

2017

绿盟科技

恶意样本分析手册

绿盟科技安全能力中心 (SAC)

[特殊方法篇]
(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

调试服务程序	1
服务简介	1
服务启动后附加	2
服务启动时附加	3
进程间通信	6
文件映射	7
共享内存	9
匿名管道	11
命名管道	12
邮槽 Mailslot	19
剪切板	22
socket	23
远程线程调试	24
定位 DllEntry	26
TLS 调试	29
windbg 调试 .NET	34
消息处理调试	41
消息机制说明	42
1. 什么是消息	42
2. 消息队列	43
3. 消息标识符	43
4. 消息的分类	43
5. 队列消息和非队列消息	44
调试示例	44

调试服务程序

服务简介

windows 服务是由三个组件构成的：服务应用，服务控制程序 SCP，以及服务控制管理器 SCM，当 SCM 启动一个服务进程时，该进程必须立即调用 `StartServiceCtrlDispatcher` 函数。`StartServiceCtrlDispatcher` 函数接受一个入口点列表，每个入口点对应于该进程中的一个服务。每个入口点是由它所对应的服务的名称来标识的。`StartServiceCtrlDispatcher` 创建了一个命名管道来跟 SCM 进行通信，在建立了该通信管道以后，它等待 SCM 通过该管道发送过来的命令。每次 SCM 启动一个属于该进程的服务时，它发送一个“服务启动”命令。`StartServiceCtrlDispatcher` 函数对于所接收到的每个启动命令，创建一个线程（服务线程），由该线程调用所启动服务的入口点函数，并实现该服务的命令循环。`StartServiceCtrlDispatcher` 一直在等待来自 SCM 的命令，只有当该进程的所有服务都停止时，它才会将控制返回至该进程的 `main` 函数，以便服务进程在退出以前做一些资源清理工作。

当 SCM 要启动一个服务时，它就调用 `ScStartService`，当 `ScStartService` 启动一个 windows 服务时，它首先读取该服务的注册表键中 `ImagePath` 值，以确定该服务进程的映像文件名。然后，它检查该服务的 `Type` 值，如果此值是 `SERVICE_WINDOWS_SHARE_PROCESS(0x20)` 那么，SCM 保证该服务运行所在的进程（如果已经启动了的话），其登录的账户一定与服务的指定启动账户相同。

SCM 在一个称为映像数据库的内部数据库中，检查是否有针对该服务的 `ImagePath` 值的条目，以便验证该服务的进程尚未在其他账户下被启动起来。如果在映像数据库中没有找到该 `ImagePath` 值的条目，则 SCM 创建一个这样的条目。当 SCM 创建一个新的条目时，它还将该账户的登录账户名，以及该服务的 `ImagePath` 值中的数据也存储起来。如果 SCM 在映像数据库中找到了一个与该服务的 `ImagePath` 数据相匹配的条目，那么，必须保证当前正在启动的服务的用户账户信息与数据库条目中存储的信息是相同的。一个进程只能以一个账户的身份来登录，所以，当一个服务指定的账户名与同一个进程中已经启动起来的其他服务的账户名不相同时，SCM 会报告一个错误。

SCM 调用 `ScLogonAndStartImage` 来登录一个服务，并启动该服务的进程。SCM 通过调用 `lsass` 函数 `LogonUserEx` 来登录那些并非运行在系统账户下的服务。

当 SCM 配置一个服务的登录信息时，SCM 利用 `LsaStorePrivateData` 函数来指示 `lsass` 将一个登录口令保存到 `Secrets` 子键下。在登陆成功后，`LogonUserEx` 给调用者返回一个句柄，指向一个访问令牌。Windows 使用访问令牌来代表一个用户的安全环境，以后 SCM 将该访问令牌与实现此服务的进程关联起来。

下一个步骤是，如果该服务的进程尚未被启动（例如，为了另一个服务），则 `ScLogonAndStartImage` 为该服务激发一个进程。SCM 通过 windows 函数 `CreateProcessAsUser` 来启动此进程，并且将该进程的状态设置为挂起状态。接下来 SCM 创建一个命名管道，以后通过该管道与服务进程进行通信，分配给管道的名称为 `\Pipe\Net\NtControlPipeX`，其中 X 是一个数字，每次 SCM 创建一个管道，该数字就会递增。然后，SCM 通过 `ResumeThread` 函数来恢复服务进程的执行，并且等待该服务连接到它的 SCM 管道上。如果注册表值 `HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout` 存在的话，则它决定了 SCM 等待又给服务调用 `StartServiceCtrlDispatcher` 并连接过来的时间长度，如果在这么长时间里没有等到，则 SCM 就会放弃，终止该进程，并得出结论：该服务未能启动。如果 `ServicesPipeTimeout` 不存在，则 SCM 使用默认的 30 秒作为超时时间间隔值。SCM 对于它所有的服务通信都是用同样的超时时间间隔值。

当一个服务通过它的管道连接到 SCM 时，SCM 向该服务发送一个启动命令。如果该服务未能在有效时间内



响应启动命令，SCM 就会放弃，转移到启动下一个服务。当一个服务没有对启动请求做出响应时，SCM 不会像一个服务在指定时间内没有调用 StartServiceCtrlDispatcher 的情形一样终止进程，相反会在系统的时间日志中记录一个错误，指明该服务未能及时启动起来。

如果 SCM 调用 ScStartService 启动的服务有一个 Type 注册表值为 SERVICE_KERNEL_DRIVER 或 SERVICE_FILE_SYSTEM_DRIVER，那么该服务是一个设备驱动程序，所以 ScStartService 调用 ScLoadDeviceDriver 来加载该驱动程序。ScLoadDeviceDriver 首先使 SCM 进程具有加载驱动程序的安全特权，然后调用内核服务 NtLoadDriver，将该驱动程序的注册表键中的 ImagePath 值的数据传递过去。与 windows 服务不同的是，驱动程序不需要指定 ImagePath 值；如果该值不存在的话，SCM 通过将驱动程序的名称附加在字符串 %SystemRoot%\System32\Drivers\ 的后面就可以构造一个映像文件路径。

ScAutostartServices 继续循环处理同一个组的服务，直到所有这些服务要么被启动起来，要么产生相依性错误。这种循环处理方式是 SCM 根据一个组中的服务的 DependOnService 相依性来自动对他们进行顺序处理的。SCM 在较早的循环中启动那些被其他服务依赖的服务，跳过那些依赖于其他服务的服务，而在后续的循环中再启动这些服务。

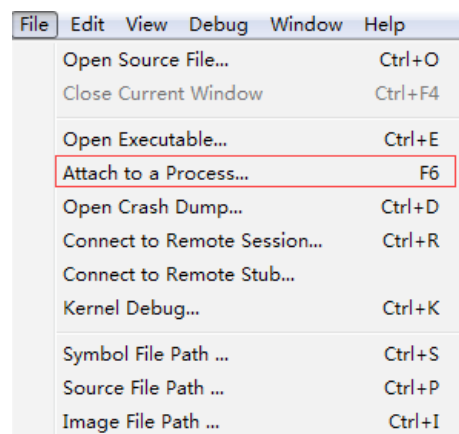
SCM 在处理了自动启动的服务以后，调用 ScInitDelayStart，该函数将一个延迟的工作项目加入队列中，该工作项目与一个专门的辅助线程相关联，它负责处理所有由于被标记为“延迟的自动启动”而被 ScAutoStartServices 忽略掉的服务。此辅助线程将在一段时间的延迟之后执行，默认的延迟是 120 秒，但那时可以通过在 HTML\SYSTEM\CurrentControlSet\Control 中创建一个 AutoStartDelay 值，可以覆盖这一默认值。

当 SCM 完成了启动所有这些自动启动的服务和驱动程序，以及设置了延迟的自动启动的工作项目时，发信号通过 \BaseNameObjects\SC_AutostartComplete 事件完成。这一事件被 windows setup 程序用于在安装过程中衡量启动过程。

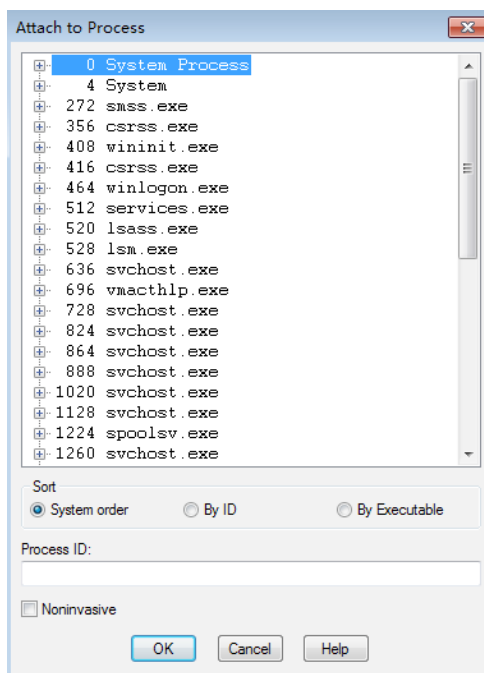
有些恶意样本会通过创建系统服务来执行自身的恶意功能，通过这种方法可以实现开机启动，也可以达到隐藏自身的目的，对于安全人员来说，调试服务程序也比较麻烦，这里介绍两种调试的方法。

服务启动后附加

通过任务管理器找到要调试的服务程序，查看目标进程的 pid，然后再启动 windbg，选择 file->Attach to a Process



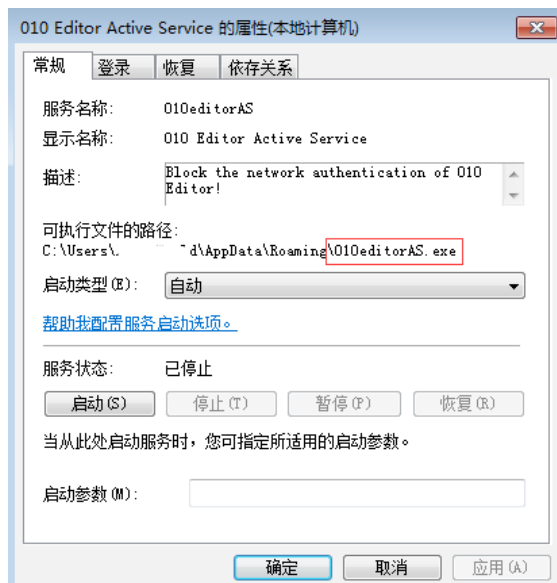
在显示的进程列表中找到目标程序，然后点击 OK 即可开始调试。（关于此工具的详细使用方法，请参考工具篇介绍）。



