

【老文】如何将.Net程序集注入非托管进程

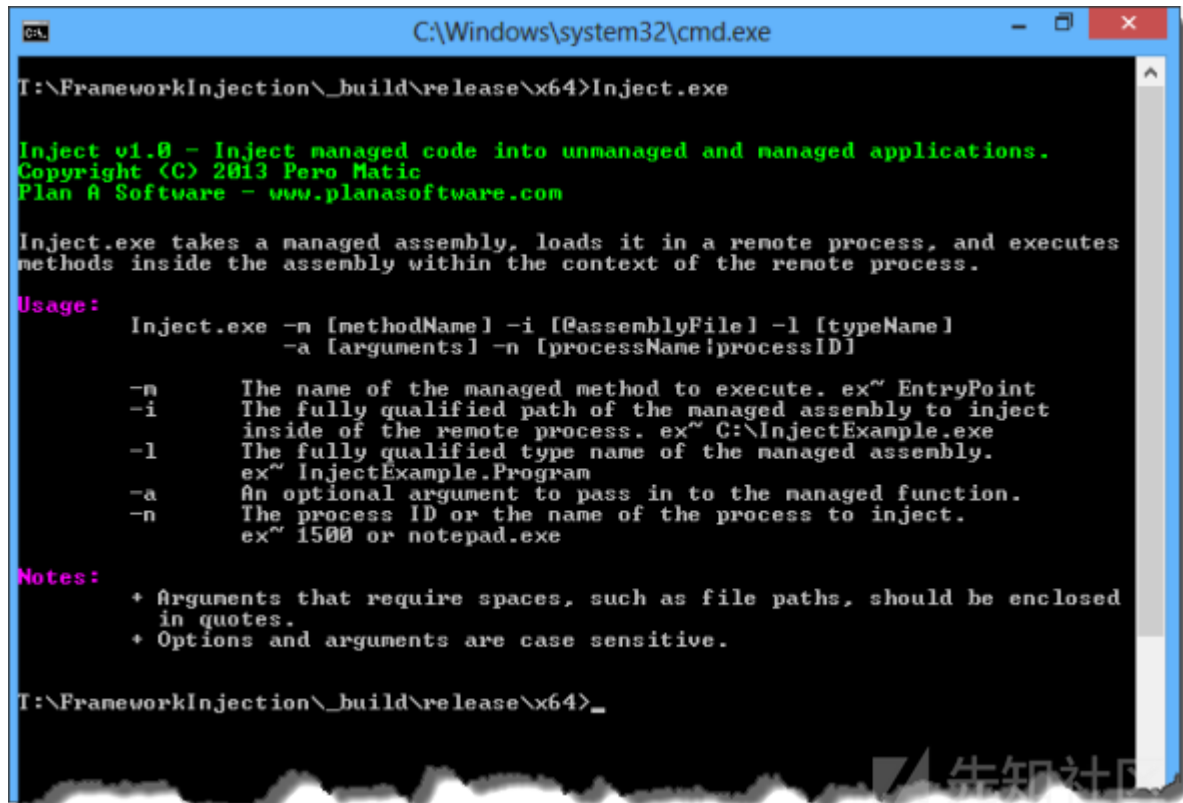
[websni****@outlo](#) / 2018-10-27 08:05:00 / 浏览数 2726 [技术文章](#) [技术文章 顶\(2\) 踩\(0\)](#)

翻译:CoolCat

原文地址：<https://www.codeproject.com/Articles/607352/Injecting-Net-Assemblies-Into-Unmanaged-Processes>

本文将详细分析如何注入运行中的.NET 程序，以及如何将任意.NET程序注入到非托管和托管的进程中；并在这些进程中执行托管代码。

[源代码下载](#)



```
C:\Windows\system32\cmd.exe
I:\FrameworkInjection\_build\release\x64>Inject.exe

Inject v1.0 - Inject managed code into unmanaged and managed applications.
Copyright (C) 2013 Pero Matic
Plan A Software - www.planasoftware.com

Inject.exe takes a managed assembly, loads it in a remote process, and executes
methods inside the assembly within the context of the remote process.

Usage:
    Inject.exe -n [methodName] -i [AssemblyFile] -l [typeName]
               -a [arguments] -n [processName|processID]

    -n      The name of the managed method to execute. ex~ EntryPoint
    -i      The fully qualified path of the managed assembly to inject
            inside of the remote process. ex~ C:\InjectExample.exe
    -l      The fully qualified type name of the managed assembly.
            ex~ InjectExample.Program
    -a      An optional argument to pass in to the managed function.
    -n      The process ID or the name of the process to inject.
            ex~ 1500 or notepad.exe

Notes:
    + Arguments that require spaces, such as file paths, should be enclosed
      in quotes.
    + Options and arguments are case sensitive.

I:\FrameworkInjection\_build\release\x64>_
```

0x1 简介

.NET

是一种易上手且可靠的编程语言。然而它并不适合于每个场景。本文将重点介绍了其中一种特殊的情况，即DLL注入。.NET的无法在其运行时在从未加载的远程进程中进行DLL中的方法呢？架构需要考虑么？64位进程是否不同于32位进程的？本文将展示如何使用文档化的api执行所有这些任务。

