2021/01/25_调试器_第1课_内联HOOK、调试器框架的编写

笔记本:	调试器
创建时间:	2021/1/25 星期一 10:14
作者:	ileemi
标签:	调试器框架的编写, 内联HOOK

API HOOK

> API Hook： 内联HOOK（Inline Hook），修改目标程序的汇编二进制，只能使用汇编代码。

监控扫雷程序中API调用情况或者修改API的参数数据信息（如："程序运行时间"、"关于扫雷"对话框上的文本数据）：

- 使用 Spy++ 定位过程函数。
- 通过过程函数定位所需要的消息（WM_COMMAND），通过 "条件断点"，示例：[esp+8] == WM_COMMAND（消息ID：0111h）。
- 在窗口过程处下条件断点，运行程序，等消息来，进行调试，定位弹出的窗口所使用的API（此程序使用 "ShellAboutW"），"ShellAboutW" 在 "Shell32.dll" 模块中。
- 通过 "ollydbg" 在 "ShellAboutW" 处下断点，并运行调试
- 通过 hook 修改该API中的参数（标题或者显示子在窗口中的内容）。

ShellAboutW参数：

- 参数1：窗口句柄
- 参数2：标题
- 参数3：显示在窗口中的内容
- 参数4：图标句柄

监控API，实现步骤：

- 在API开始位置找到一些代码（jmp xxx 所需要5个字节），将其修改为：跳转到注入程序中注入代码的位置（jmp xxxxxxxx），修改的源代码放到注入程序中进行执行，执行后再返回。
- 在注入程序中修改数据后，在跳转回到目标程序中，继续执行后面的代码。
- 在注入程序中定位目标程序中所使用API（ShellAboutW）的模块基址（通过遍历模块）
- 在注入程序中自己加载对应模块中对应的API，计算出API在模块中的偏移地址
- 通过偏移地址计算目标API在目标程序中加载的模块中的地址

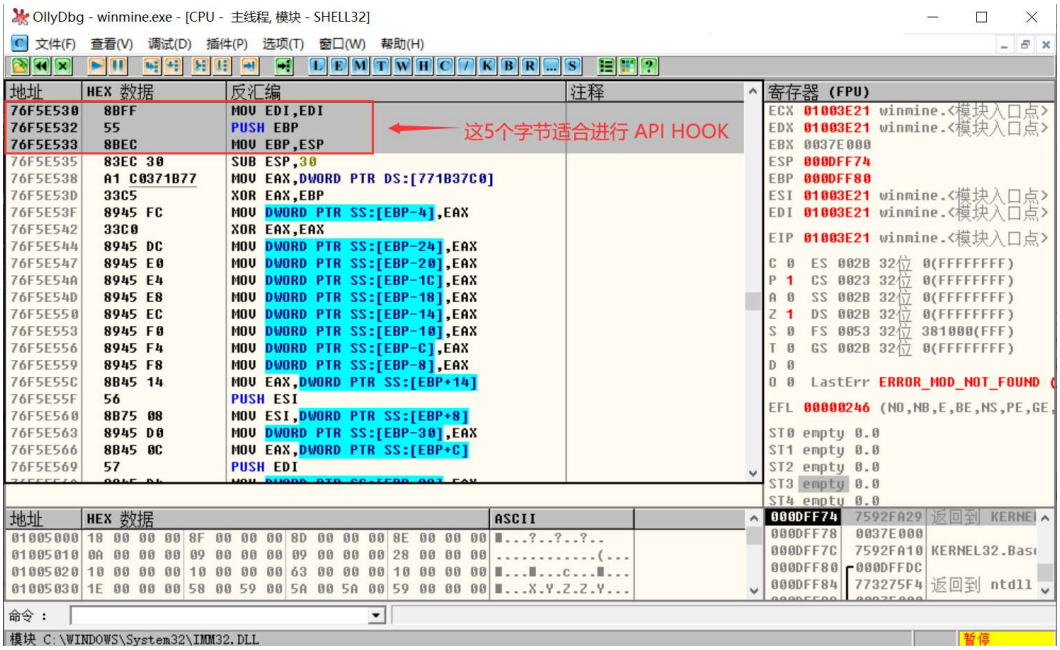
修改远程进程内存API的汇编二进制：

我们申请的地址减去目标进程中API代码的首地址，再减上所要修改汇编代码的字节数（**目标地址是加偏移之后的地址**）。之后再将计算后的值保存到目标API的首地址进行替换（注意修改API内存保护属性）。

```
;修改远程进程内存API的汇编二进制
;操作码
invoke WriteProcessMemory, @hProcess, ebx, addr @OpCode, 1, addr @dwBytes

;计算偏移
mov ecx, @lpBuf
sub ecx, ebx
sub ecx, 5
mov @Offset, ecx
inc ebx
```