服务启动时附加

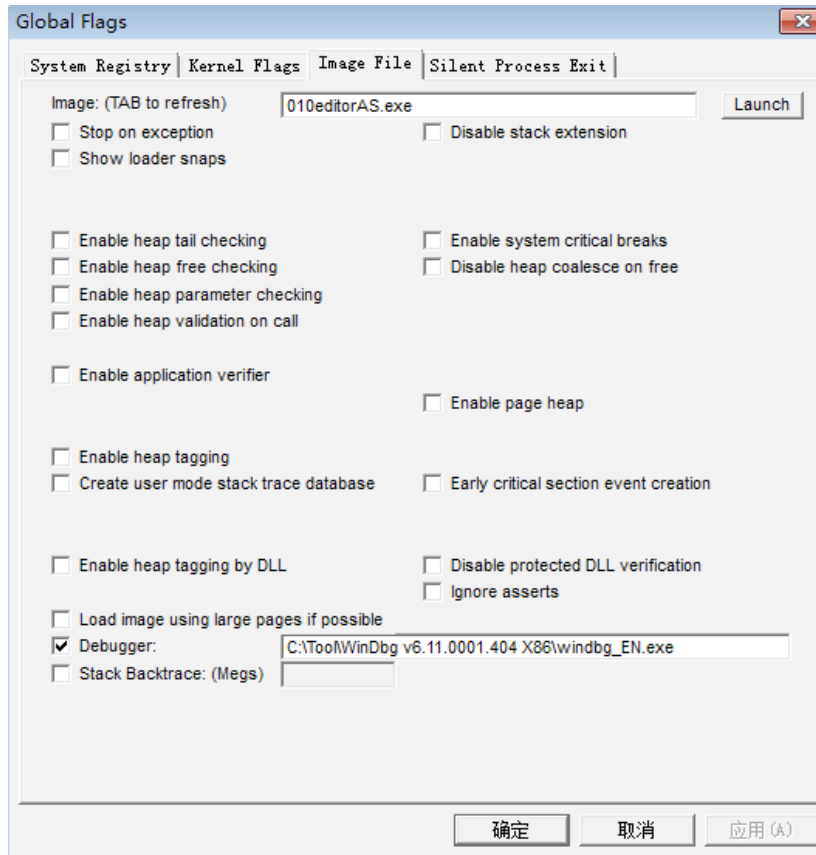
在服务启动的时候就使用 windbg 进行附加，使用这种方法需要进行简单的配置。

1. 使用管理员启动 windbg 目录下的 gflag.exe，在“Image File”标签页中填写 Image 名字（不需要写全路径）。如下图所示，服务名称是 010editorAS，可执行路径为 C:\Users\xxxxx\AppData\Roaming\010editorAS.exe。





按照上图的例子，在 Image 框中填写 010editorAS.exe，在下面的 Debugger 框中填写 windbg 的全路径。



2. 设置服务的属性，允许服务以交互形式启动。



3. 设置结束后，就可以启动服务了，系统会显示交互式服务检测，点击查看消息后，便会中断在 windbg 中。



但是这种方法有个问题，就是只能断下来 30 秒，过了这个时间就无法进行调试了。上文中提到了这个问题，查找网上的资料说，在注册表 HKLM\SYSTEM\CurrentControlSet\Control 下新建一个值 ServicesPipeTimeout，这个值表示超时时间，但是经过实践发现并没有什么效果。

另外一种方法是在服务的入口加 int3 断点，这样就能在超时间隔内使程序执行完 StartServiceCtrlDispatcher 函数，不至于使服务管理器因为长时间收不到来自服务的消息而终止服务的运行。

进程间通信

随着人们对应用程序的要求越来越高，单进程应用在许多场合已不能满足人们的要求，编写多进程 / 多线程程序成为现代程序设计的一个重要特点，恶意软件也跟随时代的脚步，通过多进程来相互保护。而多进程之间想“交流”，那么肯定少不了进程间通信的技术，如果我们熟悉进程间通信技术，就可以轻松自如的调试多进程的恶意样本。

文件映射

文件映射 (Memory-Mapped Files) 能使进程把文件内容当作进程地址区间一块内存那样来对待。因此，进程不必使用文件 I/O 操作，只需简单的指针操作就可读取和修改文件的内容。

Win32 API 允许多个进程访问同一文件映射对象，各个进程在它自己的地址空间里接收内存的指针。通过使用这些指针，不同进程就可以读或修改文件的内容，实现了对文件中数据的共享。

应用程序有三种方法来使多个进程共享一个文件映射对象。

1. **继承**：第一个进程建立文件映射对象，其子进程继承该对象的句柄。
2. **命名文件映射**：第一个进程在建立文件映射对象时可以给该对象指定一个名字 (可与文件名不同)。第二个进程可通过这个名字打开此文件映射对象。另外，第一个进程也可以通过一些其它 IPC 机制 (有名管道、邮件槽等) 把名字传给第二个进程。
3. **句柄复制**：第一个进程建立文件映射对象，然后通过其它 IPC 机制 (有名管道、邮件槽等) 把对象句柄传递给第二个进程。第二个进程复制该句柄就取得对该文件映射对象的访问权限。

文件映射是在多个进程间共享数据的非常有效方法，有较好的安全性。但文件映射只能用于本地机器的进程之间，不能用于网络中，而开发者还必须控制进程间的同步。

下面的示例代码展示了如何通过文件来进行通信：

发送方：

```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
int main()
{
    HANDLE hFile = CreateFile(TEXT("c:\\zj.dat"), GENERIC_READ | GENERIC_WRITE,
        0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == NULL)
    {
        printf("create file error!");
        return 0;
    }
    // HANDLE hFile = (HANDLE)0xffffffff; // 创建一个进程间共享的对象
    HANDLE hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 1024 * 1024, TEXT("ZJ"));
    int rst = GetLastError();
    if (hMap != NULL && rst == ERROR_ALREADY_EXISTS)
```



```
{
    printf("hMap error\n");
    CloseHandle(hMap);
    hMap = NULL;
    return 0;
}
CHAR* pszText = NULL;
pszText = (CHAR*)MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 1024 * 1024);
if (pszText == NULL)
{
    printf("view map error!");
    return 0;
}
//printf(" 写入数据 :");
sprintf(pszText, "hello my first mapping file!\n"); // 其实是向文件中（共享内存中）写入了
printf(" 写入的数据: %s", pszText);
while (1)
{
    printf(" 读取数据: ");
    printf(pszText);
    Sleep(3000);
}
getchar();
UnmapViewOfFile((LPCVOID)pszText);
CloseHandle(hMap);
CloseHandle(hFile);
return 0;
}
```

接收方:

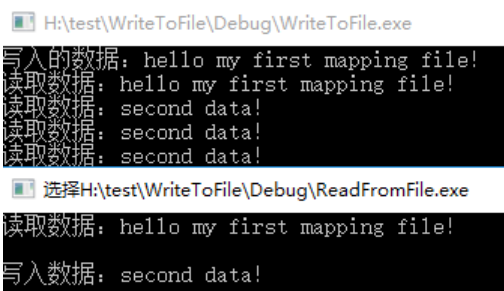
```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
int main()
{
    HANDLE hMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, TRUE, TEXT("ZJ"));
    if (hMap == NULL)
    {
        printf("open file map error!");
        return 0;
    }
    CHAR* pszText = NULL;
    pszText = (CHAR*)MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 1024 * 1024);
```

```

if (pszText == NULL)
{
    printf("map view error!\n");
    return 0;
}
printf(" 读取数据: ");
printf(pszText); // 从文件中读 ( 共享内存 )
printf("\n 写入数据: ");
sprintf(pszText, "second data!\n"); // 写入
printf("%s\r\n", pszText);
getchar();
UnmapViewOfFile(pszText);
CloseHandle(hMap);
hMap = NULL;
return 0;
}

```

首先发送方创建一个文件并进行映射，然后向映射的内存中写入数据，接着再从映射的内存中读取数据，接下来运行接收方，接收方先是读取数据，然后在往里面写入数据，所以，运行之后会看到发送方读取的数据变化了。



```

H:\test\WriteToFile\Debug\WriteToFile.exe
写入的数据: hello my first mapping file!
读取数据: hello my first mapping file!
读取数据: second data!
读取数据: second data!
读取数据: second data!

选择H:\test\WriteToFile\Debug\ReadFromFile.exe
读取数据: hello my first mapping file!
写入数据: second data!

```

使用这种方法进行通信用到的函数主要有 CreateFile, CreateFileMapping, MapViewOfFile, sprintf。也有一种情况是使用函数 WriteFile 向文件种写入数据。

通过以上代码可以知道这种机制的通信方式，然后在上述的函数上设置断点，截获程序交互的数据即可。

共享内存

共享内存主要是通过映射机制实现的。

Windows 下进程的地址空间在逻辑上是相互隔离的，但是在物理上却是重叠的。所谓的重叠是指同一块内存区域可能被多个进程同时使用。当调用 CreateFileMapping 创建命名的内存映射文件对象时，Windows 即在物理内存申请一块指定大小的内存区域，返回文件映射对象的句柄 hMap。为了能够访问这块内存区域必须调用 MapViewOfFile 函数，促使 Windows 将此内存空间映射到进程的地址空间中。当在其他进程访问这块内存区域时，则必须使用 OpenFileMapping 函数取得对象句柄 hMap，并调用 MapViewOfFile 函数得到此内存空间的一个映射。这样一来，系统就把同一块内存区域映射到了不同进程的地址空间中，从而达到共享内存的目的。

以下面的代码为例，第一次运行时，创建一个共享内存，写入数据“HelloWorld”，只要创建共享内存的进

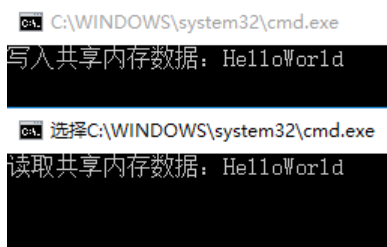


程没有关闭句柄 hMap，以后运行的程序就会读出共享内存里面的数据，并打印出来。

知道了共享内存的手法，就可以在关键函数处下断点进行调试了。

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;
int main()
{
    wchar_t strMapName[] = L"ShareMemory";           // 内存映射对象名称
    string strComData("HelloWorld");                // 共享内存中的数据
    LPVOID pBuffer;                                  // 共享内存指针
    // 首先试图打开一个命名的内存映射文件对象
    HANDLE hMap = ::OpenFileMapping(FILE_MAP_ALL_ACCESS, 0, (LPCWSTR)strMapName);
    if (NULL == hMap)
    { // 打开失败，则创建
        hMap = ::CreateFileMapping(INVALID_HANDLE_VALUE,
            NULL,
            PAGE_READWRITE,
            0,
            strComData.length() + 1,
            strMapName);
        // 映射对象的一个视图，得到指向共享内存的指针，设置里面的数据
        pBuffer = ::MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
        strcpy((char*)pBuffer, strComData.c_str());
        cout << " 写入共享内存数据: " << (char *)pBuffer << endl;
    }
    else
    { // 打开成功，映射对象的一个视图，得到指向共享内存的指针，显示出里面的数据
        pBuffer = ::MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
        cout << " 读取共享内存数据: " << (char *)pBuffer << endl;
    }
    getchar(); // 注意，进程关闭后，所有句柄自动关闭，所以要在这里暂停
    // 解除文件映射，关闭内存映射文件对象句柄
    ::UnmapViewOfFile(pBuffer);
    ::CloseHandle(hMap);
    system("pause");
    return 0;
}
```

连续运行两次程序，结果如下图所示：



这种方式 and 上述介绍的第一种方式类似，用到的函数有 `OpenFileMapping`，`CreateFileMapping`，`MapViewOfFile`。在调用 `MapViewOfFile` 后，会返回一个地址，进程交互的数据就是通过这个地址，所以在 `MapViewOfFile` 函数处设置断点，然后在内存中跟踪返回地址中的数据即可。

匿名管道

匿名管道是一种未命名的、单向管道，通常用来在一个父进程和一个子进程之间传输数据。匿名的管道只能实现本地机器上两个进程间的通信，而不能实现跨网络的通信。

匿名管道是在父进程和子进程之间，或同一父进程的两个子进程之间传输数据的无名字的单向管道。通常由父进程创建管道，然后由要通信的子进程继承通道的读端点句柄或写端点句柄，然后实现通信。父进程还可以建立两个或更多个继承匿名管道读和写句柄的子进程。这些子进程可以使用管道直接通信，不需要通过父进程。

匿名管道是单机上实现子进程标准 I/O 重定向的有效方法，它不能在网上使用，也不能用于两个不相关的进程之间。

创建匿名管道的函数为 `CreatePipe`，对管道进行读写的操作是 `ReadFile` 和 `WriteFile` 函数。

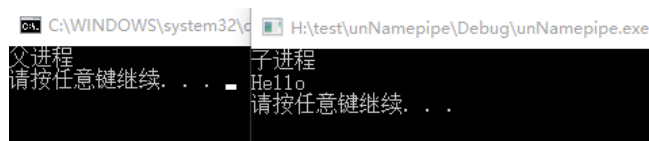
参考代码如下：