目标

- 不考虑架构在任意进程中启动.NET CLR（公共语言运行时）。
- 在任意进程中加载自定义.NET程序。
- 在任意进程中执行托管代码。

0x2 过程

篇幅问题、文章将分为五小段做精简介绍：

1. 加载CLR（初级）-介绍如何在非托管进程内部启动.NET Framework。
2. 加载 CLR（高级）-如何加载自定义.NET 程序集，并从非托管代码中调用受管理的方法。
3. DLL注入（基础）-介绍如何在远程过程中执行非托管代码。
4. DLL 注入（高级）-介绍如何在远程进程中执行任意导出的功能。
5. 综合利用。

1. 加载 CLR（初级）

目标:写一个可以同时加载运行中的.NET程序和其他任意程序集的非托管程序的程序

示例中将演示如何利用C++程序将运行中.NET加载到自身中：

```

#include <metahost.h>

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

#import "mscorlib.tlb" raw_interfaces_only \
    high_property_prefixes("_get", "_put", "_putref") \
    rename("ReportEvent", "InteropServices_ReportEvent")

int wmain(int argc, wchar_t* argv[])
{
    char c;
    wprintf(L"Press enter to load the .net runtime...");
    while (getchar() != '\n');

    HRESULT hr;
    ICLRMetaHost *pMetaHost = NULL;
    ICLRRuntimeInfo *pRuntimeInfo = NULL;
    ICLRRuntimeHost *pClrRuntimeHost = NULL;

    // build runtime
    hr = CLRCreateInstance(CLSID_CLRMetaHost, IID_PPV_ARGS(&pMetaHost));
    hr = pMetaHost->GetRuntime(L"v4.0.30319", IID_PPV_ARGS(&pRuntimeInfo));
    hr = pRuntimeInfo->GetInterface(CLSID_CLRRuntimeHost,
        IID_PPV_ARGS(&pClrRuntimeHost));

    // start runtime
    hr = pClrRuntimeHost->Start();

    wprintf(L".Net runtime is loaded. Press any key to exit...");
    while (getchar() != '\n');

    return 0;
}

```

上述代码需要关注：

CLRCreateInstance
指定CLSID_CLRMetaHost,获取指向一个实例的指针ICLRMetaHost

ICLRMetaHost::GetRuntime
获取类型的指针ICLRRunTimeInfo指向特定.NET 运行时

ICLRRunTimeInfo::GetInterface
将 CLR 加载到当前进程中，然后获取一个ICLRRuntimeHost指针

ICLRRuntimeHost::Start
显式启动 CLR，在首次加载托管代码时隐式调用 CLR

PS:需要注意，ICLRMetaHost::GetRuntime以下版本中中有效

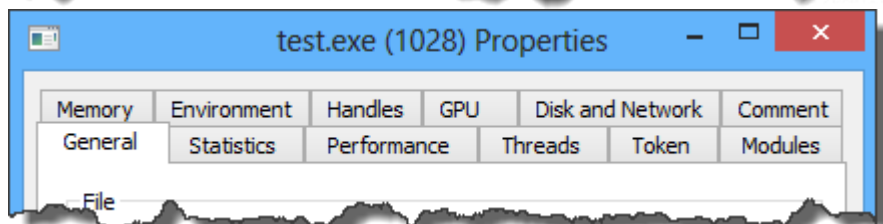
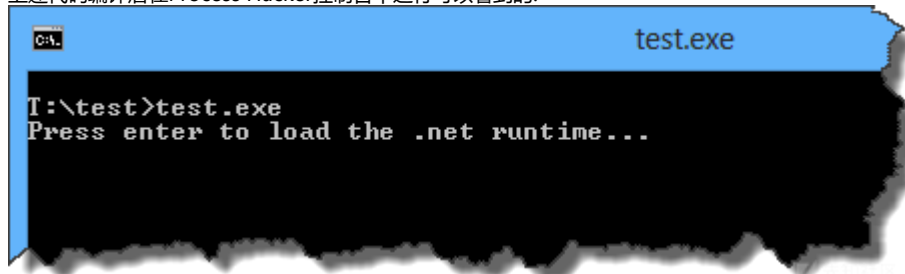
V1.0.3705
V1.1.4322
V2.0.50727
V4.0.30319

在最新的版本中运行会出现null

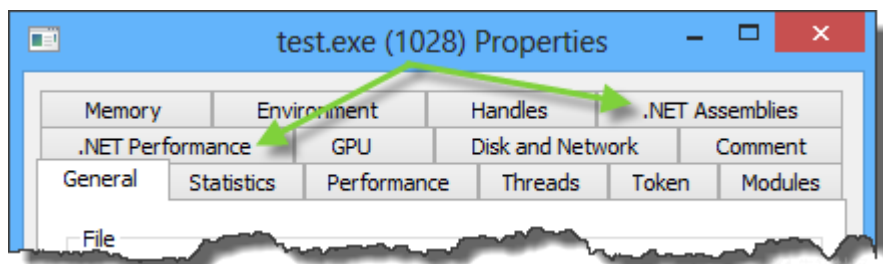
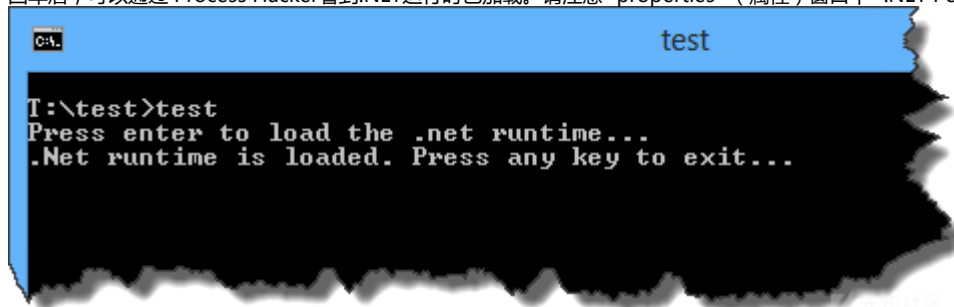
在所需系统上安装运行时 版本号可以在下列两个目录其中之一找到

