最强黑客库Blackbone使用教程

鬼手56 于 2022-01-14 19:37:49 发布 6114 收藏 43

分类专栏： 软件逆向 文章标签： 安全 web安全

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/qq_38474570/article/details/122466905

版权

软件逆向 专栏收录该内容

17 篇文章 109 订阅

订阅专栏

文章目录

- 环境搭建
  - 项目地址
  - 项目介绍
  - 项目编译
  - 项目集成
- 项目使用
  - 进程交互
    - 根据进程名枚举所有进程ID
    - 根据进程ID或进程名枚举进程信息
    - 枚举所有进程
    - 附加一个进程(打开进程)，并获取相关信息
    - 启动并附加进程
  - 进程模块管理
  - 进程内存管理
  - 进程线程管理
  - JIT汇编
  - 远程代码执行
  - 内存搜索匹配
  - 远程函数调用
  - 系统调用 只适用于x64
  - dll隐藏注入
- 总结

环境搭建

项目地址

https://github.com/DarthTon/Blackbone

项目介绍

作为Windows开发人员，经常遇到枚举进程、枚举模块、读写进程内存的操作；Windows安全开发人员更是会涉及注入、hook、操作PE文件、编写驱动。每次都要翻各种资料制造轮子，那么有没有好的开源库解决这个问题呢，目前知道的就Blackbone了，而且它的代码写的很好，完全可以作为教科书来用。

网上对于各种类型的封装库百分之99都是copy的这个项目，并且看雪很多大佬的项目都引用了其中的代码。由此可见，这个项目的重要性和牛逼程度就不言而喻了。

库内容包含

```
Process interaction（进程交互）
    操作PEB32/PEB64
    通过WOW64 barrier管理进程
Process Memory（进程内存）
    分配和释放虚拟内存
    改变内存保护属性
    读写虚拟内存
Process modules（进程模块）
    枚举所有进程加载的模块 (32/64 bit)
    获得导出函数地址
    获得主模块
    从模块列表中去除模块信息
    注入、卸载模块(支持 pure IL images)
    在WOW64进程中注入64位模块
    操作PE
Threads（线程）
    枚举线程
    创建和关闭线程
    获得线程退出码
    获得主线程
    管理 TEB32/TEB64
    join线程（等待线程退出）
    暂停、恢复线程
    设置、清除硬件断点
Pattern search
    特征码匹配（支持本进程和其他进程）
Remote code execution
    在远程进程中执行函数
    组装自己的代码并远程执行（shellcode注入）
    支持各种调用方式： cdecl/stdcall/thiscall/fastcall
    支持各种参数类型
    支持FPU
    支持在新线程或已经存在的线程中执行shellcode
Remote hooking
    通过软硬断点，在远程进程中hook函数
    Hook functions upon return（通过return挂钩函数）
Manual map features
    隐藏注入，支持x86和x64，注入任意未保护的进程，支持导入表和延迟导入等等特性，不一一列举了。
Driver features
    分配、释放、保护内存
    读写用户层、驱动层内存
    对于WOW64进程，禁用DEP
    修改进程保护标记
    修改句柄访问权限
    操作进程内存（如：将目标进程map到本进程等）
    隐藏已分配用户模式内存
    用户模式dll注入；手动mapping（手动加载模块）
    手动加载驱动
```

项目编译

需要安装下面几个组件

1. sdk、wdk版本要相同10.0.17763.0
2. cor.h文件及相关的mscoree.lib库找不到编译错误，需要安装.NET桌面开发
3. VC++ 2017 Libs for Spectre (x86 and x64)
4. Visual C++ ATL (x86/x64) with Spectre Mitigations

