2017
绿盟科技
恶意样本分析手册

绿盟科技安全能力中心(SAC)

[反调试篇]

(第二版)

关于绿盟科技

北京神州绿盟信息安全科技股份有限公司（简称绿盟科技）成立于 2000 年 4 月，总部位于北京。在国内外设有 30 多个分支机构，为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。

基于多年的安全攻防研究，绿盟科技在网络及终端安全、互联网基础安全、合规及安全管理等领域，为客户提供入侵检测 / 防护、抗拒绝服务攻击、远程安全评估以及 Web 安全防护等产品以及专业安全服务。

北京神州绿盟信息安全科技股份有限公司于 2014 年 1 月 29 日起在深圳证券交易所创业板上市交易。

股票简称：绿盟科技　股票代码：300369

目录

目录

调试器原理

调试事件与调试循环

启动被调试程序

当我们要对一个程序进行调试时，首先要做的当然是启动这个程序，这要使用 CreateProcess 这个 Windows API 来完成。

```
CreateProcess(
TEXT("Path"),
NULL,
NULL,
NULL,
FALSE,
DEBUG_ONLY_THIS_PROCESS | CREATE_NEW_CONSOLE,
NULL,
NULL,
&si,
&pi)
```

CreateProcess 的第六个参数使用了 DEBUG_ONLY_THIS_PROCESS，这意味着调用 CreateProcess 的进程成为了调试器，而它启动的子进程成了被调试的进程。除了 DEBUG_ONLY_THIS_PROCESS 之外，还可以使用 DEBUG_PROCESS，两者的不同在于：DEBUG_PROCESS 会调试被调试进程以及它的所有子进程，而 DEBUG_ONLY_THIS_PROCESS 只调试被调试进程，不调试它的子进程。一般情况下我们只想调试一个进程，所以应使用后者。

建议在第六个参数中加上 CREATE_NEW_CONSOLE 标记。因为如果被调试程序是一个控制台程序的话，调试器和被调试程序的输出都在同一个控制台窗口内，显得很混乱，加上这个标记之后，被调试程序就会在一个新的控制台窗口中输出信息。如果被调试程序是一个窗口程序，这个标记没有影响。

调试循环

一个进程成为被调试进程之后，在完成了某些操作或者发生异常时，它会发送通知给调试器，然后将自身挂起，直到调试器命令它继续执行。这有点像 Windows 窗口的消息机制。

被调试进程发送的通知称为调试事件，DEBUG_EVENT 结构体描述了调试事件的内容：

```
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;
```

dwDebugEventCode 描述了调试事件的类型，总共有 9 类调试事件：

CREATE_PROCESS_DEBUG_EVENT	创建进程之后发送此类调试事件，这是调试器收到的第一个调试事件。
CREATE_THREAD_DEBUG_EVENT	创建一个线程之后发送此类调试事件。
EXCEPTION_DEBUG_EVENT	发生异常时发送此类调试事件。
EXIT_PROCESS_DEBUG_EVENT	进程结束后发送此类调试事件。
EXIT_THREAD_DEBUG_EVENT	一个线程结束后发送此类调试事件。
LOAD_DLL_DEBUG_EVENT	装载一个 DLL 模块之后发送此类调试事件。
OUTPUT_DEBUG_STRING_EVENT	被调试进程调用 OutputDebugString 之类的函数时发送此类调试事件。
RIP_EVENT	发生系统调试错误时发送此类调试事件。
UNLOAD_DLL_DEBUG_EVENT	卸载一个 DLL 模块之后发送此类调试事件。

每种调试事件的详细信息通过联合体 u 来记录，通过 u 的字段的名称可以很快地判断哪个字段与哪种事件关联。例如 CREATE_PROCESS_DEBUG_EVENT 调试事件的详细信息由 CreateProcessInfo 字段来记录。

dwProcessId 和 dwThreadId 分别是触发调试事件的进程 ID 和线程 ID。一个调试器可能同时调试多个进程，而每个进程内又可能有多个线程，通过这两个字段就可以知道调试事件是从哪个进程的哪个线程触发的了。

调试器通过 WaitForDebugEvent 函数获取调试事件，通过 ContinueDebugEvent 继续被调试进程的执行。ContinueDebugEvent 有三个参数，第一和第二个参数分别是进程 ID 和线程 ID，表示让指定进程内的指定线程继续执行。通常这是在一个循环中完成的。

异常

对于 EXCEPTION_DEBUG_EVENT 这类调试事件，这类调试事件是调试器与被调试进程进行交互的最主要手段，调试器可以使用这类事件完成断点、单步执行等操作。综上所述，异常是调试器的重点，所以需要首先了解关于异常的知识。

异常的分类

根据异常发生时是否可以恢复执行，可以将异常分为三种类型，分别是错误异常，陷阱异常以及中止异常。

错误异常和陷阱异常一般都可以修复，并且在修复后程序可以恢复执行。两者的不同之处在于，错误异常恢复执行时，是从引发异常的那条指令开始执行；而陷阱异常是从引发异常那条指令的下一条指令开始执行。中止异常属于严重的错误，程序不可以再继续执行。

根据异常产生的原因，可以将异常分为硬件异常和软件异常。硬件异常即由 CPU 引发的异常。

软件异常即程序调用 RaiseException 函数引发的异常，C++ 的 throw 语句最终也是调用该函数来抛出异常的。软件异常的异常代码可以在调用 RaiseException 时由程序员任意指定。

异常的分发

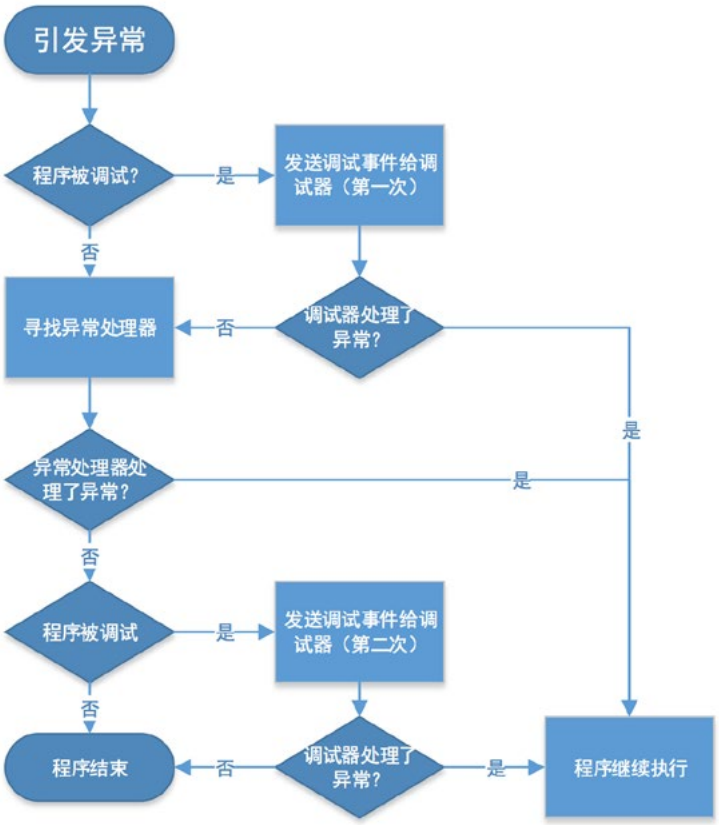
一个异常一旦发生了，就要经历一个复杂的分发过程。一般来说，一个异常有以下几种可能的结果：

1. 异常未被处理，程序因“应用程序错误”退出。
2. 异常被调试器处理了，程序在发生异常的地方继续执行（具体取决于是错误异常还是陷阱异常）。
3. 异常被程序内的异常处理器处理了，程序在发生异常的地方继续执行，或者转到异常处理块内继续执行。

下面来看一下异常的分发过程。为了突出重点，这里省略了很多细节：

1. 程序发生了一个异常，Windows 捕捉到这个异常，并转入内核态执行。
2. Windows 检查发生异常的程序是否正在被调试，如果是，则发送一个 EXCEPTION_DEBUG_EVENT 调试事件给调试器，这是调试器第一次收到该事件；如果否，则跳到第 4 步。
3. 调试器收到异常调试事件之后，如果在调用 ContinueDebugEvent 时第三个参数为 DBG_CONTINUE，即表示调试器已处理了该异常，程序在发生异常的地方继续执行，异常分发结束；如果第三个参数为 DBG_EXCEPTION_NOT_HANDLED，即表示调试器没有处理该异常，跳到第 4 步。
4. Windows 转回到用户态中执行，寻找可以处理该异常的异常处理器。如果找到，则进入异常处理器中执行，然后根据执行的结果继续程序的执行，异常分发结束；如果没找到，则跳到第 5 步。
5. Windows 又转回内核态中执行，再次检查发生异常的程序是否正在被调试，如果是，则再次发送一个 EXCEPTION_DEBUG_EVENT 调试事件给调试器，这是调试器第二次收到该事件；如果否，跳到第 7 步。
6. 调试器第二次处理该异常，如果调用 ContinueDebugEvent 时第三个参数为 DBG_CONTINUE，程序在发生异常的地方继续执行，异常分发结束；如果第三个参数为 DBG_EXCEPTION_NOT_HANDLED，跳到第 7 步。
7. 异常没有被处理，程序以“应用程序错误”结束。

下图展示了这个流程：



读取寄存器和内存

获取寄存器的值

每个线程都有一个上下文环境，它包含了有关线程的大部分信息，例如线程栈的地址，线程当前正在执行的指令地址等。上下文环境保存在寄存器中，系统进行线程调度的时候会发生上下文切换，实际上就是将一个线程的上下文环境保存到内存中，然后将另一个线程的上下文环境装入寄存器。

获取某个线程的上下文环境需要使用 GetThreadContext 函数，该函数声明如下：

BOOL WINAPI GetThreadContext(

HANDLE hThread,

LPCONTEXT lpContext);

第一个参数是线程的句柄，第二个参数是指向 CONTEXT 结构的指针。要注意，调用该函数之前需要设置 CONTEXT 结构的 ContextFlags 字段，指明你想要获取哪部分寄存器的值。该字段的取值如下：

CONTEXT_CONTROL	获取 EBP，EIP，CS，EFLAGS，ESP 和 SS 寄存器的值。
CONTEXT_INTEGER	获取 EAX，EBX，ECX，EDX，ESI 和 EDI 寄存器的值。
CONTEXT_SEGMENTS	获取 DS，ES，FS 和 GS 寄存器的值。
CONTEXT_FLOATING_POINT	获取有关浮点数寄存器的值。
CONTEXT_DEBUG_REGISTERS	获取 DR0，DR1，DR2，DR3，DR6，DR7 寄存器的值。
CONTEXT_FULL	等于 CONTEXT_CONTROL | CONTEXT_INTEGER | CONTEXT_SEGMENTS

调用 GetThreadContext 函数之后，CONTEXT 结构相应的字段就会被赋值，此时就可以输出各个寄存器的值了。

对于其它寄存器来说，直接输出它的值就可以了，但是 EFLAGS 寄存器的输出比较麻烦，因为它的每一位代表不同的含义，我们需要将这些含义也输出来。一般情况下我们只需要了解以下标志：