> 内敛Hook在内存中不是哪里都能进行修改（在API函数中找对应的字节数，可能对应的功能代码已经走完了），需要在内存中找连续的字节数（构成jmp 指令最低需要5个字节数），不能将汇编指令拆开（断指令）。微软的所有API在函数头都添加了两个的字节的汇编代码：mov edi, edi，这就导致可以在API函数头进行Hook（可以实现 jmp 指令）。



调试器

> 调试器是利用异常来实现的（和SEH没有区别）。
> 调试器的原理：让程序产生异常，调试器将异常进行修复，之后程序的代码继续往下执行。

- 单步执行程序就要使程序的每行代码都出现异常，自己实现起来不叫麻烦。为此操作系统就提供了一套调试器相关的API，方便制造异常。
- WriteProcessMemory 也是一个调试器API，让程序产生异常就需要修改代码。显示（读取）程序的汇编代码也通用需要调试器相关的API：ReadProcessMemory。

> 调试器相关的API

软件产生异常时，接收异常信息的程序优先级：

- 操作系统
- 调试器
- SEH
- 筛选器

使用操作系统提供的API：CreateProcess（创建一个调试进程（DEBUG_PROCESS），此时除了操作系统，其优先级最高，软件产生的异常会优先被当前程序接收）。

创建的程序为当前程序的子进程（调试的进程结束时，被调试的进程也跟着结束）。

调试器相关API

- **ContinueDebugEvent**：允许调试器继续先前报告调试事件的线程。
- **DebugActiveProcess**：允许调试器附加到活动进程并调试它。
- **DebugActiveProcessStop**：停止调试器调试指定的进程。
- **DebugBreak**：该函数导致当前进程中发生断点异常，这允许调用线程通知调试器处理异常。
- **DebugBreakProcess**：在指定的进程中触发断点异常，这允许调用线程通知调试器处理异常。
- **DebugSetProcessKillOnExit**：设置调试线程退出时要执行的操作。
- **FatalExit**：该函数将执行控制传递给调试器。此后调试器的行为特定于所使用的调试器类型。
- **FlushInstructionCache**：为指定的进程刷新指令缓存。
- **GetThreadContext**：获取指定线程的上下文。
- **GetThreadSelectorEntry**：为指定的选择器和线程检索描述符表项。
- **IsDebuggerPresent**：该函数确定调用进程是否在调试器的上下文中运行。
- **OutputDebugString**：该函数将一个字符串发送给调试器以进行显示。
- **ReadProcessMemory**：该函数从指定进程的内存区域中读取数据。要读取的整个区域必须是可访问的，否则操作将失败。
- **SetThreadContext**：为指定的线程设置上下文。
- **WaitForDebugEvent**：等待正在调试的进程中发生调试事件。
- **WriteProcessMemory**：。将数据写入指定进程的内存区域。要写入的整个区域必须是可访问的，否则操作将失败。

WaitForDebugEvent

> 函数声明：

```
BOOL WaitForDebugEvent(
  LPDEBUG_EVENT lpDebugEvent,   // debug event information(调试事件信息)
  DWORD dwMilliseconds          // time-out value
);
```

> 参数1是一个指向 "**DEBUG_EVENT**" 结构体的指针，该结构体接收调试事件的信息。"**DEBUG_EVENT**" 结构体如下：

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union {
      EXCEPTION_DEBUG_INFO Exception;
      CREATE_THREAD_DEBUG_INFO CreateThread;
      CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
      EXIT_THREAD_DEBUG_INFO ExitThread;
      EXIT_PROCESS_DEBUG_INFO ExitProcess;
      LOAD_DLL_DEBUG_INFO LoadDll;
      UNLOAD_DLL_DEBUG_INFO UnloadDll;
      OUTPUT_DEBUG_STRING_INFO DebugString;
      RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```