%WinDir%\Microsoft.NET\Framework
%WinDir%\Microsoft.NET\Framework64

上述代码编译后在Process Hacker控制台中运行可以看到的:



回车后, 可以通过 Process Hacker看到.NET运行时已加载。请注意“properties”(属性)窗口中“.NET Performance”(.NET偏好) 选项



示例代码未打包。建议读者自行动手编译并运行。

2. 加载 CLR (高级)

完成初级的加载 CLR后, 下一步是将任意的.NET程序集加载到其他进程中, 并调用.NET中的方法。

继续修改上面的示例, 把CLR加载到进程中。可以通过获得一个指向CLR接口的指针来实现; 该指针存储在变量pClrRuntimeHost中。使其调用ICLRRuntimeHost::Star

此时CLR已经初始化, pClrRuntimeHost可以调用ICLRRuntimeHost::ExecuteInDefaultAppDomain来加载和调用任意.NRY程序中的方法。该函数具有以下签名:

```
HRESULT ExecuteInDefaultAppDomain (
    [in] LPCWSTR pwzAssemblyPath,
    [in] LPCWSTR pwzTypeName,
    [in] LPCWSTR pwzMethodName,
    [in] LPCWSTR pwzArgument,
    [out] DWORD *pReturnValue
);
```

参数说明:

- pwzAssemblyPath : .NET程序的完整路径; 这里可以是exe或dll文件
- pwzTypeName : 要调用的方法的完整类名
- pwzMethodName : 要调用的方法的名称
- pwzArgument : 可选的参数传递到方法中
- pReturnValue : 方法的返回值

并不是.NET程序中的每个方法都可以通过ICLRRuntimeHost::ExecuteInDefaultAppDomain来调用。可用的方法必须具有以下签名:

```
static int pwzMethodName (String pwzArgument);
```

补充说明: 访问修饰符(如public、protected、private和internal)不会影响方法的可见性; 因此, 被排除在签名之外。

下面的.NET程序将被用于接下来的所有示例，用于注入到托管进程.NET程序:

```
using System;
using System.Windows.Forms;

namespace InjectExample
{
    public class Program
    {
        static int EntryPoint(String pwzArgument)
        {
            System.Media.SystemSounds.Beep.Play();

            MessageBox.Show(
                "I am a managed app.\n\n" +
                "I am running inside: [" +
                System.Diagnostics.Process.GetCurrentProcess().ProcessName +
                "]\n\n" + (String.IsNullOrEmpty(pwzArgument) ?
                "I was not given an argument" :
                "I was given this argument: [" + pwzArgument + "]"));

            return 0;
        }

        static void Main(string[] args)
        {
            EntryPoint("hello world");
        }
    }
}
```

上面的示例代码的编法可以选择调用ICLRRuntimeHost::ExecuteInDefaultAppDomain，也可以独立运行;两种方法运行的结果都差不多。最终目标是当注入到非托管

在初级部分示例代码的基础上，下面的c++程序将加载上面的.NET程序并执行EntryPoint方法:

```
#include <metahost.h>

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

#import "mscorlib.tlb" raw_interfaces_only \
    high_property_prefixes("_get", "_put", "_putref") \
    rename("ReportEvent", "InteropServices_ReportEvent")

int wmain(int argc, wchar_t* argv[])
{
    HRESULT hr;
    ICLRMetaHost *pMetaHost = NULL;
    ICLRRuntimeInfo *pRuntimeInfo = NULL;
    ICLRRuntimeHost *pClrRuntimeHost = NULL;

    // build runtime
    hr = CLRCREATEINSTANCE(CLSID_CLRMetaHost, IID_PPV_ARGS(&pMetaHost));
    hr = pMetaHost->GetRuntime(L"v4.0.30319", IID_PPV_ARGS(&pRuntimeInfo));
    hr = pRuntimeInfo->GetInterface(CLSID_CLRRuntimeHost,
        IID_PPV_ARGS(&pClrRuntimeHost));

    // start runtime
    hr = pClrRuntimeHost->Start();

    // execute managed assembly
    DWORD pReturnValue;
    hr = pClrRuntimeHost->ExecuteInDefaultAppDomain(
        L"T:\\FrameworkInjection\\_build\\debug\\anycpu\\InjectExample.exe",
        L"InjectExample.Program",
        L"EntryPoint",
        L"hello .net runtime",
        &pReturnValue);

    // free resources
    pMetaHost->Release();
}
```