```
#include "stdafx.h"
#include <Windows.h>
#include <iostream>
int main()
{
    auto numArgs = 0;
    CommandLineToArgvW(GetCommandLineW(), &numArgs);
    if (numArgs > 1)
    {
        std::cout << " 子进程 " << std::endl;
        CHAR szBuffer[16]{ 0 };
        ReadFile(GetStdHandle(STD_INPUT_HANDLE), szBuffer, sizeof(szBuffer), nullptr, nullptr);
        std::cout << szBuffer << std::endl;
    }
    else
    {
        std::cout << " 父进程 " << std::endl;
        SECURITY_ATTRIBUTES sa{ 0 };
    }
}
```



```
sa.nLength = sizeof(sa);
sa.bInheritHandle = TRUE;
HANDLE hRead;
HANDLE hWrite;
CreatePipe(&hRead, &hWrite, &sa, 0);
STARTUPINFO si{ 0 };
si.cb = sizeof(si);
si.hStdInput = hRead;
si.dwFlags = STARTF_USESTDHANDLES;
PROCESS_INFORMATION pi{ 0 };
WCHAR szCommand[512]{ 0 };
GetModuleFileNameW(nullptr, szCommand, _countof(szCommand));
wcscat(szCommand, L" test");
CreateProcessW(nullptr, szCommand, nullptr, nullptr, TRUE, CREATE_NEW_CONSOLE, nullptr, nullptr,
&si, &pi);
WriteFile(hWrite, "HelloWorld", 5, nullptr, nullptr);
}
system("pause");
return 0;
}
```

运行结果如下：



在父进程创建一个匿名管道，将管道的读句柄传给子进程，并且向管道中写入“HelloWorld”，子进程启动后，通过管道的读句柄从管道中读取父进程写入的数据，这就是简单的匿名管道通信机制，调试的时候，如果看到类似的机制，可以在 WriteFile 或者 ReadFile 上设置断点来读取程序写入管道的值。

在调试的时候，遇到函数 CreatePipe，需要重点关注它的前两个参数，这两个参数分别为数据读取句柄和数据写入句柄，记下这两个值之后，在遇到 WriteFile 或者 ReadFile 函数时，如果句柄值为 CreatePipe 函数返回的值，那么就需要截获写入或读取的数据。

命名管道

命名管道 (Named Pipe) 是服务器进程和一个或多个客户进程之间通信的单向或双向管道。不同于匿名管道的是命名管道可以在不相关的进程之间和不同计算机之间使用，服务器建立命名管道时给它指定一个名字，任何进程都可以通过该名字打开管道的另一端，根据给定的权限和服务器进程通信。

命名管道提供了相对简单的编程接口，使通过网络传输数据并不比同一计算机上两进程之间通信更困难，不过如果要同时和多个进程通信就力不从心了。

创建匿名管道的函数为 CreateNamedPipe，通过 WriteFile 或 ReadFile 向这个管道内写入数据或读取数据，

此时的客户端用 CreateFile 先打开此命名管道, 通过 ReadFile 读取管道内的数据或用 WriteFile 向管道内写入数据。

参考代码 (包含服务端和客户端) :

服务端:

```
#define READ_PIPE L"\\\\.\\pipe\\ReadPipe"
#define WRITE_PIPE L"\\\\.\\pipe\\WritePipe"
typedef struct _USER_CONTEXT_
{
    HANDLE hPipe;
    HANDLE hEvent;
}USER_CONTEXT,*PUSER_CONTEXT;
USER_CONTEXT Context[2] = {0};
HANDLE hThread[2] = {0};
BOOL WritePipe();
BOOL ReadPipe();
BOOL bOk = FALSE;
DWORD WINAPI WritePipeThread(LPVOID LPParam);
DWORD WINAPI ReadPipeThread(LPVOID LPParam);
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    int nRetCode = 0;
    HANDLE hPipe = NULL;
    if (WritePipe()==FALSE)
    {
        return -1;
    }
    if (ReadPipe()==FALSE)
    {
        return -1;
    }
    int iIndex = 0;
    while (TRUE)
    {
        if (bOk==TRUE)
        {
            SetEvent(Context[0].hEvent);
            SetEvent(Context[1].hEvent);
            Sleep(1);
        }
        iIndex = WaitForMultipleObjects(2,hThread,TRUE,5000);
        if (iIndex==WAIT_TIMEOUT)
        {
```



```
        continue;
    }
    else
    {
        break;
    }
}
int i = 0;
for (i=0;i<2;i++)
{
    CloseHandle(Context[i].hEvent);
    CloseHandle(Context[i].hPipe);
}
CloseHandle(hThread[0]);
CloseHandle(hThread[1]);
cout<<"Exit"<<endl;
return nRetCode;
}
// 创建写入数据的管道
BOOL WritePipe()
{
    HANDLE hWritePipe = NULL;
    hWritePipe = CreateNamedPipe(
        WRITE_PIPE,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_MESSAGE |
        PIPE_READMODE_MESSAGE |
        PIPE_WAIT,
        PIPE_UNLIMITED_INSTANCES,
        MAX_PATH,
        MAX_PATH,
        0,
        NULL);
    if (hWritePipe==INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }
    HANDLE hEvent = CreateEvent(NULL,FALSE,FALSE,NULL);
    Context[0].hEvent = hEvent;
    Context[0].hPipe = hWritePipe;
    hThread[0] = CreateThread(NULL,0,WritePipeThread,NULL,0,NULL);
    return TRUE;
}
```

```

}
// 创建读取数据的管道
BOOL ReadPipe()
{
    HANDLE hReadPipe = NULL;
    hReadPipe = CreateNamedPipe(
        READ_PIPE,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_MESSAGE |
        PIPE_READMODE_MESSAGE |
        PIPE_WAIT,
        PIPE_UNLIMITED_INSTANCES,
        MAX_PATH,
        MAX_PATH,
        0,
        NULL);
    if (hReadPipe==INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }
    HANDLE hEvent = CreateEvent(NULL,FALSE,FALSE,NULL);
    Context[1].hEvent = hEvent;
    Context[1].hPipe = hReadPipe;
    hThread[1] = CreateThread(NULL,0,ReadPipeThread,NULL,0,NULL);
    return TRUE;
}
DWORD WINAPI ReadPipeThread(LPVOID LPParam)
{
    HANDLE hEvent = Context[1].hEvent;
    HANDLE hReadPipe = Context[1].hPipe;
    DWORD dwReturn = 0;
    char szBuffer[MAX_PATH] = {0};
    int iIndex = 0;
    while (TRUE)
    {
        iIndex = WaitForSingleObject(hEvent,30);
        iIndex = iIndex-WAIT_OBJECT_0;
        if (iIndex==WAIT_FAILED||iIndex==0)
        {
            break;
        }
        if (ReadFile(hReadPipe,szBuffer,MAX_PATH,&dwReturn,NULL))
    }
}

```



```
{
    szBuffer[dwReturn] = '\0';
    cout<<szBuffer<<endl;
}
else
{
    if (GetLastError()==ERROR_INVALID_HANDLE)
    {
        break;
    }
}
}
return 0;
}
DWORD WINAPI WritePipeThread(LPVOID LPParam)
{
    HANDLE hEvent  = Context[0].hEvent;
    HANDLE hWritePipe = Context[0].hPipe;
    DWORD dwReturn = 0;
    char szBuffer[MAX_PATH] = {0};
    int iIndex = 0;
    while (TRUE)
    {
        iIndex = WaitForSingleObject(hEvent,30);
        iIndex = iIndex-WAIT_OBJECT_0;
        if (iIndex==WAIT_FAILED||iIndex==0)
        {
            break;
        }
        cin>>szBuffer;
        if (WriteFile(hWritePipe,szBuffer,strlen(szBuffer),&dwReturn,NULL))
        {
        }
        //Error
        else
        {
            if (GetLastError()==ERROR_INVALID_HANDLE)
            {
                break;
            }
        }
    }
}
```

```

        return 0;
    }
}

客户端：
#define WRITE_PIPE L"\\\\.\\pipe\\ReadPipe"    // 命名管道的格式
#define READ_PIPE L"\\\\.\\pipe\\WritePipe"
HANDLE hThread[2] = {0};
DWORD WINAPI ReadPipeThread(LPARAM LPParam);
DWORD WINAPI WritePipeThread(LPARAM LPParam);
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    HANDLE hReadPipe = NULL;
    HANDLE hWritePipe = NULL;
    hThread[0] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ReadPipeThread,NULL,0,NULL);
    hThread[1] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)WritePipeThread,NULL,0,NULL);
    WaitForMultipleObjects(2,hThread,TRUE,INFINITE);
    CloseHandle(hReadPipe);
    CloseHandle(hWritePipe);
    CloseHandle(hThread[0]);
    CloseHandle(hThread[1]);
    cout<<"Exit"<<endl;
    return -1;
}

DWORD WINAPI WritePipeThread(LPARAM LPParam)
{
    HANDLE hWritePipe = NULL;
    char szBuffer[MAX_PATH] = {0};
    DWORD dwReturn = 0;
    // 打开管道
    while(TRUE)
    {
        hWritePipe = CreateFile(WRITE_PIPE,GENERIC_READ | GENERIC_WRITE,
            FILE_SHARE_READ | FILE_SHARE_WRITE,
            NULL,OPEN_EXISTING,0,NULL);
        if (hWritePipe==INVALID_HANDLE_VALUE)
        {
            continue;
        }
        break;
    }
    // 向管道里写入数据
    while (TRUE)
    {

```



