windbg 调试器原理的简单分析

[Author] ReverseMan
[Blog] http://hi.baidu.com/reverseman
[QQ/MSN] 705122552 / fanxinghua2314@126.com

为了方便大家的阅读，首先列出本文的目录。

本文目录：

1. 分析原因

在一次调试中，我在 windbg 的调试窗口中在 nt!NtCreateFile 处下了个条件断点，命令大体是这样子的：

```
kd> bp nt!NtCreateFile "j([esp+4]='xxx') '';'g'"
```

然后，等我回到虚拟机中，打开 xueTr 中后发现 SSDT 钩子标签项中有如下的记录：

SSDT | Shadow SSDT | FSD | KBD | 消息钩子 | 内核钩子 | 应用层钩子 | Object钩子 | 系统中断表

序号	函数名称	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
37	NtCreateFile	0x8057264C	inline hook	0x8057264C	C:\WINDOWS\system32\ntoskrnl.exe

那么我在本机中下的条件断点如何会改变虚拟机的状态的呢？而且奇怪的是 xueTr 所发现的 inline hook 的原始函数地址和当前函数地址竟然是一样的。同样是在内核模块中。由于是初学 windbg，并且对 windbg 的调试原理一无所知。所以起了调戏一番的想法，想一探究竟，下面就是我调试的整个过程，过程中某部分可能有更快捷的调试方法和下断方法。我是一边调试一边学习，所以请高手莫要见怪。

2. 分析工具及资源

调试平台：server->windows xp sp3 ,client->Sun virtualbox3.1 + windows xp sp2
调试工具：windbg、OllyDbg1.1
参考资料：ReactOS0.3.11　SRC

3. Windbg 调试原理简介

关于 windbg 的调试原理，我之前一直认为是其所有的功能实现都是在其内部完成的。其实不然，windbg 只是整个调试实现中的接口程序。其核心功能的实现均是在被调试系统内核中实现的。也就是说，windows 系统的内核已经集成了内核调试器的功能。而 windbg 调试器做的工作就是识别用户的命令，然后根据不同命令封装成一定的数据格式，然后通过串口发送给被调试系统的内核。被调试系统内核在接受到数据包后，根据数据包的内容去修改内核中的某些数据，使得内核调试器能够捕获特定的异常。内核调试器捕获到异常后，根据异常的类型以及相关的信息封装成数据包通过串口发送给 windbg.windbg 调试器根据结果向调试者反馈结果。

以上是 windbg 调试器的简单流程介绍。下面根据 ReactOS 的源码来分析下内核调试器的某些原理实现。

NOTE：由于是参考 ReactOS 的源码，而不是真正的 windows 源码。所以分析结果的正确性依赖于 ReactOS 源码和真实 windows 源码相似度的大小。

4. ReactOS 源码剖析

本来想参考 WRK1.2 的源码来着，无奈 WRK 的源码不完整，很多功能代码看不到。所以最终使用 ReactOS 源码做一下简要分析。关于 ReactOS 这个项目的介绍，请 baidu 或者 google 下就 OK 了。下面开始分析 ReactOS 中内核调试器部分的源码。分析过程中的函数大部分是摘录，如需查看完整代码请参考 ReactOS(Version:0.3.11)源码。

首先，是系统初始化的代码，是在函数 KdInitSystem 中实现。

首先默认初始化调试分发函数为 KdpStub：

```
<Kdinit.c>
/* Set the Debug Routine as the Stub for now */
   KiDebugRoutine = KdpStub;
```

首先，KiDebugRoutine 是内核调试器的一个全局的调试函数指针。调试过程中，内核调试器始终通过 KiDebugRoutine 调用真正的调试处理函数。其定义如下：

```
<kdData.c>
  PKDEBUG_ROUTINE KiDebugRoutine = KdpStub;
```

其中，PKDEBUG_ROUTINE 是一个函数指针，定义如下：

```
<kd.h>
typedef
BOOLEAN
(NTAPI *PKDEBUG_ROUTINE)(
   IN PKTRAP_FRAME TrapFrame,
   IN PKEXCEPTION_FRAME ExceptionFrame,
   IN PEXCEPTION_RECORD ExceptionRecord, //异常发生的记录，相关信息保存在此结构中
   IN PCONTEXT Context, //异常发生的上下文
   IN KPROCESSOR_MODE PreviousMode,
   IN BOOLEAN SecondChance //内核异常会处理两次，此变量标识是第几次
);
```

继续看函数 KdpInitSystem 函数的代码：

```
<kdinit.c>
   ......//调试器数据块初始化
   /* Check if we have a loader block */
   if (LoaderBlock)//函数 KdpInitSystem 传入的参数 2，包含系统启动的参数
   {
      /* Get the image entry */
      LdrEntry = CONTAINING_RECORD(LoaderBlock->LoadOrderListHead.Flink,
                                   LDR_DATA_TABLE_ENTRY,
                                   InLoadOrderLinks);
      //CONTAINING_RECORD 宏用于根据结构体某一成员变量获取结构体的起始地址值
      /* Save the Kernel Base */
      PsNtosImageBase = (ULONG_PTR)LdrEntry->DllBase;
      KdVersionBlock.KernBase = (ULONG64)(LONG_PTR)LdrEntry->DllBase;
```

```
        /* Check if we have a command line */
        CommandLine = LoaderBlock->LoadOptions;//从系统盘下 Boot.ini 文件读取得到的
        if (CommandLine)
        {
            /* Upcase it */
            _strupr(CommandLine);
            /* Assume we'll disable KD */
            EnableKd = FALSE;
            /* Check for CRASHDEBUG, NODEBUG and just DEBUG */
            if (strstr(CommandLine, "CRASHDEBUG"))
            {
                KdPitchDebugger = FALSE;
            }
            else if (strstr(CommandLine, "NODEBUG"))
            {
                KdPitchDebugger = TRUE;
            }
            else if ((DebugLine = strstr(CommandLine, "DEBUG")) != NULL)
            {//查找系统是否是以 DEBUG 模式启动
                /* Enable KD */
                EnableKd = TRUE;//启用 KD 以及 Windbg 等调试器
                ...... //下面是额外的 options 检查及处理
            }
        }
}
```

下面继续：

```
<Kdinit.c>
/* Initialize the debugger if requested */
    if ((EnableKd) && (NT_SUCCESS(KdDebuggerInitialize0(LoaderBlock))))
    {
        /* Now set our real KD routine */
        KiDebugRoutine = KdpTrap;//更换调试函数为 kdpTrap
        /* Check if we've already initialized our structures */
        if (!KdpDebuggerStructuresInitialized)
        {
            /* Set the Debug Switch Routine and Retries*/
            KdpContext.KdpDefaultRetries = 20;
            KiDebugSwitchRoutine = KdpSwitchProcessor;

            /* Initialize the Time Slip DPC */
            KeInitializeDpc(&KdpTimeSlipDpc, KdpTimeSlipDpcRoutine, NULL);
            KeInitializeTimer(&KdpTimeSlipTimer);
            ExInitializeWorkItem(&KdpTimeSlipWorkItem, KdpTimeSlipWork, NULL);
            /* First-time initialization done! */
```

```
            KdpDebuggerStructuresInitialized = TRUE;
        }
        /* Initialize the timer */
        KdTimerStart.QuadPart = 0;//初始化 timer.调试过程中使用时钟进行通信及超时重传
        /* Officially enable KD */
        KdPitchDebugger = FALSE;
        KdDebuggerEnabled = TRUE;//启用调试器
        ......
```