安装详细信息

- Visual Studio 核心编辑器
- .NET 桌面开发
- 使用 C++ 的桌面开发
- 单个组件
  - 静态分析工具
  - VC++ 2017 version 15.9 v14.16 latest v141 to...
  - 用于 x86 和 x64 的 Visual C++ ATL
  - 用于 x86 和 x64 的 Visual C++ MFC
  - 标准库模块(实验性)
  - 带有 Spectre 缓解措施的 Visual C++ ATL (x86...
  - 带有 Spectre 缓解措施的 Visual C++ MFC for...
  - VC++ 2017 version 15.9 v14.16 Libs for Spect...
  - Windows Driver Kit

CSDN @鬼手56

```
1>c:\users\edz\desktop\blackbone-master\src\blackbone\process\memblock.cpp(192): error C2280: "blackbone::MemBlock::MemBlock(const blackbone::MemBlock &)": 尝试引用已删除的函数
1>c:\users\edz\desktop\blackbone-master\src\blackbone\process\memblock.h(280): note: 编译器已在此处生成"blackbone::MemBlock::MemBlock"
1>c:\users\edz\desktop\blackbone-master\src\blackbone\process\memblock.h(280): note: "blackbone::MemBlock::MemBlock(const blackbone::MemBlock &)": 因为"blackbone::MemBlock"有一个
1>RemoteHook.cpp
1>RemoteLocalHook.cpp
1>RemoteMemory.cpp
1>Thread.cpp
1>Threads.cpp
```

CSDN @鬼手56

我用VS2017编译会报一个引用已删除函数的错误，找不到解决方案，就直接找人帮我编译了一份。

据说是最新版本只支持VS2019，如果要使用VS2017，需要选择其他分支代码。

项目集成

```
// BlackboneLearn.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
//

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

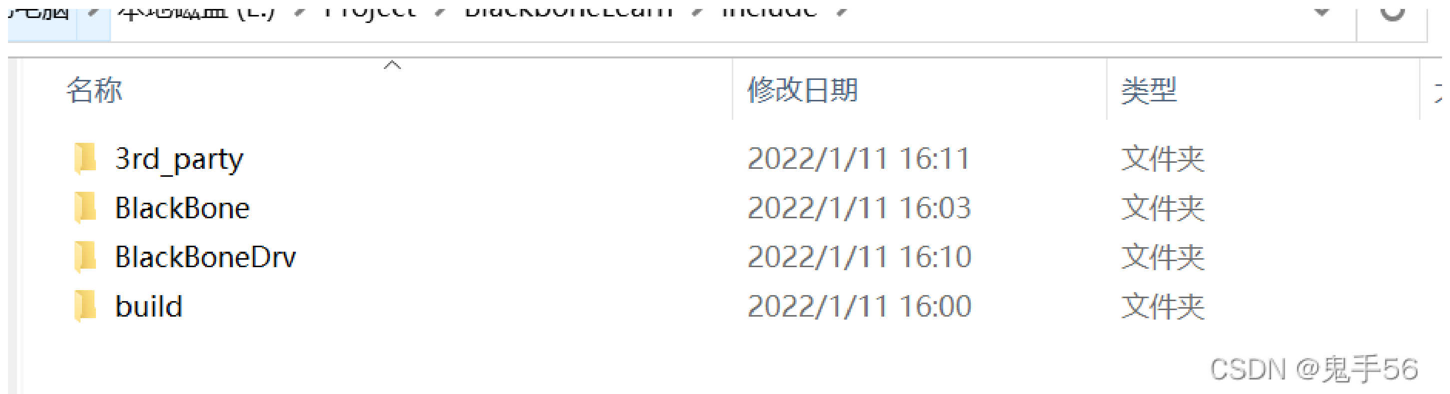
CSDN @鬼手56

新建一个空项目

名称	修改日期	类型	大小
.vs	2022/1/11 15:47	文件夹	
BlackboneLearn	2022/1/11 16:16	文件夹	
Debug	2022/1/11 16:13	文件夹	
include	2022/1/11 16:15	文件夹	
BlackboneLearn.sln	2022/1/11 15:47	Visual Studio Soluti...	2 KB

CSDN @鬼手56

然后在项目目录下新建一个`include`文件夹



把src目录下的三个文件夹复制到`include`文件夹

名称	修改日期	类型
3rd_party	2022/1/11 16:11	文件夹
BlackBone	2022/1/11 16:03	文件夹
BlackBoneDrv	2022/1/11 16:10	文件夹
build	2022/1/11 16:00	文件夹

CSDN @鬼手56

上面三个文件夹的在引用头文件的时候需要用到，`build`里面存放的是已经编译好的lib库