```
        cin>>szBuffer;
        if (WriteFile(hWritePipe,szBuffer,MAX_PATH,&dwReturn,NULL))
        {
        }
        else
        {
            if (GetLastError()==ERROR_NO_DATA)
            {
                cout<<"Write Failed"<<endl;
                break;
            }
        }
    }
    return 0;
}

DWORD WINAPI ReadPipeThread(LPARAM LPParam)
{
    HANDLE hReadPipe = NULL;
    char szBuffer[MAX_PATH] = {0};
    DWORD dwReturn = 0;
    // 打开管道
    while(TRUE)
    {
        hReadPipe = CreateFile(READ_PIPE,GENERIC_READ | GENERIC_WRITE,
            FILE_SHARE_READ | FILE_SHARE_WRITE,
            NULL,OPEN_EXISTING,0,NULL);
        if (hReadPipe==INVALID_HANDLE_VALUE)
        {
            continue;
        }
        break;
    }
    // 读取数据
    while (TRUE)
    {
        if (ReadFile(hReadPipe,szBuffer,MAX_PATH,&dwReturn,NULL))
        {
            szBuffer[dwReturn] = '\0';
            cout<<szBuffer;
        }
        else
        {

```

```

        cout<<"Read Failed"<<endl;
        break;
    }
}
return 0;
}

```

命名管道的创建函数为 `CreateNamedPipe`，这个函数会返回一个句柄，进行数据传输时，就调用 `WriteFile` 或者 `ReadFile` 向这个句柄中写入或读取数据。原理基本和匿名管道一致，不再进行赘述。

邮槽 Mailslot

邮件槽 (Mailslots) 提供进程间单向通信能力，任何进程都能建立邮件槽成为邮件槽服务器。其它进程，称为邮件槽客户，可以通过邮件槽的名字给邮件槽服务器进程发送消息。进来的消息一直放在邮件槽中，直到服务器进程读取它为止。一个进程既可以是邮件槽服务器也可以是邮件槽客户，因此可建立多个邮件槽实现进程间的双向通信。

邮件槽与命名管道相似，不过它传输数据是通过不可靠的数据报 (如 TCP/IP 协议中的 UDP 包) 完成的，一旦网络发生错误则无法保证消息正确地接收，而命名管道传输数据则是建立在可靠连接基础上的。不过邮件槽有简化的编程接口和给指定网络区域内的所有计算机广播消息的能力，所以邮件槽不失为应用程序发送和接收消息的另一种选择。

创建邮槽的函数为 `CreateMailslot`，使用这个函数可以创建一个命名的邮槽，名字格式为 `\\.\mailslot\Hello`。

下面通过程序示例代码进行原理说明：

服务端：

```

#include "stdafx.h"
#include<Windows.h>
#include<stdio.h>
HANDLE hSlot;
char lpszSlotName[] = "\\.\mailslot\sample_mailslot";
int main()
{
    DWORD cbMessage, cMessage, cbRead, cAllMessages;
    BOOL bResult;
    char lpszBuffer[1000] = {0};
    char achID[80];
    cbMessage = cMessage = cbRead = 0;
    hSlot = CreateMailslotA(
        lpszSlotName, // mailslot 名
        0, // 不限制消息大小
        MAILSLOT_WAIT_FOREVER, // 无超时
        NULL);
    if(hSlot == INVALID_HANDLE_VALUE)

```



```
{
    printf("CreateMailslot failed with %d\n", GetLastError());
    return 0;
}
else
    printf("Mailslot created successfully.\n");
while (1)
{
    // 获取 mailslot 信息
    bResult = GetMailslotInfo(hSlot, // mailslot 句柄
        (LPDWORD)NULL, // 无最大消息限制
        &cbMessage, // 下一条消息的大小
        &cMessage, // 消息的数量
        (LPDWORD)NULL); // 无时限
    if(!bResult)
    {
        printf("GetMailslotInfo failed with %d.\n", GetLastError());
        return 0;
    }
    printf("MAILSLOT_NO_MESSAGE\n");
    if(cbMessage == MAILSLOT_NO_MESSAGE)
    {
        // 没有消息，过一段时间再去读
        Sleep(3000);
        continue;
    }
    printf("has MAILSLOT_NO_MESSAGE\n");
    cAllMessages = cMessage;
    while(cMessage != 0) // 获取全部消息，有可能不只一条
    {
        // 提示信息
        sprintf(achID, "\nMessage #%d of %d\n", cAllMessages- cMessage + 1, cAllMessages);
        // 读取消息
        if (!ReadFile(hSlot, lpszBuffer, cbMessage, &cbRead, NULL))
        {
            printf("ReadFile failed with %d.\n", GetLastError());
            return 0;
        }
        //lstrcat(lpszBuffer, achID); // 连接
        printf("Contents of the mailslot:%s\n", lpszBuffer); // 显示
        bResult = GetMailslotInfo(hSlot, NULL, &cbMessage, &cMessage, NULL); // 计算剩余的消息数，若
        cMessage=0, 则退出子循环
        if(!bResult)
```



```

        {
            printf("GetMailslotInfo failed (%d)\n", GetLastError());
            return 0;
        }
    }
}
return 0;
}

```

客户端：

```

#include "stdafx.h"
#include<Windows.h>
#include<stdio.h>
char lpszSlotName[] = "\\.\mailslot\sample_mailslot"; //Mailslot 名
char lpszMessage[] = "HelloWorld"; // 通信的内容
int main()
{
    DWORD cbWritten;
    DWORD cbMessage;
    // 打开 mailslot
    HANDLE hFile = CreateFileA(lpszSlotName,
        GENERIC_WRITE, // 可写
        FILE_SHARE_READ,
        (LPSECURITY_ATTRIBUTES)NULL,
        OPEN_EXISTING, // 打开一个已经存在的 mailslot, 应该由服务端已经创建
        FILE_ATTRIBUTE_NORMAL,
        (HANDLE)NULL);
    if(hFile == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with %d.\n", GetLastError());
        return 0;
    }
    // 向 mailslot 写入
    int num = 50;
    while (num)
    {
        Sleep(2000);
        num--;
        if (!WriteFile(hFile,
            lpszMessage,
            (DWORD)(strlen(lpszMessage) + 1),
            &cbWritten,
            (LPOVERLAPPED)NULL))

```



```
{  
    printf("WriteFile failed with %d.\n", GetLastError());  
    return 0;  
}  
printf("Slot written to successfully.\n");  
}  
CloseHandle(hFile); // 结束  
return 0;  
}
```

运行结果如下：

```
H:\test\MyMailslot\Debug\MyMailslot.exe  
Mailslot created successfully.  
MAILSLOT_NO_MESSAGE  
MAILSLOT_NO_MESSAGE  
MAILSLOT_NO_MESSAGE  
has MAILSLOT_NO_MESSAGE  
Contents of the mailslot:HelloWorld  
Contents of the mailslot:HelloWorld  
MAILSLOT_NO_MESSAGE  
MAILSLOT_NO_MESSAGE  
has MAILSLOT_NO_MESSAGE  
Contents of the mailslot:HelloWorld  
MAILSLOT_NO_MESSAGE  
server  
H:\test\MyMailslot\Debug\MyMailslotClient.exe  
Slot written to successfully.  
Slot written to successfully.  
Slot written to successfully.  
Slot written to successfully.  
client
```

代码说明: 服务端使用函数 `CreateMailslotA` 创建一个邮槽, 然后使用函数 `GetMailslotInfo` 获取 mailslot 消息, 如果有消息的话使用 `ReadFile` 函数读取邮槽中的数据。客户端通过邮槽的名字使用函数 `CreateFileA` 来打开邮槽, 直接调用 `WriteFile` 函数向邮槽中写入数据, 邮槽的命名格式为 `\\\\.\\mailslot\\sample_mailslot`。

调试的时候, 如果看到 `CreateMailslotA`, 需要注意它的返回值—邮槽的句柄, 后期程序会通过这个句柄来向邮槽中写入或读取数据, 所以服务端需要注意的函数有 `CreateMailslotA`, `GetMailslotInfo` 和 `ReadFile`, 客户端需要注意的函数有 `CreateFileA`, `WriteFile`。

剪切板

剪贴板 (Clipped Board) 实质是 Win32 API 中一组用来传输数据的函数和消息, 为 Windows 应用程序之间进行数据共享提供了一个中介, Windows 已建立的剪切 (复制) — 粘贴的机制为不同应用程序之间共享不同格式数据提供了一条捷径。当用户在应用程序中执行剪切或复制操作时, 应用程序把选取的数据用一种或多种格式放在剪贴板上。然后任何其它应用程序都可以从剪贴板上拾取数据, 从给定格式中选择适合自己的格式。

使用剪切板进行通信的过程大致如下:

发送端:

首先 `OpenClipboard` 打开剪切板, 打开剪切板后其他应用程序不能修改剪切板, 直到调用了 `CloseClipboard`。然后调用函数 `EmptyClipboard` 清空剪切板并释放剪切板中的句柄, 将使用权给当前打开剪切板的窗口。接着使用 `SetClipboardData` 向剪切板中放置数据。

参考代码:

```

if(OpenClipboard()) // 打开剪切板
{
    CString str;
    HANDLE hClip;
    TCHAR *pBuf;
    EmptyClipboard(); // 清空剪切板
    GetDlgItemText(IDC_EDIT_SEND,str); // 读取编辑框中的内容
    hClip=GlobalAlloc(GMEM_MOVEABLE,str.GetLength()*2+2); // 为数据分配内存
    pBuf=(TCHAR *)GlobalLock(hClip); // 锁定内存, 并将返回的句柄转换为指针
    //str2=str1;
    _tcscpy(pBuf,(TCHAR *) (LPCTSTR)str);
    GlobalUnlock(hClip); // 解除内存锁定
    SetClipboardData(CF_TEXT,hClip); // 将指定格式的数据写入剪切板
    CloseClipboard(); // 关闭剪切板否则其它进程打不开剪切板
}

```

接收端：

打开剪切板, 这时不再需要清空剪切板内容了, 调用函数 `GetClipboardData` 获取剪切板的数据。参考代码：

```

if(OpenClipboard())
{
    if(IsClipboardFormatAvailable(CF_TEXT)) // 判断剪切板中数据是否是想要格式的数据
    {
        HANDLE hClip;
        TCHAR *pBuf;
        hClip=GetClipboardData(CF_TEXT);
        pBuf=(TCHAR *)GlobalLock(hClip);
        GlobalUnlock(hClip);
        SetDlgItemText(IDC_EDIT_RECV,pBuf);
        CloseClipboard();
    }
}