至此，内核调试模块的初始化就已经完成了。其流程大致如此：

首先，初始化调试函数为默认的 KdpStub 和其他一些相关的初始化工作。

然后，扫描启动参数是否有关键字"DEBUG"来决定是否启用 KD 等调试器。如果有则更换内核调试函数为 kdpTrap。

最后，保存调试相关数据，初始化 Timer 并结束内核调试的初始化工作。

由此可知，函数 KdpTrap 是整个调试过程的处理函数。下面分析该函数：

```
BOOLEAN
NTAPI
KdpTrap(IN PKTRAP_FRAME TrapFrame,
        IN PKEXCEPTION_FRAME ExceptionFrame,
        IN PEXCEPTION_RECORD ExceptionRecord,
        IN PCONTEXT ContextRecord,
        IN KPROCESSOR_MODE PreviousMode,
        IN BOOLEAN SecondChanceException)
{
    BOOLEAN Unload = FALSE;
    ULONG_PTR ProgramCounter;
    BOOLEAN Handled;
    NTSTATUS ReturnStatus;
    USHORT ReturnLength;
    if ((ExceptionRecord->ExceptionCode == STATUS_BREAKPOINT) &&
        (ExceptionRecord->ExceptionInformation[0] != BREAKPOINT_BREAK))
    {//处理 STATUS_BREAKPOINT 异常，包括 Print、Prompt、Load/Unload sysmbols 等
        /* Save Program Counter */
        ProgramCounter = KeGetContextPc(ContextRecord);
        /* Check what kind of operation was requested from us */
        switch (ExceptionRecord->ExceptionInformation[0])
        {
            /* DbgPrint */
            case BREAKPOINT_PRINT://DbgPrint 调用
                /* Call the worker routine */
                ReturnStatus                                                    =
KdpPrint((ULONG)KdpGetParameterThree(ContextRecord),
                                    (ULONG)KdpGetParameterFour(ContextRecord),
                                    (LPSTR)ExceptionRecord->
```

```
                                ExceptionInformation[1],
                                (USHORT)ExceptionRecord->
                                ExceptionInformation[2],
                                PreviousMode,
                                TrapFrame,
                                ExceptionFrame,
                                &Handled);
            /* Update the return value for the caller */
            KeSetContextReturnRegister(ContextRecord, ReturnStatus);
            break;
         /* DbgPrompt */
         case BREAKPOINT_PROMPT://Prompt 调用
            /* Call the worker routine */
            ReturnLength = KdpPrompt((LPSTR)ExceptionRecord->
                                ExceptionInformation[1],
                                (USHORT)ExceptionRecord->
                                ExceptionInformation[2],
(LPSTR)KdpGetParameterThree(ContextRecord),
(USHORT)KdpGetParameterFour(ContextRecord),
                                PreviousMode,
                                TrapFrame,
                                ExceptionFrame);
            Handled = TRUE;
            /* Update the return value for the caller */
            KeSetContextReturnRegister(ContextRecord, ReturnLength);
            break;
         /* DbgUnLoadImageSymbols */
         case BREAKPOINT_UNLOAD_SYMBOLS:
            /* Drop into the load case below, with the unload parameter */
            Unload = TRUE;
         /* DbgLoadImageSymbols */
         case BREAKPOINT_LOAD_SYMBOLS://加载符号表
            /* Call the worker routine */
            KdpSymbol((PSTRING)ExceptionRecord->
                    ExceptionInformation[1],
                    (PKD_SYMBOLS_INFO)ExceptionRecord->
                    ExceptionInformation[2],
                    Unload,
                    PreviousMode,
                    ContextRecord,
                    TrapFrame,
                    ExceptionFrame);
            Handled = TRUE;
            break;
```

```
         /* DbgCommandString */
         case BREAKPOINT_COMMAND_STRING:
            /* Call the worker routine */
            KdpCommandString((PSTRING)ExceptionRecord->
                        ExceptionInformation[1],
                        (PSTRING)ExceptionRecord->
                        ExceptionInformation[2],
                        PreviousMode,
                        ContextRecord,
                        TrapFrame,
                        ExceptionFrame);
            Handled = TRUE;
         /* Anything else, do nothing */
         default:
            /* Invalid debug service! Don't handle this! */
            Handled = FALSE;
            break;
      }
      if (ProgramCounter == KeGetContextPc(ContextRecord))
      {
         /* Update it */
         KeSetContextPc(ContextRecord,
                     ProgramCounter + KD_BREAKPOINT_SIZE);
      }
   }
   else
   {//调用 KdpReport 处理包括 INT3 等软件中断
      /* Call the worker routine */
      Handled = KdpReport(TrapFrame,
                      ExceptionFrame,
                      ExceptionRecord,
                      ContextRecord,
                      PreviousMode,
                      SecondChanceException);
   }
   /* Return TRUE or FALSE to caller */
   return Handled;
}
```

从 KdpTrap 函数的源码可知，KdpTrap 负责处理了 STATUS_BREAKPOINT 和 STATUS_SINGLE_STEP 异常。对于 STATUS_SINGLE_STEP 异常，KdpTrap 调用 KdpReport 处理该异常。KdpReport 的分析如下：

```
BOOLEAN
NTAPI
KdpReport(IN PKTRAP_FRAME TrapFrame,
```

```
        IN PKEXCEPTION_FRAME ExceptionFrame,
        IN PEXCEPTION_RECORD ExceptionRecord,
        IN PCONTEXT ContextRecord,
        IN KPROCESSOR_MODE PreviousMode,
        IN BOOLEAN SecondChanceException)
{
   ......//判断异常类型以及异常处理的方式(包括查找异常处理函数或者直接 pass 给调试器)
   //下面直接 pass 给调试器
   Enable = KdEnterDebugger(TrapFrame, ExceptionFrame);
   /*
    * Get the KPRCB and save the CPU Control State manually instead of
    * using KiSaveProcessorState, since we already have a valid CONTEXT.
    */
   Prcb = KeGetCurrentPrcb();
   KiSaveProcessorControlState(&Prcb->ProcessorState);
   RtlCopyMemory(&Prcb->ProcessorState.ContextFrame,
                 ContextRecord,
                 sizeof(CONTEXT));
   /* Report the new state */
   Handled = KdpReportExceptionStateChange(ExceptionRecord,
                                           &Prcb->ProcessorState.
                                           ContextFrame,
                                           SecondChanceException);
   /* Now restore the processor state, manually again. */
   RtlCopyMemory(ContextRecord,
                 &Prcb->ProcessorState.ContextFrame,
                 sizeof(CONTEXT));
   KiRestoreProcessorControlState(&Prcb->ProcessorState);
   /* Exit the debugger and clear the CTRL-C state */
   KdExitDebugger(Enable);
   KdpControlCPressed = FALSE;
   return Handled;
}
```