```
#include <iostream>
#include "../include/BlackBone/Config.h"
#include "../include/BlackBone/Process/Process.h"
#include "../include/BlackBone/Process/MultPtr.hpp"
#include "../include/BlackBone/Process/RPC/RemoteFunction.hpp"
#include "../include/BlackBone/PE/PEImage.h"
#include "../include/BlackBone/Misc/Utils.h"
#include "../include/BlackBone/Misc/DynImport.h"
#include "../include/BlackBone/Syscalls/Syscall.h"
#include "../include/BlackBone/Patterns/PatternSearch.h"
#include "../include/BlackBone/Asm/LDasm.h"
#include "../include/BlackBone/localHook/VTableHook.hpp"
#include "../include/BlackBone/DriverControl/DriverControl.h"


#ifdef _DEBUG
#pragma comment(lib, "..\\include\\build\\Win32\\Debug\\BlackBone.lib")
#else
#pragma comment(lib, "..\\include\\build\\Win32\\Release\\BlackBone.lib")
#endif
```

CSDN @鬼手56

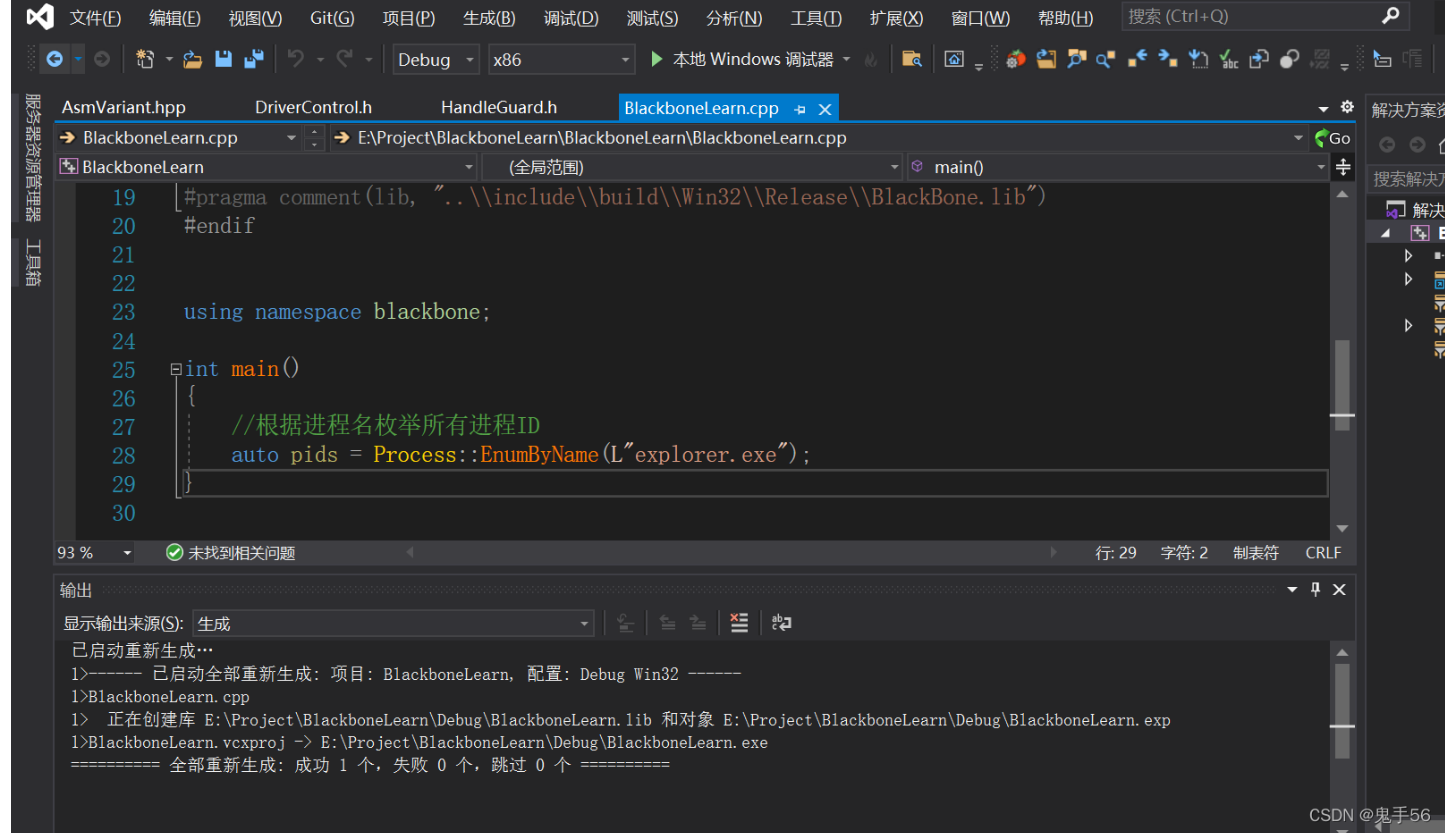
接着包含lib库和头文件