标志	位	含义
CF	0	进位标志。无符号数发生溢出时，该标志为 1，否则为 0。
PF	2	奇偶标志。运算结果的最低字节中包含偶数个 1 时，该标志为 1，否则为 0。
AF	4	辅助进位标志。运算结果的最低字节的第三位向高位进位时，该标志为 1，否则为 0。
ZF	6	0 标志。运算结果未 0 时，该标志为 1，否则为 0。
SF	7	符号标志。运算结果未负数时，该标志为 1，否则为 0。
DF	10	方向标志。该标志为 1 时，字符串指令每次操作后递减 ESI 和 EDI，为 0 时递增。
OF	11	溢出标志。有符号数发生溢出时，该标志为 1，否则为 0。

读取内存内容

读取进程的内存使用 ReadProcessMemory 函数，该函数声明如下：

```
BOOL WINAPI ReadProcessMemory(
    HANDLE hProcess,              // 进程句柄
    LPCVOID lpBaseAddress,        // 要读取的地址
    LPVOID lpBuffer,              // 一个缓冲区的指针，保存读取到的内容
    SIZE_T nSize,                 // 要读取的字节数
    SIZE_T* lpNumberOfBytesRead   // 一个变量的指针，保存实际读取到的字节数
);
```

要想成功读取到进程的内存，需要两个条件：一是 hProcess 句柄具有 PROCESS_VM_READ 的权限；二是由 lpBaseAddress 和 nSize 指定的内存范围必须位于用户模式地址空间内，而且是已分配的。

对于调试器来说，第一个条件很容易满足，因为调试器对被调试进程具有完整的权限，可以对其进行任意操作。

第二个条件意味着我们不能读取进程任意地址的内存，而是有一个限制。Windows 将进程的虚拟地址空间分成了四个分区，如下表所示：

分区	地址范围
空指针赋值分区	0x00000000~0x0000FFFF
用户模式分区	0x00010000~0x7FFEFFFF
64KB 禁入分区	0x7FFF0000~0x7FFFFFFF
内核模式分区	0x80000000~0xFFFFFFFF

空指针赋值分区主要为了帮助程序员检测对空指针的访问，任何对这一分区的读取或写入操作都会引发异常。64KB 禁入分区正如其名字所言，是禁止访问的，由 Windows 保留。内核模式分区由 Windows 的内核部分使用，运行于用户态的进程不能访问这一区域。进程只能访问用户模式分区的内存，对于其它分区的访问将会引发 ACCESS_VIOLATION 异常。

另外，并不是用户模式分区的任意部分都可以访问。我们知道，在 32 位保护模式下，进程的 4GB 地址空间是虚拟的，在物理内存中不存在。如果要使用某一部分地址空间的话，必须先向操作系统提交申请，让操作系统

为这部分地址空间分配物理内存。只有经过分配之后的地址空间才是可访问的，试图访问未分配的地址空间仍然会引发 ACCESS_VIOLATION 异常。

调试符号

什么是调试符号

一个调试器应该可以跟踪被调试程序执行到了什么地方，显示下一条将要执行的语句，显示各个变量的值，设置断点，进行单步执行等等，这些功能都需要一个基础设施的支持，那就是调试符号。

所谓符号，简单来说就是源代码中每个对象的名称。例如变量、函数、类型等，它们都有一个名称，以及其它的相关信息：变量有类型、地址等信息；函数有返回值类型、参数类型、地址等信息；类型有长度等信息。编译器在编译每个源文件的时候都会收集该源文件中的符号的信息，在生成目标文件的时候将这些信息保存到符号表中。链接器使用符号表中的信息将各个目标文件链接成可执行文件，同时将多个符号表整合成一个文件，这个文件就是用于调试的符号文件，它既可以嵌入可执行文件中，也可以独立存在。

加载调试符号

一个进程会有多个模块，每个模块都有它自己的符号文件，有关符号文件的信息保存在模块的可执行文件中。DbgHelp 通过符号处理器（Symbol Handler）来处理模块的符号文件。符号处理器位于调试器进程中，每个被调试的进程对应一个符号处理器。通常，调试器在被调试进程启动的时候创建符号处理器，在被调试进程结束的时候清理相应符号处理器占用的资源。

创建一个符号处理器使用 SymInitialize 函数，该函数声明如下：

```
BOOL WINAPI SymInitialize(
HANDLE hProcess,
PCTSTR UserSearchPath,
fInvadeProcess);
```

第一个参数是被调试进程的句柄，它是符号管理器的标识符，其它的 DbgHelp 函数都需要这样一个参数值指明使用哪个符号管理器。实际上这个参数不一定是句柄：当 fInvadeProcess 参数为 TRUE 时，它必须是一个有效的进程句柄；当 fInvadeProcess 为 FALSE 时，它可以是任意一个唯一的数值。

fInvadeProcess 的作用是指示是否加载进程所有模块的调试符号，如果该参数为 FALSE，那么 SymInitialize 只是创建一个符号处理器，不加载任何模块的调试符号，此时需要我们自己调用 SymLoadModule64 函数来加载模块；如果为 TRUE，SymInitialize 会遍历进程的所有模块，并加载其调试符号，所以在这种情况下 hProcess 必须是一个有效的进程句柄。

当 fInvadeProcess 为 TRUE 时，第二个参数 UserSearchPath 指示 SymInitialize 函数去哪里寻找符号文件。使用 PDB 符号文件的可执行文件中已包含有符号文件的绝对路径，如果符号文件不存在，SymInitialize 就会使用 UserSearchPath 指定的路径去寻找符号文件。该参数可指定多个路径，以分号（;）分割。如果该参数为 NULL，那么 SymInitialize 会按照以下的顺序寻找符号文件：

调试器进程的工作目录；

_NT_SYMBOL_PATH 环境变量指定的路径；

_NT_ALTERNATE_SYMBOL_PATH 环境变量指定的路径。

如果在以上路径中仍然找不到符号文件，SymInitialize 并不会返回 FALSE，而是返回 TRUE。也就是说，它成功创建了符号处理器，并且加载了模块的信息，但是没有加载调试符号（关于如何判断某个模块是否加载了调试符号，下文会有讲解）。实际上，SymInitialize 几乎不会返回 FALSE，然而在某种情况下它会这么做，下面会有关于这方面的说明。

根据对 SymInitialize 的描述，有两种方法可以加载调试符号。第一种方法是在调用 SymInitialize 的时候第三个参数传入 TRUE，由它负责加载每个模块的调试符号。这种方法的好处是方便，但是有一个前提：被调试进程必须初始化完毕。由于每个进程在初始化完毕之后都会引发一个断点异常，所以加载调试符号的最好的时机就是在处理这个初始断点的时候。

第二种方法是在调用 SymInitialize 的时候第三个参数传入 FALSE，然后对每个模块调用 SymLoadModule64 函数加载调试符号。我们可以在处理 CREATE_PROCESS_DEBUG_EVENT 和 LOAD_DLL_DEBUG_EVENT 事件时分别加载 exe 文件和 dll 文件的调试符号。SymLoadModule64 函数的声明如下：

```
DWORD64 WINAPI SymLoadModule64(
    HANDLE hProcess,
    HANDLE hFile,
    PCSTR ImageName,
    PCSTR ModuleName,
    DWORD64 BaseOfDll,
    DWORD SizeOfDll
);
```

第一个参数是符号处理器的标识符，也就是在调用 SymInitialize 时第一个参数的值。

第二个参数是模块文件的句柄，该函数通过这个文件句柄来获取有关符号文件的信息。你可能记得在 CREATE_PROCESS_DEBUG_INFO 和 LOAD_DLL_DEBUG_INFO 结构体中都有一个 hFile 的字段，这个字段刚好可以用在 SymLoadModule64 函数上。

第三个参数 ImageName 用于指定模块文件的路径和名称，当第二个参数为 NULL 时，SymLoadModule64 会通过这里指定的路径和名称去寻找模块文件。一般情况下都不会使用这个参数，因为我们可以使用更可靠的 hFile 参数。

第四个参数 ModuleName 为该模块赋予一个名称，在使用其它 DbgHelp 函数的时候可以通过这个名称来引用模块。如果该参数为 NULL，SymLoadModule64 会使用符号文件的文件名作为模块名称。

第五个参数 BaseOfDll 是模块加载到进程地址空间之后的基地址。这个参数很重要，因为符号文件中每个符号的地址都是相对于模块基地址的偏移地址，而不是绝对地址，这样的话，不论模块被加载到哪个地址，它的符号文件都是可用的。当然，这一切的前提是你将正确的模块基地址传给了 SymLoadModule64 函数。幸运的是，

CREATE_PROCESS_DEBUG_INFO 和 LOAD_DLL_DEBUG_INFO 结构体中已包含了一个 lpBaseOfImage 字段，我们直接使用即可，不必为了获取模块基地址而大动干戈。

至于最后一个参数 SizeOfDll，表示模块文件的大小。

判断符号文件的格式

前面说过，SymInitialize 在找不到符号文件的情况下仍然会返回 TRUE，此时它只加载了模块的信息，而没有加载调试符号。SymLoadModule64 函数同样如此。那么，如何知道某个模块是否含有调试信息呢？或者，如何知道某个模块的符号文件使用哪种格式呢？可以通过调用 SymGetModuleInfo64 函数来获取这些信息。该函数的声明如下：

```
BOOL WINAPI SymGetModuleInfo64(

HANDLE hProcess,

DWORD64 dwAddr,

PIMAGEHLP_MODULE64 ModuleInfo);
```

第一个参数是符号处理器的标识符，现在你应该对它很熟悉了。第二个参数是模块的基地址，也就是在调用 SymLoadModule64 时传给 BaseOfDll 参数的值。第三个参数是指向 IMAGEHLP_MODULE64 结构体的指针，调用函数完成之后模块的信息将会保存到这个结构体中。

IMAGEHLP_MODULE64 结构体含有非常多的字段，不过我们一般只关心其中的一个：SymType。这个字段指示模块使用的是哪种格式的符号文件，其可能的取值如下：

SymCoff	COFF 格式。
SymCv	CodeView 格式。
SymDeferred	调试符号是延迟加载的。下文会提及。
SymDia	DIA 格式。
SymExport	符号是从 DLL 文件的导出表中生成的。
SymNone	没有调试符号。
SymPdb	PDB 格式。
SymSym	使用 .sym 类型的符号文件。
SymVirtual	与 SymLoadModuleEx 函数的最后一个参数有关，还未知道什么意思。

在调用 SymGetModuleInfo64 之前需要将 IMAGEHLP_MODULE64 结构体的 SizeOfStruct 字段设置为 sizeof(IMAGEHLP_MODULE64)；

清理调试符号

在被调试进程结束的时候必须删除与之对应的符号处理器，以及清理它占用的资源。只要在处理 EXIT_PROCESS_DEBUG_EVENT 事件的时候调用 SymCleanup 函数就可以完成这个操作，该函数接受一个符号处理器的标识符。

另外，在 dll 文件卸载的时候也应该清理与之相关的调试符号，避免占用内存。这要在处理 UNLOAD_DLL_DEBUG_EVENT 事件时调用 SymUnloadModule64 函数。该函数接受一个符号处理器的标识符，以及模块的基地址，我们可以直接使用 UNLOAD_DLL_DEBUG_INFO 结构体中唯一的字段 lpBaseOfDll。