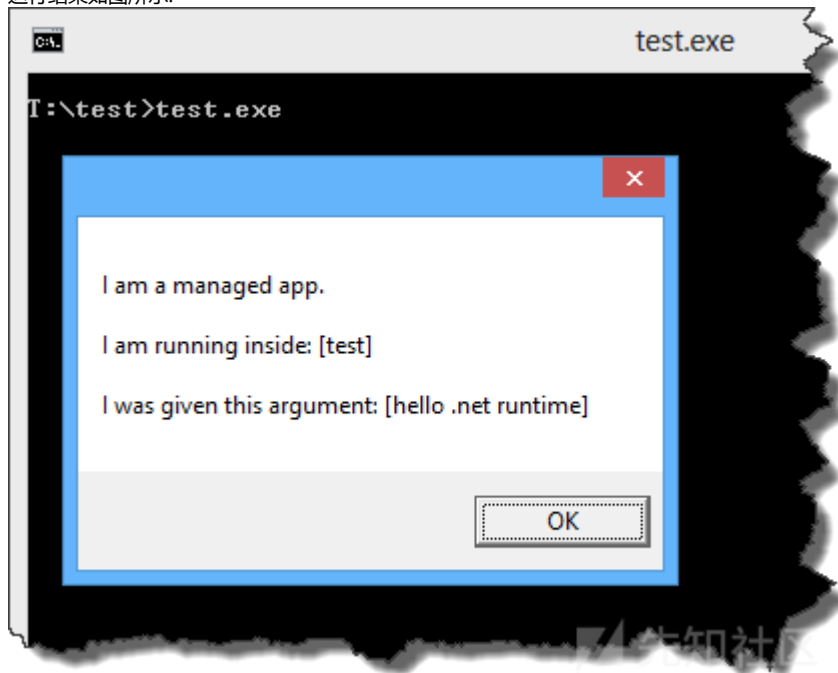
```

pRuntimeInfo->Release();
pClrRuntimeHost->Release();

return 0;
}

```

运行结果如图所示:



到目前为止，这部分两个目标都已实现。现在我们将继续尝试如何从非托管代码中加载CLR并执行任意方法。但是如何在任意过程中实现呢？

3. DLL注入（初级）

DLL注入是一种在远程进程中加载DLL来执行远程进程内部代码的技术。很多DLL注入策略集中于DllMain内部的代码执行。缺陷是从DllMain中启动CLR会导致Windows加载

- [loaderLock MDA.aspx](#))
- [Initialization of Mixed Assemblies](#)
- [Preventing Hangs in Windows Applications.aspx](#))

难以避免的是当Windows加载程序初始化另一个模块时，CLR无法启动。每个锁都是特定于进程的，由Windows托管且一个锁已经被获取时，任何试图在加载器上获取多个

上述关于问题似乎很麻烦,那么我们将问题分解开来，比如从将一个最起码的DLL注入到远程进程中作为开始。示例代码:

```

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

```

上面的代码实现了一个简单的DLL。要想把这个DLL注入到远程进程中，需要以下Windows api:

- OpenProcess：获取进程的句柄
- GetModuleHandle：获取给定模块的句柄
- LoadLibrary：在调用进程的地址空间中加载库
- GetProcAddress：从库中获取导出函数的虚拟地址
- VirtualAllocEx：在给定进程中分配空间
- WriteProcessMemory：在给定地址将字节写入进程
- CreateRemoteThread：在远程进程中派生一个线程

```

DWORD_PTR Inject(const HANDLE hProcess, const LPVOID function,
    const wstring& argument)
{
    // allocate some memory in remote process
    LPVOID baseAddress = VirtualAllocEx(hProcess, NULL, GetStringAllocSize(argument),
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    // write argument into remote process
    BOOL isSucceeded = WriteProcessMemory(hProcess, baseAddress, argument.c_str(),
        GetStringAllocSize(argument), NULL);

    // make the remote process invoke the function
    HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
        (LPTHREAD_START_ROUTINE)function, baseAddress, NULL, 0);

    // wait for thread to exit
    WaitForSingleObject(hThread, INFINITE);

    // free memory in remote process
    VirtualFreeEx(hProcess, baseAddress, 0, MEM_RELEASE);

    // get the thread exit code
    DWORD exitCode = 0;
    GetExitCodeThread(hThread, &exitCode);

    // close thread handle
    CloseHandle(hThread);

    // return the exit code
    return exitCode;
}

```

本节的目的是在远程进程中加载库。下一个问题是如何利用上述函数在远程进程中注入DLL

?假设kernel32.dll映射在每个进程的地址空间内。LoadLibrary是kernel32的导出函数。它有一个与LPTHREAD_START_ROUTINE匹配的函数签名，所以它可以作为开始例程。

```

// get handle to remote process with PID 2432
HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, 2432);

// get address of LoadLibrary
FARPROC fnLoadLibrary = GetProcAddress(GetModuleHandle(L"Kernel32"), "LoadLibraryW");

// inject test.dll into the remote process
Inject(hProcess, fnLoadLibrary, L"T:\\test\\test.dll");

```

继续修改前面的示例代码，一旦调用CreateRemoteThread，就会调用WaitForSingleObject来等待线程退出。接下来调用GetExitCodeThread以获得结果。巧合的是，当线程退出时，hThread已经变为INVALID_HANDLE_VALUE。

在此为止，这个部分的三个问题已经解决:

1. 使用非托管代码加载CLR
2. 从非托管代码执行任意.NET程序
3. DLL注入
4. DLL注入（高级）

前面已经说到了如何在远程进程中加载DLL了;关于如何在远程进程中启动CLR的问题可以继续探讨。

当LoadLibrary返回时，加载器上的锁将被释放。利用远程进程地址空间中的DLL，可以通过对CreateRemoteThread的后续调用调用导出函数;假设函数签名匹配LPTHREAD_START_ROUTINE。

再次分解问题。下面的函数可以返回x86和x64系统上给定进程中给定模块的句柄(碰巧是基本地址):

```

DWORD_PTR GetRemoteModuleHandle(const int processId, const wchar_t* moduleName)
{
    MODULEENTRY32 me32;
    HANDLE hSnapshot = INVALID_HANDLE_VALUE;

    // get snapshot of all modules in the remote process
    me32.dwSize = sizeof(MODULEENTRY32);
    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, processId);

    // can we start looking?

