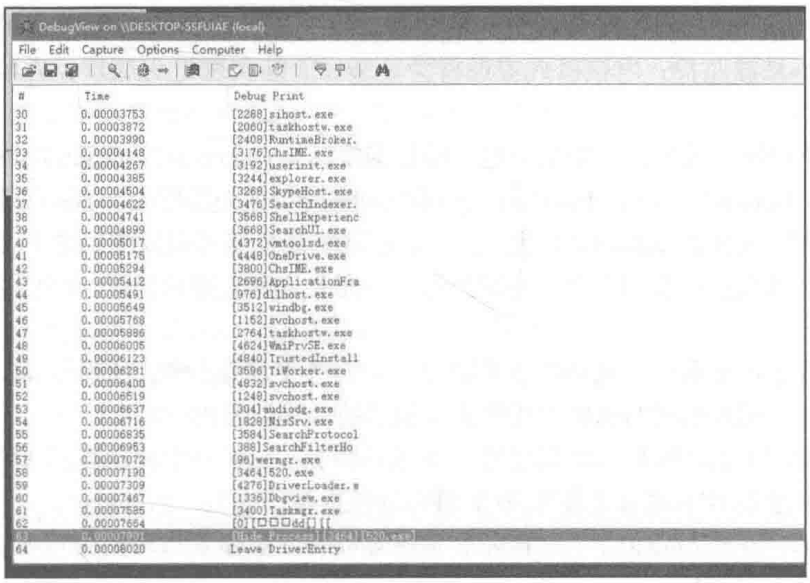

图 18-2 绕过 Patch Guard 隐藏指定进程

18.2.4 小结

实现进程隐藏的关键在于理解系统进程遍历的原理，系统是通过遍历进程结构 EPROCESS 中的活动进程双向链表 ActiveProcessLinks 来实现进程遍历的。所以，进程隐藏的实现思路便是在活动双向链表中进行摘链处理，使系统获取不到进程信息，从而实现进程隐藏。

其中，不能通过 RemoveEntryList 这个普通的摘链函数去实现摘链，而是要通过未导出函数 MiProcessLoaderEntry 实现摘链，该函数执行的摘链操作不会触发 Patch Guard。

对于这个程序的理解并不难，关键是要理解双向链表的思想。双向链表遍历进程并从中摘链实现进程隐藏。

需要注意一点，在不同系统上，ActiveProcessLinks 在 EPROCESS 进程结构中的偏移位置并不是固定不变的。我们可以借助 WinDbg 来获取在不同系统中 EPROCESS 进程结构的定义。

安全小贴士

使用暴力搜索内存，根据 EPROCESS 结构中 PEB 等特征值定位 EPROCESS 结构，从而获取内存中所有进程信息，包括隐藏进程。

18.3 TDI 网络通信

Windows 操作系统提供了两种网络编程模式，它们分别为用户模式和内核模式。顾名思义，用户模式是主要是通过调用用户层的 API 函数实现的用户程序，内核模式主要是通过

调用内核层的内核 API 函数或是自定义的通信协议实现的内核驱动程序。用户模式虽然易于开发，但是容易被监控；内核模式实现较为复杂，但是实现更为底层，通信更为隐蔽，较难监控。

内核模式下的网络编程，主要是通过传输数据接口（Transport Data Interface, TDI）和网络驱动接口规范（Network Driver Interface Specification, NDIS）实现的。TDI 是直接使用现有的 TCP/IP 来通信的，无需重新编写新的协议，所以，网络防火墙可以检测到基于 TDI 开发的网络通信。而 NDIS 可以直接在网络上读写原始报文，需要自定义通信协议，所以它能够绕过网络防火墙的检测。

数据回传对于病毒木马来说是极为关键的一步，稍有差池，则会原形毕露。所以，内核下的网络通信，会使病毒木马的通信方式更为底层隐蔽，难以检测。

接下来，本节将介绍基于 TDI 实现的 TCP 网络通信，它使一个驱动客户端程序能够与用户层的服务端程序建立 TCP 连接并使用 TCP 进行通信。

18.3.1 实现过程

前面用户篇的时候介绍过 Socket 编程之 TCP 通信的相关内容，Socket 编程中调用的 API 函数比较容易理解，然而，内核下的 TDI 并没有现成封装好的函数接口供程序调用。为了方便读者类比 Socket 编程来理解 TDI 编程，所以，接下来将基于 TDI 实现 TCP 客户端的实现原理分成 5 个部分来介绍，它们分别为 TDI 初始化、TDI TCP 连接、TDI TCP 数据发送、TDI TCP 数据接收以及 TDI 关闭。

1. TDI 初始化

在调用 TDI 进行 TCP 数据通信之前，需要先初始化 TDI 操作。初始化操作主要包括创建本地地址对象、创建端点对象以及将端点对象与本地地址对象进行关联。因此，具体的 TDI 初始化实现步骤如下所示。

首先，在创建本地地址对象之前，先构建本地地址拓展属性结构 PFILE_FULL_EA_INFORMATION。设置该拓展属性结构的名称为 TdiTransportAddress，拓展属性结构的内容是 TA_IP_ADDRESS，它里面存储着通信协议类型、本地 IP 地址及端口等信息。TA_IP_ADDRESS 中的 AddressType 表示通信协议类型，TDI_ADDRESS_TYPE_IP 则支持 UDP 和 TCP 等 IP。将 IP 地址以及端口都置为 0，这表示为本机本地 IP 地址和随机端口。

在本地地址拓展属性结构构建完成之后，就可以调用 ZwCreateFile 函数根据本地地址拓展属性结构创建本地地址对象。该函数打开的设备名称为“\\Device\\Tcp”，即打开了 TCP 设备驱动服务。ZwCreateFile 函数中的重要参数是拓展属性（Extended Attributes），通过拓展属性可以向其他的驱动程序传递信息。所以，驱动程序会将本地地址拓展属性结构的数据传递 TCP 设备驱动，以创建本地地址对象，并获取对象句柄。在获取本地地址对象句柄后，调用 ObReferenceObjectByHandle 函数可获取本地地址对象的文件对象，并根据得到的文件对象调用 IoGetRelatedDeviceObject 函数以获取对应本地地址对象的驱动设备指针，以方便后续操作。

然后，开始创建端点对象。同样，在创建端点对象之前，应先构建上下文拓展属性结构 PFILE_FULL_EA_INFORMATION。设置该拓展属性结构的名称为 TdiConnectionContext，拓展属性结构的内容为 CONNECTION_CONTEXT。本节并没有用到 CONNECTION_CONTEXT 结构里的数据，所以都置为零。

上下文拓展属性结构构建完成后，同样是调用 ZwCreateFile 函数根据上下文拓展属性结构来创建端点对象。仍是打开 TCP 设备驱动服务，向 TCP 设备驱动传递上下文结构数据，以创建端点对象，并获取端点对象句柄。在获取端点对象句柄之后，直接调用 ObReferenceObjectByHandle 函数来获取端点对象的文件对象，以方便后续操作。

最后，在创建了本地地址对象和端点对象后，将两者关联起来，没有关联地址的端点是没有任何用处的。其中，本地地址对象存储的信息向系统表明驱动程序使用的是本地 IP 地址和端口。直接调用 TdiBuildInternalDeviceControlIrp 函数创建 TDI 的 I/O 请求包（IRP），消息类型为 TDI_ASSOCIATE_ADDRESS，这表示端点对象关联到本地地址对象，它需要用到上述获取的本地地址驱动设备对象指针以及端点文件对象指针作为参数。TdiBuildInternalDeviceControlIrp 实际是一个宏，它在内部调用了 IoBuildDeviceIoControlRequest，这个函数将这个宏的一些参数忽略了。所以，调用 TdiBuildAssociateAddress 宏，将获取的本地地址驱动设备对象指针以及端点文件对象指针添加到 IRP 的 I/O 堆栈空间中。