显示源代码

实现显示源代码的功能大概需要一下几个步骤：

1. 获取下一条要执行的指令的地址。
2. 通过调试符号获取该地址对应哪个源文件的哪一行。
3. 对于其它的行，通过调试符号获取它对应的地址。

第一步可以通过获取 EIP 寄存器的值来完成，相关的内容已经在上文中进行了讲解，这里不再重复。下面讲一下如何实现第二个和第三个步骤。

获取源文件以及行号

在调试符号中，记录了每一行源代码对应的地址。通过 DbgHelp 的 SymGetLineFromAddr64 函数可以由地址获取源文件路径以及行号。该函数的声明如下：

```
BOOL WINAPI SymGetLineFromAddr64(
HANDLE hProcess,
DWORD64 dwAddr,
PDWORD pdwDisplacement,
PIMAGEHLP_LINE64 Line);
```

hProcess 参数是符号处理器的标识符，dwAddr 是指令的地址。pdwDisplacement 是一个输出参数，用于获取 dwAddr 相对于它所在行的起始地址的偏移量，以字节为单位。之所以需要这么一个参数，是因为一行代码可能对应多条汇编指令，有了它就可以知道下一条要执行的指令位于这一行代码的哪个位置。

第四个参数是指向 IMAGEHLP_LINE64 结构体的指针，该结构体用来保存有关于行的信息，其声明如下：

```
typedef struct _IMAGEHLP_LINE64 {
    DWORD SizeOfStruct;
    PVOID Key;
    DWORD LineNumber;
    PTSTR FileName;
    DWORD64 Address;
} IMAGEHLP_LINE64, *PIMAGEHLP_LINE64;
```

SizeOfStruct 字段保存结构体的大小，在调用 SymGetLineFromAddr64 之前需要初始化这个字段，否则函数调用会失败。Key 字段是由操作系统保留的，我们不需要使用它。FileName 和 LineNumber 字段分别是源文件的绝对路径以及行号。Address 是该行的起始地址。

要注意，FileName 字段是一个指向字符串的指针，而这个字符串的存储空间并不需要我们自己分配，我们

也不需要释放这个指针指向的内存。实际上这个指针指向了调试符号内的某个地方，我们可以读取这些数据，但是不能修改其中的数据，一旦这些数据被修改，其它的 DbgHelp 函数可能会出现奇怪的问题。如果一定要修改这个字符串，要先将它复制到另一个地方再进行操作。我很奇怪为什么这个字段不是 PCTSTR 类型的，这样的话就不必担心这个字符串被修改了。

调用 SymGetLineFromAddr64 成功的条件有两个：一是 dwAddr 的值所在的模块已经通过 SymLoadModule64 函数加载到符号处理器中；二是该模块含有 SymGetLineFromAddr64 所需的调试符号信息。如果第一个条件没有满足，GetLastError 返回 126；如果第二个条件没有满足，GetLastError 返回 487。

获取行的地址

通过 SymGetLineFromAddr64 可以获取指令对应的源文件以及行号，那么能不能根据源文件路径以及行号获取行的地址呢？当然可以，SymGetLineFromName64 函数就是用作此目的的。该函数的声明如下：

```
BOOL WINAPI SymGetLineFromName64(
    HANDLE hProcess,
    PCTSTR ModuleName,
    PCTSTR FileName,
    DWORD dwLineNumber,
    PLONG lpDisplacement,
    PIMAGEHLP_LINE64 Line
);
```

该函数与 SymGetLineFromAddr64 很相似，都是通过 IMAGEHLP_LINE64 结构体来返回行的信息，并且都有一个 displacement 输出参数，不过这个参数在两个函数中的意义大不相同，下面将会详述。首先来看一下其它参数的含义。

ModuleName 用于指定模块的名称，上文中讲解 SymLoadModule64 函数时提到的 ModuleName 参数就可以用在这个地方（奇怪的是 SymLoadModule64 的 ModuleName 参数是 PCSTR 类型，而 SymGetLineFromName64 的 ModuleName 参数却是 PCTSTR 类型）。当 FileName 参数只指定了文件名，而多个模块中含有同名的源文件时，SymGetLineFromName64 就使用这个参数确定使用哪个模块的源文件。如果各个模块都没有同名的源文件，或者 FileName 指定的是绝对路径时，这个参数就没有必要了，指定为 NULL 即可。

FileName 和 dwLineNumber 参数分别指定源文件和行号。FileName 可以是文件名，也可以是绝对路径，正如上面的描述那样。dwLineNumber 是任意非零值，即使行号在源文件中不存在，甚至是负数，SymGetLineFromName64 也会返回 TRUE！那么我们如何知道指定的行号是否有效呢？只要检查 displacement 的值即可。大多数情况下，displacement 表示指定行与最接近该行的有效行的行号之差，而且有效行的行号要小于等于指定行的行号。所谓有效行即能够产生汇编指令的行（能产生汇编指令才会有对应的地址），可以用下面的式子表示（式中的变量均使用函数参数的名字）：

```
*lpDisplacement = dwLine - Line->LineNumber  (dwLine >= Line->LineNumber)
```

断点

断点是最基本和最重要的调试技术之一，下面讲解如何在调试器中实现断点功能。

什么是断点

在进行调试的时候，只有被调试进程暂停执行时调试器才可以对它执行操作，例如观察内存内容等。如果被调试进程不停下来的话，调试器是什么也做不了的。要使被调试进程停下来，除了几个在特定时刻才发生的调试事件外，唯一的途径就是引发异常。

断点正是用来达到上述目的的异常，在上文中的异常代码表中，有一种 EXCEPTION_BREAKPOINT 异常，它就是断点异常。虽然断点是一种异常，但并不意味着被调试进程发生了问题，它只是用来调试的一种手段，所以调试器应该将它和别的异常明显区分开来。实际上 Windows 对断点异常的处理也有一些微妙的不同，下文将会讲到。

断点有软件断点和硬件断点之分。硬件断点是通过 CPU 的寄存器来设置的，它的功能很强大，既可以在代码中设置断点，也可以在数据中设置断点，但是可以设置的数量有限。软件断点即通过 int 3 指令引发的断点，机器码是 0xCC，它只能设置在代码中，但没有数量的限制。本文只关注软件断点。

陷阱标志

除了断点之外，CPU 本身提供了一个单步执行的功能，也可以使程序在某处中断。在 CPU 的标志寄存器中，有一个 TF（Trap Flag）位，当该位为 1 时，CPU 每执行一条指令就会引发一次中断，Windows 以单步执行异常来通知调试器，异常代码为 EXCEPTION_SINGLE_STEP。每引发一次中断，CPU 都会自动将 TF 位设为 0，所以如果想连续单步执行多条指令，需要在每次处理单步执行异常时都重新设置 TF 位。

单步执行异常属于错误异常，引发异常的地址与 EIP 指向的地址相同。

断点功能原理

为了可以在任何指令处设置断点，调试器要将指令的第一个字节替换成 0xCC（int 3 的机器码），接收到断点异常之后，再替换回原来的那个字节，从该指令开始继续执行。这样就实现了在任意指令处中断，并对原程序毫无影响。

例如，下面的赋值语句对应一条汇编指令：

```
int b = 2;

C7 45 F8 02 00 00 00    mov    dword ptr [b],2
```

这条指令有 7 个字节。假如调试器想要在这条语句设置断点，它首先将指令的第一个字节 0xC7 保存起来，然后替换成 0xCC：

```
CC 45 F8 02 00 00 00
```

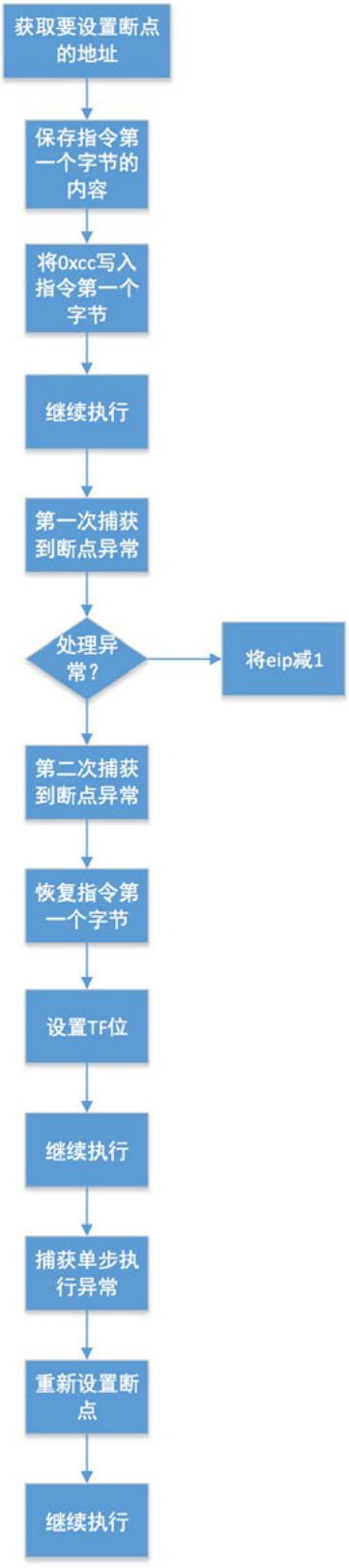
此时原来的 mov 指令变成了一条 int 3 指令和 6 个字节的垃圾数据。被调试程序不知道这个变化，它逐条指令地执行，到了 int 3 指令之后引发断点异常，暂停执行。此时被调试程序不能再往下执行了，因为接下来的 6 个字节是垃圾数据，尝试执行的话肯定会失败。

调试器可以选择在第一次或第二次接收断点异常时进行处理。如果在第一次接收时处理，它就要主动将被调

试进程的 EIP 减。如果在第二次接收时处理，就不需要修改被调试进程的 EIP 了，因为正如上文所说，第二次接收断点异常时 Windows 已经将 EIP 减 1 了。无论何时处理异常，调试器都要将 0xCC 替换回原来的 0xC7，然后以 DBG_CONTINUE 继续被调试进程执行。

最后还有一个问题需要留意，如果断点设置在循环的内部，或者设置在一个被多次调用的函数中，那么该断点只会中断一次，因为它在第一次中断之后就被取消了。为了让它持续有效，我们需要一种机制，让断点所在的指令执行完之后重新设置该断点。这可以借助 TF 位的帮助：处理断点异常的时候，在取消断点之后立即设置 TF 位，然后继续执行；在捕捉到单步执行异常时重新设置断点。

完整的断点功能流程图如下：


获取要设置断点的地址
保存指令第一个字节的内容
将0xcc写入指令第一个字节
继续执行
第一次捕获到断点异常
处理异常？
将eip减1
第二次捕获到断点异常
恢复指令第一个字节
设置TF位
继续执行
捕获单步执行异常
重新设置断点
继续执行

实现断点功能

了解了断点功能的原理，下面就来逐步实现这个功能。这里只描述大概的思路

首先是要确定断点的地址，注意，断点只能设置在指令的第一个字节，否则会破坏指令的结构，导致被调试进程无法执行。

确定地址之后就要替换指令第一个字节。读取这个字节可以使用 ReadProcessMemory 函数，写入字节可以使用 WriteProcessMemory 函数。

调试器必须保存一份断点列表，最好用一个结构体来表示断点，例如：

```
typedef struct {
    DWORD address;    // 断点地址
    BYTE content;     // 原指令第一个字节
} BREAK_POINT;
```

接下来是处理断点异常的方式。应该将断点分成三种类型：初始断点，被调试进程中的断点，以及调试器设置的断点。对于初始断点，不需要进行任何处理，因为它是由 Windows 管理的。如果对初始断点应用了以上的处理过程，被调试进程会无法运行。被调试进程中的断点即代码中显式加入的断点，对于这类断点，只要在第一次接收断点异常时报告给用户即可，不需要进行其它处理。而调试器设置的断点就要按照上文所说的方法来处理了。

