8种HOOK技术

转载

昵称正在加载 于 2020-09-08 13:53:49 发布 819 收藏 1

1.IAT_HOOK

IAT是程序中存储导入函数地址的数据结构，如果HOOK了导入函数地址。就可以在函数调用的时候，将函数流程HOOK到我们指定的流程。但是我个人觉得这种方式最好要结合DLL注入的方式，如果单纯的使用HOOK，那么就需要将需要执行的操作的shellcode写入目标进程，如果操作复杂，可能需要的shellcode量特别大，**所以我们需要借助DLL注入**，这样就将我们需要执行的代码写入进程内部，在HOOK的Detour函数只需要实现LoadLibrary的操作。

IATHOOK的基本原理就是通过修改程序IAT数据结构，将原始调用API函数地址Target函数地址修改为Detour函数地址。所以IAT_HOOK需要实现以下几个步骤:

1)、构造Detour函数
2)、获取Target函数地址
3)、通过PE获取Target函数所在的IAT的地址
4)、保存原始的IAT地址和IAT地址所存储的内容
5)、修改IAT地址中的数据
6)、如果需要调用原来API函数，可以直接使用保存的API地址，可以就保证了HOOK的有效性

2.EAT_HOOK

使用EAT_HOOK需要注意一下两点：第一：EAT存储的是函数地址的偏移，**所以在HOOK EAT的时候需要加上基地址**，在写入EAT的时候，Detour地址需要减去BaseAddress。第二，EAT不对隐式链接起作用，**只对显示链接起作用**，也就是说对于那种**GetProcAddress的那种调用**起作用。

EAT_HOOK的原理和IAT_HOOK类似，都是通过修改函数地址数据从而HOOK。EAT_HOOK，也需要进行以下步骤：

1)、获取Target函数在HookModule上的RVA
2)、获取导出函数数组首地址
3)、遍历查找Target函数RVA
4)、切记在修改函数地址之前，需要保存EAT地址和原函数地址
5)、将Detour函数地址写入EAT

下面用代码来实现x64下的IAT HOOK和EAT HOOK

```
#include <stdio.h>#include <Windows.h>#include <Psapi.h>//#include "pe.h" #pragma comment(lib,"user32.lib")
```

3.VirtualFunctionHook

C++虚函数存在的意义是为了方便使用多态性。在实现虚函数Hook的时候需要注意如下问题：1.在构建DetourFun函数的时候，一定要构造DetourClass，因为在调用虚函数的时候使用了Thiscall的函数调用约定，如果直接调用detourfun函数应该使用的标准调用约定，两者不统一，会出错。2.当使用Trampolinefun回调的时候，需要重新实例化一个TrampolineClass。

实现代码可以参考链接：https://blog.csdn.net/ab7936573/article/details/65967178

4.inline hook

下面以x64内核为例，hook PsLookupProcessByProcessId函数，使得不能打开计算器。代码如下：

```
#include <ntddk.h>#include "LDE64x64.h" KIRQL WPOFF(){ KIRQL irql = KeRaiseIrqlToDpcLevel(); UINT64 cr0 = _
```

头文件 LDE64x64.h 百度搜一下，有很多，我就不帖了

5.VEH_HOOK

VEH技术的主要原理是利用异常处理改变程序指令流程。通过主动抛出异常，使程序触发异常，控制权交给异常处理例程的这一系列操作来实现HOOK。

这里简单提一下VEH，向量异常处理，基于VEH链表而不是栈，这样的话其作用范围是进程全局，而不是线程。且优先级也高于SEH，这也是VEH_HOOK的优势所在。

VEH_HOOK通过异常机制实现HOOK，必不可少需要构造异常处理函数，同时也需要人为的构造异常，同时为了实现永久化机制，保证执行原操作需要实现TrampolineFun函数。所以总结VEH_HOOK步骤如下：

1)、构造TrampolineFun
2)、构造异常处理函数，即Detour函数
3)、人为构造异常。

用软件断点实现如下

```
#include <Windows.h> LPVOID Checkaddr = NULL;BYTE oldbyte = 0; DWORD WINAPI ExceptionHandle(EXCEPTION_POINT
```