```
#include <iostream>
#include "../include/BlackBone/Config.h"
#include "../include/BlackBone/Process/Process.h"
#include "../include/BlackBone/Process/MultPtr.hpp"
#include "../include/BlackBone/Process/RPC/RemoteFunction.hpp"
#include "../include/BlackBone/PE/PEImage.h"
#include "../include/BlackBone/Misc/Utils.h"
#include "../include/BlackBone/Misc/DynImport.h"
#include "../include/BlackBone/Syscalls/Syscall.h"
#include "../include/BlackBone/Patterns/PatternSearch.h"
#include "../include/BlackBone/Asm/LDasm.h"
#include "../include/BlackBone/localHook/VTableHook.hpp"
#include "../include/BlackBone/DriverControl/DriverControl.h"


#ifdef _DEBUG
#pragma comment(lib, "..\\include\\build\\Win32\\Debug\\BlackBone.lib")
#else
#pragma comment(lib, "..\\include\\build\\Win32\\Release\\BlackBone.lib")
#endif
```

使用命名空间

```
using namespace blackbone;
```



随意调用一个函数，即可编译通过。编译通过说明集成成功。

接着需要把build生成的三个文件拷贝到项目目录下，否则会报下面的错误

```
E:\Project\BlackboneLearn\Debug\BlackboneLearn.exe
BlackBone: PDB: Failed to load msdia140.dll, error 0x0000007e
BlackBone: PDB: blackbone::PDBHelper::Init: (CoCreateDiaDataSource()) failed with HRESULT 0x8007007e
```

CSDN @鬼手56

项目使用

下面讲解每一个模块的代码功能使用。

进程交互

根据进程名枚举所有进程ID

```
auto pids = Process::EnumByName(L"explorer.exe");
```

根据进程ID或进程名枚举进程信息

```
auto procInfo = Process::EnumByNameOrPID(4596, L"");
for (auto info : procInfo.result())
{
 wcout << info.imageName << info.pid << info.threads.size();
}
```

枚举所有进程

```
auto all = Process::EnumByNameOrPID(0, L"");
for (auto info : all.result())
{
 wcout << info.imageName << "\t" << info.pid << info.threads.size() << endl;
}
```

附加一个进程(打开进程)，并获取相关信息

```
Process process;
vector<DWORD> vecPid = Process::EnumByName(L"QQ.exe");

//附加一个进程
if (!vecPid.empty()&&NT_SUCCESS(process.Attach(vecPid.front())))
{
 //ProcessCore类
 ProcessCore& processCore = process.core();