如果选择在第一次接收断点异常时进行处理，那么需要使用 SetThreadContext 函数设置被调试进程的 EIP，该函数的参数与 GetThreadContext 完全一致。为了避免修改 EIP 而影响到其它的寄存器，应该先调用 GetThreadContext 填充 CONTEXT 结构，再调用 SetThreadContext。例如：

```
CONTEXT context;
context.ContextFlags = CONTEXT_CONTROL;
GetThreadContext(g_hThread, &context);
context.Eip -= 1;
SetThreadContext(g_hThread, &context);
```

设置 TF 位的方法与设置 EIP 的方法一致，同样是先调用 GetThreadContext，然后修改 Eflags 字段的值，再调用 SetThreadContext。TF 位是 EFLAGS 寄存器中的第 8 位（从 0 开始算），通过下面的语句可以设置 TF 位：

```
ontext.EFlags |= 0x100;
```

在处理单步执行异常时，不能简单认为 EIP 减 1 就是原断点的地址，因为断点所在指令的长度是不确定的。为了重新设置断点，需要保存该断点的地址，或者干脆将所有断点都重新设置一次。

在 main 函数设置断点

如果按照上面的处理方法将初始断点忽略之后，带来了一个新的问题：被调试进程此时不会在初始断点发生时暂停，而是一直运行到结束，我们根本没机会对它进行任何操作。解决这个问题的方法就是在 Main 函数的入

口处设置断点。这里所说的 Main 函数是一个统称，指代下面四个入口函数：

main

wmain

WinMain

wWinMain

一个 C/C++ 应用程序的入口函数必定是上面四个的其中之一。

为了在 Main 函数处设置断点，首先要知道它的地址，这就需要调试符号的帮助了。一个函数是一个符号，可以通过 SymFromName 函数根据符号名称获取符号的信息。该函数的声明如下：

```
BOOL WINAPI SymFromName(
HANDLE hProcess,
PCTSTR Name,
PSYMBOL_INFO Symbol);
```

第一个参数是符号处理器的标识符；第二个参数是符号的名称；第三个参数是指向 SYMBOL_INFO 结构体的指针，函数调用成功后符号的信息就保存在这个结构体中。该结构体的定义如下：

```
typedef struct _SYMBOL_INFO {
    ULONG SizeOfStruct;
    ULONG TypeIndex;
    ULONG64 Reserved[2];
    ULONG Index;
    ULONG Size;
    ULONG64 ModBase;
    ULONG Flags;
    ULONG64 Value;
    ULONG64 Address;
    ULONG Register;
    ULONG Scope;
    ULONG Tag;
    ULONG NameLen;
    ULONG MaxNameLen;
```

```
    TCHAR Name[1];
} SYMBOL_INFO, *PSYMBOL_INFO;
```

这个结构体有很多字段，但目前我们只关注 Address，它就是符号的起始地址。关于 SYMBOL_INFO 这个结构体，在后文中还会提到。

单步执行

单步执行有三种类型：StepIn，StepOver 和 StepOut，下面进行讲解每种单步执行的原理

StepIn 原理

StepIn 即逐条语句执行，遇到函数调用时进入函数内部。当用户对调试器下达 StepIn 命令时，调试器的实现方式如下：

1. 通过调试符号获取当前指令对应的行信息，并保存该行的信息。
2. 设置 TF 位，开始 CPU 的单步执行。
3. 在处理单步执行异常时，获取当前指令对应的行信息，与 1 中保存的行信息进行比较。如果相同，表示仍然在同一行上，转到 2；如果不相同，表示已到了不同的行，结束 StepIn。

行信息可以使用上文中介绍过的 SymGetLineFromAddr64 函数来获取。StepIn 的原理比较简单，需要注意的一点是，断点也是使用 TF 位来达到重新设置的目的，所以在处理单步执行异常时需要判断是否重新设置断点。另外，如果进行 StepIn 的那行代码存在断点，调试器只要恢复该断点所在指令就可以了，不需要通知用户；如果那行代码含有 __asm { int 3}; 语句，调试器只要忽略它即可，也不需要通知用户。

StepOver 原理

StepOver 即逐条语句执行，遇到函数调用时不进入函数内部。当用户对调试器下达 StepOver 命令时，调试器的实现方式如下：

1. 通过调试符号获取当前指令对应的行信息，并保存该行的信息。
2. 检查当前指令是否 CALL 指令。如果是，则在下一条指令设置一个断点，然后让被调试进程继续运行；如果不是，则设置 TF 位，开始 CPU 的单步执行，跳到 4。
3. 处理断点异常时，恢复断点所在指令第一个字节的内容。然后获取当前指令对应的行信息，与 1 中保存的行信息进行比较，如果相同，跳到 2；否则停止 StepOver。
4. 处理单步执行异常时，获取当前指令对应的行信息，与 1 中保存的行信息进行比较。如果相同，跳到 2；否则停止 StepOver。

StepOver 的原理与 StepIn 基本相同，唯一的不同就是遇到 CALL 指令时跳过，跳过的方法是在下一条指令设置断点，然后令被调试进程全速执行，触发断点之后再设置 TF 位，进行 CPU 的单步执行。在全速执行时，用户设置的断点和被调试进程中的断点不应该忽略，而应该像正常的执行那样中断并通知用户，此时应该停止 StepOver。在进行 CPU 的单步执行时，则应该像 StepIn 那样忽略断点。StepOver 需要断点来实现，所以应该将 StepOver 使用的断点和其它类型的断点区分开来。

StepOut 原理

StepOut 即跳出当前正在执行的函数，立即回到上一层函数。当用户对调试器下达 StepOut 命令时，调试器的实现方式如下：

1. 获取当前函数的 RET 指令地址，并在该指令设置断点，让被调试进程继续执行。
2. 处理断点异常时，恢复断点所在指令的第一个字节。从线程栈的顶部中获取返回地址，在该地址设置断点，然后让被调试进程继续执行。
3. 再次处理断点异常，恢复断点所在指令的第一个字节，结束 StepOut。

StepOut 有两个难点：一是如何获取 RET 指令的地址，二是如何获取函数的返回地址。对于第一个问题，可以使用 SymFromAddr 函数来获取函数的相关信息。SymFromAddr 函数的作用是由地址获取相应符号的信息，符号可以是变量或者函数。其声明如下：

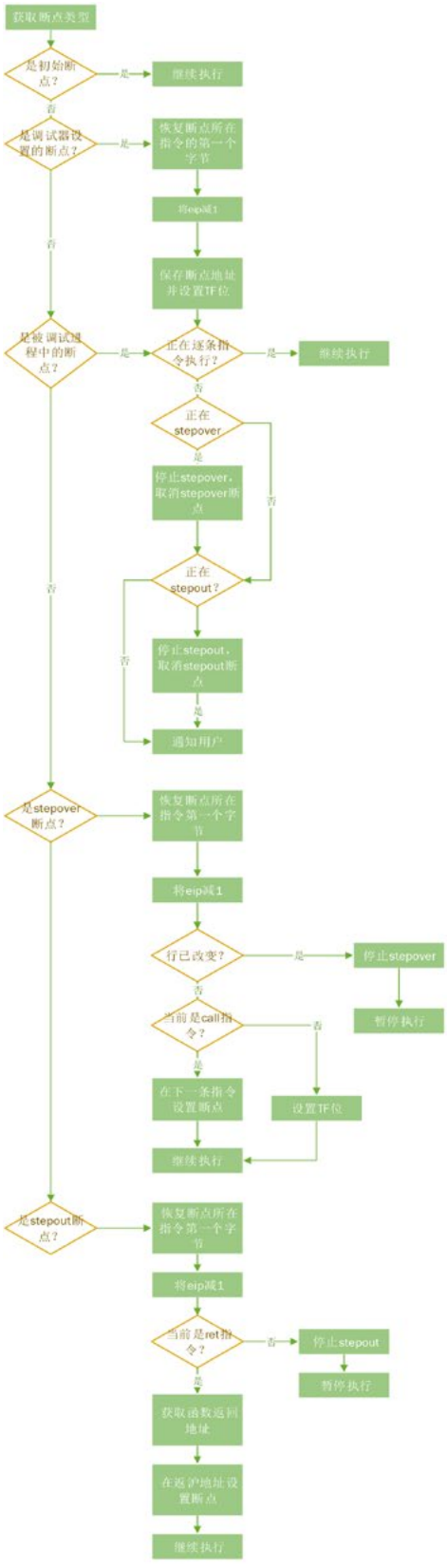
```
BOOL WINAPI SymFromAddr(
HANDLE hProcess,
DWORD64 Address,
PDWORD64 Displacement,
PSYMBOL_INFO Symbol );
```

第一个参数是符号处理器的标识符。第二个参数是地址。第三个参数是输出参数，函数调用成功之后返回 Address 相对于符号起始地址的偏移。第四个参数是指向 SYMBOL_INFO 结构体的指针，函数调用成功后与符号相关的信息都保存在这个结构体中。

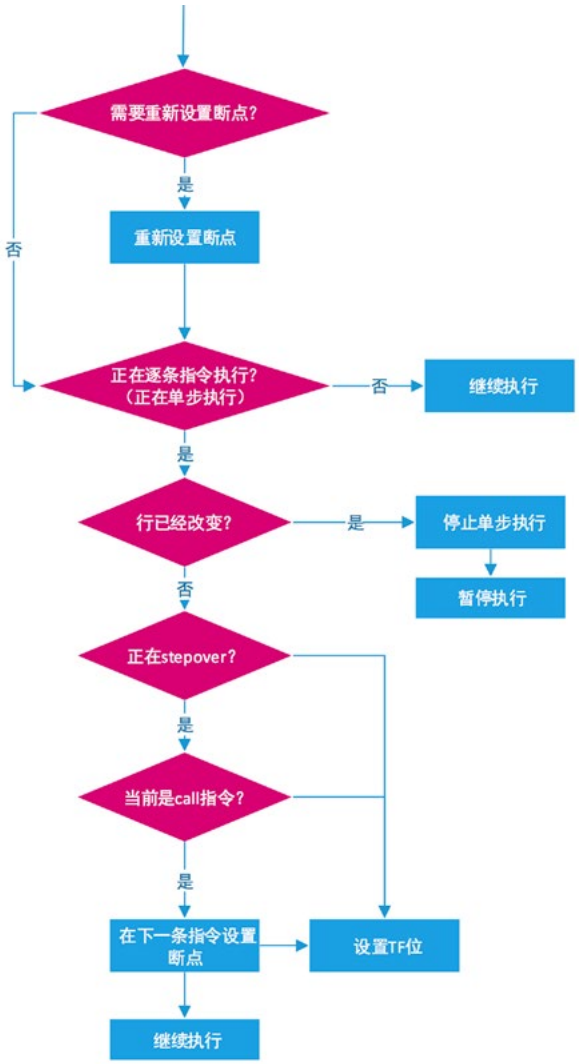
我们可以将当前被调试进程的 EIP 作为地址传给 SymFromAddr，以得到当前函数的信息。SYMBOL_INFO 的 Address 字段保存了函数第一条指令的地址，Size 字段保存了函数所有指令的字节长度，由于 RET 指令是函数的最后一条指令，所以 Address 加上 Size 再减去 RET 指令的长度就是 RET 指令的地址。那么接下来的问题就是要获得 RET 指令的长度。RET 指令比 CALL 指令简单得多，只有两种形式，一种只有一个字节长，十六进制值为 0xC3 或 0xCB；另一种有三个字节长，第一个字节的十六进制值为 0xC2 或 0xCA，后面两个字节是 ESP 将要加上的值。