用硬件断点实现如下

```
#include <windows.h>#include <tlhelp32.h>DWORD ThreadID;HANDLE hThread;PVOID ExceptionHandle = NULL;PVOID T
```

6. SSDT_HOOK

SSDT中文全称为**系统服务描述符表**，其作用是作为R3和R0层的通道，将用户态API函数和内核函数联系起来。用简单的API函数举例子，我们调用了CreateFile,其会调用ZwCreateFile,然后调用NtCreateFile，经过参数和模式的检查，然后调用系统服务分发函数KiSystemService进入内核。在R0中通过传入的系统服务号(函数索引)得到系统服务的地址，然后调用该系统服务即可。

所以，根据上述，我们可以知道SSDT其实是一个存储系统服务的数组。SSDT_HOOK其实就是在内核层的AddressHook。只不过他修改是系统服务描述符表数据。

因为SSDT的索引号和系统服务内核地址是一一对应的，所以不需要向普通的AddressHook一一对比函数地址。所以让我们来屡一下执行SSDT的操作。我们有目的向原因开始。如果我们需要执行SSDT_HOOK的话，首先需要修改为与SSDT中的系统服务地址，但又由于系统服务地址是和服务索引是保持对应关系的，所以我们还需要获取索引号。

根据上面的分析，我们知道首先需要获取服务索引号。但是服务索引号和函数地址对应的，**在X86系统中，相对于导出函数偏移量1的地址往后读四个字节就是SSDT服务索引号。但是对于X64位的系统，却是函数地址偏移为4的地址读取四个字节。**所以需要得到服务索引号，就需要得到导出函数地址。

我们现在总结一下得到服务索引的步骤：

Step1：将Ntdll.dll载入内存
Step2：获取导出函数地址
Step3：计算函数索引

下面以x64内核为例，进行ssdt hook

```
#include <ntddk.h> typedef struct _SYSTEM_SERVICE_TABLE { PVOID    ServiceTableBase; PVOID    ServiceCounte
```

7. IRP_HOOK

IRP全称是IO请求包，发送到设备驱动程序的大多数请求都打包在IRP中。操作系统组件或驱动程序通过调用IoCallDriver将IRP发送给驱动程序。

大概的执行流程是这样的：IO管理器创建一个IRP来代表一个IO操作，并且将该IRP传递给正确的驱动程序，当此IO操作完成时再处理该请求包。相对的，驱动程序(上层的虚拟设备驱动或者底层的真实设备驱动)接收一个IRP，执行该IRP指定的操作，然后将IRP传回给IO管理器，告诉它，该操作已经完成，或者应该传给另一个驱动以进行进一步处理。

IO管理器可以使用一下三个函数创建IRP。**但此时，IRP堆栈还没有被初始化，难以进行拦截。**然后使用你可以调用IoGetNextIrpStackLocation函数获得该IRP第一个堆栈单元的指针。然后初始化这个堆栈单元。当初始化完成之后，就可以调用IoCallDriver函数把IRP发送到设备驱动程序了。**这就可以在中途进行拦截啦。**

- IoBuildAsynchronousFsdRequest 创建异步IRP
- IoBuildSynchronousFsdRequest 创建同步IRP
- IoBuildDeviceIoControlRequest 创建一个同步IRP_MJ_DEVICE_CONTROL或IRP_MJ_INTERNAL_DEVICE_CONTROL请求。

根据上述流程，执行IrpHook可以在三个地址进行，第一：在Irp初始化之后，第二：在发往派遣例程过程中，第三，直接修改需要拦截驱动对象派遣例程函数表。

通过查看 IofCallDriver函数发现，在函数开头存在一个jmp指令。ff2500c85480其中ff25是jmp的机器码，后面的机器码是跳转的绝对地址。可以使用InlineHook直接修改跳转地址即可

```
void HookpIofCallDriver(){    KIRQL oldIrql;    ULONG addr = (ULONG)IofCallDriver;    //保存原始的IofCallDri
```

给一个更详细的链接：https://bbs.pediy.com/thread-60022.htm

8.Object HOOK

```
NTSTATUS Hook(){    NTSTATUS  Status;    HANDLE hFile;    UNICODE_STRING Name;    OBJECT_ATTRIBUTES Attr;
```