完成上述 3 个操作之后，就可以调用 IoCallDriver 函数向驱动设备发送 TDI 的 I/O 请求包了。其中，驱动程序需要等待系统执行 IRP，所以，需要调用 IoSetCompletionRoutine 设置完成回调函数，通知程序 IRP 执行完成。这样，TDI 的初始化操作到此结束了。

TDI 初始化的具体实现代码如下所示。

```
    // TDI 初始化设置
    NTSTATUS        TdiOpen(PDEVICE_OBJECT        *ppTdiAddressDevObj,        PFILE_OBJECT
*ppTdiEndPointFileObject, HANDLE *phTdiAddress, HANDLE *phTdiEndPoint)
    {
        // 变量（略）
        // 为本地地址拓展属性结构申请内存及初始化
        ulAddressEaBufferLength = sizeof(FILE_FULL_EA_INFORMATION) + TDI_TRANSPORT_ ADDRESS_
LENGTH + sizeof(TA_IP_ADDRESS);
        pAddressEaBuffer     =     (PFILE_FULL_EA_INFORMATION)ExAllocatePool(NonPagedPool,
ulAddressEaBufferLength);
        RtlZeroMemory(pAddressEaBuffer, ulAddressEaBufferLength);
        RtlCopyMemory(pAddressEaBuffer->EaName, TdiTransportAddress, (1 + TDI_TRANSPORT_
ADDRESS_LENGTH));
        pAddressEaBuffer->EaNameLength = TDI_TRANSPORT_ADDRESS_LENGTH;
        pAddressEaBuffer->EaValueLength = sizeof(TA_IP_ADDRESS);
        // 初始化本机 IP 地址与端口
        pTaIpAddr = (PTA_IP_ADDRESS)((PUCHAR)pAddressEaBuffer->EaName + pAddressEaBuffer->
EaNameLength + 1);
        pTaIpAddr->TAAddressCount = 1;
        pTaIpAddr->Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
        pTaIpAddr->Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
        pTaIpAddr->Address[0].Address[0].sin_port = 0;     // 0 表示为任意端口
        pTaIpAddr->Address[0].Address[0].in_addr = 0;     // 0 表示本机本地 IP 地址
        RtlZeroMemory(pTaIpAddr->Address[0].Address[0].sin_zero,
```

```
sizeof(pTaIpAddr->Address[0].Address[0].sin_zero));
        // 创建 TDI 驱动设备字符串与初始化设备对象
        RtlInitUnicodeString(&ustrTDIDevName, COMM_TCP_DEV_NAME);
        InitializeObjectAttributes(&ObjectAttributes, &ustrTDIDevName, OBJ_CASE_INSENSITIVE
| OBJ_KERNEL_HANDLE, NULL, NULL);
        // 根据本地地址拓展属性结构创建本地地址对象
        status = ZwCreateFile(&hTdiAddress, GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
            &ObjectAttributes, &iosb, NULL, FILE_ATTRIBUTE_NORMAL,
            FILE_SHARE_READ, FILE_OPEN, 0, pAddressEaBuffer, ulAddressEaBufferLength);
        // 根据本地地址对象句柄获取对应的本地地址文件对象
        status = ObReferenceObjectByHandle(hTdiAddress,
            FILE_ANY_ACCESS, 0, KernelMode, &pTdiAddressFileObject, NULL);
        // 获取本地地址文件对象对应的驱动设备
        pTdiAddressDevObj = IoGetRelatedDeviceObject(pTdiAddressFileObject);

        // 为上下文拓展属性申请内存并初始化
        ulContextEaBufferLength = FIELD_OFFSET(FILE_FULL_EA_INFORMATION, EaName) +
TDI_CONNECTION_CONTEXT_LENGTH + 1 + sizeof(CONNECTION_CONTEXT);
        pContextEaBuffer = (PFILE_FULL_EA_INFORMATION)ExAllocatePool(NonPagedPool,
ulContextEaBufferLength);
        RtlZeroMemory(pContextEaBuffer, ulContextEaBufferLength);
        RtlCopyMemory(pContextEaBuffer->EaName, TdiConnectionContext, (1 + TDI_CONNECTION_
CONTEXT_LENGTH));
        pContextEaBuffer->EaNameLength = TDI_CONNECTION_CONTEXT_LENGTH;
        pContextEaBuffer->EaValueLength = sizeof(CONNECTION_CONTEXT);
        // 根据上下文创建 TDI 端点对象
        status = ZwCreateFile(&hTdiEndPoint, GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
            &ObjectAttributes, &iosb, NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
            FILE_OPEN, 0, pContextEaBuffer, ulContextEaBufferLength);
        // 根据 TDI 端点对象句柄获取对应的端点文件对象
        status = ObReferenceObjectByHandle(hTdiEndPoint,
            FILE_ANY_ACCESS, 0, KernelMode, &pTdiEndPointFileObject, NULL);

        // 设置事件
        KeInitializeEvent(&irpCompleteEvent, NotificationEvent, FALSE);
        // 将 TDI 端点与本地地址对象关联，创建 TDI 的 I/O 请求包:TDI_ASSOCIATE_ADDRESS
        pIrp = TdiBuildInternalDeviceControlIrp(TDI_ASSOCIATE_ADDRESS,
            pTdiAddressDevObj, pTdiEndPointFileObject, &irpCompleteEvent, &iosb);
        // 拓展 I/O 请求包
        TdiBuildAssociateAddress(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL,
hTdiAddress);

        // 设置完成实例的回调函数
        IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &irpCompleteEvent, TRUE, TRUE,
TRUE);
        // 发送 I/O 请求包并等待执行
        status = IoCallDriver(pTdiAddressDevObj, pIrp);
        if (STATUS_PENDING == status)
        {
            KeWaitForSingleObject(&irpCompleteEvent, Executive, KernelMode, FALSE, NULL);
        }
        // 返回数据 (略)
        // 释放内存 (略)
    }
```

2. TDI TCP 连接

在 TDI 初始化完成之后，驱动程序便向 TCP 服务端发送连接请求，并建立 TCP 连接。主要操作就是要构造一个包含服务器 IP 地址以及监听端口的 IRP，然后发送给驱动程序执行。基于 TDI 的 TCP 连接的具体实现流程如下所示。

首先，直接调用 TdiBuildInternalDeviceControlIrp 宏创建 IRP，设置 IRP 的消息类型为 TDI_CONNECT，这表示建立 TCP 连接。

然后，构建 TDI 连接信息结构 TDI_CONNECTION_INFORMATION，设置 IP 地址的相关信息 TA_IP_ADDRESS，它包括通信协议类型、服务器的 IP 地址以及服务器监听端口号等。并调用 TdiBuildConnect 宏将 TDI 连接信息的结构数据添加到 IRP 的 I/O 堆栈空间中。

最后，调用 IoCallDriver 函数向驱动程序发送已构建好的 IRP，并创建完成回调函数，等待系统处理 IRP。系统处理完毕后，驱动程序便与服务端程序成功地建立了 TCP 连接。

TCP 连接的具体实现代码如下所示。

```
    // TDI TCP 连接服务器
    NTSTATUS        TdiConnection(PDEVICE_OBJECT        pTdiAddressDevObj,        PFILE_OBJECT
pTdiEndPointFileObject, LONG *pServerIp, LONG lServerPort)
    {
        // 变量 (略)
        // 创建连接事件
        KeInitializeEvent(&connEvent, NotificationEvent, FALSE);
        // 创建 TDI 连接 I/O 请求包:TDI_CONNECT
        pIrp     =     TdiBuildInternalDeviceControlIrp(TDI_CONNECT,     pTdiAddressDevObj,
pTdiEndPointFileObject, &connEvent, &iosb);
        // 初始化服务器 IP 地址与端口
        serverIpAddr = INETADDR(pServerIp[0], pServerIp[1], pServerIp[2], pServerIp[3]);
        serverPort = HTONS(lServerPort);
        serverTaIpAddr.TAAddressCount = 1;
        serverTaIpAddr.Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
        serverTaIpAddr.Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
        serverTaIpAddr.Address[0].Address[0].sin_port = serverPort;
        serverTaIpAddr.Address[0].Address[0].in_addr = serverIpAddr;
        serverConnection.UserDataLength = 0;
        serverConnection.UserData = 0;
        serverConnection.OptionsLength = 0;
        serverConnection.Options = 0;
        serverConnection.RemoteAddressLength = sizeof(TA_IP_ADDRESS);
        serverConnection.RemoteAddress = &serverTaIpAddr;
        // 把上述的地址与端口信息增加到 I/O 请求包中,增加连接信息
        TdiBuildConnect(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL, NULL,
&serverConnection, 0);

        // 设置完成实例回调函数
        IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &connEvent, TRUE, TRUE, TRUE);
        // 发送 I/O 请求包并等待执行
        status = IoCallDriver(pTdiAddressDevObj, pIrp);
        if (STATUS_PENDING == status)
        {
            KeWaitForSingleObject(&connEvent, Executive, KernelMode, FALSE, NULL);
        }
```

```
    return status;
}
```