函数直接调用 KdpReportExceptionStateChange 来通知异常状态改变的消息。之前，KdpReport 会调用 KdEnterDebugger 初始化 KD 以及 windbg 等接管异常所需要的条件，包括保存 CPU 的某些状态,锁定串口以方便 windbg 调试器使用串口传递数据等。代码详见 ReactOS 源码。KdpReportExceptionStateChange 的代码如下：

```
BOOLEAN
NTAPI
KdpReportExceptionStateChange(IN PEXCEPTION_RECORD ExceptionRecord,
                              IN OUT PCONTEXT Context,
                              IN BOOLEAN SecondChanceException)
{
   STRING Header, Data;
```

```
    DBGKD_ANY_WAIT_STATE_CHANGE WaitStateChange;
    KCONTINUE_STATUS Status;
    /* Start report loop */
    do
    {
       KdpSetCommonState(DbgKdExceptionStateChange, Context, &WaitStateChange);
     //数据包通用部分的初始化，这些数据是大部分异常的数据包都拥有的。
       /* Copy the Exception Record and set First Chance flag */
#if !defined(_WIN64)
       ExceptionRecord32To64((PEXCEPTION_RECORD32)ExceptionRecord,
                             &WaitStateChange.u.Exception.ExceptionRecord);
#else
       RtlCopyMemory(&WaitStateChange.u.Exception.ExceptionRecord,
                     ExceptionRecord,
                     sizeof(EXCEPTION_RECORD));
#endif
       WaitStateChange.u.Exception.FirstChance = !SecondChanceException;
       /* Now finish creating the structure */
       KdpSetContextState(&WaitStateChange, Context);
       //上面这些代码是初始化结构 WaitStateChange 中针对特定异常的数据的部分
       /* Setup the actual header to send to KD */
       Header.Length = sizeof(DBGKD_ANY_WAIT_STATE_CHANGE);
       Header.Buffer = (PCHAR)&WaitStateChange;//数据包封装完成，赋值给 STRING
       /* Setup the trace data */
       DumpTraceData(&Data);
       Status = KdpSendWaitContinue(PACKET_TYPE_KD_STATE_CHANGE64,
                                    &Header,
                                    &Data,
                                    Context);
    } while (Status == ContinueProcessorReselected);
    /* Return */
    return Status;
}
```

KdpReportExceptionStateChange 根据异常的上下文来初始化数据包，并调用 KdpSendWaitContinue 通过串口发送数据包给 windbg 并等待来自 windbg 的处理结果。

```
KCONTINUE_STATUS
NTAPI
KdpSendWaitContinue(IN ULONG PacketType,
                    IN PSTRING SendHeader,
                    IN PSTRING SendData OPTIONAL,
                    IN OUT PCONTEXT Context)
{
    STRING Data, Header;
    DBGKD_MANIPULATE_STATE64 ManipulateState;
```

```
    ULONG Length;
    KDSTATUS RecvCode;
    /* Setup the Manipulate State structure */
    Header.MaximumLength = sizeof(DBGKD_MANIPULATE_STATE64);
    Header.Buffer = (PCHAR)&ManipulateState;
    Data.MaximumLength = sizeof(KdpMessageBuffer);
    Data.Buffer = KdpMessageBuffer;
    KdpContextSent = FALSE;
SendPacket:
    /*调用 KdSendPacket 发送异常数据包给 windbg 调试器 */
    KdSendPacket(PacketType, SendHeader, SendData, &KdpContext);
    /* If the debugger isn't present anymore, just return success */
    if (KdDebuggerNotPresent) return ContinueSuccess;
    /*使用 For 循环等待来自 windbg 的数据包处理结果，接受数据是调用 KdReceivePacket 接受数据包*/
    for (;;)//循环条件为空
    {
        do
        {
            /* Wait to get a reply to our packet */
            RecvCode = KdReceivePacket(PACKET_TYPE_KD_STATE_MANIPULATE,
                                       &Header,
                                       &Data,
                                       &Length,
                                       &KdpContext);

            /*Windbg 使用确认机制来保证数据包被处理，否则重传*/
            if (RecvCode == KdPacketNeedsResend) goto SendPacket;
        } while (RecvCode == KdPacketTimedOut);
        /*根据接收到的数据包做相应的处理*/
        switch (ManipulateState.ApiNumber)
        {
            case DbgKdReadVirtualMemoryApi:
                /* 读取虚拟内存*/
                KdpReadVirtualMemory(&ManipulateState, &Data, Context);
                break;
            case DbgKdWriteVirtualMemoryApi:
                /*写虚拟内存*/
                KdpWriteVirtualMemory(&ManipulateState, &Data, Context);
                break;
            case DbgKdGetContextApi:
                 ......
            case DbgKdSetContextApi:
                 ......
```

```
            case DbgKdWriteBreakPointApi:
                /*写入断点，后面以此为例做实验*/
                KdpWriteBreakpoint(&ManipulateState, &Data, Context);
                break;
            ....../*处理了包括读写内存、搜索内存、设置/恢复断点、继续执行、重启等所有 windbg
                  的功能实现。*/
            default:
                /*错误的参数，直接返回失败*/
                KdpDprintf("Received        Unhandled        API        %lx\n",
ManipulateState.ApiNumber);
                Data.Length = 0;
                ManipulateState.ReturnStatus = STATUS_UNSUCCESSFUL;
                /* Send it */
                KdSendPacket(PACKET_TYPE_KD_STATE_MANIPULATE,
                             &Header,
                             &Data,
                             &KdpContext);
                break;
        }
    }
}
```