```

```

if (!Module32First(hSnapshot, &me32))
{
    CloseHandle(hSnapshot);
    return 0;
}

// enumerate all modules till we find the one we are looking for
// or until every one of them is checked
while (wcscmp(me32.szModule, moduleName) != 0 && Module32Next(hSnapshot, &me32));

// close the handle
CloseHandle(hSnapshot);

// check if module handle was found and return it
if (wcscmp(me32.szModule, moduleName) == 0)
    return (DWORD_PTR)me32.modBaseAddr;

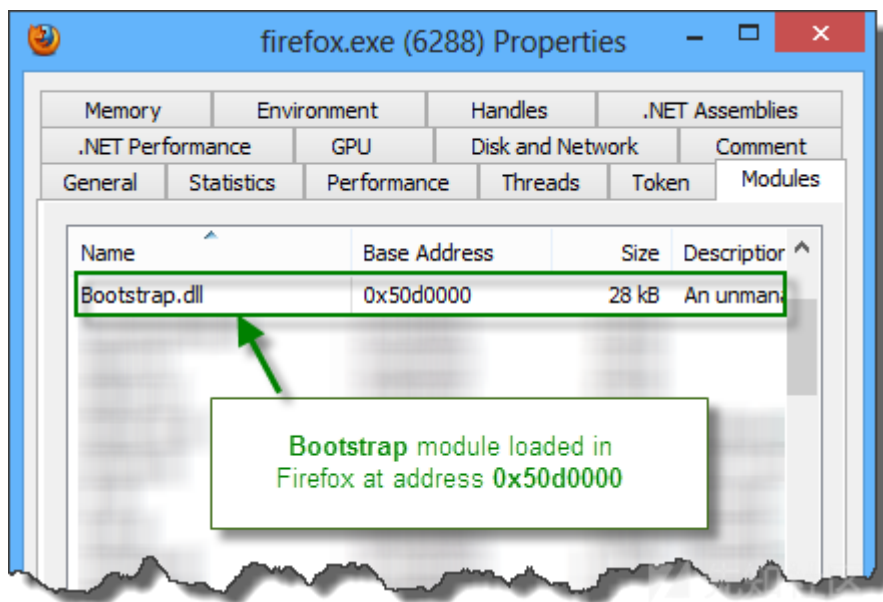
return 0;
}

```

搞清楚远程进程中DLL的基本地址是朝着正确方向迈出的一步。接下来是写出一个可以获取任意导出函数地址的策略的时候了。回顾一下，我们知道如何调用loadlibrary并在DLL项目中可以找到以下导出函数:

```
__declspec(dllexport) HRESULT ImplantDotNetAssembly(_In_ LPCTSTR lpCommand)
```

在远程调用此函数之前，Bootstrap.dll模块必须首先在远程进程中加载。使用Process Hacker注入Firefox时内存中的dll模块。如图。



继续我们的思路，下面是加载引导程序的示例程序Bootstrap.dll模块到本地(调用到进程内):

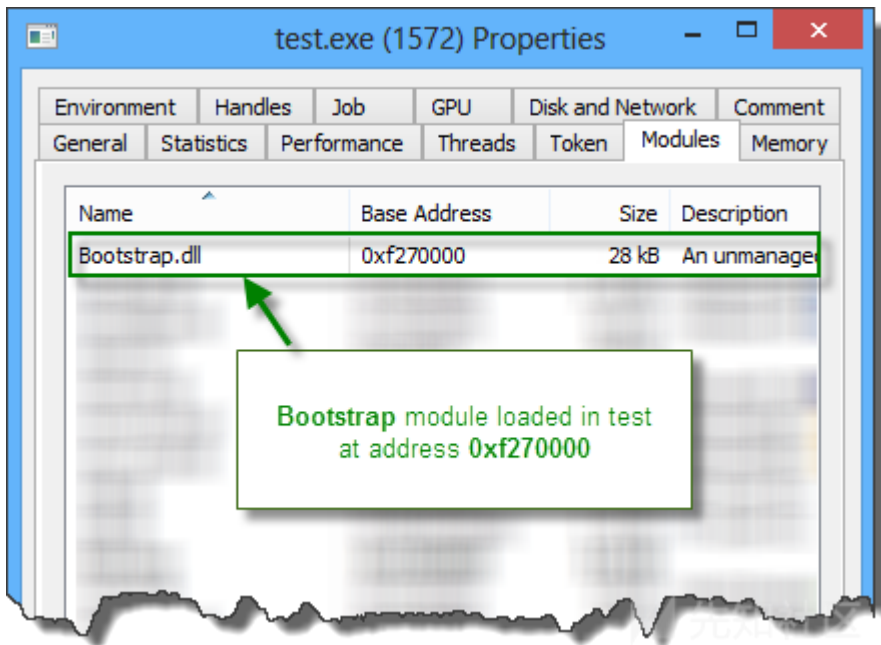
```

#include <windows.h>

int wmain(int argc, wchar_t* argv[])
{
    HMODULE hLoaded = LoadLibrary(
        L"T:\\FrameworkInjection\\_build\\release\\x86\\Bootstrap.dll");
    system("pause");
    return 0;
}