3. TDI TCP 数据发送

在成功建立 TCP 连接之后，客户端程序与服务端程序可以相互通信进行数据交互了。基于 TDI 实现的 TCP 数据发送的主要操作便是构造一个发送数据的 IRP，并向 IRP 添加要发送的数据，然后发送给驱动程序处理即可。具体的实现流程如下所示。

首先，直接调用 TdiBuildInternalDeviceControlIrp 宏创建 IRP，设置 IRP 的消息类型为 TDI_SEND，这表示发送数据。

然后，调用 IoAllocateMdl 函数将要发送的数据创建一份新的映射并获取分配到的 MDL 结构，因为驱动程序接下来需要调用 TdiBuildSend 宏来将 MDL 结构数据添加到 IRP 的 I/O 堆栈空间中，以此传递发送的数据信息。

最后，调用 IoCallDriver 函数向驱动程序发送已构建好的 IRP，并创建完成回调函数以等待系统处理 IRP。处理完毕后，要记得调用 IoFreeMdl 函数来释放 MDL。

TCP 数据发送的具体实现代码如下所示。

```
// TDI TCP 发送信息
NTSTATUS TdiSend(PDEVICE_OBJECT pTdiAddressDevObj, PFILE_OBJECT pTdiEndPointFileObject,
PUCHAR pSendData, ULONG ulSendDataLength)
{
    // 变量 (略)
    // 初始化事件
    KeInitializeEvent(&sendEvent, NotificationEvent, FALSE);
    // 创建 I/O 请求包:TDI_SEND
    pIrp      =      TdiBuildInternalDeviceControlIrp(TDI_SEND,      pTdiAddressDevObj,
pTdiEndPointFileObject, &sendEvent, &iosb);
    // 创建 MDL
    pSendMdl = IoAllocateMdl(pSendData, ulSendDataLength, FALSE, FALSE, pIrp);
    MmProbeAndLockPages(pSendMdl, KernelMode, IoModifyAccess);
    // 拓展 I/O 请求包,添加发送信息
    TdiBuildSend(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL, pSendMdl,
0, ulSendDataLength);

    // 设置完成实例回调函数
    IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &sendEvent, TRUE, TRUE, TRUE);
    // 发送 I/O 请求包并等待执行
    status = IoCallDriver(pTdiAddressDevObj, pIrp);
    if (STATUS_PENDING == status)
    {
        KeWaitForSingleObject(&sendEvent, Executive, KernelMode, FALSE, NULL);
    }
    // 释放 MDL (略)
    return status;
}
```

4. TDI TCP 数据接收

基于 TDI 实现的 TCP 数据接收的具体实现流程和数据发送类似，同样是构造数据接收的 IRP，设置接收数据缓冲区，将 IRP 发送给驱动程序处理即可。具体的数据接收实现流程如下

所示。

首先，直接调用 TdiBuildInternalDeviceControlIrp 宏创建 IRP，设置 IRP 的消息类型为 TDI_RECV，这表示接收数据。

然后，调用 IoAllocateMdl 函数来将缓冲区中的数据创建一份新的映射并获取分配到的 MDL 结构，因为驱动程序接下来需要调用 TdiBuildReceive 宏来将 TDI 接收数据缓冲区的 MDL 结构数据添加到 IRP 的 I/O 堆栈空间中，以此传递接收数据缓冲区的信息。

最后，调用 IoCallDriver 函数向驱动程序发送已构建好的 IRP，并创建完成回调函数以等待系统处理 IRP。处理完毕后，要记得调用 IoFreeMdl 函数来释放 MDL。

TCP 数据接收的具体实现代码如下所示。

```
    // TDI TCP 接收信息
    ULONG_PTR TdiRecv(PDEVICE_OBJECT pTdiAddressDevObj, PFILE_OBJECT pTdiEndPointFileObject,
PUCHAR pRecvData, ULONG ulRecvDataLength)
    {
        // 变量 (略)
        // 初始化事件
        KeInitializeEvent(&recvEvent, NotificationEvent, FALSE);
        // 创建 I/O 请求包:TDI_SEND
        pIrp      =      TdiBuildInternalDeviceControlIrp(TDI_RECV,      pTdiAddressDevObj,
pTdiEndPointFileObject, &recvEvent, &iosb);
        // 创建 MDL
        pRecvMdl = IoAllocateMdl(pRecvData, ulRecvDataLength, FALSE, FALSE, pIrp);
        MmProbeAndLockPages(pRecvMdl, KernelMode, IoModifyAccess);
        // 拓展 I/O 请求包,添加发送信息
        TdiBuildReceive(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL, pRecvMdl,
TDI_RECEIVE_NORMAL, ulRecvDataLength);

        // 设置完成实例回调函数
        IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &recvEvent, TRUE, TRUE, TRUE);
        // 发送 I/O 请求包并等待执行
        status = IoCallDriver(pTdiAddressDevObj, pIrp);
        if (STATUS_PENDING == status)
        {
            KeWaitForSingleObject(&recvEvent, Executive, KernelMode, FALSE, NULL);
        }
        // 获取实际接收的数据大小
        ulRecvSize = pIrp->IoStatus.Information;
        // 释放 MDL (略)
        return status;
    }
```

5. TDI 关闭

所谓的关闭 TDI，主要是负责资源数据的释放和清理工作。调用 ObDereferenceObject 函数释放端点文件对象资源，调用 ZwClose 函数关闭端点对象句柄以及本地地址对象句柄。

TDI 关闭的具体实现代码如下所示。

```
    // TDI 关闭释放
    VOID   TdiClose(PFILE_OBJECT   pTdiEndPointFileObject,   HANDLE   hTdiAddress,   HANDLE
hTdiEndPoint)
    {
```

```
    if (pTdiEndPointFileObject)
    {
        ObDereferenceObject(pTdiEndPointFileObject);
    }
    if (hTdiEndPoint)
    {
        ZwClose(hTdiEndPoint);
    }
    if (hTdiAddress)
    {
        ZwClose(hTdiAddress);
    }
}
```

18.3.2 测试

在 64 位 Windows 10 操作系统上，先运行 TCP 服务端程序 ChatServer.exe，设置服务端程序的 IP 地址以及监听端口分别为 127.0.0.1 和 12345，并开始监听。然后，直接加载并运行上述驱动程序，程序连接监听状态的服务端，并向服务端程序发送数据“I am Demon`Gan--->From TDI”。服务端程序成功与驱动程序建立 TCP 连接，并成功接收到来自驱动程序发送的数据。处于用户层的服务端程序向驱动程序发送数据“nice to meet you, Demon”，驱动程序也能成功接收。所以，基于 TDI 通信的驱动程序测试成功，如图 18-3 所示。