可以说 KdpSendWaitContinue 函数是整个内核调试器的大管家，它根据异常的类型来调用不同的函数进行处理，是整个调试流程的分发地所在。被调试系统和调试器进行交互式通过 KdSendPacket 和 KdReceivePacket 函数进行的。下面首先分析下 KdReceivePacket 的源码：

```
KDP_STATUS
NTAPI
KdReceivePacket(
    IN ULONG PacketType,
    OUT PSTRING MessageHeader,
    OUT PSTRING MessageData,
    OUT PULONG DataLength,
    IN OUT PKD_CONTEXT KdContext)
{
    UCHAR Byte = 0;
    KDP_STATUS KdStatus;
    KD_PACKET Packet;
    ULONG Checksum;
    /* Special handling for breakin packet */
    if(PacketType == PACKET_TYPE_KD_POLL_BREAKIN)
    {//包类型是中断包.内核调试器中的包类型共有 13 种
        return KdpPollBreakIn();
    }
    for (;;)
    {/*下面 Step1-Step5 依次读取数据包头中的 PacketLeader、PacketType、ByteCount、
```

```
PackID 以及 CheckSum*/
      /* Step 1 - Read PacketLeader */
      KdStatus = KdpReceivePacketLeader(&Packet.PacketLeader);
      if (KdStatus != KDP_PACKET_RECEIVED)
      {
         /* Check if we got a breakin  */
         if (KdStatus == KDP_PACKET_RESEND)
         {
            KdContext->KdpControlCPending = TRUE;
         }
         return KdStatus;
      }
      /* Step 2 - Read PacketType */
      KdStatus = KdpReceiveBuffer(&Packet.PacketType, sizeof(USHORT));
      if (KdStatus != KDP_PACKET_RECEIVED)
      {
         /* Didn't receive a PacketType. */
         return KdStatus;
      }
      /* Check if we got a resend packet */
      if (Packet.PacketLeader == CONTROL_PACKET_LEADER &&
         Packet.PacketType == PACKET_TYPE_KD_RESEND)
      {
         return KDP_PACKET_RESEND;
      }
      /* Step 3 - Read ByteCount */
      KdStatus = KdpReceiveBuffer(&Packet.ByteCount, sizeof(USHORT));
      if (KdStatus != KDP_PACKET_RECEIVED)
      {
         /* Didn't receive ByteCount. */
         return KdStatus;
      }
      /* Step 4 - Read PacketId */
      KdStatus = KdpReceiveBuffer(&Packet.PacketId, sizeof(ULONG));
      if (KdStatus != KDP_PACKET_RECEIVED)
      {
         /* Didn't receive PacketId. */
         return KdStatus;
      }
      /*Step 5 -Read CheckSum*/
      KdStatus = KdpReceiveBuffer(&Packet.Checksum, sizeof(ULONG));
      if (KdStatus != KDP_PACKET_RECEIVED)
      {
         /* Didn't receive Checksum. */
```

```
        return KdStatus;
    }
    //下面根据接收的数据包头部的信息来做相应的处理工作
    /* Step 6 - Handle control packets */
    if (Packet.PacketLeader == CONTROL_PACKET_LEADER)
    {
        switch (Packet.PacketType)
        {
            case PACKET_TYPE_KD_ACKNOWLEDGE:
               //确认包，调试器没发一个包，都等待一个确认包
                /* Are we waiting for an ACK packet? */
                if (PacketType == PACKET_TYPE_KD_ACKNOWLEDGE &&
                    Packet.PacketId == (CurrentPacketId & ~SYNC_PACKET_ID))
                {
                    /* Remote acknowledges the last packet */
                    CurrentPacketId ^= 1;
                    return KDP_PACKET_RECEIVED;
                }
                /* That's not what we were waiting for, start over. */
                continue;
            case PACKET_TYPE_KD_RESET://重置包
                KDDBGPRINT("KdReceivePacket - got a reset packet\n");
                KdpSendControlPacket(PACKET_TYPE_KD_RESET, 0);
                CurrentPacketId = INITIAL_PACKET_ID;
                RemotePacketId = INITIAL_PACKET_ID;
                /* Fall through */

            case PACKET_TYPE_KD_RESEND://重发包
                KDDBGPRINT("KdReceivePacket - got PACKET_TYPE_KD_RESEND\n");
                /* Remote wants us to resend the last packet */
                return KDP_PACKET_RESEND;
            default:
                KDDBGPRINT("KdReceivePacket - got unknown control packet\n");
                return KDP_PACKET_RESEND;
        }
    }
    /* Did we wait for an ack packet? */
    if (PacketType == PACKET_TYPE_KD_ACKNOWLEDGE)
    {
        /* We received something different */
        KdpSendControlPacket(PACKET_TYPE_KD_RESEND, 0);
        CurrentPacketId ^= 1;
        return KDP_PACKET_RECEIVED;
    }
```

```
.....//省略内容为检查包的完整性--是否有丢失等情况发生
/* Receive the message header data */
KdStatus = KdpReceiveBuffer(MessageHeader->Buffer,
                      MessageHeader->Length);
if (KdStatus != KDP_PACKET_RECEIVED)
{//要求重新发送，调用 KdpSendControlPacket，最终调用 KdSendPacket 实现发包
   KDDBGPRINT("KdReceivePacket - Didn't receive message header data.\n");
   KdpSendControlPacket(PACKET_TYPE_KD_RESEND, 0);
   continue;
}
/*计算校验和是否正确 */
Checksum = KdpCalculateChecksum(MessageHeader->Buffer,
                           MessageHeader->Length);
/* Calculate the length of the message data */
*DataLength = Packet.ByteCount - MessageHeader->Length;
/* Shall we receive messsage data? */
if (MessageData)
{
   /* Set the length of the message data */
   MessageData->Length = *DataLength;

   /* Do we have data? */
   if (MessageData->Length)
   {
      KDDBGPRINT("KdReceivePacket - got data\n");

      /* Receive the message data */
      KdStatus = KdpReceiveBuffer(MessageData->Buffer,
                            MessageData->Length);
      if (KdStatus != KDP_PACKET_RECEIVED)
      {
         /* Didn't receive data. Start over. */
         KDDBGPRINT("KdReceivePacket - Didn't receive message data.\n");
         KdpSendControlPacket(PACKET_TYPE_KD_RESEND, 0);
         continue;
      }

      /* Add cheksum for message data */
      Checksum += KdpCalculateChecksum(MessageData->Buffer,
                                  MessageData->Length);
   }
}
/*包尾部必须是一个以 0xAA 结尾的包，否则不合法。这个是调试器的规定*/
KdStatus = KdpReceiveBuffer(&Byte, sizeof(UCHAR));//接收一个字节
```

```
      if (KdStatus != KDP_PACKET_RECEIVED || Byte != PACKET_TRAILING_BYTE)
      {// PACKET_TRAILING_BYTE==0xAA
         KDDBGPRINT("KdReceivePacket - wrong trailing byte (0x%x), status
0x%x\n", Byte, KdStatus);
         KdpSendControlPacket(PACKET_TYPE_KD_RESEND, 0);
         continue;
      }
      /*对比校验和，看包中是否有差错*/
      if (Packet.Checksum != Checksum)
      {
         KDDBGPRINT("KdReceivePacket - wrong cheksum, got %x, calculated %x\n",
                    Packet.Checksum, Checksum);
         KdpSendControlPacket(PACKET_TYPE_KD_RESEND, 0);
         continue;
      }
      /*发送确认包，告诉 windbg 包成功接收 */
      KdpSendControlPacket(PACKET_TYPE_KD_ACKNOWLEDGE, Packet.PacketId);
      /* Check if the received PacketId is ok */
      if (Packet.PacketId != RemotePacketId)
      {
         /* Continue with next packet */
         continue;
      }
      /* Did we get the right packet type? */
      if (PacketType == Packet.PacketType)
      {
         /* Yes, return success */
         //KDDBGPRINT("KdReceivePacket - all ok\n");
         RemotePacketId ^= 1;
         return KDP_PACKET_RECEIVED;
      }
      /* We received something different, ignore it. */
      KDDBGPRINT("KdReceivePacket - wrong PacketType\n");
   }
    return KDP_PACKET_RECEIVED;
}
```