```

当上面的程序运行时，下面是Windows加载Bootstrap.dll模块的截图:



下一步，我们在wmain函数中调用GetProcAddress来获取ImplantDotNetAssembly函数的地址:

```
#include <windows.h>

int wmain(int argc, wchar_t* argv[])
{
    HMODULE hLoaded = LoadLibrary(
        L"T:\\FrameworkInjection\\_build\\debug\\x86\\Bootstrap.dll");

    // get the address of ImplantDotNetAssembly
    void* lpInject = GetProcAddress(hLoaded, "ImplantDotNetAssembly");

    system("pause");
    return 0;
}
```

模块中的函数的地址总是比模块的基本地址要高。这就是初等数学发挥作用的地方。下面是一个表格来帮助说明:

	Firefox.exe	Bootstrap.dll @ 0x50d0000	ImplantDotNetAssembly @ ?
test.exe		Bootstrap.dll @ 0xf270000	ImplantDotNetAssembly @ 0xf271490 (lpInject)

test.exe显示了Bootstrap.dll
在地址0xf270000处加载，ImplantDotNetAssembly可以在内存地址0xf271490中找到。从引导程序的地址中减去植入式网络装配的地址。dll会给出函数相对于模块的基本地址 - 0xf270000) = 0x1490字节进入模块。然后可以将此偏移量添加到引导程序的基本地址。dll模块在远程进程中可靠地给出相对于远程进程的植入式网络程序集的地址。在Firefox中计算ImplantDotNetAssembly的地址: 0x50d1490 + 0x1490 = 0x50d1490。下面的函数计算给定模块中给定函数的偏移:

```
DWORD_PTR GetFunctionOffset(const wstring& library, const char* functionName)
{
    // load library into this process
    HMODULE hLoaded = LoadLibrary(library.c_str());

    // get address of function to invoke
    void* lpInject = GetProcAddress(hLoaded, functionName);

    // compute the distance between the base address and the function to invoke
    DWORD_PTR offset = (DWORD_PTR)lpInject - (DWORD_PTR)hLoaded;

    // unload library from this process
    FreeLibrary(hLoaded);

    // return the offset to the function
    return offset;
}
```

值得注意的是ImplantDotNetAssembly故意匹配LPTHREAD_START_ROUTINE的签名;所有传递给CreateRemoteThread的方法都应该这样。具有在远程DLL中执行任意函数的能力。

5. 综合利用

使用一个非托管DLL(Bootstrap.dll)加载CLR的主要原因是如果CLR在远程进程中运行,唯一的方式开始从非托管代码开始(除非使用Python脚本语言等,否则都有自己的依赖关

另外,对于Inject应用程序来说,最好能够灵活地接受命令行上的输入;避免重新编译。

Inject应用程序的相关命令参数:

- m 要执行的托管方法的名称。如EntryPoint
- i 被注入到远程进程内部托管程序的完整路径。如 C:\InjectExample.exe
- l 托管程序集的完整类名。如InjectExample.Program
- a 一个可选的参数传递给托管函数。
- n 进程ID或要注入的进程名称。如1500或notepad.exe