第二个难点是获取函数的返回地址。如果你熟悉函数的调用过程，肯定知道在即将执行 RET 指令时，线程栈的顶部，也就是 ESP 所指的内存位置，必定是函数的返回地址。这就是在 RET 指令设置断点的目的。

断点异常处理过程图


获取断点类型
是初始断点？
是
继续执行
否
是调试器设置的断点？
是
恢复断点所在指令的第一个字节
将eip减1
保存断点地址并设置TF位
是被调试进程中的断点？
是
正在逐条指令执行？
是
继续执行
否
正在stepover
是
停止stepover，取消stepover断点
否
正在stepout？
否
停止stepout，取消stepout断点
是
通知用户
否
是stepover断点？
恢复断点所在指令第一个字节
将eip减1
行已改变？
是
停止stepover
暂停执行
否
当前是call指令？
否
是
在下一条指令设置断点
设置TF位
继续执行
是stepout断点？
恢复断点所在指令第一个字节
将eip减1
当前是ret指令？
否
停止stepout
暂停执行
是
获取函数返回地址
在返回地址设置断点
继续执行

单步执行异常处理过程图



显示函数调用栈

原理

首先我们来看一下显示调用栈所依据的原理。每个线程都有一个栈结构，用来记录函数的调用过程，这个栈是由高地址向低地址增长的，即栈底的地址比栈顶的地址大。ESP 寄存器的值是栈顶的地址，通过增加或减小 ESP 的值可以缩减或扩大栈的大小。如下所示为详细过程：

1. 在栈上压入参数。
2. 执行 CALL 指令，在栈上压入函数的返回地址。
3. 压入 EBP 寄存器的值。
4. 将 ESP 寄存器的值赋给 EBP 寄存器。

5. 减小 ESP 寄存器的值，为局部变量分配空间。

6. 执行函数代码。

7. 将 EBP 寄存器的值赋给 ESP 寄存器，等于回收了局部变量的空间。

8. 弹出栈顶的值，赋给 EBP，即将第 3 步中压入的值重新赋给 EBP。

9. 执行 RET 指令，弹出栈顶的返回地址。如果被调用函数负责回收参数的空间，则需要增加 ESP 的值。

完成第 3 步的指令是 push ebp，它是所有函数的第一条指令，因此每个函数在栈上都会保存有一个 EBP 值，标志了一个函数调用的开始，这就像分界线一样，将每个函数调用区分开来。从一个分界线开始，到下一个分界线之间的部分称作“栈帧”，一个栈帧代表一个函数调用。

在压入了 EBP 的值之后，第 4 步立即将 ESP 的值赋给了 EBP，此时 ESP 和 EBP 的值都是刚刚压入的值的地址。从此之后，ESP 的值随着指令的执行不断变化，而 EBP 的值在当前栈帧中永远不会改变，一直指向当前栈帧的起始地址，所以 EBP 也被称为“栈帧指针”。

函数返回的时候，第 7 步将 EBP 的值赋给 ESP，此时 ESP 指向第 3 步压入的值，然后第 8 步弹出这个值，赋给 EBP，恢复 EBP 在上一个函数调用中的值。

函数调用过程的第 3 步和第 4 步使得各个压入的 EBP 值形成了一个链表的结构，而 EBP 寄存器是链表的表头

正是这种链表结构的存在，使得获取函数调用栈成为可能。只要从 EBP 寄存器开始，沿着链表层层往上，就可以得到函数调用的轨迹。

由于 EBP 在当前函数调用中的不变性，调试版的程序都使用 EBP 作为变量和参数的基址，将 EBP 的值与一个偏移值相加就可以得到变量或参数的地址。有些发行版的程序会对函数的调用过程进行优化，省略了压入 EBP 的步骤，因此不能再使用 EBP 作为变量和参数的基址，也不能使用 EBP 链表来获取函数调用栈。

在 DbgHelp 中主要使用 StackWalk64 函数来获取函数调用栈，该函数的声明如下：

```
BOOL WINAPI StackWalk64(
    DWORD MachineType,
    HANDLE hProcess,
    HANDLE hThread,
    LPSTACKFRAME64 StackFrame,
    PVOID ContextRecord,
    PREAD_PROCESS_MEMORY_ROUTINE64 ReadMemoryRoutine,
    PFUNCTION_TABLE_ACCESS_ROUTINE64 FunctionTableAccessRoutine,
    PGET_MODULE_BASE_ROUTINE64 GetModuleBaseRoutine,
    PTRANSLATE_ADDRESS_ROUTINE64 TranslateAddress
```

);

该函数的参数比较多，这意味着灵活性，同时也意味着复杂性。事实上 StackWalk64 有多种不同的使用方式，使用何种方式由传入的参数决定。在这里我只介绍一种最简单的方式，这种方式已经足够了。如果想了解更多有关 StackWalk64 的信息，请参考 MSDN。

MachineType 参数指定 CPU 的类型，它的取值范围以及意义如下表所示（摘自 MSDN）：

Value	Meaning
IMAGE_FILE_MACHINE_I386	Intel x86
IMAGE_FILE_MACHINE_IA64	Intel Itanium Processor Family (IPF)
IMAGE_FILE_MACHINE_AMD64	x64 (AMD64 or EM64T)

调试器需要根据 CPU 的类型来设置该参数的值。在目前，大部分情况下都是设置为 IMAGE_FILE_MACHINE_I386。

hProcess 和 hThread 分别指定被调试进程的进程句柄以及线程句柄。而且在当前所使用的方式下，hProcess 必须是符号处理器的标识符。如果在调用 SymInitialize 创建符号处理器时使用的就是进程的句柄，那么在这里不会有任何问题；如果不是使用进程句柄，那么就必须用另一种方式调用 StackWalk64 了。

StackFrame 参数是一个 STACKFRAME64 结构体的指针，在调用 StackWalk64 之前需要初始化这个结构体，函数调用成功后，前一个栈帧的信息会保存到该结构体中；然后用这些信息再次调用 StackWalk64，以获取再前一个栈帧的信息……由此看出，StackWalk64 的工作就是获取指定栈帧的前一个栈帧，所以，必须要在循环中获取所有栈帧。STACKFRAME64 结构体中需要初始化的字段有三个：AddrPC，AddrStack 和 AddrFrame，它们分别表示程序计数器，线程栈顶以及栈帧指针，也是 EIP，ESP 和 EBP 的用途。这三个字段又分别是一个 ADDRESS64 结构体，这个结构体可以表示多种不同类型的地址，但 Windows 应用程序只会使用虚拟地址，所以 Mode 字段应设为 AddrModeFlat，Offset 字段设为上述寄存器的值。STACKFRAME64 结构体的初始化代码如下所示（context 为 CONTEXT 结构）：

```
STACKFRAME64 stackFrame = { 0 };
stackFrame.AddrPC.Mode = AddrModeFlat;
stackFrame.AddrPC.Offset = context.Eip;
stackFrame.AddrStack.Mode = AddrModeFlat;
stackFrame.AddrStack.Offset = context.Esp;
stackFrame.AddrFrame.Mode = AddrModeFlat;
stackFrame.AddrFrame.Offset = context.Ebp;
```

第一次成功调用 StackWalk64 后，STACKFRAME64 结构体的其它字段会被设置为适当的值，而上述的三个字段不会改变。从第二次调用开始才会真正获取前一个栈帧，这三个字段才会改变。

ContextRecord 参数是指向 CONTEXT 结构体的指针，调用之前需要使用 GetThreadContext 初始化该结构体，StackWalk64 函数会使用里面的值，并有可能会修改它。

ReadMemoryRoutine 是一个回调函数的指针，当 StackWalk64 函数需要读取被调试进程的内存时会调用该函数。如果不想提供这样的函数，最简单的方法就是设置该参数为 NULL，这样 StackWalk64 就会使用默认的函数，

此时 hProcess 必须是一个有效的进程句柄。

FunctionTableAccessRoutine 也是一个回调函数的指针，当 StackWalk64 需要访问函数表时会调用该函数。简单来说，函数表保存了每一个函数的信息，比如起始地址，长度等。这个参数不能为 NULL，但是我们可以将它设置为一个已有的函数，这个函数就是 SymFunctionTableAccess64。此时 hProcess 必须是符号处理器的标识符。

GetModuleBaseRoutine 又是一个回调函数的指针，当 StackWalk64 需要获取模块的基地址时会调用该函数。将该参数设置为 GetModuleBase64 函数即可，此时 hProcess 也必须是符号处理器的标识符。

最后的参数 TranslateAddress 仍然是回调函数的指针，不过该参数只用于 16 位地址的转换，几乎不会用到，设置为 NULL 即可。

可以看到，选择 StackWalk64 的何种使用方式由后面的四个参数决定。最简单的使用方式就是直接使用 NULL 或 DbgHelp 提供的函数作为这几个参数的值，不过此时对 hProcess 和 hThread 的限制最大。我们也可以自己提供这几个回调参数，此时 hProcess 和 hThread 几乎没有什么限制，它们只是作为唯一标识符。

StackWalk64 调用成功后，STACKFRAME64 结构体被赋值，在众多的字段中，我们只需要关心 AddrPC，它表示栈帧的返回地址（除了第一次调用 StackWalk64 之外），即 CALL 指令下一条指令的地址，本文开头的图片中显示的地址就是 AddrPC.Offset 的值。由于返回地址是指向前一个栈帧的，所以每次调用 StackWalk64 都会使 STACKFRAME64 结构体填充前一个栈帧的信息。StackWalk64 只能获取用户模式下的栈帧，如果栈帧遍历完毕，它会返回 FALSE。

获取模块名称

显示函数调用栈时最好同时显示函数所在的模块，这样可以方便知道每个函数位于哪个模块。有一个 SymGetModuleInfo64 函数可以获取模块的信息，但是却不可以获取模块的名称，而另一个 SymEnumerateModules64 函数可以做到这点，虽然它的使用方式比较麻烦。SymEnumerateModules64 用于枚举所有已经加载到符号处理器中的模块，它的声明如下：

```
BOOL WINAPI SymEnumerateModules64(
HANDLE hProcess,
PSYM_ENUMMODULES_CALLBACK64 EnumModulesCallback,
PVOID UserContext);
```

第一个参数是符号处理器的标识符。第二个参数是一个回调函数的指针，对于每个模块都会调用这个函数。该回调函数的声明如下：

```
BOOL CALLBACK SymEnumerateModulesProc64(
PCSTR ModuleName,
DWORD64 BaseOfDll,
PVOID UserContext);
```

ModuleName 是模块文件的绝对路径；BaseOfDll 是模块的基地址；UserContext 就是 SymEnumerateModules64 的第三个参数，可以通过这个参数给回调函数传递更多信息。

反调试技术总结

使用 Windows API 函数检测调试器是否存在是最简单的反调试技术。Windows 操作系统中提供了一些这样的 API，应用程序可以通过调用这些 API 来探测自己是否正在被调试。这些 API 有些是专门用来探测调试器的存在的。而另外一些 API 是处于其他目的而设计的，但也可以被改造用来探测调试器的存在。