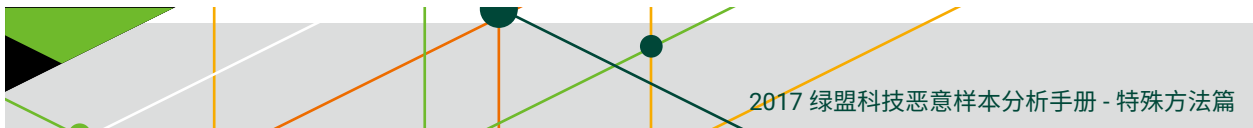
```

从示例代码中可以知道, 需要重点关注的函数有: `OpenClipboard`, `CloseClipboard`, `EmptyClipboard`, `SetClipboardData`, `GetClipboardData`。其中 `SetClipboardData` 用来往剪切板里设置数据, `GetClipboardData` 用来获取剪切板中的数据, 所以在这两个函数上设置断点即可截获进程间传输的数据。

socket

关于 socket 通信大家应该都不陌生, 我们只需要在关键函数上设置断点即可截获程序间传输的数据, 创建套接字的函数为 `socket`, 可以在这个函数上设置断点, 然后逐步跟踪, 服务端方面后期会调用 `bind` 绑定特定端口, `listen` 等待客户端连接, `accept` 来接收客户端的连接, 返回客户端套接字, 客户端方面生成套接字后直接调用 `connect` 与服务端进行连接, 连接成功后就进行数据传输, 发送数据的函数为 `recv`、`recvfrom`, 接收数据的函数为 `send`、`sendto`。当断在这些关键函数上时, 就可以获取他们传输的数据了。

远程线程调试



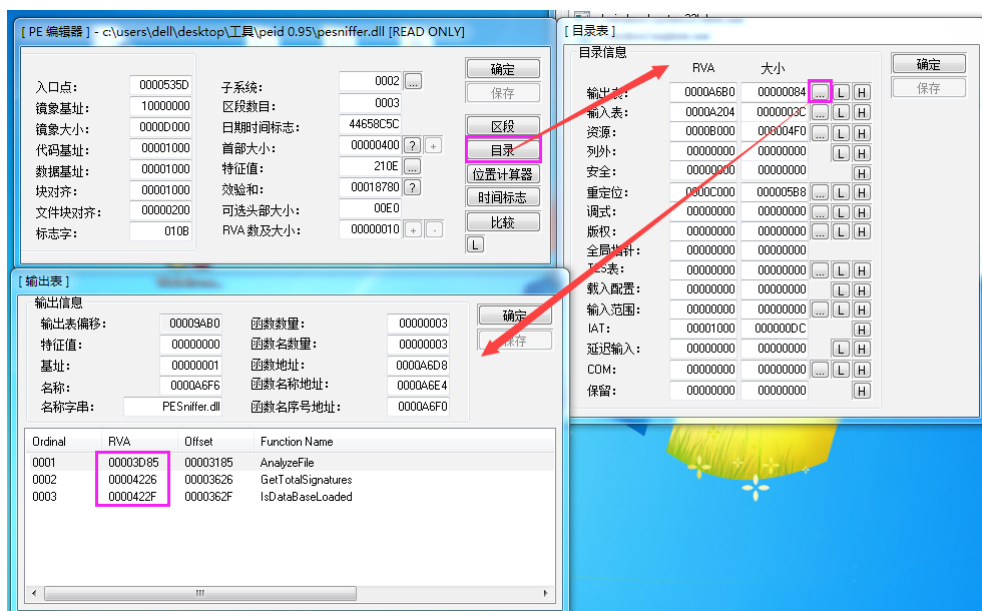
远程线程是攻击者常用的手法，通过这种方式，可以达到隐藏自身的目的，也可以注入代码或者动态库。

攻击者如果使用远程线程进行注入，肯定会在目标进程中写代码，使用的函数是 `WriteProcessMemory`，这个函数的第二个参数给出了写入的目标地址，记下这个地址。后面攻击者会调用函数 `RtlCreateUserThread`，`ResumeThread` 恢复线程运行。在它调用 `ResumeThread` 前，附加到目标进程中，点击菜单栏的“M”选项可以看到此进程的内存信息。在主进程 `WriteProcessMemory` 写入的内存地址处设置内存访问断点，然后 F9 让其运行，接着在回到主进程中让其调用 `ResumeThread`，这样，就可以在目标进程中中断下来。

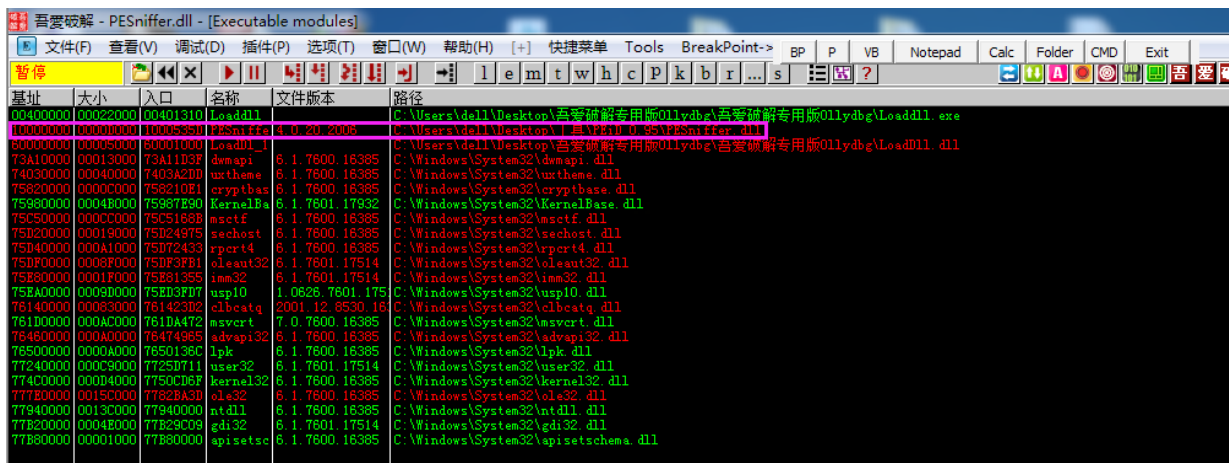
定位 DllEntry

调试动态库文件的时候，最简单的方式是使用 OD 载入，OD 中的 LoadDll.exe 会自动加载动态库文件，并定位到入口处，虽然有时候不太准确。这里讲解的是如何手动定位到 DllEntry。

先借助 LordPE 获取输出表的方法获取 RVA 地址



通过 OD 获取动态库文件的加载基地址



通过基址和 RVA 计算出来 VA，定位到入口函数进行分析



绿盟科技安全能力中心 (SAC)

Address	Disassembly	Comment	Register/Value
10003D7E	5E	pop esi	
10003D80	5B	pop ebx	
10003D81	C9	leave	
10003D82	C2 0C00	ret 0xC	
10003D85	55	push ebp	
10003D86	8BEC	mov ebp,esp	
10003D88	6A FF	push -0x1	
10003D8A	68 D0380010	push PESniffe.100038D0	
10003D8F	68 F84D0010	push PESniffe.10004DF8	
10003D94	64:A1 000000	mov eax,dword ptr fs:[0]	
10003D9A	50	push eax	
10003D9B	64:8925 0000	mov dword ptr fs:[0],esp	
10003DA2	81EC 1C01000	sub esp,0x11C	
10003DA8	53	push ebx	
10003DA9	56	push esi	
10003DAA	57	push edi	
10003DAB	8965 E8	mov dword ptr ss:[ebp-	
10003DAE	33DB	xor ebx,ebx	
10003DB0	895D E4	mov dword ptr ss:[ebp-	
10003DB3	889D D4FEFF	mov byte ptr ss:[ebp-	
10003DB9	6A 43	push 0x43	
10003DBB	59	pop ecx	

寄存器 (FPU)

EAX 10000000 PESniffe.

ECX 0000535D

EDX 00000000

EBX 00000001

ESP 0012FC34

EBP 0012FC48

ESI 00000001

EDI 0012FD18

EIP 1000535D PESniffe.

C 0 ES 0023 32位 0(FF

P 0 CS 001B 32位 0(FF

A 0 SS 0023 32位 0(FF

Z 0 DS 0023 32位 0(FF

0 FS 003B 32位 7FFD

0 GS 0000 NULL

0

LastErr ERROR_INU

FL 00000202 (NO,NB,NE

T0 empty 0.0

ST1 empty 0.0

ST2 empty 0.0

ST3 empty 0.0

ST4 empty 0.0

SE 处理程序安装

PESniffe.10000000

输入要跟随的表达式

10003d85

确定 取消

ebp=0012FC48

TLS 调试



TLS 的全称是 Thread Local Storage，是 Windows 为解决一个进程中多个线程同时访问全局变量而提供的机制。TLS 可以简单的由操作系统代为完成整个互斥过程，也可以由用户自己编写控制信号量的函数。当进程中的线程访问预先指定的内存空间时，操作系统会调用系统默认的或用户自定义的信号量函数，保证数据的完整性和正确性。

当用户选择使用自己编写的信号量函数时，在应用程序初始化阶段，系统将要调用一个由用户编写的初始化函数以完成信号量的初始化以及其他的一些初始化工作。此调用必须在侧滑盖内需真正开始执行到入口点之前就完成了，以保证程序执行的正确性。

TLS 回调函数的函数原型如下：

```
void NTAPI TlsCallbackFunction(PVOID Handle,DWORD Reason,PVOID Reserve);
```

Windows 的可执行文件为 PE 格式，在 PE 格式中，专门为 TLS 数据开辟了一段空间，具体位置为 IMAGE_NT_HEADERS.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS]。其中 Data Directory 的元素具有如下结构：

```
typedef struct _IMAGE_DATA_DIRECTORY{  
    DWORD VirtualAddress;  
    DWORD Size;  
}IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

对于 TLS 的 DataDirectory 元素，VirtualAddress 成员指向一个结构体，结构体中定义了访问需要互斥的内存地址，TLS 回调函数地址以及其他一些信息。

由于 TLS 回调函数是在 main 函数之前执行的，所以有些恶意软件将恶意代码写入到 TLS 回调函数中，我们调试的时候，在我们找到 main 函数时，攻击者已经执行完了它的恶意功能，所以我们要尽快下手，在回调函数执行前就断下来。