从源码可以看出，KdReceivePacket 是调用 KdpReceiveBuffer 来完成数据的接收的。而 KdpReceiveBuffer 则是通过调用 KdpReceiveByte, KdpReceiveByte 再调用 KdpPollByte，然后进入硬件抽象层调用 READ_PORT_UCHAR 函数读取串口来完成数据接收的。函数源码依次如下：

```
KDP_STATUS
NTAPI
KdpReceiveBuffer(
   OUT PVOID Buffer,
```

```
    IN  ULONG Size)
{
    ULONG i;
    PUCHAR ByteBuffer = Buffer;
    KDP_STATUS Status;
    for (i = 0; i < Size; i++)
    {  /*一字节一字节的读取数据，大小为 Size*/
        Status = KdpReceiveByte(&ByteBuffer[i]);
        if (Status != KDP_PACKET_RECEIVED)
        {return Status;}
    }
    return KDP_PACKET_RECEIVED;
}
KDP_STATUS
NTAPI
KdpReceiveByte(OUT PBYTE OutByte)
{    ULONG Repeats = REPEAT_COUNT;//尝试读取的重复次数
    while (Repeats--)
    { /* Check if data is available */
        if (KdpPollByte(OutByte) == KDP_PACKET_RECEIVED)
        { /* We successfully got a byte */
            return KDP_PACKET_RECEIVED;
        }
    }
   /* Timed out */
    return KDP_PACKET_TIMEOUT;
}
KDP_STATUS
NTAPI
KdpPollByte(OUT PBYTE OutByte)
{
    READ_PORT_UCHAR(ComPortBase + COM_MSR); // Timing
    /*调用 HAL 函数读取串口*/
    if ((READ_PORT_UCHAR(ComPortBase + COM_LSR) & LSR_DR))
    {   /* Yes, return the byte */
        *OutByte = READ_PORT_UCHAR(ComPortBase + COM_DAT);
        return KDP_PACKET_RECEIVED;
    }
    /*返回超时*/
    return KDP_PACKET_TIMEOUT;
}
```

上述函数都较为简单，不做分析。

至此，整个接收的流程都分析完毕。下面简单分析下发送的流程，发送过程首先调用 KdSendPacket 发送数据包：

```
VOID
NTAPI
KdSendPacket(
   IN ULONG PacketType,
   IN PSTRING MessageHeader,
   IN PSTRING MessageData,
   IN OUT PKD_CONTEXT KdContext)
{
   KD_PACKET Packet;
   KDP_STATUS KdStatus;
   ULONG Retries;

   /*封包过程，依次初始化数据包包头，包括数据包标识、数据包类型、数据包长度、校验和*/
   Packet.PacketLeader = PACKET_LEADER;
   Packet.PacketType = PacketType;
   Packet.ByteCount = MessageHeader->Length;
   Packet.Checksum = KdpCalculateChecksum(MessageHeader->Buffer,
                                          MessageHeader->Length);
   /*添加额外数据 */
   if (MessageData)
   {    Packet.ByteCount += MessageData->Length;
      Packet.Checksum += KdpCalculateChecksum(MessageData->Buffer,
                                              MessageData->Length);
   }
   Retries = KdContext->KdpDefaultRetries;
   do
   {  /* Set the packet id */
      Packet.PacketId = CurrentPacketId;
      /* Send the packet header to the KD port */
      KdpSendBuffer(&Packet, sizeof(KD_PACKET));
      /* Send the message header */
      KdpSendBuffer(MessageHeader->Buffer, MessageHeader->Length);
      /* If we have meesage data, also send it */
      if (MessageData)
      {
         KdpSendBuffer(MessageData->Buffer, MessageData->Length);
      }
      /*发送结束符 0xAA */
      KdpSendByte(PACKET_TRAILING_BYTE);
      /*等待接收确认包*/
      KdStatus = KdReceivePacket(PACKET_TYPE_KD_ACKNOWLEDGE,
                                 NULL,
                                 NULL,
                                 0,
```

```
                            KdContext);

      /* Did we succeed? */
      if (KdStatus == KDP_PACKET_RECEIVED)
      {
         CurrentPacketId &= ~SYNC_PACKET_ID;
         break;
      }
      /* PACKET_TYPE_KD_DEBUG_IO is allowed to instantly timeout */
      if (PacketType == PACKET_TYPE_KD_DEBUG_IO)
      {
         /* No response, silently fail. */
         return;
      }
      if (KdStatus == KDP_PACKET_TIMEOUT)
      {
         Retries--;
      }
      /* Packet timed out, send it again */
      KDDBGPRINT("KdSendPacket got KdStatus 0x%x\n", KdStatus);
   }
   while (Retries > 0);
}
```

下面的过程就是和 KdReceivePacket 想法的过程了。在此不再赘述，详细请参考 ReactOS 源码。

5. Windbg 命令实战之条件断点

为了方便调试过程中的分析，首先把调试器使用到的一些常量罗列一下：

- 数据包的包头(数据包包头大小：0x10)

```
typedef struct _KD_PACKET
{
   ULONG PacketLeader;//共有四种标识
   USHORT PacketType;//共有十三种包类型，某些包类型具有共同的标示
   USHORT ByteCount;//紧随包头的数据部分的大小
   ULONG PacketId;//数据包的 ID
   ULONG Checksum;//数据包的校验和
} KD_PACKET, *PKD_PACKET;
```

- PacketLeader 的种类

```
#define BREAKIN_PACKET                  0x62626262 //中断包
#define BREAKIN_PACKET_BYTE              0x62
#define PACKET_LEADER                   0x30303030 //普通的功能包
#define PACKET_LEADER_BYTE               0x30
#define CONTROL_PACKET_LEADER            0x69696969//控制包
#define CONTROL_PACKET_LEADER_BYTE        0x69
#define PACKET_TRAILING_BYTE             0xAA
```

- PacketType 的种类

```
#define PACKET_TYPE_UNUSED              0
#define PACKET_TYPE_KD_STATE_CHANGE32     1
#define PACKET_TYPE_KD_STATE_MANIPULATE    2
#define PACKET_TYPE_KD_DEBUG_IO          3
#define PACKET_TYPE_KD_ACKNOWLEDGE        4
#define PACKET_TYPE_KD_RESEND            5
#define PACKET_TYPE_KD_RESET             6
#define PACKET_TYPE_KD_STATE_CHANGE64     7
#define PACKET_TYPE_KD_POLL_BREAKIN       8
#define PACKET_TYPE_KD_TRACE_IO          9
#define PACKET_TYPE_KD_CONTROL_REQUEST     10
#define PACKET_TYPE_KD_FILE_IO           11
#define PACKET_TYPE_MAX                 12
```