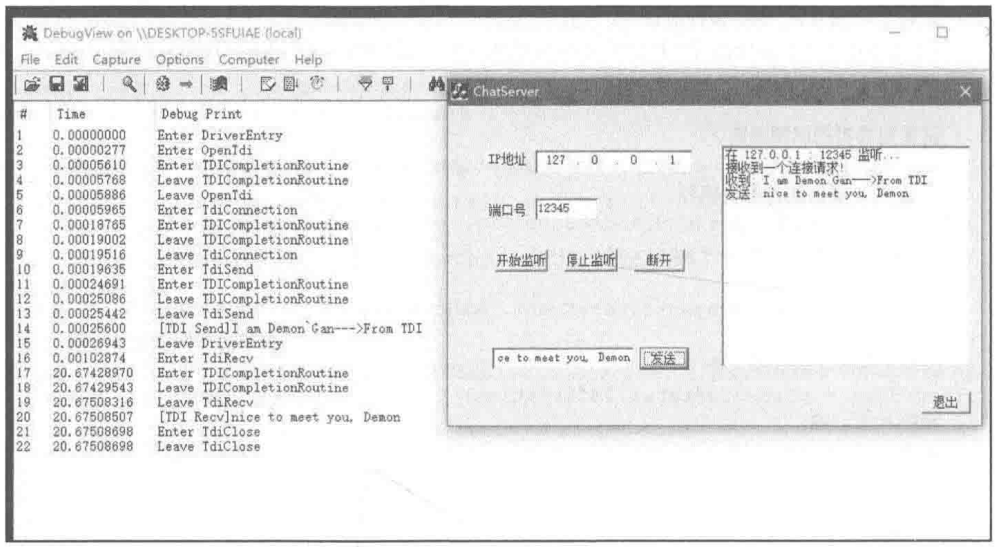

图 18-3 基于 TDI 的 TCP 通信测试

打开 cmd.exe 命令行窗口后，输入命令 netstat -ano 来查看网络连接情况以及对应进程的 PID，命令结果如图 18-4 所示，从中可以知道，与服务端程序建立通信连接的进程 PID 为 4，即为 system.exe 进程，因为是驱动程序与用户程序建立的 TCP 连接，所以，进程 PID 显示为 4。

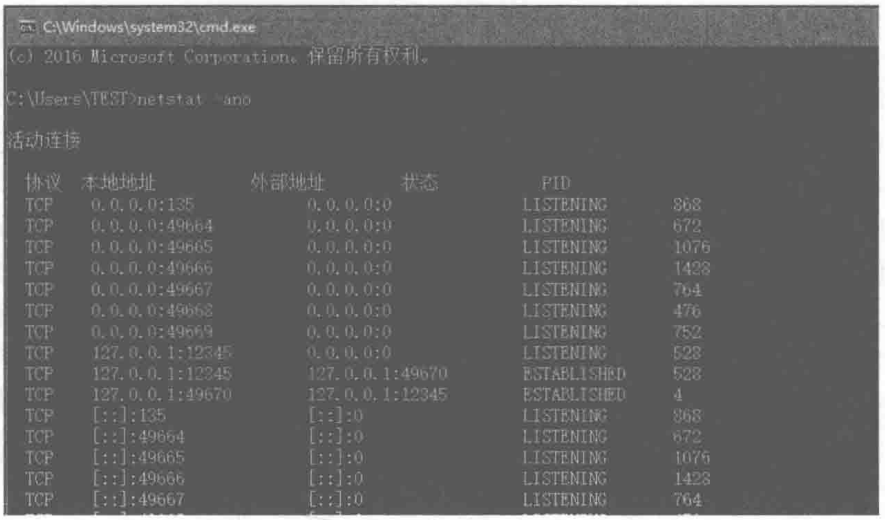

图 18-4 建立连接的 TCP 及其进程 PID 信息

18.3.3 小结

基于 TDI 的 TCP 客户端的实现原理实际上就是通过构造不同信息的 TDI 的 I/O 请求包，携带不同的参数数据，发送给驱动函数进行处理来实现的。实现该程序的关键在于 TDI 的 I/O 请求包的构建上。

在通信的过程中，要注意及时调用 IoFreeMdl 函数来释放创建的 MDL。同时，驱动程序可以通过调用 PsCreateSystemThread 函数创建一个多线程，循环接收来自服务端程序的数据。

18.4 强制结束进程

无论是哪一款杀毒软件，进程保护总是一项必不可少的功能，至少要对杀毒软件自身的进程进行保护。设想一下，如果杀毒软件没有对自己的进程进行保护，那么任意一个程序都能直接强制结束杀毒软件的进程，即使这个杀毒软件拥有强大的扫描检测能力，它也不能发挥杀毒软件的作用。所以，自保对于任何一个杀毒软件来说都是至关重要的。

病毒木马要与杀毒软件进行对抗，最正面、最暴力的无过于关掉杀毒软件进程，删除杀毒软件中的文件数据。但这样会引起用户注意，增加暴露的风险。

实现进程强制结束的方法有很多，常用的有 ZwTerminateProcess 内核函数方法，也有进程内存清零方法。接下来，本节将介绍一种更加有效的强制结束进程方法，它是利用未导出函数 PspTerminateThreadByPointer 实现的。

18.4.1 实现原理

线程是进程中执行运算的最小单位，是进程中的一个实体，是系统独立调度和分派的基本单位。线程自己不拥有系统资源，在运行中只拥有一点必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程，同一进程中的多个线程间可以并发执行。

也就是说，当一个进程中的所有线程都结束的时候，这个进程也就没有了存在的意义，它也会随之结束了。这便是本书介绍的强制结束进程的实现原理，即把进程中的线程都杀掉，从而让进程消亡，实现强制结束进程的效果。

Windows 提供了一个可导出的内核函数 PsTerminateSystemThread，它可帮助程序结束线程的操作。但是，由于该导出函数功能具有特殊性，所以它会成为杀毒软件重点监控的对象，防止杀毒软件结束自己的线程。

通过对 PsTerminateSystemThread 导出函数进行逆向跟踪，发现该函数实际上通过调用未导出的内核函数 PspTerminateThreadByPointer 实现结束线程操作的。如果程序可以获取到未导出函数 PspTerminateThreadByPointer 的地址，就可以直接调用它来结束线程。由于这个操作较为底层，所以可以绕过绝大部分的线程保护，从而实现强制结束线程。

PspTerminateThreadByPointer 的函数声明为：

```
NTSTATUS PspTerminateThreadByPointer(
    PETHREAD pEThread,
    NTSTATUS ntExitCode,
    BOOLEAN bDirectTerminate)
```

但要注意，对 PspTerminateThreadByPointer 的函数指针声明的调用约定：

```
// 32 位系统中
typedef NTSTATUS(*PSPTERMINATETHREADBYPOINTER_X86) (
    PETHREAD pEThread,
    NTSTATUS ntExitCode,
    BOOLEAN bDirectTerminate);

// 64 位系统中
typedef NTSTATUS(__fastcall *PSPTERMINATETHREADBYPOINTER_X64) (
    PETHREAD pEThread,
    NTSTATUS ntExitCode,
    BOOLEAN bDirectTerminate);
```

如何获取未导出函数 PspTerminateThreadByPointer 的地址呢？上面提到过，导出函数 PsTerminateSystemThread 会调用 PspTerminateThreadByPointer 未导出函数实现具体功能。

其中，在 64 位 Windows 10 系统下，使用 WinDbg 查看导出的内核函数 PsTerminate SystemThread 的逆向分析代码如下所示。

```
nt!PsTerminateSystemThread:
fffff800`83904518 8bd1          mov     edx, ecx
fffff800`8390451a 65488b0c2588010000 mov   rcx, qword ptr gs : [188h]
fffff800`83904523 f7417400080000  test    dword ptr[rcx + 74h], 800h
```

```
fffff800`8390452a 7408      je      nt!PsTerminateSystemThread + 0x1c (fffff800`83904534)
fffff800`8390452c 41b001        mov     r8b, 1
fffff800`8390452f e978d9fcff   jmp     nt!PspTerminateThreadByPointer(fffff800`838d1eac)
fffff800`83904534 b80d0000c0     mov     eax, 0C000000Dh
fffff800`83904539 c3            ret
```