要在执行前设置断点，首要的任务就是找到回调函数的地址。

首先我们写一个带有回调函数的程序，参考代码如下：

```
#include "stdafx.h"  
#include <windows.h>  
#include <stdlib.h>  
#include <time.h>  
DWORD g_StartAddressOfRawData = 0;  
DWORD g_EndAddressOfRawData = 0;  
DWORD g_AddressOfIndex = 0;        // PDWORD  
DWORD g_SizeOfZeroFill = 0;  
DWORD g_Characteristics = 0;  
DWORD g_dwData1 = 0;  
DWORD g_dwData2 = 0;  
VOID NTAPI tlsCb1(PVOID DllHandle, DWORD Reason, PVOID Reserved);  
VOID NTAPI tlsCb2(PVOID DllHandle, DWORD Reason, PVOID Reserved);  
// 声明 g_tlsCbAry, 关键是要加 extern "C"  
extern "C" PIMAGE_TLS_CALLBACK g_tlsCbAry[] = {
```

```

    tlsCb1,
    tlsCb2,
    NULL
};

extern "C" IMAGE_TLS_DIRECTORY32 _tls_used = {
    (DWORD)& g_StartAddressOfRawData,
    (DWORD)& g_EndAddressOfRawData,
    (DWORD)& g_AddressOfIndex,
    (DWORD)g_tlsCbAry, // 这必须直接赋值为 tls 回调数组, PIMAGE_TLS_CALLBACK *
    g_SizeOfZeroFill,
    g_Characteristics
};

#pragma comment(linker, "/INCLUDE: __tls_used")
int main()
{
    if (g_dwData1 > 0) {
        printf("tlscb1 was called before main\r\n");
    }
    if (g_dwData2 > 0) {
        printf("tlscb2 was called before main\r\n");
    }
    system("pause");
    return 0;
}

VOID NTAPI tlsCb1(PVOID DllHandle, DWORD Reason, PVOID Reserved)
{
    MSG msg;
    if (DLL_PROCESS_ATTACH == Reason) {
        PeekMessageA(&msg, NULL, WM_KEYFIRST, WM_KEYLAST, PM_NOREMOVE);
        g_dwData1++;
    }
    else if (DLL_PROCESS_DETACH == Reason) {
        g_dwData1--;
    }
}

VOID NTAPI tlsCb2(PVOID DllHandle, DWORD Reason, PVOID Reserved)
{
    if (DLL_PROCESS_ATTACH == Reason) {
        g_dwData2++;
    }
    else if (DLL_PROCESS_DETACH == Reason) {
        g_dwData2--;
    }
}

```



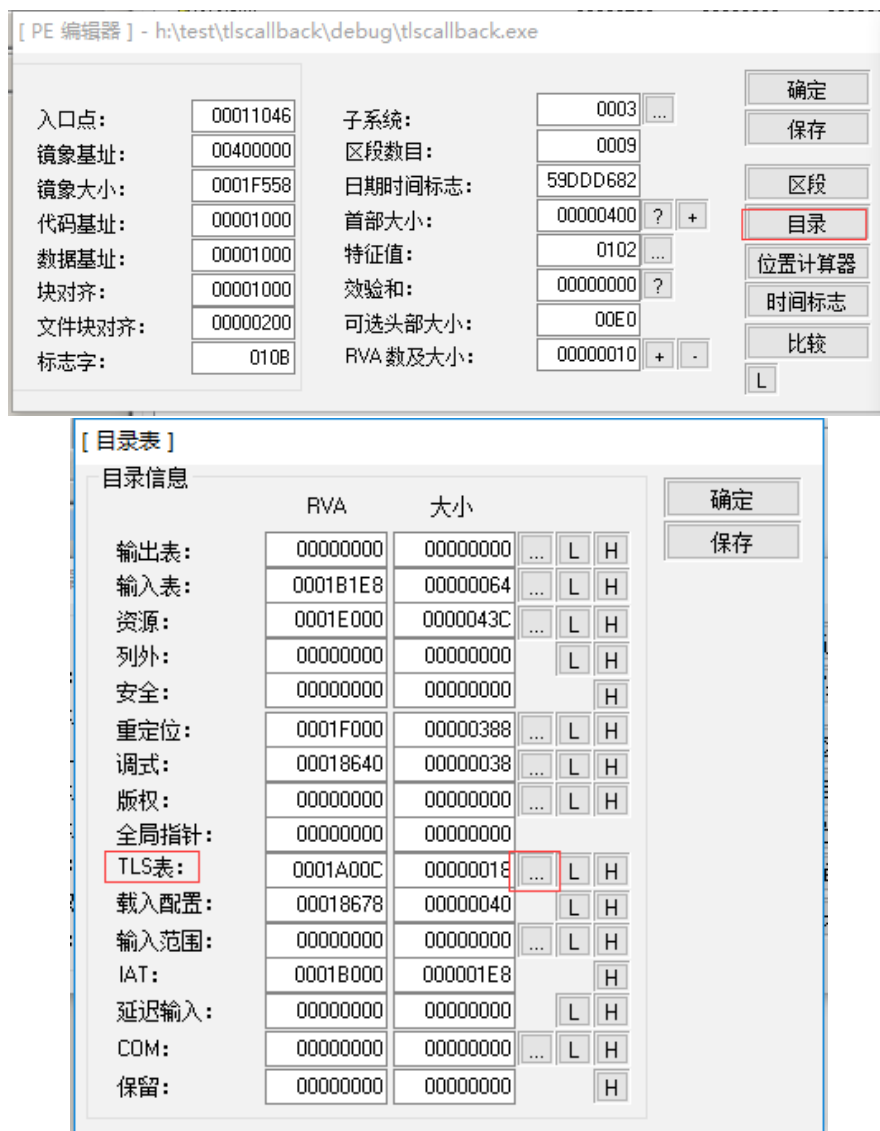
```
}  
}
```

运行结果如下：

```
C:\> 选择C:\WINDOWS\system32\cmd.exe  
tlscbl was called before main  
tlscb2 was called before main  
请按任意键继续. . .
```

可以看到，在 main 函数前，两个回调函数已经执行过了。下面介绍定位回调函数的方法。

使用 CFF，Load PE 或者其他工具，查看 PE 文件的数据目录表可以看到是否存在 TLS 回调函数



[TLS 表]

数据块开始 VA:	0041A16C	确定
数据块结束 VA:	0041A170	
索引变量 VA:	0041A174	
回调表 VA:	0041A000	
填零大小:	00000000	
特征值:	00000000	

然后打开 OD 或者 IDA 定位到这个地址，就可以看到回调函数了。

```
.data:0041A000 ; void ( _stdcall *g_tlsCbAry[3])(void *, unsigned int, void *)
.data:0041A000 g_tlsCbAry dd offset TlsCallback_0 ; DATA XREF: .data:TlsCallbacks_ptr+0
.data:0041A004 dd offset TlsCallback_1
.data:0041A008 dd 0
```

使用 IDA 找回调函数的一个简单的方法是，使用快捷键 Ctrl+E，就可以在弹出的窗口中看到所有的回调函数。

Choose an entry point

Name	Address	Ordinal
TlsCallback_0	004111C7	
TlsCallback_1	00411316	
j__mainCRTStartup	00411046	[main entry]

OK Cancel Search Help

Line 2 of 3

定位到回调函数后就可以像普通函数一样进行调试了。

windbg 调试 .NET

对于 .net 程序的调试，首选的工具还是建议使用 Reflector 或者 dnspy，这两款工具结合了静态反编译和动态调试两项功能。如果样本没有进行混淆的话，可以使用这两款工具直接看到程序源码，对于动态调试具有极大的帮助。但是对于一些特殊的情况，必须恶意样本在内存中释放的一段 .net 程序，这并不是一个完整的 .net 文件，所有就无法用上述两款工具进行查看，这样的话，就可以使用 windbg 进行调试。

这里使用自己写的一段简单的 .net 代码进行演示说明：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace FirstNet
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 3;
            int b = 4;
            int c = a + b;
            Console.WriteLine("a={0},b={1},c={2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

首先使用 windbg 打开可执行文件，然后加载动态库 sos.dll 和 clr.dll。需要注意的是，版本一定要对应，不然会出现 The call to LoadLibrary(C:\Windows\Microsoft.NET\Framework\v4.0.30319\sos.dll) failed, Win32 error 0n193 这样的错误。这两个动态库的路径一般为 C:\Windows\Microsoft.NET\Framework\v4.0.30319，如果是 64 位程序，使用 64 的 windbg 打开，就需要加载 64 位下的动态库。加载命令为：

```
.load C:\Windows\Microsoft.NET\Framework\v4.0.30319\sos.dll
```

```
.load C:\Windows\Microsoft.NET\Framework\v4.0.30319\clr.dll
```

本地磁盘 (C:) > WINDOWS > Microsoft.NET		
名称	修改日期	类型
assembly	2016/7/16 19:47	文件夹
authman	2016/7/16 19:47	文件夹
Framework	2017/10/10 9:27	文件夹
Framework64	2016/10/11 11:09	文件夹

接着输入 g 运行程序，在程序断下来时加载调试扩展

```
.loadby sos clr
```



然后使用命令 !CLRStack 仅为托管代码提供真实的调用栈信息

```
0:000> !CLRStack
OS Thread Id: 0x2e94 (0)
Child SP      IP Call Site
006fee90 7763e7fc [PrestubMethodFrame: 006fee90] FirstNet.Program.Main(System.String[])
006ff068 7763e7fc [GCFrame: 006ff068]
```

命令 Name2EE 用于将给定的类名转换为 Method Table 或 EEClass 的地址，或将方法名称转换为 MethodDesc。

```
0:000> !Name2EE FirstNet!FirstNet.Program.Main
Module:      008b3ffc
Assembly:    FirstNet.exe
Token:       06000001
MethodDesc:  008b4d0c
Name:        FirstNet.Program.Main(System.String[])
Not JITTED yet. Use !bpmd -md 008b4d0c to break on run.
找到 MethodDesc 的地址之后，就可以使用命令 DumpIL 显示 IL 指令
```

```
0:000> !DumpIL 008b4d0c
iAddr = 00332050
IL_0000: nop
IL_0001: ldc.i4.3
IL_0002: stloc.0
IL_0003: ldc.i4.4
IL_0004: stloc.1
IL_0005: ldloc.0
IL_0006: ldloc.1
IL_0007: add
IL_0008: stloc.2
IL_0009: ldstr "a={0},b={1},c={2}"
IL_000e: ldloc.0
IL_000f: box System.Int32
IL_0014: ldloc.1
IL_0015: box System.Int32
IL_001a: ldloc.2
IL_001b: box System.Int32
IL_0020: call System.Console::WriteLine
IL_0025: nop
IL_0026: call System.Console::ReadLine
IL_002b: pop
IL_002c: ret
```

使用命令 !u Address 来查看对应的汇编代码

```
0:000> !u 01684d0c
```


Normal JIT generated code

FirstNet.Program.Main(System.String[])

Begin 03230448, size c6

H:\test\FirstNet\FirstNet\Program.cs @ 12:

```

03230448 55          push    ebp
03230449 8bec        mov     ebp,esp
0323044b 57          push    edi
0323044c 56          push    esi
0323044d 83ec28      sub     esp,28h
03230450 8bf1        mov     esi,ecx
03230452 8d7dd0      lea     edi,[ebp-30h]
03230455 b906000000  mov     ecx,6
0323045a 33c0        xor     eax,eax
0323045c f3ab        rep stos dword ptr es:[edi]
0323045e 8bce        mov     ecx,esi
03230460 894df4      mov     dword ptr [ebp-0Ch],ecx
03230463 833da842680100 cmp     dword ptr ds:[16842A8h],0
0323046a 7405        je      03230471
0323046c e83f125a6f  call    clr!JIT_DbglJustMyCode (727d16b0)
03230471 33d2        xor     edx,edx
03230473 8955ec      mov     dword ptr [ebp-14h],edx
03230476 33d2        xor     edx,edx
03230478 8955f0      mov     dword ptr [ebp-10h],edx
0323047b 33d2        xor     edx,edx
0323047d 8955e8      mov     dword ptr [ebp-18h],edx
03230480 90          nop

```

H:\test\FirstNet\FirstNet\Program.cs @ 13:

```

03230481 c745f003000000 mov     dword ptr [ebp-10h],3