注入的wmain方法如下:

```
int wmain(int argc, wchar_t* argv[])
{
    // parse args (-m -i -l -a -n)
    if (!ParseArgs(argc, argv))
    {
        PrintUsage();
        return -1;
    }

    // enable debug privileges
    EnablePrivilege(SE_DEBUG_NAME, TRUE);

    // get handle to remote process
    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, g_processId);

    // inject bootstrap.dll into the remote process
    FARPROC fnLoadLibrary = GetProcAddress(GetModuleHandle(L"Kernel32"),
        "LoadLibraryW");
    Inject(hProcess, fnLoadLibrary, GetBootstrapPath());

    // add the function offset to the base of the module in the remote process
    DWORD_PTR hBootstrap = GetRemoteModuleHandle(g_processId, BOOTSTRAP_DLL);
    DWORD_PTR offset = GetFunctionOffset(GetBootstrapPath(), "ImplantDotNetAssembly");
    DWORD_PTR fnImplant = hBootstrap + offset;

    // build argument; use DELIM as tokenizer
    wstring argument = g_moduleName + DELIM + g_typeName + DELIM +
        g_methodName + DELIM + g_Argument;

    // inject the managed assembly into the remote process
    Inject(hProcess, (LPVOID)fnImplant, argument);

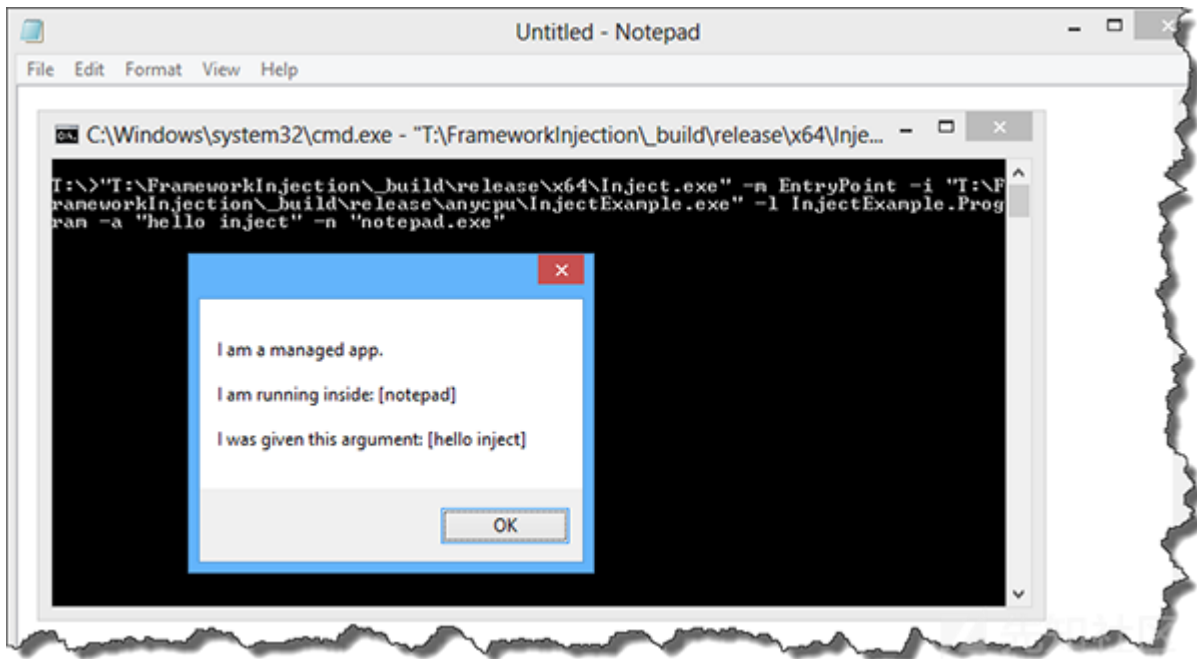
    // unload bootstrap.dll out of the remote process
    FARPROC fnFreeLibrary = GetProcAddress(GetModuleHandle(L"Kernel32"),
        "FreeLibrary");
    CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)fnFreeLibrary,
        (LPVOID)hBootstrap, NULL, 0);

    // close process handle
    CloseHandle(hProcess);

    return 0;
}
```

下面是Inject.exe的截图。调用exe应用程序注入.NET程序InjectExample.exe注入到notepad.exe 以及所用到的命令:

```
C:\Inject.exe -m EntryPoint -i «C:\InjectExample.exe» -l InjectExample.Program -a «hello inject» -n «notepad.exe»
```



值得一提的是，在注入一个基于x86、x64或其他CPU构建的DLL时，应该区分它们之间的区别。正常情况下，x86架构的Inject.exe和Bootstrap.dll用于注入x86进程，x64架构用于注入x64进程。调用方的责任是确保正确地使用二进制文件。其他CPU是.NET中可用的平台。为任何cpu设定目标，告诉CLR为适当的体系结构。

代码环境：

继续有趣的事情!使用默认设置运行代码有几个先决条件。

编译环境：

Visual Studio 2012 Express+
Visual Studio 2012 Express Update 1 +

运行环境：

.Net Framework 4.0 +
Visual C++ Redistributable for Visual Studio 2012 Update 1 +
Windows XP SP3 +

为了简化编译，给出压缩包下载后有一个叫“build.bat”，点击后将解决前期麻烦的安装任务。它将编译调试和发布版本以及相应的x86、x64和其他CPU版本。每个项目也可用Visual Studio编译。build.bat将把二进制文件放在一个名为_build的文件夹中。

代码里的注释也蛮齐全的。此外，请使用C++ 11.0和.NET4.0。因为这两个版本从XP SP3 x86 到 Windows 8 x64的所有Windows操作系统上都可以顺利运行。另外再提一下微软在VS 2012 U1中增加了对c++ 11.0运行时的XP SP3支持。

0x3彩蛋

正如文章中提到的，.NET是一门强大的语言。比如可以利用.NET中的Reflection API（反射API）来获取关于程序的类型信息。这样做的意义在于.NET可以用来扫描程序集并返回可用于注入的有效方法!下载的源代码包含一个.NET项目，名为InjectGUI。

还有一个助手类叫MethodItem，它的定义如下:

```
public class MethodItem
{
    public string TypeName { get; set; }
    public string Name { get; set; }
    public string ParameterName { get; set; }
}
```

以下来自ExtractInjectableMethods方法的代码片段将获得一个Collection of type List<MethodItem>，它匹配所需的方法签名:

```
// find all methods that match:
// static int pwzMethodName (String pwzArgument)

private void ExtractInjectableMethods()
{
    // ...

    // open assembly
```

```

Assembly asm = Assembly.LoadFile(ManagedFilename);

// get valid methods
InjectableMethods =
    (from c in asm.GetTypes()
     from m in c.GetMethods(BindingFlags.Static |
                             BindingFlags.Public | BindingFlags.NonPublic)
     where m.ReturnType == typeof(int) &&
           m.GetParameters().Length == 1 &&
           m.GetParameters().First().ParameterType == typeof(string)
     select new MethodItem
     {
         Name = m.Name,
         ParameterName = m.GetParameters().First().Name,
         TypeName = m.ReflectedType.FullName
     }).ToList();

// ...
}