 //进程WOW64信息
 Wow64Barrier barrier = process.barrier();

 //获取进程ID和进程句柄
 DWORD dwPid = processCore.pid();
 HANDLE hProcess = processCore.handle();

 //获取进程PEB
 PEB_T peb = {};
 ptr_t ptrPEB = processCore.peb(&peb);

 //获取进程的所有句柄
 for (HandleInfo handleInfo:process.EnumHandles().result())
 {
  wcout << handleInfo.name << handleInfo.handle << handleInfo.typeName << endl;
 }
```

启动并附加进程

```
//启动并附加进程
Process process;
process.CreateAndAttach(L"D:\\InjectTool.exe", true);
{
 //恢复进程
 process.Resume();

 //延迟100毫秒
 std::this_thread::sleep_for(std::chrono::milliseconds(100));
}
```

进程模块管理

```
//进程模块管理
Process process;
if (NT_SUCCESS(process.Attach(L"WeChat.exe")))
{
 //模块管理
 ProcessModules& proModules = process.modules();

 //获取所有模块 包括x86跟x64模块
 auto mapModule = proModules.GetAllModules();

 //获取主模块
 ModuleDataPtr mainModule = proModules.GetMainModule();

 //主模块基地址
 module_t baseAddress = mainModule->baseAddress;

 //获取导出函数地址
 call_result_t<exportData> exportFunAddr = proModules.GetExport(L"kernel32.dll", "LoadLibraryW");
 if (exportFunAddr.success())
 {
  cout << "LoadLibraryW" << exportFunAddr->procAddress << endl;
 }

 //模块断链
 proModules.Unlink(mainModule);


 auto mapModule2 = proModules.GetAllModules();

       //注入模块
 call_result_t<ModuleDataPtr> ptr= proModules.Inject(L"E:\\Dll1.dll");

 //卸载模块
 proModules.Unload(ptr);
}
```

里面有一个很牛逼的功能，模块断链，实测可行~

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-oc7z6bW0-1642037164410)(Blackbone黑客库的使用.assets/image-20220112125146182.png)]

进程内存管理

```
Process process;
if (NT_SUCCESS(process.Attach(L"QQ.exe")))
{
 //操作内存类
 ProcessMemory& memory = process.memory();

 //主模块
 ModuleDataPtr  mainModoulePtr = process.modules().GetMainModule();

 //PE头信息
 IMAGE_DOS_HEADER dosHeader = {};

 //读取内存 模块的开始信息是PE头信息
 //第一种方法
 memory.Read(mainModoulePtr->baseAddress, dosHeader);

 //第二种方法
 memory.Read(mainModoulePtr->baseAddress, sizeof(dosHeader),&dosHeader);

 //第三种方法
 call_result_t<IMAGE_DOS_HEADER> dosHeaderResult = memory.Read<IMAGE_DOS_HEADER>(mainModoulePtr->baseAddress);

 //改变内存读写属性
 if (NT_SUCCESS(memory.Protect(mainModoulePtr->baseAddress,sizeof(dosHeader), PAGE_READWRITE)))
 {
  //写内存第一种方法
  memory.Write(mainModoulePtr->baseAddress, dosHeader);

  //写内存第二种方法
  memory.Write(mainModoulePtr->baseAddress, sizeof(dosHeader),&dosHeader);
 }

 //分配内存
 call_result_t<MemBlock> memBlockResult = memory.Allocate(0x1000, PAGE_READWRITE);
 if (memBlockResult.success())
 {
  //写内存
  memBlockResult->Write(0x10, 12.0);
  //读内存
  memBlockResult->Read<double>(0x10, 0.0);
 }

 //枚举所有有效内存区域
 vector<MEMORY_BASIC_INFORMATION64> region = memory.EnumRegions();
}
```

进程线程管理

```
Process process;
if (NT_SUCCESS(process.Attach(L"QQ.exe")))
{
 //枚举所有线程
 vector<ThreadPtr> vecTheadPtr = process.threads().getAll();