```

H:\test\FirstNet\FirstNet\Program.cs @ 14:

```

03230488 c745ec04000000 mov     dword ptr [ebp-14h],4

```

H:\test\FirstNet\FirstNet\Program.cs @ 15:

```

0323048f 8b45f0      mov     eax,dword ptr [ebp-10h]
03230492 0345ec      add     eax,dword ptr [ebp-14h]
03230495 8945e8      mov     dword ptr [ebp-18h],eax

```

H:\test\FirstNet\FirstNet\Program.cs @ 16:

```

03230498 b9043c1570  mov     ecx,offset mscorlib_ni+0x503c04 (70153c04) (MT: System.Int32)
0323049d e8522c44fe  call    016730f4 (JitHelp: CORINFO_HELP_NEWSFAST)

```



```

032304a2 8945e4      mov     dword ptr [ebp-1Ch],eax
032304a5 b9043c1570    mov     ecx,offset mscorlib_ni+0x503c04 (70153c04) (MT: System.Int32)
032304aa e8452c44fe     call    016730f4 (JitHelp: CORINFO_HELP_NEWSFAST)
032304af 8945e0      mov     dword ptr [ebp-20h],eax
032304b2 b9043c1570    mov     ecx,offset mscorlib_ni+0x503c04 (70153c04) (MT: System.Int32)
032304b7 e8382c44fe     call    016730f4 (JitHelp: CORINFO_HELP_NEWSFAST)
032304bc 8945dc      mov     dword ptr [ebp-24h],eax
032304bf 8b05bc224b04  mov     eax,dword ptr ds:[44B22BCh] ("a={0},b={1},c={2}")
032304c5 8945d4      mov     dword ptr [ebp-2Ch],eax
032304c8 8b45e4      mov     eax,dword ptr [ebp-1Ch]
032304cb 8b55f0      mov     edx,dword ptr [ebp-10h]
032304ce 895004      mov     dword ptr [eax+4],edx
032304d1 8b45e4      mov     eax,dword ptr [ebp-1Ch]
032304d4 8945d0      mov     dword ptr [ebp-30h],eax
032304d7 8b45e0      mov     eax,dword ptr [ebp-20h]
032304da 8b55ec      mov     edx,dword ptr [ebp-14h]
032304dd 895004      mov     dword ptr [eax+4],edx
032304e0 8b45e0      mov     eax,dword ptr [ebp-20h]
032304e3 50          push    eax
032304e4 8b45dc      mov     eax,dword ptr [ebp-24h]
032304e7 8b55e8      mov     edx,dword ptr [ebp-18h]
032304ea 895004      mov     dword ptr [eax+4],edx
032304ed 8b45dc      mov     eax,dword ptr [ebp-24h]
032304f0 50          push    eax
032304f1 8b4dd4      mov     ecx,dword ptr [ebp-2Ch]
032304f4 8b55d0      mov     edx,dword ptr [ebp-30h]
032304f7 e8506ade6c    call    mscorlib_ni+0x3c6f4c (70016f4c) (System.Console.WriteLine(System.String,
System.Object, System.Object, System.Object), mdToken: 06000b51)
032304fc 90          nop

```

H:\test\FirstNet\FirstNet\Program.cs @ 17:

```

032304fd e87a71566d    call    mscorlib_ni+0xb4767c (7079767c) (System.Console.ReadLine(), mdToken:
06000b40)
03230502 8945d8      mov     dword ptr [ebp-28h],eax
03230505 90          nop

```

H:\test\FirstNet\FirstNet\Program.cs @ 18:

```

03230506 90          nop
03230507 8d65f8      lea     esp,[ebp-8]
0323050a 5e          pop     esi
0323050b 5f          pop     edi

```

```
0323050c 5d          pop     ebp
```

```
0323050d c3          ret
```

其中的 @ 17, @ 18 表示源码中的行号。

这种情况下, 就可以直接使用 bp Address 在目标地址出设置断点了。

sos 扩展命令如下所示:

Object Inspection	Examining code and stacks
-----	-----
DumpObj (do)	Threads
DumpArray (da)	ThreadState
DumpStackObjects (dso)	IP2MD
DumpHeap	U
DumpVC	DumpStack
GCRoot	EESStack
ObjSize	CLRStack
FinalizeQueue	GCInfo
PrintException (pe)	EHInfo
TraverseHeap	BPMD
	COMState
Examining CLR data structures	Diagnostic Utilities
-----	-----
DumpDomain	VerifyHeap
EEHeap	VerifyObj
Name2EE	FindRoots
SyncBlk	HeapStat
DumpMT	GCWhere
DumpClass	ListNearObj (lno)
DumpMD	GCHandles
Token2EE	GCHandleLeaks
EEVersion	FinalizeQueue (fq)
DumpModule	FindAppDomain
ThreadPool	SaveModule
DumpAssembly	ProcInfo
DumpSigElem	StopOnException (soe)
DumpRuntimeTypes	DumpLog
DumpSig	VMMMap
RCWCleanupList	VMStat
DumpIL	MinidumpMode
DumpRCW	AnalyzeOOM (ao)
DumpCCW	



绿盟科技安全能力中心 (SAC)

Examining the GC history

Other

HistInit

FAQ

HistRoot

HistObj

HistObjFind

HistClear

如果要查看某个命令的详细信息，可以使用 !help <functionname> 进行查看。

消息处理调试



Windows 程序和 MFC 程序是靠消息驱动的，他们对于消息的处理本质上是相同的。只是 Windows 程序对于消息处理的过程十分清晰明了，MFC 程序则掩盖了消息处理的过程，以消息映射的方式呈现在开发者面前，使得开发消息的处理十分简单。当然，在分析消息类程序之前首先对 Windows 程序的消息机制进行一个简单的描述。

消息机制说明

1. 什么是消息

消息对于 Win32 程序来说十分重要，它是一个程序运行的动力源泉。一个消息，是系统定义的一个 32 位的值，他唯一的定义了一个事件，向 Windows 发出一个通知，告诉应用程序某个事情发生了。例如，单击鼠标、改变窗口尺寸、按下键盘上的一个键都会使 Windows 发送一个消息给应用程序的消息队列中，然后应用程序再从消息队列中取出消息并进行相应的响应。在这个处理的过程中，操作系统也会给应用程序“发送消息”，而所谓的发送消息 -- 实际上就是操作系统调用程序中的一个专门负责处理消息的函数，这个函数称为窗口过程。

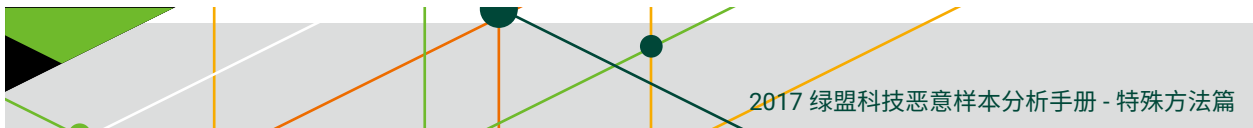
消息本身是作为一个记录传递给应用程序的，这个记录中包含了消息的类型以及其他信息。例如，对于单击鼠标所产生的消息来说，这个记录中包含了单击鼠标时的坐标。这个记录类型叫做 MSG，MSG 含有来自 windows 应用程序消息队列的消息信息，在 Windows 中 MSG 结构体定义如下：

```
typedef struct tagMsg
{
    HWND    hwnd;        // 接受该消息的窗口句柄
    UINT    message;      // 消息常量标识符，也就是我们通常所说的消息号
    WPARAM  wParam;       // 32 位消息的特定附加信息，确切含义依赖于消息值
    LPARAM  lParam;       // 32 位消息的特定附加信息，确切含义依赖于消息值
    DWORD   time;         // 消息创建时的时间
    POINT   pt;           // 消息创建时的鼠标 / 光标在屏幕坐标系中的位置
} MSG;
```

hwnd：窗口句柄，它表示的是消息所属的窗口。我们通常开发的程序都是窗口应用程序，一般一个消息都是和某个窗口相关联的。比如我们在某个活动窗口按下鼠标右键，此时产生的消息就是发送给该活动窗口的。窗口可以是任何类型的屏幕对象，因为 Win32 能够维护大多数可视对象的句柄（窗口、对话框、按钮、编辑框等）。

message：一个消息的标识符，用于区别其他消息的常量值，这些常量可以是 Windows 单元中预定义的常量，也可以是自定义的常量。在 Windows 中消息是由一个数值表示的，不同的消息对应不同的数值。但由于当这些消息种类多到足以挑战我们的 IQ，所以聪明的程序开发者便想到将这些数值定义为 WM_XXX 宏的形式。例如，鼠标左键按下的消息 --WM_LBUTTONDOWN，键盘按下消息 --WM_KEYDOWN，字符消息 --WM_CHAR，等等.. 消息标识符以常量命名的方式指出消息的含义。当窗口过程接收到消息之后，他就会使用消息标识符来决定如何处理消息。例如、WM_PAINT 告诉窗口过程窗体客户区被改变了需要重绘。符号常量指定系统消息属于的类别，其前缀指明了处理解释消息的窗体的类型。

wParam 和 lParam-- 用于指定消息的附加信息。例如，当我们收到一个键盘按下消息的时候，message 成员变量的值就是 WM_KEYDOWN，但是用户到底按下的是哪一个按键，我们就得拜托这二位，由他们来告知我们具体的信息。



time 和 pt- - 这两兄弟分别被用来表示消息投递到消息队列中的时间和鼠标当前的位置，一般情况下不怎么使用（但不代表没用）。

2. 消息队列

在 Windows 编程中，每一个 Windows 应用程序开始执行后，系统都会为该程序创建一个消息队列，这个消息队列用来存放该应用程序所创建的窗口的信息。例如，当我们按下鼠标右键的时候，这时会产生一个 WM_RBUTTONDOWN 消息，系统会自动将这个消息放进当前窗口所属的应用程序的消息队列中，等待应用程序的结束。Windows 将产生的消息以此放进消息队列中，应用程序则通过一个消息循环不断的从该消息队列中读取消息，并做出响应。

3. 消息标识符

系统保留消息标识符的值在 0x0000 在 0x03ff(WM_USER-1) 范围。这些值被系统定义消息使用。应用程序不能使用这些值给自己的消息。应用程序消息从 WM_USER (0x0400) 到 0x7FFF，或 0xC000 到 0xFFFF；WM_USER 到 0x7FFF 范围的消息由应用程序自己使用；0xC000 到 0xFFFF 范围的消息用来和其他应用程序通信，在此只是罗列一些具有标志性的消息值：

WM_NULL---0x0000	空消息。
0x0001----0x0087	主要是窗口消息。
0x00A0----0x00A9	非客户区消息
0x0100----0x0108	键盘消息
0x0111----0x0126	菜单消息
0x0132----0x0138	颜色控制消息
0x0200----0x020A	鼠标消息
0x0211----0x0213	菜单循环消息
0x0220----0x0230	多文档消息
0x03E0----0x03E8	DDE 消息
0x0400	WM_USER
0x8000	WM_APP
0x0400----0x7FFF	应用程序自定义私有消息