由上面代码可以知道，通过扫描 PsTerminateSystemThread 内核函数中的特征码，可获取 PspTerminateThreadByPointer 函数的地址偏移，再根据偏移计算出该函数的地址。

其中，在不同系统中，PspTerminateThreadByPointer 函数的特征码也会不同。下面使用 WinDbg 在 Windows 7、Windows 8.1 以及 Windows 10 系统上逆向分析导出函数 PsTerminateSystemThread，并总结得到特征码，如下所示。

	Windows 7	Windows 8.1	Windows 10
32 位	E8	E8	E8
64 位	E8	E9	E9

强制结束进程的具体实现流程如下所示。

首先，根据特征码扫描内存，获取 PspTerminateThreadByPointer 函数地址。

然后，调用 PsLookupProcessByProcessId 函数，根据将要结束的进程 ID 获取对应的进程结构对象 EPROCESS。

最后，遍历所有的线程 ID，并调用 PsLookupThreadByThreadId 函数根据线程 ID 获取对应的线程结构 ETHREAD。再调用函数 PsGetThreadProcess 获取线程结构 ETHREAD 对应的进程结构 EPROCESS。这时，要判断该进程是否为指定要结束的进程，若是，则调用 PspTerminateThreadByPointer 函数结束线程；否则，继续遍历下一个线程 ID。重复遍历线程 ID 的操作，直到把指定进程的所有线程都结束掉。

这样，就可以查杀指定进程的所有线程，线程全部结束之后，进程也随之结束。

18.4.2 编码实现

获取 PspTerminateThreadByPointer 函数地址的具体实现代码如下所示。

```
// 根据特征码获取 PspTerminateThreadByPointer 数组地址
PVOID SearchPspTerminateThreadByPointer(PUCHAR pSpecialData, ULONG ulSpecialDataSize)
{
    // 变量 (略)
    // 先获取 PsTerminateSystemThread 函数地址
    RtlInitUnicodeString(&ustrFuncName, L"PsTerminateSystemThread");
    pPsTerminateSystemThread = MmGetSystemRoutineAddress(&ustrFuncName);
    if (NULL == pPsTerminateSystemThread)
    {
        ShowError("MmGetSystemRoutineAddress", 0);
        return pPspTerminateThreadByPointer;
    }
    // 然后，查找 PspTerminateThreadByPointer 函数地址
    pAddress = SearchMemory(pPsTerminateSystemThread,
        (PVOID)((PUCHAR)pPsTerminateSystemThread + 0xFF),
```

```
            pSpecialData, ulSpecialDataSize);
        if (NULL == pAddress)
        {
            ShowError("SearchMemory", 0);
            return pPspTerminateThreadByPointer;
        }
        // 先获取偏移，再计算地址
        lOffset = *(PLONG)pAddress;
        pPspTerminateThreadByPointer = (PVOID)((PUCHAR)pAddress + sizeof(LONG) + lOffset);
        return pPspTerminateThreadByPointer;
    }
```

强制结束指定进程的具体实现代码如下所示。

```
    // 强制结束指定进程
    NTSTATUS ForceKillProcess(HANDLE hProcessId)
    {
        // 变量（略）
    #ifdef _WIN64
        // 64 位系统中
        typedef NTSTATUS(__fastcall *PSPTERMINATETHREADBYPOINTER) (PETHREAD pEThread,
NTSTATUS ntExitCode, BOOLEAN bDirectTerminate);
    #else
        // 32 位系统中
        typedef NTSTATUS(*PSPTERMINATETHREADBYPOINTER) (PETHREAD pEThread, NTSTATUS
ntExitCode, BOOLEAN bDirectTerminate);
    #endif
        // 获取 PspTerminateThreadByPointer 函数地址
        pPspTerminateThreadByPointerAddress = GetPspLoadImageNotifyRoutine();
        if (NULL == pPspTerminateThreadByPointerAddress)
        {
            ShowError("GetPspLoadImageNotifyRoutine", 0);
            return FALSE;
        }
        // 获取结束进程的进程结构对象 EPROCESS
        status = PsLookupProcessByProcessId(hProcessId, &pEProcess);
        if (!NT_SUCCESS(status))
        {
            ShowError("PsLookupProcessByProcessId", status);
            return status;
        }
        // 遍历所有线程，并结束所有指定进程的线程
        for (i = 4; i < 0x80000; i = i + 4)
        {
            status = PsLookupThreadByThreadId((HANDLE)i, &pEThread);
            if (NT_SUCCESS(status))
            {
                // 获取线程对应的进程结构对象
                pThreadEProcess = PsGetThreadProcess(pEThread);
                // 结束指定进程的线程
                if (pEProcess == pThreadEProcess)
                {
                    ((PSPTERMINATETHREADBYPOINTER)pPspTerminateThreadByPointerAddress)(pET
hread, 0, 1);
                    DbgPrint("PspTerminateThreadByPointer Thread:%d\n", i);
```

```
            }
            // 凡是 Lookup...，必须 Dereference，否则在某些时候会造成蓝屏
            ObDereferenceObject(pEThread);
        }
    }
    // 凡是 Lookup...，必须 Dereference，否则在某些时候会造成蓝屏
    ObDereferenceObject(pEProcess);
    return status;
}
```

18.4.3 测试

在 64 位 Windows 10 系统下，直接加载并运行上述驱动程序。程序强制结束了指定进程，如图 18-5 所示。

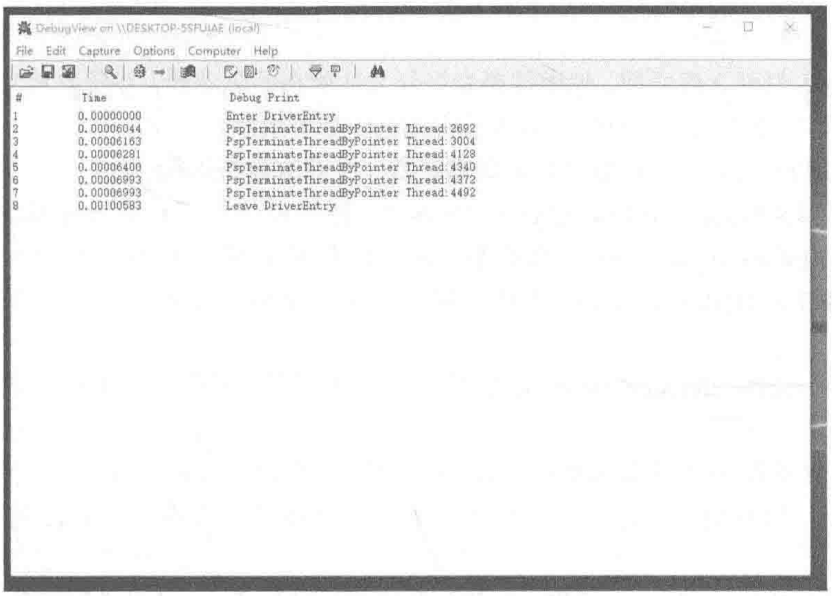

图 18-5 强制结束指定进程

18.4.4 小结

强制结束进程的原理就是通过强制结束进程中的所有线程，从而让进程结束。其中，强制结束线程的操作是通过未导出内核函数 PspTerminateThreadByPointer 实现的。由于该函数较为底层，所以可以绕过绝大部分的线程保护。

当驱动程序调用 PsLookupProcessByProcessId 和 PsLookupThreadByThreadId 等 LookupXXX 系列函数获取对象的时候，都需要调用 ObDereferenceObject 函数释放对象，否则在某些时候会造成蓝屏。

18.5 文件保护

文件保护技术不论是对于病毒木马还是杀毒软件来说，都至关重要。要想在内核层下实现文件防删除，通常是通过挂钩内核函数来实现的，它通过判断是否要删除保护文件，从而拒绝操作。本节将介绍一种不需要挂钩操作即可实现的文件保护方法，它通过发送 IRP 信息打开文件并获取文件句柄，但不关闭文件句柄，使文件句柄一直保持打开状态。在文件句柄没有关闭释放的情况，不能删除文件，从而实现文件防删除。