 //获取主线程
 ThreadPtr mainThread = process.threads().getMain();

 //根据线程ID获取线程信息
 ThreadPtr threadPtr = process.threads().get(mainThread->id());

 //获取线程上下文
 CONTEXT_T ctx = {};
 if (threadPtr->GetContext(ctx, CONTEXT_FLOATING_POINT))
 {
           //这里设置Dr寄存器就可以清除硬件断点
           //......
  //设置线程上下文
  threadPtr->SetContext(ctx);
 }

 //等待线程退出
 threadPtr->Join(100);
}
```

JIT汇编

```
AsmHelper32 asmPtr = AsmFactory::GetAssembler();
 IAsmHelper& a = *asmPtr;
 //生成函数代码
 a->GenPrologue();
 a->add(a->zcx, a->zdx);
 a->mov(a->zax, a->zdx);
 a.GenEpilogue();
 auto func = reinterpret_cast<uintptr_t(__fastcall*)(uintptr_t, uintptr_t)>(a->make());
 uintptr_t r = func(10, 5);
```

远程代码执行

```
Process process;
if (NT_SUCCESS(process.Attach(L"QQ.exe")))
{
  //远程执行类
  RemoteExec& remoteExe = process.remote();

  //创建远程执行环境
  remoteExe.CreateRPCEnvironment(Worker_None, true);

  //获取导出函数地址
  call_result_t<exportData> GetModuleHandleWPtr = process.modules().GetExport(L"kernel32.dll", "GetModuleHandleW
");
  if (GetModuleHandleWPtr.success())
  {
    //使用指定的入口点和参数创建新线程
    DWORD mod = remoteExe.ExecDirect(GetModuleHandleWPtr->procAddress, 0);

    AsmHelperPtr asmPtr = AsmFactory::GetAssembler();

    if (asmPtr)
    {
      blackbone::IAsmHelper& asmHelper = *asmPtr;
      asmHelper.GenPrologue();
      asmHelper.GenCall(static_cast<uintptr_t>(GetModuleHandleWPtr->procAddress), { nullptr }, blackbone::cc_stdca
ll);
      asmHelper.GenEpilogue();
      uint64_t result = 0;
      //创建新线程并在其中执行代码。 等待执行结束
      remoteExe.ExecInNewThread(asmHelper->make(), asmHelper->getCodeSize(), result);
    }

    //在主线程执行
    blackbone::ThreadPtr mainThreadPtr = process.threads().getMain();
    blackbone::AsmHelperPtr asmHelperPtr = blackbone::AsmFactory::GetAssembler();
    if (asmHelperPtr)
    {
      blackbone::IAsmHelper& asmHelper = *asmHelperPtr;
      asmHelper.GenPrologue();
      asmHelper.GenCall(static_cast<uintptr_t>(GetModuleHandleWPtr->procAddress), { nullptr }, blackbone::cc_stdca
ll);
      asmHelper.GenEpilogue();

      uint64_t result = 0;
      remoteExe.ExecInAnyThread(asmHelper->make(), asmHelper->getCodeSize(), result, mainThreadPtr);
    }

  }

}
```

内存搜索匹配

```
Process process;
if (NT_SUCCESS(process.Attach(L"QQ.exe")))
{
 //匹配模版
 PatternSearch ps{ 1,2,3,4,5,6,7,8,9 };
 //搜索结果
 vector<ptr_t> results;
 //内存搜索
 ps.SearchRemote(process, false, 0, results);

}
```

上面的`SearchRemote`函数是远程搜索

```
BLACKBONE_API size_t Search(
        uint8_t wildcard,
        void* scanStart,
        size_t scanSize,
        std::vector<ptr_t>& out,
        ptr_t value_offset = 0,
  size_t maxMatches = SIZE_MAX
        ) const;
```

另外还提供了一个`Search`函数支持本进程搜索吗，而且还支持通配符

远程函数调用