- DBGKD_MANIPULATE_STATE64 结构的具体构成如下：

```
typedef struct _DBGKD_MANIPULATE_STATE64
{
   ULONG ApiNumber;
   USHORT ProcessorLevel;
   USHORT Processor;
   NTSTATUS ReturnStatus;
   union
   {
      DBGKD_READ_MEMORY64 ReadMemory;
      DBGKD_WRITE_MEMORY64 WriteMemory;
      DBGKD_GET_CONTEXT GetContext;
      DBGKD_SET_CONTEXT SetContext;
      DBGKD_WRITE_BREAKPOINT64 WriteBreakPoint;
      DBGKD_RESTORE_BREAKPOINT RestoreBreakPoint;
      DBGKD_CONTINUE Continue;
      DBGKD_CONTINUE2 Continue2;
      DBGKD_READ_WRITE_IO64 ReadWriteIo;
      DBGKD_READ_WRITE_IO_EXTENDED64 ReadWriteIoExtended;
      DBGKD_QUERY_SPECIAL_CALLS QuerySpecialCalls;
      DBGKD_SET_SPECIAL_CALL64 SetSpecialCall;
      DBGKD_SET_INTERNAL_BREAKPOINT64 SetInternalBreakpoint;
      DBGKD_GET_INTERNAL_BREAKPOINT64 GetInternalBreakpoint;
      DBGKD_GET_VERSION64 GetVersion64;
      DBGKD_BREAKPOINTEX BreakPointEx;
      DBGKD_READ_WRITE_MSR ReadWriteMsr;
      DBGKD_SEARCH_MEMORY SearchMemory;
      DBGKD_GET_SET_BUS_DATA GetSetBusData;
      DBGKD_FILL_MEMORY FillMemory;
      DBGKD_QUERY_MEMORY QueryMemory;
```

```
      DBGKD_SWITCH_PARTITION SwitchPartition;
   } u;
} DBGKD_MANIPULATE_STATE64, *PDBGKD_MANIPULATE_STATE64;
```

上述的这个结构体，就是交换数据过程中所使用的结构。其中的 union 根据不同的功能做不同的解释。

- Manipulate 类型，在 KdSendWaitContinue 中用作区分不同的操作，即结构体 DBGKD_MANIPULATE_STATE64 中的 ApiNumber。

```
#define DbgKdMinimumManipulate            0x00003130
#define DbgKdReadVirtualMemoryApi         0x00003130
#define DbgKdWriteVirtualMemoryApi        0x00003131
#define DbgKdGetContextApi                0x00003132
#define DbgKdSetContextApi                0x00003133
#define DbgKdWriteBreakPointApi           0x00003134
#define DbgKdRestoreBreakPointApi         0x00003135
#define DbgKdContinueApi                  0x00003136
#define DbgKdReadControlSpaceApi          0x00003137
#define DbgKdWriteControlSpaceApi         0x00003138
#define DbgKdReadIoSpaceApi               0x00003139
#define DbgKdWriteIoSpaceApi              0x0000313A
#define DbgKdRebootApi                    0x0000313B
#define DbgKdContinueApi2                 0x0000313C
#define DbgKdReadPhysicalMemoryApi        0x0000313D
#define DbgKdWritePhysicalMemoryApi       0x0000313E
#define DbgKdQuerySpecialCallsApi         0x0000313F
#define DbgKdSetSpecialCallApi            0x00003140
#define DbgKdClearSpecialCallsApi         0x00003141
#define DbgKdSetInternalBreakPointApi     0x00003142
#define DbgKdGetInternalBreakPointApi     0x00003143
#define DbgKdReadIoSpaceExtendedApi       0x00003144
#define DbgKdWriteIoSpaceExtendedApi      0x00003145
#define DbgKdGetVersionApi                0x00003146
#define DbgKdWriteBreakPointExApi         0x00003147
#define DbgKdRestoreBreakPointExApi       0x00003148
#define DbgKdCauseBugCheckApi             0x00003149
#define DbgKdSwitchProcessor              0x00003150
#define DbgKdPageInApi                    0x00003151
#define DbgKdReadMachineSpecificRegister  0x00003152
#define DbgKdWriteMachineSpecificRegister 0x00003153
#define OldVlm1                           0x00003154
#define OldVlm2                           0x00003155
#define DbgKdSearchMemoryApi              0x00003156
#define DbgKdGetBusDataApi                0x00003157
#define DbgKdSetBusDataApi                0x00003158
#define DbgKdCheckLowMemoryApi            0x00003159
```

```
#define DbgKdClearAllInternalBreakpointsApi 0x0000315A
#define DbgKdFillMemoryApi                0x0000315B
#define DbgKdQueryMemoryApi               0x0000315C
#define DbgKdSwitchPartition              0x0000315D
#define DbgKdMaximumManipulate            0x0000315E
```

好的，常量介绍完毕。下面以 windbg 的调试断点来简要实验下。

首先，打开 OllyDbg，Attach 到 Windbg 上。由于要分析 windbg 对命令的处理，自然要首先断在 windbg 获取命令的地方。所以，首先使用 Spy++或者直接从 OD 中获取输入框的句柄值（0x2038A），然后在 OD 的命令行插件中对 GetWindowTextW 下条件断点：

```
bp GetWindowTextW [esp+4]==1EC
```

一定要下条件断点，否则，OllyDbg 会不断的断在 GetWindowTextW 处。然后，在 windbg 的命令输入框中输入以下条件断点命令：

```
bp nt!NtCreateFile "j([esp+4]='xxx') '';'g'"
```