```

既然已经提取了有效的(可注入的)方法，UI还应该知道要注入的进程是32位还是64位。要做到这一点，需要一些Windows API的协助:

```

[DllImport("kernel32.dll", SetLastError = true, CallingConvention =
    CallingConvention.Winapi)]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern bool IsWow64Process([In] IntPtr process,
    [Out] out bool wow64Process);

```

IsWow64Process只在64位操作系统上定义，如果应用程序是32位，则返回true。在.NET4.0中，引入了以下特性:Environment.Is64BitOperatingSystem。这可以用

```

private static bool IsWow64Process(int id)
{
    if (!Environment.Is64BitOperatingSystem)
        return true;

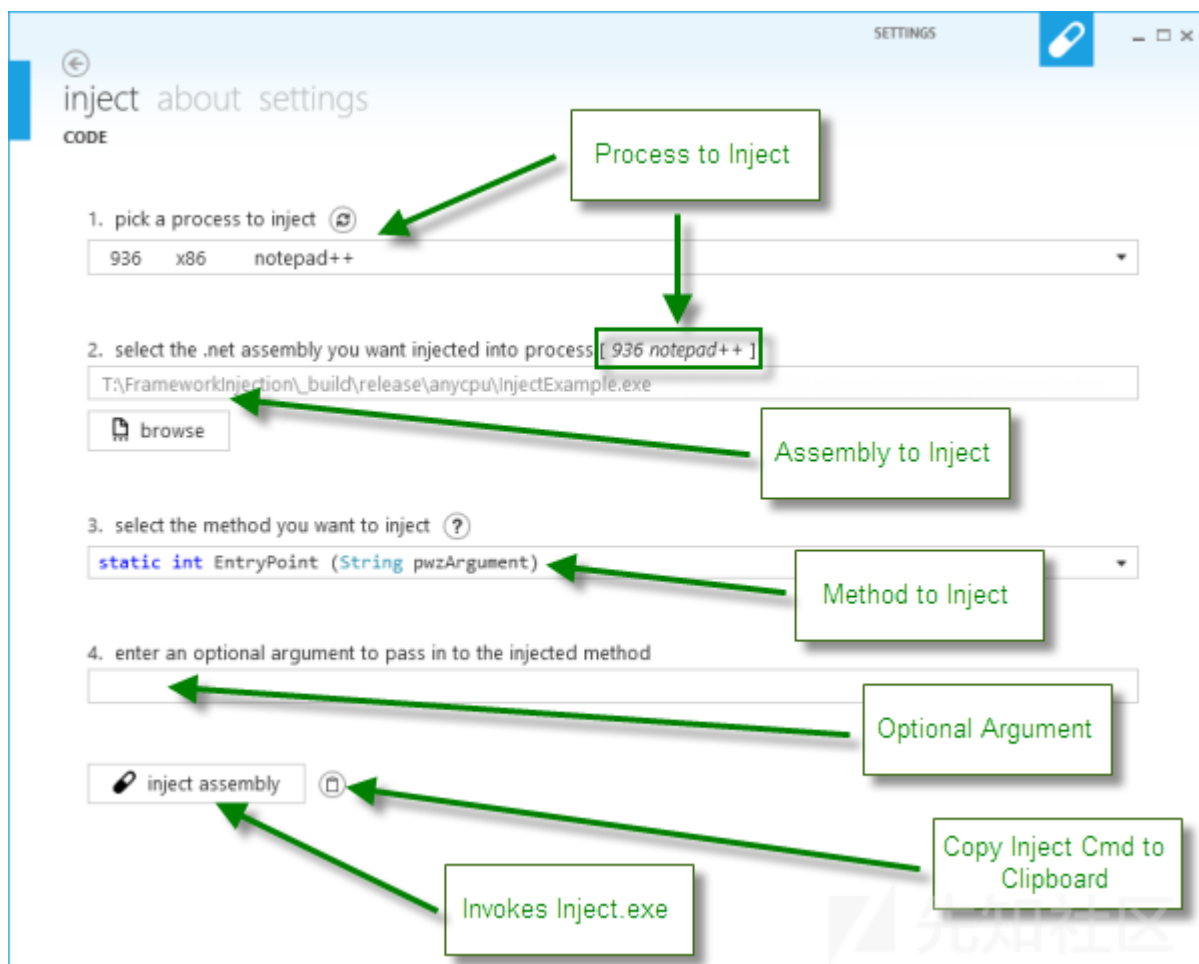
    IntPtr processHandle;
    bool retVal;

    try
    {
        processHandle = Process.GetProcessById(id).Handle;
    }
    catch
    {
        return false; // access is denied to the process
    }

    return IsWow64Process(processHandle, out retVal) && retVal;
}

```

InjectGUI项目中的逻辑相当简单。了解WPF和依赖属性对于理解UI是必要的，然而，所有与注入相关的逻辑都在InjectWrapper类中。UI是使用WPF的现代UI构建的，图标



相关文献

- [托管程序与非托管程序的区别](#)
- [.NET CLR是什么](#)

点击收藏 | 0 关注 | 1

[上一篇：看！这里有三种非Web型的XSS注入漏洞](#) [下一篇：WebExec之技术纲要\(权限提升...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)