```
Process process;
if (NT_SUCCESS(process.Attach(L"QQ.exe")))
{
  //1.简单直接调用
  //获取远程函数对象
  RemoteFunction<decltype(&MessageBoxW)> pMessageBoxW = MakeRemoteFunction<decltype(&MessageBoxW)>(process, L"user32.dll", "MessageBoxW");
  if (pMessageBoxW.valid())
  {
   //远程调用
   auto result = pMessageBoxW(HWND_DESKTOP, L"HelloWorld", L"Title", MB_OK);
  }

  //2.使用特定线程调用
  auto mainThread = process.threads().getMain();
  if (auto pIsGuiThread= MakeRemoteFunction<decltype(&IsGUIThread)>(process, L"user32.dll", "IsGUIThread"); pIsGuiThread && mainThread)
  {
   auto result = pIsGuiThread.Call({ FALSE }, mainThread);
   if (*result)
   {
   }

  }

  //3.给定参数调用
  if (auto pMultiByteToWideChar = blackbone::MakeRemoteFunction<decltype(&MultiByteToWideChar)>(process, L"kernel32.dll", "MultiByteToWideChar"))
  {
   auto args = pMultiByteToWideChar.MakeArguments({ CP_ACP, 0, "Sample text", -1, nullptr, 0 });
   std::wstring converted(32, L'\0');

   // Set buffer pointer and size manually
   args.set(4, blackbone::AsmVariant(converted.data(), converted.size() * sizeof(wchar_t)));
   args.set(5, converted.size());

   auto length = pMultiByteToWideChar.Call(args);
   if (length)
    converted.resize(*length - 1);
  }

}
```

系统调用 只适用于x64

```
{
        uint8_t buf[32] = { };
        uintptr_t bytes = 0;

        NTSTATUS status = syscall::nt_syscall(
            syscall::get_index( "NtReadVirtualMemory" ),
            GetCurrentProcess(),
            GetModuleHandle( nullptr ),
            buf,
            sizeof(buf),
            &bytes
        );

        if (NT_SUCCESS( status ))
        {
        }
    }
```

dll隐藏注入

```
//隐藏注入dll 进程名 dll路径 适配32位和64位
void HideInject(const std::wstring& processName, const std::wstring& dllPath)
{
 vector<DWORD> vecPid = Process::EnumByName(processName);

 if (vecPid.empty())
 {
  MessageBoxA(0, "不存在目标进程", "提示", 0);
  return;
 }


 //首先要拿到目标进程的信息
 Process proc;
 proc.Attach(vecPid.front());

 //确保LdrInitializeProcess被调用
 proc.EnsureInit();

 //将PE文件映射到目标进程 1.PE文件路径 flags(手动映射导入函数) 回调函数
 auto image = proc.mmap().MapImage(dllPath, ManualImports, &MapCallback2);

 //获取导出函数地址
 auto g_loadDataPtr = proc.modules().GetExport(image.result(), "g_LoadData");

 //调用导出函数
 auto g_loadData = proc.memory().Read<DllLoadData>(g_loadDataPtr->procAddress);
}
```

`ManualMap`模块的代码利用反射型dll注入的原理，将PE文件不借助`LoadLibrary`在目标进程中展开，从而实现隐藏注入。

```
https://github.com/DarthTon/Xenos
```

另外这个作者还有一个非常强大的注入工具，支持三种零环的注入方式和两种三环的注入。不过这个工具基本都已经被国外的各大游戏检测的死死的。

总结

这个项目功能看似很强大，包含有各个模块的封装。但实际使用的时候往往只会用到其中一两个模块的功能。我只需要里面的特征码搜索和dll隐藏注入这两个功能，然后顺便学习了一下其他模块。

没想到最后集成的时候，头文件之间互相包含依赖，而且数据类型都是作者自定义的。如果想拿出其中的一个模块，最后会因为头文件包含的问题，从而把整个工程代码都包含进来。

为了一个功能而把剩下几个不需要的模块代码也加进来实在是有点鸡肋。总的来说，学习价值拉满，实用性就不得而知了。