然后回车，并 OllyDbg 断下。Alt+F9 返回用户空间。F7 几下来到如下地方：

```
01014330    8BFF                    mov edi,edi
01014332    55                      push ebp
01014333    8BEC                    mov ebp,esp
01014335    B8 48200000             mov eax,2048
0101433A    E8 01400400             call windbg.01058340
0101433F    A1 20450601             mov eax,dword ptr ds:[1064520]
01014344    33C5                    xor eax,ebp
01014346    8945 DC                 mov dword ptr ss:[ebp-24],eax
01014349    8B45 08                 mov eax,dword ptr ss:[ebp+8]
0101434C    0FB708                  movzx ecx,word ptr ds:[eax]
0101434F    51                      push ecx
01014350    FF15 6C140001           call dword ptr ds:[<&msvcrt.iswspace>]     ; msvcrt.iswspace
01014356    83C4 04                 add esp,4
01014359    85C0                    test eax,eax
0101435B    74 0B                   je short windbg.01014368
0101435D    8B55 08                 mov edx,dword ptr ss:[ebp+8]
01014360    83C2 02                 add edx,2
01014363    8955 08                 mov dword ptr ss:[ebp+8],edx
01014366  ^ EB E1                   jmp short windbg.01014349
01014368    8B45 08                 mov eax,dword ptr ss:[ebp+8]
0101436B    8945 F4                 mov dword ptr ss:[ebp-C],eax
……
01014416    66:894D F0              mov word ptr ss:[ebp-10],cx
0101441A    68 4C250001             push windbg.0100254C                       ; .beep
0101441F    8B55 08                 mov edx,dword ptr ss:[ebp+8]
01014422    52                      push edx
01014423    FF15 60140001           call dword ptr ds:[<&msvcrt._wcsicmp>]     ; msvcrt._wcsicmp
01014429    83C4 08                 add esp,8
0101442C    85C0                    test eax,eax
0101442E    75 2F                   jnz short windbg.0101445F
```

```
01014430    8B45 F8                 mov eax,dword ptr ss:[ebp-8]
01014433    50                       push eax
01014434    0FB74D F0               movzx ecx,word ptr ss:[ebp-10]
01014438    51                       push ecx
01014439    8B55 08                 mov edx,dword ptr ss:[ebp+8]
0101443C    52                       push edx
0101443D    68 2C250001              push windbg.0100252C                ; windbg> %s%c%s\n
01014442    E8 C9FBFFFF             call windbg.01014010
01014447    83C4 10                 add esp,10
0101444A    68 2C010000              push 12C
0101444F    68 EE020000             push 2EE
01014454    FF15 20110001           call dword ptr ds:[<&KERNEL32.Beep>]    ; kernel32.Beep
0101445A    E9 DB090000              jmp windbg.01014E3A
0101445F    68 1C250001             push windbg.0100251C                ; .browse
01014464    8B45 08                 mov eax,dword ptr ss:[ebp+8]
01014467    50                       push eax
01014468    FF15 60140001           call dword ptr ds:[<&msvcrt._wcsicmp>]    ; msvcrt._wcsicmp
0101446E    83C4 08                 add esp,8
01014471    85C0                     test eax,eax
01014473    75 2C                   jnz short windbg.010144A1
01014475    8B4D F8                 mov ecx,dword ptr ss:[ebp-8]
01014478    51                       push ecx
01014479    0FB755 F0               movzx edx,word ptr ss:[ebp-10]
0101447D    52                       push edx
0101447E    8B45 08                 mov eax,dword ptr ss:[ebp+8]
01014481    50                       push eax
01014482    68 2C250001              push windbg.0100252C                ; windbg> %s%c%s\n
01014487    E8 84FBFFFF             call windbg.01014010
0101448C    83C4 10                 add esp,10
0101448F    6A 00                   push 0
01014491    6A 00                   push 0
01014493    8B4D F8                 mov ecx,dword ptr ss:[ebp-8]
01014496    51                       push ecx
01014497    E8 34AFFFFF             call windbg.0100F3D0
0101449C    E9 99090000             jmp windbg.01014E3A
010144A1    68 10250001              push windbg.01002510                ; .cls
010144A6    8B55 08                 mov edx,dword ptr ss:[ebp+8]
010144A9    52                       push edx
010144AA    FF15 60140001           call dword ptr ds:[<&msvcrt._wcsicmp>]    ; msvcrt._wcsicmp
010144B0    83C4 08                 add esp,8
010144B3    85C0                     test eax,eax
010144B5    75 55                   jnz short windbg.0101450C
010144B7    C645 EB 01              mov byte ptr ss:[ebp-15],1
010144BB    837D FC 00              cmp dword ptr ss:[ebp-4],0
```

```
010144BF    74 1A                   je short windbg.010144DB
010144C1    68 08250001             push windbg.01002508          ; /s
010144C6    8B45 FC                 mov eax,dword ptr ss:[ebp-4]
010144C9    50                      push eax
010144CA    FF15 60140001           call dword ptr ds:[<&msvcrt._wcsicmp>]; msvcrt._wcsicmp
010144D0    83C4 08                 add esp,8
010144D3    85C0                    test eax,eax
010144D5    75 04                   jnz short windbg.010144DB
010144D7    C645 EB 00              mov byte ptr ss:[ebp-15],0
010144DB    0FB64D EB               movzx ecx,byte ptr ss:[ebp-15]
010144DF    51                      push ecx
010144E0    E8 FBFAFFFF             call windbg.01013FE0
010144E5    0FB655 EB               movzx edx,byte ptr ss:[ebp-15]
010144E9    85D2                    test edx,edx
010144EB    75 1A                   jnz short windbg.01014507
010144ED    8B45 F8                 mov eax,dword ptr ss:[ebp-8]
010144F0    50                      push eax
010144F1    0FB74D F0               movzx ecx,word ptr ss:[ebp-10]
010144F5    51                      push ecx
010144F6    8B55 08                 mov edx,dword ptr ss:[ebp+8]
010144F9    52                      push edx
010144FA    68 2C250001             push windbg.0100252C                    ; windbg> %s%c%s\n
010144FF    E8 0CFBFFFF             call windbg.01014010
01014504    83C4 10                 add esp,10
01014507    E9 2E090000             jmp windbg.01014E3A
0101450C    68 F4240001             push windbg.010024F4                    ; .cmdtree
01014511    8B45 08                 mov eax,dword ptr ss:[ebp+8]
01014514    50                      push eax
01014515    FF15 60140001           call dword ptr ds:[<&msvcrt._wcsicmp>]      ; msvcrt._wcsicmp
0101451B    83C4 08                 add esp,8
0101451E    85C0                    test eax,eax
01014520    0F85 C9000000           jnz windbg.010145EF
01014526    C645 EA 00              mov byte ptr ss:[ebp-16],0
0101452A    8B4D F8                 mov ecx,dword ptr ss:[ebp-8]
0101452D    51                      push ecx
0101452E    0FB755 F0               movzx edx,word ptr ss:[ebp-10]
01014532    52                      push edx
01014533    8B45 08                 mov eax,dword ptr ss:[ebp+8]
01014536    50                      push eax
01014537    68 2C250001             push windbg.0100252C      ; windbg> %s%c%s\n
0101453C    E8 CFFAFFFF             call windbg.01014010
01014541    83C4 10                 add esp,10
01014544    837D FC 00              cmp dword ptr ss:[ebp-4],0
01014548    74 7F                   je short windbg.010145C9
```

```
0101454A    8B4D FC                 mov ecx,dword ptr ss:[ebp-4]
0101454D    0FB711                  movzx edx,word ptr ds:[ecx]
01014550    83FA 2D                 cmp edx,2D
01014553    74 0B                   je short windbg.01014560
01014555    8B45 FC                 mov eax,dword ptr ss:[ebp-4]
01014558    0FB708                  movzx ecx,word ptr ds:[eax]
0101455B    83F9 2F                 cmp ecx,2F
0101455E    75 69                   jnz short windbg.010145C9
01014560    8B55 FC                 mov edx,dword ptr ss:[ebp-4]
01014563    0FB742 02               movzx eax,word ptr ds:[edx+2]
01014567    83F8 72                 cmp eax,72
0101456A    75 5D                   jnz short windbg.010145C9
0101456C    8B4D FC                 mov ecx,dword ptr ss:[ebp-4]
0101456F    0FB751 04               movzx edx,word ptr ds:[ecx+4]
01014573    85D2                    test edx,edx
01014575    74 15                   je short windbg.0101458C
01014577    8B45 FC                 mov eax,dword ptr ss:[ebp-4]
0101457A    0FB748 04               movzx ecx,word ptr ds:[eax+4]
0101457E    51                      push ecx
0101457F    FF15 6C140001           call dword ptr ds:[<&msvcrt.iswspace>]    ; msvcrt.iswspace
01014585    83C4 04                 add esp,4
01014588    85C0                    test eax,eax
0101458A    74 3D                   je short windbg.010145C9
0101458C    C645 EA 01              mov byte ptr ss:[ebp-16],1
01014590    8B55 FC                 mov edx,dword ptr ss:[ebp-4]
01014593    83C2 04                 add edx,4
01014596    8955 FC                 mov dword ptr ss:[ebp-4],edx
01014599    8B45 FC                 mov eax,dword ptr ss:[ebp-4]
0101459C    0FB708                  movzx ecx,word ptr ds:[eax]
0101459F    51                      push ecx
010145A0    FF15 6C140001           call dword ptr ds:[<&msvcrt.iswspace>]    ; msvcrt.iswspace
010145A6    83C4 04                 add esp,4
010145A9    85C0                    test eax,eax
010145AB    74 0B                   je short windbg.010145B8
010145AD    8B55 FC                 mov edx,dword ptr ss:[ebp-4]
010145B0    83C2 02                 add edx,2
010145B3    8955 FC                 mov dword ptr ss:[ebp-4],edx
010145B6  ^ EB E1                   jmp short windbg.01014599
010145B8    8B45 FC                 mov eax,dword ptr ss:[ebp-4]
010145BB    0FB708                  movzx ecx,word ptr ds:[eax]
010145BE    85C9                    test ecx,ecx
010145C0    75 07                   jnz short windbg.010145C9
010145C2    C745 FC 00000000        mov dword ptr ss:[ebp-4],0
010145C9    837D FC 00              cmp dword ptr ss:[ebp-4],0
```

```
010145CD    75 09           jnz short windbg.010145D8
010145CF    6A 01           push 1
010145D1    E8 2A8E0400     call windbg.0105D400
010145D6    EB 12           jmp short windbg.010145EA
010145D8    8D55 E4         lea edx,dword ptr ss:[ebp-1C]
010145DB    52              push edx
010145DC    0FB645 EA       movzx eax,byte ptr ss:[ebp-16]
010145E0    50              push eax
010145E1    8B4D FC         mov ecx,dword ptr ss:[ebp-4]
010145E4    51              push ecx
010145E5    E8 36D0FFFF     call windbg.01011620
010145EA    E9 4B080000     jmp windbg.01014E3A
010145EF    68 D4240001     push windbg.010024D4    ; .flash_on_break
010145F4    8B55 08         mov edx,dword ptr ss:[ebp+8]
010145F7    52              push edx
```

