获取数据时调用send函数将数据发送至UI。

4 系统测试

4.1 测试样例

测试程序包含了个人Debug时编写的测试样例与小组同学编写的测试样例，涵盖了大部分待截获函数与异常行为。

测试样例详细信息如下表所示

程序名称	消息框	堆	注册表	文件	网络
file.exe				√	√
messagebox.exe	√	√		√	
heap.exe		√			
register.exe			√		
copyfile.exe				√	
socket.exe				√	√