通常，防止恶意代码使用 API 进行反调试的最简单的方法是在恶意代码运行期间修改恶意代码，使其不能调用探测调试器的 API 函数，或者修改这些 API 函数的返回值，确保恶意代码执行合适的路径。与这些方法相比，较复杂的方法是挂钩这些函数，比如使用 rootkit 技术。

1.IsDebuggerPresent

探测调试器是否存在的最简单的 API 函数是 IsDebuggerPresent。它会查询进程环境块（PEB）中的 IdDebugged 标志，如果进程没有运行在调试器环境中，函数返回 0；如果调试附加了进程，函数返回一个非零值。

PEB 结构体如下所示：

```
typedef struct _PEB {
    BYTE                    Reserved1[2];
    BYTE                    BeingDebugged;
    BYTE                    Reserved2[1];
    PVOID                   Reserved3[2];
    PPEB_LDR_DATA           Ldr;
    PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
    BYTE                    Reserved4[104];
    PVOID                   Reserved5[52];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE                    Reserved6[128];
    PVOID                   Reserved7[1];
    ULONG                   SessionId;
} PEB, *PPEB;
```

2.FindWindow

使用此函数查找目标窗口，如果找到，可以禁用窗口，也可以直接退出程序

函数声明：

```
HWND FindWindow
(
```

LPCSTR lpClassName,

LPCSTR lpWindowName

);

参数表：

lpClassName：指向一个以 NULL 字符结尾的、用来指定类名的字符串或一个可以确定类名字符串的原子。如果这个参数是一个原子，那么它必须是一个在调用此函数前已经通过 GlobalAddAtom 函数创建好的全局原子。这个原子（一个 16bit 的值），必须被放置在 lpClassName 的低位字节中，lpClassName 的高位字节置零。

如果该参数为 null 时，将会寻找任何与 lpWindowName 参数匹配的窗口

lpWindowName：指向一个以 NULL 字符结尾的、用来指定窗口名（即窗口标题）的字符串。如果此参数为 NULL，则匹配所有窗口名。

```
void CAntiDebugDlg::OnBnClickedBtnFindwindow()
{
    HWND Hwnd = NULL;
    Hwnd = ::FindWindow(L"OllyDbg", NULL);
    if (Hwnd == NULL) {
        MessageBoxW(L"Not Being Debugged!");
    }
    else {
        MessageBoxW(L"Being Debugged!");
    }
}
```

3. 枚举窗口

使用 EnumWindow 函数枚举窗口，并且为每一窗口调用一次回调函数，在回调函数中可以调用 GetWindowText 获取窗口的标题。与目标窗口名进行比对，如果比对成功，则说明发现调试器。

函数声明：

WINUSERAPI

BOOL

WINAPI

EnumWindows(

```
    _In_ WNDENUMPROC lpEnumFunc,
    _In_ LPARAM lParam
    );
```

参数表：

lpEnumFunc：回调函数指针。

lParam：指定一个传递给回调函数的应用程序定义值。

回调函数原型：

BOOL CALLBACK EnumWindowsProc(HWND *hwnd*,LPARAM *lParam*);

参数表：

Hwnd：顶层窗口的句柄。

Lparam：应用程序定义的一个值（即 EnumWindows 中的 lParam）。

Int GetWindowText(HWND hWnd,LPTSTR lpString,Int nMaxCount);

参数表：

hWnd：带文本的窗口或控件的句柄。

lpString：指向接收文本的缓冲区的指针。

nMaxCount：指定要保存在缓冲区内的字符的最大个数，其中包含 NULL 字符。如果文本超过界限，它就被截断。

```
BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam)
{
    WCHAR wzChar[100] = { 0 };
    CStringW strData = L"OllyDbg";
    if (IsWindowVisible(hwnd)) {
        GetWindowText(hwnd, wzChar, 100);
        if (wcsstr(wzChar, strData)) {
            MessageBoxW(NULL,L"Being Debugged!",NULL,0);
            g_bDebugged = TRUE;
            return FALSE;
        }
    }
```

```
    return TRUE;
}
void CAntiDebugDlg::OnBnClickedBtnEnumwindow()
{
    EnumWindows(EnumWindowsProc, NULL);
    if (g_bDebugged == FALSE) {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

4. 枚举进程

枚举进程列表，查看是否有调试器进程

函数声明：

```
HANDLE WINAPI CreateToolhelp32Snapshot(
DWORD dwFlags,
    DWORD th32ProcessID
);
```

通过获取进程信息为指定的进程、进程使用的堆 [HEAP]、模块 [MODULE]、线程建立一个快照。

参数表：

dwFlags：用来指定“快照”中需要返回的对象，可以是 TH32CS_SNAPPROCESS 等

th32ProcessID: 一个进程 ID 号，用来指定要获取哪一个进程的快照，当获取系统进程列表或获取 当前进程快照时可以设为 0

```
BOOL
WINAPI
Process32FirstW(
    HANDLE hSnapshot,
    LPPROCESSENTRY32W lppe
        );
```

process32First 是一个进程获取函数，当我们利用函数 CreateToolhelp32Snapshot() 获得当前运行进程的快照后，我们可以利用 process32First 函数来获得第一个进程的句柄。

```
BOOL
WINAPI
Process32NextW(
    HANDLE hSnapshot,
    LPPROCESSENTRY32W lppe
        );
```

Process32Next 是一个进程获取函数，当我们利用函数 CreateToolhelp32Snapshot() 获得当前运行进程的快照后 , 我们可以利用 Process32Next 函数来获得下一个进程的句柄。

```
void CAntiDebugDlg::OnBnClickedBtnEnumprocess()
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE hwnd = NULL;
    PROCESSENTRY32W pe32 = { 0 };
    pe32.dwSize = sizeof(pe32);// 如果没有这句，得出的路径不对
    WCHAR str[] = L"OLLYDBG";
    CStringW strTemp;
    BOOL bOK = FALSE;
    hwnd = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
    if (hwnd != INVALID_HANDLE_VALUE) {
        bool bMore = Process32FirstW(hwnd, &pe32);
        do {
            strTemp = pe32.szExeFile;
            // 统一转换为大写进行比较
            strTemp.MakeUpper();
            if (wcsstr(strTemp, str)) {
                MessageBoxW(L"Being Debugged!");
                bOK = TRUE;
                break;
            }
```

```
            else if (wcsstr(pe32.szExeFile, L"WINDBG")) {
                MessageBoxW(L"Being Debugged!");
                bOK = TRUE;
                break;
            }
        } while (Process32NextW(hwnd, &pe32));
    }
    if (bOK == FALSE) {
        MessageBoxW(L"Not Being Debugged!");
    }
    CloseHandle(hwnd);
}
```

5. 查看父进程是不是 Explorer

当我们双击运行应用程序的时候，父进程都是 Explorer，如果是通过调试器启动的，父进程就不是 Explorer。

通过 GetCurrentProcessId() 获得当前进程的 ID

通过桌面窗口类和名称获得 Explorer 进程的 ID

使用 Process32First/Next() 函数枚举进程列表，通过 PROCESSENTRY32.th32ParentProcessID 获得的当前进程的父进程 ID 与 Explorer 的 ID 进程比对。如果不一样的很可能被调试器附加

函数声明：

```
DWORD GetWindowThreadProcessId(
HWND hWnd,
LPDWORD lpdwProcessId
);
```

找出某个窗口的创建者（线程或进程），返回创建者的标志符。

参数说明：

hWnd：（向函数提供的）被查找窗口的句柄 .

lpdwProcessId：进程号的存放地址（变量地址）

使用方法：

```
void CAntiDebugDlg::OnBnClickedBtnExplorer()
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE hwnd = NULL;
    HANDLE hexplorer = NULL;
    PROCESSENTRY32 pe32 = { 0 };
    pe32.dwSize = sizeof(pe32);
    CStringW str = L"explorer";
    DWORD ExplorerId = 0;
    DWORD SelfId = 0;
    DWORD SelfParentId = 0;
    SelfId = GetCurrentProcessId();
    hexplorer = ::FindWindowW(L"Progman",NULL);
    GetWindowThreadProcessId((HWND)hexplorer, &ExplorerId);
    hwnd = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
    if (hwnd != INVALID_HANDLE_VALUE)
    {
        Process32FirstW(hwnd, &pe32);
        do
        {
            if (SelfId == pe32.th32ProcessID)
            {
                SelfParentId = pe32.th32ParentProcessID;
            }
        } while (Process32NextW(hwnd,&pe32));
    }
    if (ExplorerId == SelfParentId) {
        MessageBoxW(L"Not Being Debugged!");
    }
```

```
    else
    {
        MessageBoxW(L"Being Debugged!");
    }
    CloseHandle(hwnd);
}
```

6. 检测系统时钟

当进程被调试时，调试器事件处理代码、步过指令等将占用 CPU 循环。如果相邻指令之间所花费的时间如果大大超出常规，就意味着进程很可能是在被调试。有如下两种时钟检测来探测调试器存在的方法：

- 记录执行一段操作前后的时间戳，然后比较这两个时间戳，如果存在滞后，则可以认为存在调试器
- 记录触发一个异常前后的时间戳。如果不调试进程，可以很快处理完异常，因为调试器处理异常的速度非常慢。默认情况下，调试器处理异常时需要认为干预，这导致大量延迟。虽然很多调试器允许我们忽略异常，将异常直接返回给程序，但这样操作仍然存在不小的延迟

首先我们可以使用 rdtsc 指令，它返回至系统重新启动以来的时钟数，并且将其作为一个 64 位的值存入 edx:eax 中。恶意代码运行两次 rdstc 指令，然后比较两次读取之间的差值来判断是否存在调试器。

另外也可以使用函数 QueryPerformanceCounter 和 GetTickCount。同 rdstc 指令一样，这两个 API 函数也可以被用来执行一个反调试的时钟检测。为了获取比较的时间差，调用两次函数查询这个计数器，如果两次调用时间话费时间过于长，则可以认为存在调试器。

GetTickCount 返回从操作系统启动所经过的毫秒数。调用两次此函数，如果返回值的差值相差反常，就说明在调试状态。

QueryPerformanceCounter 和 GetTickCount 的原理类似，这里以 GetTickCount 为例说明使用方法：

```
void CAntiDebugDlg::OnBnClickedBtnGettickcount()
{
    // TODO: 在此添加控件通知处理程序代码
    DWORD dwTime1 = 0;
    DWORD dwTime2 = 0;
    dwTime1 = GetTickCount();
    GetCurrentProcessId();
    GetCurrentProcessId();
    GetCurrentProcessId();
```

```
    dwTime2 = GetTickCount();
    if (dwTime2 - dwTime1 > 100)
    {
        MessageBoxW(L"Being Debugged!");
    }
    else {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

7. 查看 StartupInfo 结构

在 windows 操作系统中，Explorer 创建进程的时候会把 STARTUPINFO 结构中的某些值设为 0，非 Explorer 创建进程的时候会忽略这个结构中的值，所以可以通过查看这个结构中的值是不是为 0 来判断是否在调试状态。

函数说明：

```
WINBASEAPI
VOID
WINAPI
GetStartupInfoW(
    _Out_ LPSTARTUPINFOW lpStartupInfo
        );
```