18.5.1 实现过程

在前面章节就已经介绍过可发送 IRP 管理操作文件。直接发送 IRP 对文件进行操作可以避免一些钩子的干扰。

在发送 IRP 打开文件之前，程序需要先打开文件所在的驱动器，并获取其文件系统驱动的设备对象以及物理磁盘的设备对象。具体的实现流程如下所示。

首先，调用 IoCreateFile 函数打开磁盘驱动器，并获取设备句柄。

然后，调用 ObReferenceObjectByHandle 函数，根据设备句柄获取设备对象的主体内容。本节获取的是*IoFileObjectType 对象类型，对应的数据类型为 PFILE_OBJECT。这样，程序就可以从 PFILE_OBJECT 的 Vpb 中获取其文件系统驱动的设备对象以及物理磁盘的设备对象。

最后，调用 ObDereferenceObject 函数释放文件对象内容，调用 ZwClose 函数关闭获取的设备句柄。

当获取了驱动器中文件系统驱动的设备对象以及物理磁盘的设备对象之后，程序就可以创建 IRP_MJ_CREATE 消息，发送到系统中以打开相应的文件了。具体的实现流程如下所示。

首先，调用 ObCreateObject 函数，创建一个*IoFileObjectType 文件类型对象。

然后，调用 IoAllocateIrp 函数，根据文件系统驱动的设备对象中栈大小 DeviceObject->StackSize 申请一块 IRP 数据空间。

接着，调用 KeInitializeEvent 初始化一个内核同步事件。并开始构造一个 IRP，其过程如下所示。

（1）对文件对象数据结构 FILE_OBJECT 进行设置。

（2）调用 SeCreateAccessState 函数创建访问状态。

（3）设置安全上下文 IO_SECURITY_CONTEXT 数据，并调用 IoGetNextIrpStackLocation 函数获取 IRP 的堆栈空间数据，并对堆栈空间数据进行设置。

最后，调用 IoSetCompletionRoutine 为 IRP 设置一个完成回调函数，以用于 IRP 的清理工作。再调用 IofCallDriver 函数，将 IRP 发送到系统中进行处理，并打开文件。保存文件对象，在后续关闭文件的时候，需要用到该文件对象。

这样，程序就可以打开文件，获取文件对象了。在没有调用 ObDereferenceObject 函数释放文件对象之前，文件一直处于打开状态，不能被删除。从而，实现了文件。

发送 IRP 打开文件的具体代码实现，可以参考前面章节中介绍的通过 IRP 创建或打开文件的实现代码，在此就不重复介绍了。保护文件的具体实现代码如下所示。

```
PFILE_OBJECT ProtectFile(UNICODE_STRING ustrFileName)
{
    PFILE_OBJECT pFileObject = NULL;
    IO_STATUS_BLOCK iosb = { 0 };
    NTSTATUS status = STATUS_SUCCESS;
    // 创建或者打开文件
    status = IrpCreateFile(&pFileObject, DELETE | FILE_READ_ATTRIBUTES | SYNCHRONIZE,
        &ustrFileName, &iosb, NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ |
        FILE_SHARE_WRITE, FILE_OPEN, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return pFileObject;
    }
    return pFileObject;
}
```

关闭文件保护的实现代码如下所示。

```
BOOLEAN UnprotectFile(PFILE_OBJECT pFileObject)
{
    // 释放文件对象
    if (pFileObject)
    {
        ObDereferenceObject(pFileObject);
    }
    return TRUE;
}
```

18.5.2 测试

在 64 位 Windows 10 操作系统上，直接加载并运行上述驱动程序，以保护 C:\520.exe 文件不被删除。在驱动程序成功执行后，直接删除 520.exe 文件，删除失败，系统提示文件正在使用，如图 18-6 所示。关闭文件保护后，文件可正常删除。

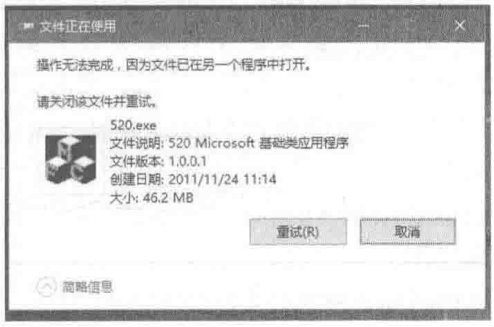

图 18-6 文件删除失败

18.5.3 小结

该文件保护程序的核心原理是通过发送 IRP 打开文件而使文件句柄处于打开状态，系统在删除文件的时候，会检查文件打开的句柄数是否为零，若不为零，则拒绝删除操作。

在编码实现的过程中需要注意的是，在构造打开文件 IRP 的时候，文件对象的文件名 pFileObj->FileName，需要使用 ExAllocPool 申请 MAX_NAME_SPACE 大小字节的非分页内存空间来存储，这样，程序才能正常执行。否则，可能会导致蓝屏。

18.6 文件强删

在平时使用计算机的过程中，遇到删不掉的文件的时候，我们会通过一些软件提供的强制删除文件功能来删除顽固的文件。文件强删技术对于杀毒软件来说是清除病毒木马的武器，在扫描器检测出恶意文件的时候，就需要强删功能来清除恶意文件。而对于病毒木马来说，反过来可以用强删技术来强制删除系统保护文件或是受杀毒软件保护的文件。

本节将会介绍文件强删技术，即使文件正在运行，它也能强制删除本地文件。

18.6.1 实现过程

当文件是 PE 文件而且已经加载到内存中的时候，在正常情况下这是无法通过资源管理器 explorer.exe 来删除本地文件的，因为在删除运行的文件或者加载的 DLL 文件的时候，系统会调用 MmFlushImageSection 内核函数来检测文件是否处于运行状态，若是，则拒绝删除操作。其原理主要是检查文件对象中的 PSECTION_OBJECT_POINTERS 结构数据来判断该文件是否处于运行状态、是否可以删除。

在发送 IRP 删除文件的时候，系统同样会判断文件的属性是否为只读，若是只读属性则会拒绝删除操作。

根据上述的删除原理，文件强制删除的具体实现流程如下所示。

首先，发送 IRP 打开删除文件，并获取文件对象。

然后，发送 IRP 设置文件属性，属性类型为 FileBasicInformation，将文件属性重新设置为 FILE_ATTRIBUTE_NORMAL，以防止原来的文件属性为只读属性。并保存文件对象中 PSECTION_OBJECT_POINTERS 结构的值，保存完成后，再对该结构进行清空处理。