4. 消息的分类

Windows 消息大体上可以分为 3 类：窗口消息，命令消息和空间通知消息。

窗口消息是系统中最常见的消息，它是指由操作系统和控制其他窗口的窗口所使用的消息。例如 CreateWindow、DestroyWindow 和 MoveWindow 等都会激发窗口消息，还有我们在上面谈到的单击鼠标所产生的消息也是一种窗口消息。

命令消息是一种特殊的窗口消息，他用来处理从一个窗口发送到另一个窗口的用户请求，例如按下一个按钮，他就会向主窗口发送一个命令消息。

控件消息：其实它是这样的，当一个窗口内的子控件发生了一些事情，而这些是需要通知父窗口的，此刻它就上场啦。通知消息只适用于标准的窗口控件如按钮、列表框、组合框、编辑框，以及 Windows 公共控件如树状视图、列表视图等。



例如，单击或双击一个控件、在控件中选择部分文本、操作控件的滚动条都会产生通知消息 -- 她类似于命令消息，那么控件通知消息就会从控件窗口发送到它的主窗口。但是这种消息的存在并不是为了处理用户命令，而是为了让主窗口能够改变控件，例如加载、显示数据。

5. 队列消息和非队列消息

从消息的发送途径来看，Windows 程序中的消息可以分成 2 种：队列消息和非队列消息，也有叫“进队消息”和“不进队消息”。

消息队列可以分成系统消息队列和线程消息队列。系统消息队列由 Windows 维护，线程消息队列则由每个 GUI 线程自己进行维护，为避免给 non-GUI 现成创建消息队列，所有线程产生时并没有消息队列，仅当线程第一次调用 GDI 函数时系统才给线程创建一个消息队列。

(1) 队列消息送到系统消息队列，然后到线程消息队列；

对于队列消息，最常见的是鼠标和键盘触发的消息，例如 WM_MOUSEMOVE、WM_CHAR 等消息，还有一些其它的消息，例如：WM_PAINT、WM_TIMER 和 WM_QUIT。当鼠标、键盘事件被触发后，相应的鼠标或键盘驱动程序就会把这些事件转换成相应的消息，然后输送到系统消息队列，由 Windows 系统去进行处理。Windows 系统则在适当的时机，从系统消息队列中取出一个消息，根据前面我们所说的 MSG 消息结构确定消息是要被送往那个窗口，然后把取出的消息送往创建窗口的线程的相应队列，下面的事情就该由线程消息队列操心了，Windows 开始忙自己的事情去了。线程看到自己的消息队列中有消息，就从队列中取出来，通过操作系统发送到合适的窗口过程去处理。

一般来讲，系统总是将消息 Post 在消息队列的末尾。这样保证窗口以先进先出的顺序接受消息。然而，WM_PAINT 是一个例外，同一个窗口的多个 WM_PAINT 被合并成一个 WM_PAINT 消息，合并所有的无效区域到一个无效区域。合并 WM_PAINT 的目的是为了减少刷新窗口的次数。

(2) 非队列消息直接送给目的窗口过程。

非队列消息将会绕过系统队列和消息队列，直接将消息发送到窗口过程。系统发送非队列消息通知窗口，系统发送消息通知窗口。例如，当用户激活一个窗口系统发送 WM_ACTIVATE、WM_SETFOCUS，and WM_SETCURSOR。这些消息通知窗口它被激活了。非队列消息也可以由当应用程序调用系统函数产生。例如，当程序调用 SetWindowPos 系统发送 WM_WINDOWPOSCHANGED 消息。一些函数也发送非队列消息。

调试示例

以一个简单的 win32 程序进行说明（直接使用 VS 生成 win32 项目即可）。由于代码量比较大，这里只展现关键代码。

注册窗口类：

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEXW wcex;
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style      = CS_HREDRAW | CS_VREDRAW;
```



```

wce.x.lpfnWndProc    = WndProc;    // 关联窗口过程函数
wce.x.cbClsExtra     = 0;
wce.x.cbWndExtra     = 0;
wce.x.hInstance     = hInstance;
wce.x.hIcon         = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_WIN32PROJECT1));
wce.x.hCursor       = LoadCursor(nullptr, IDC_ARROW);
wce.x.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wce.x.lpszMenuName  = MAKEINTRESOURCEW(IDC_WIN32PROJECT1);
wce.x.lpszClassName = szWindowClass;
wce.x.hIconSm       = LoadIcon(wce.x.hInstance, MAKEINTRESOURCE(IDI_SMALL));
return RegisterClassExW(&wce.x);
}

```

注册完之后创建窗口：

```

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    hInst = hInstance; // 将实例句柄存储在全局变量中
    HWND hWnd = CreateWindowW(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, nullptr, nullptr, hInstance, nullptr);
    if (!hWnd)
    {
        return FALSE;
    }
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    return TRUE;
}

```

消息循环：

```

while (GetMessage(&msg, nullptr, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

```

消息处理函数：

```

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_COMMAND:

```



```
{
    int wmlId = LOWORD(wParam);
    // 分析菜单选择 :
    switch (wmlId)
    {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
}
break;
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    // TODO: 在此处添加使用 hdc 的任何绘图代码 ...
    EndPaint(hWnd, &ps);
}
break;
case WM_DESTROY:
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
```

这个 win32 程序，我们没有写任何代码，运行之后显示一个窗口，关闭窗口程序结束运行。借助这个程序说一下与消息相关的函数。

把一个消息发送到窗口有三种方式：发送，寄送和广播。

发送消息的函数有 SendMessage、SendMessageCallback、SendNotifyMessage、SendMessageTimeout。

寄送消息的函数有 PostMessage、PostThreadMessage、PostQuitMessage。

广播消息的函数有：BroadcastSystemMessage、BroadcastSystemMessageEx。

消息的接收主要由 3 个函数：GetMessage、PeekMessage、WaitMessage。

关于函数的信息可以参考 API 函数篇或自行网上查阅。

消息循环：

```
while (GetMessage(&msg, nullptr, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

在消息循环中，GetMessage 从进程的主线程的消息队列中获取一个消息并将它复制到 MSG 结构，如果队列中没有消息，则 GetMessage 函数将等待一个消息的到来以后才返回。如果你将一个窗口句柄作为第二个参数传入 GetMessage，那么只有指定窗口的消息可以从队列中获得。GetMessage 也可以从消息队列中过滤消息只接受消息队列中落在范围内的消息。这时候就要利用 GetMessage / PeekMessage 指定一个消息过滤器。这个过滤器是一个消息标识符的范围或者是一个窗体句柄，或者两者同时指定。当应用程序要查找一个后入消息队列的消息是很有用。WM_KEYFIRST 和 WM_KEYLAST 常量用于接受所有的键盘消息。WM_MOUSEFIRST 和 WM_MOUSELAST 常量用于接受所有的鼠标消息。

然后 TranslateAccelerator 判断该消息是不是一个按键消息并且是一个加速键消息，如果是，则该函数将把几个按键消息转换成一个加速键消息传递给窗口的回调函数。处理了加速键之后，函数 TranslateMessage 将把两个按键消息 WM_KEYDOWN 和 WM_KEYUP 转换成一个 WM_CHAR，不过需要注意的是，消息 WM_KEYDOWN, WM_KEYUP 仍然将传递给窗口的回调函数。

处理完之后，DispatchMessage 函数将把此消息发送给该消息指定的窗口中已设定的回调函数。如果消息是 WM_QUIT，则 GetMessage 返回 0，从而退出循环体。应用程序可以使用 PostQuitMessage 来结束自己的消息循环。通常在主窗口的 WM_DESTROY 消息中调用。

消息处理函数：

窗口过程是一个用于处理所有发送到这个窗口的消息的函数。任何一个窗口类都有一个窗口过程。同一个类的窗口使用同样的窗口过程来响应消息。系统发送消息给窗口过程将消息作为参数传递给他，消息到来之后，按照消息类型排序进行处理，其中的参数则用来区分不同的消息，窗口过程使用参数产生对应行为。

一个窗口过程不经常忽略消息，如果它不处理，他会将消息传回执行到默认的处理。窗口过程通过调用 DefWindowProc 来做这个处理。窗口过程必须 return 一个值作为它的消息处理结果。大多数窗口只处理小部分消息和将其他的通过 DefWindowProc 传递给系统做默认的处理。窗口过程被素有属于同一个类的窗口共享，能为不同点窗口处理消息。

知道了消息机制的原理，下面简单说明一下如何使用 OD 调试。调试 Win32 程序，重点关注的是消息的处理。而消息过程函数是在注册类的时候初始化的，所以我们需要在 RegisterClassEx 函数设置断点，它的参数即为窗口类，结构体声明如下：

```
typedef struct tagWNDCLASSEXW {
```



```
UINT        cbSize;
/* Win 3.x */
UINT        style;
WNDPROC      lpfnWndProc;
int          cbClsExtra;
int          cbWndExtra;
HINSTANCE    hInstance;
HICON        hIcon;
HCURSOR      hCursor;
HBRUSH       hbrBackground;
LPCWSTR      lpzMenuName;
LPCWSTR      lpzClassName;
/* Win 4.0 */
HICON        hIconSm;
} WNDCLASSEXW, *PWNDCLASSEXW, NEAR *NPWNDCLASSEXW, FAR *LPWNDCLASSEXW;
第三个参数就是我们需要关注的重点函数。
```

地址	HEX 数据	ASCII	CALL 到 RegisterClassExW 来自 Win32Pro.00EB19D5
00E4F5F4	30 00 00 00 03 00 00 00 52 13 EB 00 00 00 00 00	0... ..RM?...	00E4F51C 00EB19DB CALL 到 RegisterClassExW 来自 Win32Pro.00EB19D5
00E4F604	00 00 00 00 00 00 EA 00 05 08 54 07 03 00 01 00?eMT.左	00E4F520 00E4F5F4 lpWndClassEx = 00E4F5F4
00E4F614	06 00 00 00 60 00 00 00 10 92 EB 00 DF 05 6F 04?eMT.左	00E4F524 00E4F738 ASCII "P" "P"
00E4F624	CC CC CC CC 77 56 EC 88 38 F7 E4 00 D9 1C EB 00?eMT.左	00E4F528 00E4F638
00E4F634	00 00 EA 00 72 11 EB 00 72 11 EB 00 00 00 12 01?eMT.左	00E4F52C 0112B000
			00E4F530 CCCCCCCC

程序断在 RegisterClassEx 后，左边为 pWndClassEx 参数的数据部分，可以看到第三个参数 0xEB1352 就是消息过程函数，再在这个函数设置断点，运行程序，在有消息处理的时候就会停留在过程处理函数中。然后调试方法就和普通程序的调试方法一样了。

《2017 绿盟科技恶意样本分析手册 - 特殊方法篇》

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

本文中出现的任何文字叙述、文档格式、插图、照片、方法、过程等内容，除另有特别注明，版权均属绿盟科技所有，受到有关产权及版权法保护。任何个人、机构未经绿盟科技的书面授权许可，不得以任何方式复制或引用本文的任何片断。



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