取得进程在启动时被指定的 STARTUPINFO 结构

```
typedef struct _STARTUPINFOW {
    DWORD   cb;
    LPWSTR  lpReserved;
    LPWSTR  lpDesktop;
    LPWSTR  lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
```

```
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    LPBYTE  lpReserved2;
    HANDLE  hStdInput;
    HANDLE  hStdOutput;
    HANDLE  hStdError;
} STARTUPINFOW, *LPSTARTUPINFOW;
```

使用方法：

```
void CAntiDebugDlg::OnBnClickedBtnStartupinfor()
{
    // TODO: 在此添加控件通知处理程序代码
    STARTUPINFO info = { 0 };
    GetStartupInfo(&info);
    if (info.dwX != 0 || info.dwY != 0 || info.dwXCountChars != 0 || info.dwYCountChars != 0
        || info.dwFillAttribute != 0 || info.dwXSize != 0 || info.dwYSize != 0)
    {
        MessageBoxW(L"Being Debugged!");
    }
    else {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

8.BeingDebugged，NTGlobalFlag

IsDebuggerPresent() API 检测进程环境块 (PEB) 中的 BeingDebugged 标志检查这个标志以确定进程是否正在被用户模式的调试器调试。通过 fs:[30] 可以获得 PEB 地址。然后通过不同的偏移访问不同的值

通常程序没有被调试时， PEB 另一个成员 NtGlobalFlag（偏移 0x68）值为 0，如果进程被调试通常值为 0x70（代表下述标志被设置）：

FLG_HEAP_ENABLE_TAIL_CHECK(0X10)

FLG_HEAP_ENABLE_FREE_CHECK(0X20)

FLG_HEAP_VALIDATE_PARAMETERS(0X40)

9.CheckRemoteDebuggerPresent

这个函数同 IsDebuggerPresent 函数几乎一致，它用来检测本机器中的一个进程是否运行在调试器中。同时，它也检查 PEB 结构中的 IsDebugged 属性。他不仅可以探测进程自身是否被调试，同时可以探测系统其他进程是否被调试。这个函数将一个进程句柄作为参数，检查这个句柄对应的进程是否被调试器附加，同时，CheckRemoteDebuggerPresent 也可以通过传递自身进程句柄探测自己是否被调试。

函数声明：

BOOL CheckRemoteDebuggerPresent(

HANDLE hProcess,

PBOOL pbDebuggerPresent

)

此函数用来确定是否有调试器附加到进程。

参数表：

hProcess：进程句柄

pbDebuggerPresent：指向一个 BOOL 的变量，如果进程被调试，此变量被赋值为 TRUE。

使用方法：

typedef *BOOL*(*WINAPI* *CHECK_REMOTE_DEBUGGER_PRESENT)(*HANDLE*, *PBOOL*);

void CAntiDebugDlg::OnBnClickedBtnCheckremote()

{

```
// TODO: 在此添加控件通知处理程序代码
HANDLE      hProcess;
HINSTANCE   hModule;
BOOL        bDebuggerPresent = FALSE;
CHECK_REMOTE_DEBUGGER_PRESENT CheckRemoteDebuggerPresent;
hModule = GetModuleHandleA("Kernel32");
```

```
        CheckRemoteDebuggerPresent =
            (CHECK_REMOTE_DEBUGGER_PRESENT)GetProcAddress(hModule,
"CheckRemoteDebuggerPresent");
        hProcess = GetCurrentProcess();
        CheckRemoteDebuggerPresent(hProcess, &bDebuggerPresent);
        if (bDebuggerPresent == TRUE)
        {
            MessageBoxW(L"Being Debugged!");
        }
        else
        {
            MessageBoxW(L"Not Being Debugged!");
        }
    }
```

10.NtQueryInformationProcess

ntdll!NtQueryInformationProcess() 有 5 个参数。为了检测调试器的存在，需要将 ProcessInformationclass 参数设为 ProcessDebugPort(7)。NtQueryInformationProcess() 检索内核结构 EPROCESS 的 DebugPort 成员，这个成员是系统用来与调试器通信的端口句柄。非 0 的 DebugPort 成员意味着进程正在被用户模式的调试器调试。如果是这样的话， ProcessInformation 将被置为 0xFFFFFFFF ，否则 ProcessInformation 将被置为 0。

函数声明：

```
NTSTATUS WINAPI NtQueryInformationProcess(
    _In_        HANDLE              ProcessHandle,
    _In_        PROCESSINFOCLASS    ProcessInformationClass,
    _Out_       PVOID               ProcessInformation,
    _In_        ULONG               ProcessInformationLength,
    _Out_opt_   PULONG              ReturnLength
);
```

参数表：

ProcessHandle：进程句柄

ProcessInformationClass：信息类型

ProcessInformation： 缓冲指针

ProcessInformationLength： 以字节为单位的缓冲大小

ReturnLength：写入缓冲的字节数

使用方法：

```
typedef NTSTATUS(_stdcall *ZW_QUERY_INFORMATION_PROCESS)(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass, // 该参数也需要上面声明的数据结构
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength
    ); // 定义函数指针
void CAntiDebugDlg::OnBnClickedBtnQueryinforpro()
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE      hProcess;
    HINSTANCE   hModule;
    DWORD       dwResult;
    ZW_QUERY_INFORMATION_PROCESS ZwQueryInformationProcess;
    hModule = GetModuleHandleW(L"ntdll.dll");
    ZwQueryInformationProcess = (ZW_QUERY_INFORMATION_PROCESS)GetProcAddress(hModule,"ZwQueryInformationProcess");
    hProcess = GetCurrentProcess();
    ZwQueryInformationProcess(
        hProcess,
        ProcessDebugPort,
        &dwResult,
        4,
        NULL);
```

```
    if (dwResult != 0)
    {
        MessageBoxW(L"Being Debugged!");
    }
    else
    {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

11.SetUnhandledExceptionFilter

调试器中步过 INT3 和 INT1 指令的时候，由于调试器通常会处理这些调试中断，所以设置的异常处理例程默认情况下不会被调用， Debugger Interrupts 就利用了这个事实。这样我们可以在异常处理例程中设置标志，通过 INT 指令后如果这些标志没有被设置则意味着进程正在被调试。

函数说明：

LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter (_In_LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter

) ;

设置异常捕获函数 .，当异常没有处理的时候 , 系统就会调用 SetUnhandledExceptionFilter 所设置异常处理函数 .

参数表：

lpTopLevelExceptionFilter：函数指针。当异常发生时，且程序不处于调试模式（在 vs 或者别的调试器里运行）则首先调用该函数。

使用方法：

```
static DWORD lpOldHandler;
static DWORD NewEip;
typedef LPTOP_LEVEL_EXCEPTION_FILTER(_stdcall  *pSetUnhandledExceptionFilter)(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
    );
pSetUnhandledExceptionFilter lpSetUnhandledExceptionFilter;
LONG WINAPI TopUnhandledExceptionFilter(
```

```
        struct _EXCEPTION_POINTERS *ExceptionInfo
    )
    {
        _asm pushad
        lpSetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)lpOldHandler);
        ExceptionInfo->ContextRecord->Eip = NewEip;// 转移到安全位置
        _asm popad
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    void CAntiDebugDlg::OnBnClickedBtnSetunhandleexcfilter()
    {
        // TODO: 在此添加控件通知处理程序代码
        bool isDebugged = 0;
        // TODO: Add your control notification handler code here
        lpSetUnhandledExceptionFilter = (pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary((L"kernel32.dll")),
            "SetUnhandledExceptionFilter");
        // 当异常没有处理的时候，系统就会调用此函数所设置的异常处理函数，此函数返回以前设置的回调函数
        lpOldHandler = (DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilter);
        _asm {  // 获取这个安全地址
            call me;
    me :
            pop NewEip;
            mov NewEip, offset safe;
            int 3;  // 触发异常 , 如果被调试，不会走自己定义的异常处理函数
        }
    //    MessageBoxW(L"Being Debugged!");
        isDebugged = 1;
```

```
    _asm {
safe:
    }
    if (1 == isDebugged) {
        MessageBoxW(L"Being Debugged!");
    }
    else {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

12.SeDebugPrivilege 进程权限

默认情况下进程没有 SeDebugPrivilege 权限，调试时，会从调试器继承这个权限，可以通过打开 CSRSS.EXE 进程间接地使用 SeDebugPrivilege 来判断进程是否被调试。

使用方法：

```
void CAntiDebugDlg::OnBnClickedBtnSedebugpre()
{
    // TODO: 在此添加控件通知处理程序代码
    HANDLE hProcessSnap;
    HANDLE hProcess;
    PROCESSENTRY32 tp32 = { 0 };  // 结构体
    tp32.dwSize = sizeof(tp32);
    CString str = L"csrss.exe";
    hProcessSnap = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
    if (INVALID_HANDLE_VALUE != hProcessSnap)
    {
        Process32First(hProcessSnap, &tp32);
        do {
            if (0 == lstrcmpi(str, tp32.szExeFile))
            {
```

```
            hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, NULL, tp32.th32ProcessID);
            if (NULL != hProcess)
            {
                MessageBoxW(L"Being Debugged!");
            }
            else
            {
                MessageBoxW(L"Not Being Debugged!");
            }
            CloseHandle(hProcess);
        }
    } while (Process32Next(hProcessSnap, &tp32));
  }
  CloseHandle(hProcessSnap);
}
```

13.GuardPages

这个检查是针对 OllyDbg 的，因为它和 OllyDbg 的内存访问 / 写入断点特性相关。除了硬件断点和软件断点外， OllyDbg 允许设置一个内存访问 / 写入断点，这种类型的断点是通过页面保护来实现的。简单地说，页面保护提供了当应用程序的某块内存被访问时获得通知这样一个途径。

页面保护是通过 PAGE_GUARD 页面保护修改符来设置的，如果访问的内存地址是受保护页面的一部分，将会产生一个 STATUS_GUARD_PAGE_VIOLATION(0x80000001) 异常。如果进程被 OllyDbg 调试并且受保护的页面被访问，将不会抛出异常，访问将会被当作内存断点

使用方法：

```
static bool isDebugged = 1;
LONG WINAPI TopUnhandledExceptionFilter2(
    struct _EXCEPTION_POINTERS *ExceptionInfo
)
{
    _asm pushad
```

```
        lpSetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)lpOldHandler);
        ExceptionInfo->ContextRecord->Eip = NewEip;
        isDebugged = 0;
        _asm popad
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    void CAntiDebugDlg::OnBnClickedBtnGuidpages()
    {
        // TODO: 在此添加控件通知处理程序代码
        ULONG dwOldType;
        DWORD dwPageSize;
        LPVOID lpvBase;                    // 获取内存的基地址
        SYSTEM_INFO sSysInfo;              // 系统信息
        GetSystemInfo(&sSysInfo);          // 获取系统信息
        dwPageSize = sSysInfo.dwPageSize;  // 系统内存页大小
        lpSetUnhandledExceptionFilter = (pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(L"kern
el32.dll"),
            "SetUnhandledExceptionFilter");
        lpOldHandler = (DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilter2);
        // 分配内存
        lpvBase = VirtualAlloc(NULL, dwPageSize, MEM_COMMIT, PAGE_READWRITE);
        if (lpvBase == NULL)
            MessageBoxW(L"VirtualAlloc Error");
        _asm {
            mov   NewEip, offset safe // 方式二，更简单
            mov   eax, lpvBase
            push  eax
            mov   byte ptr[eax], 0C3H // 写一个 RETN 到保留内存，以便下面的调用
        }
```