接着，发送 IRP 设置文件属性，属性类型为 FileDispositionInformation，实现删除文件操作。这样，即使是运行中的文件也能被强制删除。

最后，还原文件对象中的 PSECTION_OBJECT_POINTERS 结构，并调用 ObDereferenceObject 函数释放文件对象，完成清理工作。

实现文件强制删除的实现代码如下所示。

```
// 强制删除文件
```

```
NTSTATUS ForceDeleteFile(UNICODE_STRING ustrFileName)
{
    // 变量 (略)
    // 发送 IRP 打开文件
    status = IrpCreateFile(&pFileObject, GENERIC_READ | GENERIC_WRITE, &ustrFileName,
        &iosb, NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ | FILE_SHARE_WRITE |
FILE_SHARE_DELETE,
        FILE_OPEN, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        DbgPrint("IrpCreateFile Error[0x%X]\n", status);
        return FALSE;
    }
    // 发送 IRP 设置文件属性，去掉只读属性，修改为 FILE_ATTRIBUTE_NORMAL
    RtlZeroMemory(&fileBaseInfo, sizeof(fileBaseInfo));
    fileBaseInfo.FileAttributes = FILE_ATTRIBUTE_NORMAL;
    status = IrpSetInformationFile(pFileObject, &iosb, &fileBaseInfo,
sizeof(fileBaseInfo), FileBasicInformation);
    if (!NT_SUCCESS(status))
    {
        DbgPrint("IrpSetInformationFile[SetInformation] Error[0x%X]\n", status);
        return status;
    }
    // 清空 PSECTION_OBJECT_POINTERS 结构
    if (pFileObject->SectionObjectPointer)
    {
        // 保存旧值
        pImageSectionObject = pFileObject->SectionObjectPointer->ImageSectionObject;
        pDataSectionObject = pFileObject->SectionObjectPointer->DataSectionObject;
        pSharedCacheMap = pFileObject->SectionObjectPointer->SharedCacheMap;
        // 置为空
        pFileObject->SectionObjectPointer->ImageSectionObject = NULL;
        pFileObject->SectionObjectPointer->DataSectionObject = NULL;
        pFileObject->SectionObjectPointer->SharedCacheMap = NULL;
    }
    // 发送 IRP 设置文件属性，设置删除文件操作
    RtlZeroMemory(&fileDispositionInfo, sizeof(fileDispositionInfo));
    fileDispositionInfo.DeleteFile = TRUE;
    status = IrpSetInformationFile(pFileObject, &iosb, &fileDispositionInfo,
sizeof(fileDispositionInfo), FileDispositionInformation);
    if (!NT_SUCCESS(status))
    {
        DbgPrint("IrpSetInformationFile[DeleteFile] Error[0x%X]\n", status);
        return status;
    }
    //还原旧值
    if (pFileObject->SectionObjectPointer)
    {
        pFileObject->SectionObjectPointer->ImageSectionObject = pImageSectionObject;
        pFileObject->SectionObjectPointer->DataSectionObject = pDataSectionObject;
        pFileObject->SectionObjectPointer->SharedCacheMap = pSharedCacheMap;
    }
    // 关闭文件对象
    ObDereferenceObject(pFileObject);
    return status;
```

```
}
```

18.6.2 测试

在 64 位 Windows 10 操作系统上，运行 C:\520.exe 程序后，直接加载并运行上述驱动程序，强制删除正在运行的 520.exe 文件。520.exe 本地文件成功删除，如图 18-7 所示。

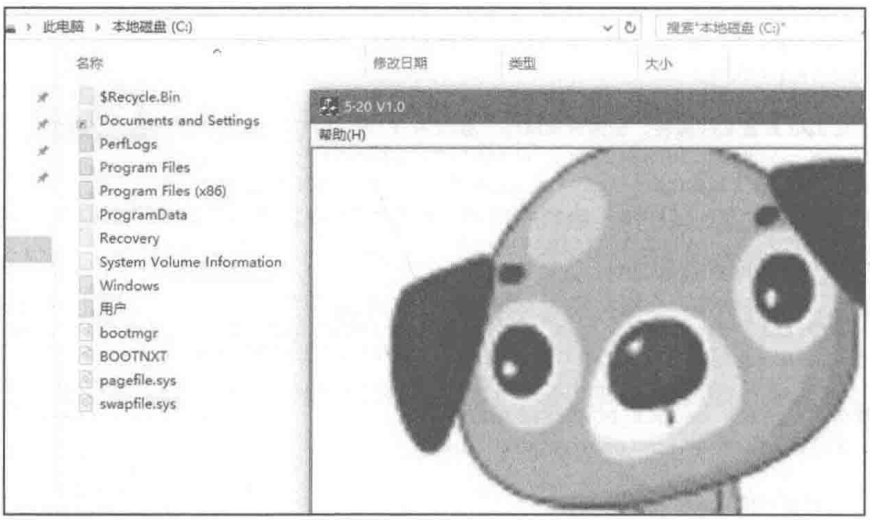

图 18-7 文件强制删除

18.6.3 小结

本节介绍的文件强删技术的实现原理是，通过发送 IRP 打开文件，获取文件对象；发送 IRP 设置文件属性，去掉只读属性；清空文件对象中的 PSECTION_OBJECT_POINTERS 结构，使从而删除运行中的程序文件；最后发送 IRP 删除文件并释放文件对象。

本节介绍的文件强删技术可以删除正在运行的 exe 文件，但是不适用于前面介绍的发送 IRP 打开文件而不关闭的文件。要想强制删除发送 IRP 打开文件不关闭的文件，需要关闭打开的文件对象。可以利用 XCB 解锁技术，通过硬编码定位出文件对象中 SCB（Stream Control Block）、FCB（File Control Blokc）、CCB（Context Control Block）结构的 CleanupCount 变量，并将 CleanupCount 都置为 1，再调用 ObDereferenceObject 函数关闭文件对象并释放资源。XCB 解锁技术不仅可以解锁发送 IRP 打开文件的保护方式，同样适用于硬链接文件保护。

附录

函数一览表

序号	章节	函数	作用
用户篇			
1	2.1	CreateMutex	创建或打开一个已命名或未命名的互斥对象
2	2.3	FindResource	确定模块中指定类型和名称的资源所在位置
3	2.3	SizeofResource	获取指定资源的字节数
4	2.3	LoadResource	装载指定资源到全局存储器
5	2.3	LockResource	锁定资源并得到资源在内存中第一个字节的指针
6	3.1	SetWindowsHookEx	将程序定义的钩子函数安装到挂钩链中
7	3.2	OpenProcess	打开现有的本地进程对象
8	3.2	VirtualAllocEx	在指定进程的虚拟地址空间内申请内存
9	3.2	WriteProcessMemory	在指定的进程中将数据写入内存
10	3.2	CreateRemoteThread	在另一个进程的虚拟地址空间中创建运行的线程
11	3.4	QueueUserAPC	将用户模式中的异步过程调用（APC）对象添加到指定线程的 APC 队列中
12	4.1	WinExec	运行指定的应用程序
13	4.1	ShellExecute	运行一个外部程序
14	4.1	CreateProcess	创建一个新进程及主线程
15	4.2	WTSGetActiveConsoleSessionId	检索控制台会话的标识符 Session Id
16	4.2	WTSQueryUserToken	获取由 Session Id 指定的登录用户的主访问令牌
17	4.2	DuplicateTokenEx	创建一个新的访问令牌
18	4.2	CreateEnvironmentBlock	检索指定用户的环境变量
19	4.2	CreateProcessAsUser	创建一个新进程及主线程
20	5.1	RegOpenKeyEx	打开一个指定的注册表键
21	5.1	RegSetValueEx	在注册表项下设置指定值的数据和类型
22	5.2	SHGetSpecialFolderPath	获取指定的系统路径

（续表）

序号	章节	函数	作用
用户篇			
23	5.4	OpenSCManager	建立一个到服务控制管理器的连接，并打开指定的数据库
24	5.4	CreateService	创建一个服务对象，并将其添加到指定的服务控制管理器中
25	5.4	OpenService	打开一个已经存在的服务
26	5.4	StartService	启动服务
27	5.4	StartServiceCtrlDispatcher	将服务进程的主线程连接到服务控制管理器
28	6.1	OpenProcessToken	打开与进程关联的访问令牌
29	6.1	LookupPrivilegeValue	查看系统权限的特权值
30	6.1	AdjustTokenPrivileges	启用或禁用指定访问令牌中的权限
31	7.1	NtQueryInformationProcess	获取指定进程的信息
32	7.2	GetThreadContext	获取指定线程的上下文
33	7.2	SetThreadContext	设置指定线程的上下文
34	7.2	ResumeThread	减少线程的暂停计数
35	7.3	ZwQuerySystemInformation	获取指定的系统信息
36	8.1	RtlGetCompressionWorkSpace Size	确定 RtlCompressBuffer 和 RtlDecompressFragment 函数的工作空间缓冲区的正确大小
37	8.1	RtlCompressBuffer	压缩一个缓冲区
38	8.1	RtlDecompressBuffer	解压缩整个压缩缓冲区
39	9.1.1	CryptAcquireContext	用于获取特定加密服务提供程序（CSP）内特定密钥容器的句柄
40	9.1.1	CryptCreateHash	创建一个空 HASH 对象
41	9.1.1	CryptHashData	将数据添加到 HASH 对象，并进行 HASH 计算
42	9.1.1	CryptGetHashParam	从 HASH 对象中获取指定参数值
43	9.1.2	CryptDeriveKey	从基础数据值派生出的加密会话密钥
44	9.1.2	CryptEncrypt	由 CSP 模块保存的密钥指定的加密算法来加密数据
45	9.1.2	CryptDecrypt	解密数据
46	9.1.3	CryptGenKey	随机生成加密会话密钥或公钥/私钥对
47	9.1.3	CryptExportKey	以安全的方式从加密服务提供程序中导出加密密钥或密钥对
48	9.1.3	CryptImportKey	将密钥从密钥 BLOB 导入到加密服务提供程序中
49	10.1.1	Socket	根据指定的地址族、数据类型和协议来分配一个套接口函数

（续表）

序号	章节	函数	作用
用户篇			
50	10.1.1	bind	将本地地址与套接字相关联
51	10.1.1	htons	将整型变量从主机字节顺序转变成网络字节顺序
52	10.1.1	inet_addr	将一个点分十进制的 IP 转换成一个长整型数
53	10.1.1	listen	将一个套接字置于正在监听传入连接的状态
54	10.1.1	accept	允许在套接字上尝试连接
55	10.1.1	send	在建立连接的套接字上发送数据
56	10.1.1	recv	从连接的套接字或绑定的无连接套接字中接收数据
57	10.1.2	sendto	将数据发送到特定的目的地
58	10.1.2	recvfrom	接收数据报并存储源地址
59	10.2.1	InternetOpen	初始化一个应用程序，以使用 WinInet 函数
60	10.2.1	InternetConnect	建立互联网的连接
61	10.2.1	FtpOpenFile	访问 FTP 服务器上的远程文件以执行读取或写入
62	10.2.1	InternetWriteFile	将数据写入打开的互联网文件中
63	10.2.2	InternetReadFile	从打开的句柄中读取数据
64	10.3.1	HttpOpenRequest	创建一个 HTTP 请求句柄
65	10.3.1	HttpSendRequestEx	将指定的请求发送到 HTTP 服务器
66	10.3.1	HttpQueryInfo	该函数查询有关 HTTP 请求的信息
67	11.1	CreateToolhelp32Snapshot	获取进程信息为指定的进程、进程使用的堆、模块以及线程建立一个快照
68	11.1	Process32First	检索系统快照中遇到的第一个进程信息
69	11.1	Process32Next	检索系统快照中记录的下一个进程信息
70	11.2	FindFirstFile	搜索与特定名称匹配的文件名称或子目录
71	11.2	FindNextFile	继续搜索文件
72	11.3	GetDC	检索指定窗口的客户区域或整个屏幕上显示设备上下文环境的句柄
73	11.3	BitBlt	对指定的源设备环境区域中的像素进行位块（Bit Block）转换
74	11.4	RegisterRawInputDevices	注册提供原始输入数据的设备
75	11.4	GetRawInputData	从指定的设备中获取原始输入
76	11.5	CreatePipe	创建一个匿名管道，并从中得到读写管道的句柄
77	11.7	ReadDirecotryChangesW	监控文件目录
78	11.8	MoveFileEx	使用各种移动选项移动现有的文件或目录

（续表）

序号	章节	函数	作用
内核篇			
79	13.1.1	ZwCreateFile	创建一个新文件或者打开一个已存在的文件
80	13.1.2	ZwDeleteFile	删除指定文件
81	13.1.3	ZwQueryInformationFile	获取有关文件对象的各种信息
82	13.1.4	ExAllocatePool	分配指定类型的池内存
83	13.1.4	ZwReadFile	从打开的文件中读取数据
84	13.1.4	ZwWriteFile	向一个打开的文件写入数据
85	13.1.5	ZwSetInformationFile	更改有关文件对象的各种信息
86	13.1.6	ZwQueryDirectoryFile	在由给定文件句柄指定的目录中获取文件的各种信息
87	13.2.1	IoAllocateIrp	申请创建一个 IRP
88	13.2.1	IoCallDriver	发送一个 IRP 给与指定设备对象相关联的驱动程序
89	14.1.1	ZwCreateKey	创建一个新的注册表项或打开一个现有的注册表项
90	14.1.2	ZwDeleteKey	从注册表中删除一个已打开的注册表键
91	14.1.2	ZwDeleteValueKey	从已打开的注册表键中删除名称的匹配键值
92	14.1.3	ZwSetValueKey	创建或替换注册表键值
93	14.1.4	ZwQueryValueKey	获取注册表键值
94	15.1.1	ZwCreateSection	创建一个节对象
95	15.1.1	ZwMapViewOfSection	将一个节表的视图映射到内核的虚拟地址空间
96	16.1	PsSetCreateProcessNotifyRoutineEx	设置进程回调监控的创建与退出，而且还能控制是否允许进程创建
97	16.2	PsSetLoadImageNotifyRoutine	设置模块加载回调函数，只要完成模块加载就会通知回调函数
98	16.3	CmRegisterCallback	注册一个 RegistryCallback 例程
99	16.4	ObRegisterCallbacks	注册线程、进程和桌面句柄操作的回调函数
100	16.5	FltGetFileNameInformation	返回文件或目录的名称信息
101	16.5	FltParseFileNameInformation	解析 FLT_FILE_NAME_INFORMATION 结构中的内容
102	17.6	FltEnumerateFilters	列举系统中所有注册的 Minifilter 驱动程序

本书详解了注入、启动、权限、HOOK、监控等技术，从技术知识体系上对其细节（如实现原理、编码实战、案例测试等）进行了剖析，降低了学习难度，因此非常适合 Windows 安全、二进制安全、逆向工程等相关领域的从业人员、爱好者阅读。大家可以通过本书对自己的 Windows 安全知识体系进行补充与巩固。值得阅读！

—— 孔韬循（K0r4dji），丁牛科技有限公司首席安全官，破晓安全团队创始人

"黑客"当今已经不再是神秘的代言词，而是攻防技术的象征——如何利用简短的代码实现意想不到的结果，各类病毒木马是如何利用 Windows 的相关 API 技术实现传播、隐藏、启动、复制等操作——黑客把 Windows 的机制运用得淋漓尽致。这本《Windows 黑客编程技术详解》带领读者从用户态到内核态一步一步地了解黑客如何巧妙地利用各类 Windows 机制达成自己的目的，以及从攻与防的角度了解黑客是怎样的一个“神秘群体”。值得推荐！

——朱利军，四叶草信息技术有限公司首席安全官

阐述 Windows 环境下黑客编程常用技术的书籍有很多，但或涉猎不全，或代码健壮性及兼容性较差，而能将以上技术按照难易梯度依次汇集在一本书内，并且所有示例代码均是多系统全平台兼容的，我个人知道的仅此一本。本书内部穿插的精巧的例子值得每个人细细品味，对于软件安全领域的工作人员来说，本书不失为一本案头必备的代码、案例参考工具书。

——任晓珲，十五派信息安全教育创始人，《黑客免杀攻防》作者

在当前网络安全逐渐娱乐化的时代，本书就像一股清流，让我想起 10 余年前和小伙伴一起彻夜研究各种植入、逃逸技术的一幕幕。本书教会你制作网络空间的利刃的方法，这对于技术流的小伙伴来说是一门必修的功课，不过切记要把它用在正道。

——王珩，赛宁网安副总经理，信息安全漏洞门户（vulhub.org.cn）创始人

作者简介

甘迪文，北京邮电大学网络空间安全学院在读研究生，2019 年秋季即将步入清华大学攻读软件工程专业的博士学位，Write-Bug 技术共享平台（www.write-bug.com）创始人。对信息安全领域兴趣颇深，常利用课余时间自学和钻研安全开发技术。擅长 Windows 系统安全程序开发，熟悉 Windows 内核编程，闲来无事之时喜欢开发功能各异的小软件。

异步社区 人民邮电出版社 www.epubit.com

异步社区 www.epubit.com
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：邓洁

分类建议：计算机/软件开发/信息安全
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-49924-0

ISBN 978-7-115-49924-0
定价：108.00 元