```
    if (0 == ::VirtualProtect(lpvBase, dwPageSize, PAGE_EXECUTE_READ | PAGE_GUARD, &dwOldType)) {
        MessageBoxW(L"VirtualProtect Error");
    }
    _asm {
        pop    ecx
        call     ecx              // 调用时压栈
        safe :
        pop     ecx              // 堆栈平衡，弹出调用时的压栈
    }
    if (1 == isDebugged) {
        MessageBoxW(L"Being Debugged!");
    }
    else {
        MessageBoxW(L"Not Being Debugged!");
    }
    VirtualFree(lpvBase, dwPageSize, MEM_DECOMMIT);
}
```

14. 软件断点

调试器设置断点的基本机制是用软件中断指令 INT3. 临时替换运行程序中的一条指令，然后当程序运行这条指令时，调用调试异常处理例程，INT 3 指令的机器码是 0xCC，因此无论何时，使用调试器设置一个断点，它都会插入一个 0xCC 来修改代码，恶意代码常用的一种反调试技术是在它的代码中查找机器码 0xCC 来扫描调试器对它代码的 INT 3 修改。所以可以通过在受保护的代码段和（或） API 函数中扫描字节 0xCC 来识别软件断点，可以搜索在一般地方下的断点，或者是在函数中下的断点

使用方法：

普通断点：

```
BOOL DetectBreakPoints()
{
    BOOL bDebugged = FALSE;
    _asm
```

```
    {
        jmp CodeEnd;
    CodeStart:
        mov eax, ecx;
        nop;
        push eax;
        push ecx;
        pop ecx;
        pop eax;
    CodeEnd:
        cld;
        mov edi, offset CodeStart;
        mov edx, offset CodeStart;
        mov ecx, offset CodeEnd;
        sub ecx, edx;
        mov al, 0CCH;
        repne scasb;
        jnz NotDebugged;
        mov bDebugged, 1;
    NotDebugged:
    }
    return bDebugged;
}
void CAntiDebugDlg::OnBnClickedBtnBreakpoint()
{
    // TODO: 在此添加控件通知处理程序代码
    if (DetectBreakPoints())
    {
        MessageBoxW(L"Being Debugged!");
```

```
    }
    else
    {
        MessageBoxW(L"Not Being Debugged");
    }
}
```

函数断点：

```
BOOL DetectFuncBreakpoints()
{
    BOOL bFoundOD;
    bFoundOD = FALSE;
    DWORD dwAddr;
    dwAddr = (DWORD)::GetProcAddress(LoadLibrary(L"user32.dll"), "MessageBoxA");
    __asm
    {
        cld; 检测代码开始
        mov    edi, dwAddr
        mov    ecx, 100; 100bytes
        mov    al, 0CCH
        repne   scasb
        jnz      ODNotFound
        mov bFoundOD, 1
        ODNotFound:
    }
    return bFoundOD;
}
void CAntiDebugDlg::OnBnClickedBtnFuncbreakpoint()
{
    // TODO: 在此添加控件通知处理程序代码
```

```
    if (DetectFuncBreakpoints())
    {
        MessageBoxW(L"Being Debugged!");
    }
    else
    {
        MessageBoxW(L"Not Being Debugged!");
    }
}
```

15. 硬件断点

硬件断点是通过设置名为 Dr0 到 Dr7 的调试寄存器来实现的。 Dr0-Dr3 包含至多 4 个断点的地址， Dr6 是个标志，它指示哪个断点被触发了， Dr7 包含了控制 4 个硬件断点诸如启用 / 禁用或者中断于读 / 写的标志。

由于调试寄存器无法在 Ring3 下访问，硬件断点的检测需要执行一小段代码。可以利用含有调试寄存器值的 CONTEXT 结构，该结构可以通过传递给异常处理例程的 ContextRecord 参数来访问。

使用方法：

```
static bool isDebuggedHBP = 0;
LONG WINAPI TopUnhandledExceptionFilterHBP(
    struct _EXCEPTION_POINTERS *ExceptionInfo
)
{
    _asm pushad
    //AfxMessageBox(" 回调函数被调用 ");
    ExceptionInfo->ContextRecord->Eip = NewEip;
    if (0 != ExceptionInfo->ContextRecord->Dr0 || 0 != ExceptionInfo->ContextRecord->Dr1 ||
        0 != ExceptionInfo->ContextRecord->Dr2 || 0 != ExceptionInfo->ContextRecord->Dr3)
        isDebuggedHBP = 1; // 检测有无硬件断点
    ExceptionInfo->ContextRecord->Dr0 = 0; // 禁用硬件断点，置 0
    ExceptionInfo->ContextRecord->Dr1 = 0;
    ExceptionInfo->ContextRecord->Dr2 = 0;
```

```
        ExceptionInfo->ContextRecord->Dr3 = 0;
        ExceptionInfo->ContextRecord->Dr6 = 0;
        ExceptionInfo->ContextRecord->Dr7 = 0;
        ExceptionInfo->ContextRecord->Eip = NewEip; // 转移到安全位置
        _asm popad
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    void CAntiDebugDlg::OnBnClickedBtnHdbreakpoint()
    {
        // TODO: 在此添加控件通知处理程序代码
        lpSetUnhandledExceptionFilter = (pSetUnhandledExceptionFilter)GetProcAddress(LoadLibrary(L"kern
el32.dll"),
            "SetUnhandledExceptionFilter");
        lpOldHandler = (DWORD)lpSetUnhandledExceptionFilter(TopUnhandledExceptionFilterHBP);
        _asm {
            mov   NewEip, offset safe // 方式二，更简单
            int     3
            mov   isDebuggedHBP, 1 // 调试时可能也不会触发异常去检测硬件断点
            safe:
        }
        if (1 == isDebuggedHBP) {
            MessageBoxW(L"Being Debugged!");
        }
        else {
            MessageBoxW(L"Not Being Debugged!");
        }
    }
```

16. 封锁键盘，鼠标输入

WINUSERAPI

```
BOOL
WINAPI
BlockInput(
    BOOL fBlockIt);
```

BlockInput 函数阻塞键盘及鼠标事件到达应用程序。该参数指明函数的目的。如果参数为 TRUE，则鼠标和键盘事件将被阻塞。如果参数为 FALSE， 则鼠标和键盘事件不被阻塞。

可以在代码中的关键位置调用此函数。

使用方法：

```
void CAntiDebugDlg::OnBnClickedBtnBlockinput()
{
    // TODO: 在此添加控件通知处理程序代码
    DWORD dwNoUse;
    DWORD dwNoUse2;
    ::BlockInput(TRUE);
    dwNoUse = 2;
    dwNoUse2 = 3;
    dwNoUse = dwNoUse2;
    ::BlockInput(FALSE);
}
```

17. 禁用窗口

与 BlockInput 函数的功能类似，用来禁用窗口

函数说明：

BOOL EnableWindow（HWND hWnd，BOOL bEnable）

hWnd：被允许 / 禁止的窗口句柄

bEnable: 定义窗口是被允许，还是被禁止。若该参数为 TRUE，则窗口被允许。若该参数为 FALSE，则窗口被禁止。

Windows API 函数。该函数允许 / 禁止指定的窗口或控件接受鼠标和键盘的输入，当输入被禁止时，窗口不响应鼠标和按键的输入，输入允许时，窗口接受所有的输入。

使用方法：

```
void CAntiDebugDlg::OnBnClickedBtnEnbalewindow()
{
    // TODO: 在此添加控件通知处理程序代码
    CWnd *wnd;
    wnd = GetForegroundWindow();
    wnd->EnableWindow(FALSE);
    DWORD dwNoUse;
    DWORD dwNoUse2;
    dwNoUse = 2;
    dwNoUse2 = 3;
    dwNoUse = dwNoUse2;
    wnd->EnableWindow(TRUE);
}
```

18.ThreadHideFromDebugger

NtSetInformationThread() 用来设置一个线程的相关信息。把 ThreadInformationClass 参数设为 ThreadHideFromDebugger(11H) 可以禁止线程产生调试事件。

函数声明：

```
NTSTATUS NTAPI NtSetInformationThread(
IN HANDLE ThreadHandle,
IN THREAD_INFORMATION_CLASS ThreadInformaitonClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
);
```

ThreadHideFromDebugger 内部设置内核结构 ETHREAD 的 HideThreadFromDebugger 成员。一旦这个成员设置以后，主要用来向调试器发送事件的内核函数 _DbgkpSendApiMessage() 将不再被调用

使用方法：

```
void CAntiDebugDlg::OnBnClickedBtnSetinforthread()
{
    HANDLE hwnd;
```

```
    HMODULE hModule;
    hwnd = GetCurrentThread();
    hModule = LoadLibrary(L"ntdll.dll");
    pfnZwSetInformationThread ZwSetInformationThread;
    ZwSetInformationThread = (pfnZwSetInformationThread)GetProcAddress(hModule,
"ZwSetInformationThread");
    ZwSetInformationThread(hwnd, ThreadHideFromDebugger, NULL, NULL);
}
```

19.OutputDebugString

OutputDebugString 函数用于向调试器发送一个格式化的字符串， Ollydbg 会在底端显示相应的信息。OllyDbg 存在格式化字符串溢出漏洞，非常严重，轻则崩溃，重则执行任意代码。这个漏洞是由于 Ollydbg 对传递给 kernel32!OutputDebugString() 的字符串参数过滤不严导致的，它只对参数进行那个长度检查，只接受 255 个字节，但没对参数进行检查，所以导致缓冲区溢出。

使用方法：

```
// 能够让 OD 崩溃
void CAntiDebugDlg::OnBnClickedBtnOutputdebugstring()
{
    // TODO: 在此添加控件通知处理程序代码
    ::OutputDebugString(L"%s%s%s");
}
```

20. 使用 TLS 回调

TLS 回调被用来在程序入口点执行之前运行代码，因此这些代码可以在调试器中秘密执行。TLS 是 windows 的一个存储类，其中数据对象不是一个自动的堆栈变量，而是代码中运行的每个线程的一个本地变量。大致而言，TLS 允许每个线程维护一个 TLS 声明的专有变量。在应用程序实现 TLS 的情况下，可执行程序的 PE 头部会包含一个 .tls 段。TLS 提供了初始化和终止 TLS 数据对象的回调函数。windows 系统在执行程序正常的入口点之前运行这些回调函数。

可以使用 PEview 查看应用程序的 .tls 段，可以发现 TLS 回调函数。通常情况下，正常程序不适用 .tls 段，如果看到了程序的 .tls 段，就可以怀疑它使用了反调试技术。

使用 IDA Pro 可以很容易的分析 TLS 回调函数。一旦 IDA Pro 完成对应用程序的分析，可以通过 Ctrl+E 看到二进制的入口点，该组合键的作用是显示应用程序的所有入口点，其中包含 TLS 回调。

《2017 绿盟科技恶意样本分析手册 - 反调试篇》
由如下部门撰写

- 绿盟科技安全能力中心（SAC）

如需了解更多，请联系：

官方网站

技术博客

微信公众号

特别声明

为避免客户数据泄露，所有数据在进行分析前都已经匿名化处理，不会在中间环节出现泄露，任何与客户有关的具体信息，均不会出现在本报告中。

版权声明

THE EXPERT
BEHIND GIANTS
巨人背后的专家

多年以来，绿盟科技致力于安全攻防的研究，
为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供
具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。
在这些巨人的背后，他们是备受信赖的专家。

www.nsfocus.com