……

可以发现，上面的函数就是在分析用户的输入，并且判断是否有 .beep、.cmdtree、.cls 等不需要和被调试系统交互的命令。继续调试，我发现，windbg 内部的处理流程还是很复杂的。尤其是进入到 windbg 的引擎模块后。所以，为了简单起见，我直接对 WriteFile 下断点，即可截获 windbg 写串口的数据，对于 windbg 内部的实现，等搞明白了再详述。

OK！下面在插件中对 WriteFile 下断：

```
bp  WriteFile
```

然后，F9 运行。栈视图如下：

```
Address   Value     Comment
02CDE488  022D4F4C  CALL to WriteFile from dbgeng.022D4F46
02CDE48C  000001EC  hFile = 000001EC (window)
02CDE490  02CDE54C  Buffer = 02CDE54C
02CDE494  00000010  nBytesToWrite = 10 (16.)
02CDE498  02CDE4FC  pBytesWritten = 02CDE4FC
02CDE49C  007E77F0  pOverlapped = 007E77F0
02CDE4A0  00000297
```

查看 buffer 0x02CDE54C(长度为 0x10)的数据如下：

```
Address   Hex dump                                          ASCII
02CDE54C  30 30 30 30 02 00 38 00 01 00 80 80 50 0D 00 00  0000□.8.□.€€P...
```

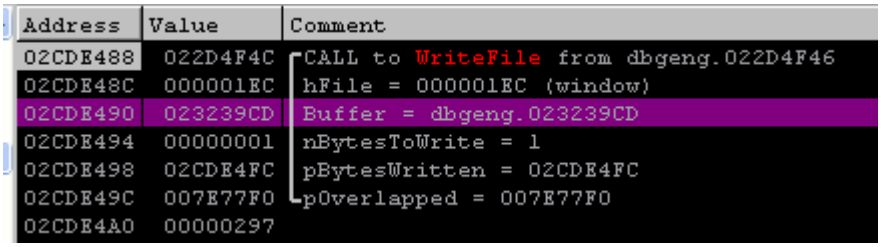
然后继续 F9.栈视图如下：

```
Address   Value     Comment
02CDE488  022D4F4C  CALL to WriteFile from dbgeng.022D4F46
02CDE48C  000001EC  hFile = 000001EC (window)
02CDE490  02CDE6E4  Buffer = 02CDE6E4
02CDE494  00000038  nBytesToWrite = 38 (56.)
02CDE498  02CDE4FC  pBytesWritten = 02CDE4FC
02CDE49C  007E77F0  pOverlapped = 007E77F0
02CDE4A0  00000297
```

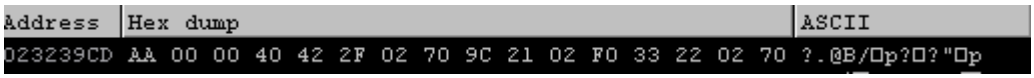
查看 buffer 0x02CDE6E4(长度为 0x38)的数据如下：

```
Address   Hex dump                                          ASCII
02CDE6E4  5C 31 00 00 70 1F D2 00 FF FF FF 00 00 C6 6E 80  \1..p?      ..芷
02CDE6F4  4C 26 57 80 FF FF FF FF 00 00 00 00 00 00 00 00  L&W€        ....
02CDE704  00 00 00 00 00 00 00 00 38 71 7E 00 00 00 00 00  ........8q~.....
02CDE714  44 00 00 00 01 00 00 00 E8 F8 7E 00 04 1F D2 00  D...□...鉧~.□?
02CDE724  34 E7 CD 02 50 E7 CD 02 58 E7 CD 02 E1 AD 0D 02  4瘁□P瘁□X瘁□幌.□
02CDE734  50 75 80 02 4C 26 57 80 FF FF FF FF 00 00 00 00  Pu?L&W€
```

继续 F9，栈视图如下：



查看 buffer 0x023239CD(长度为 0x1)的数据如下：



根据上述的截图发现：

第一次调用 WriteFile 的数据长度为 16 字节，恰好是 KD_PACKET 结构的大小。并且 buffer 中的数据很有特点：

- 开始四个字节是四个 0x30,和 PacketLeader 中的功能包相同。
- 后面的两个字节 00 02 则可能代表 PacketType 中的 PACKET_TYPE_KD_STATE_MANIPULATE
- 在后面的两个字节 00 38 则可代表了后面数据包的长度，第二次调用 WriteFile 的 buffer 的长度正是 0x38。

据此，可以推断第一次调用 WriteFile 是在发送一个 KD_PACKET 类型的数据包包头。

第二次调用 WriteFile 则正是在发送真正的数据，数据包的开头是 0x00 00 31 5C。这个恰好是 Manipulate 类型中的 DbgKdQueryMemoryApi。Windbg 在对目标系统的特定 API 函数下断点之前，要首先查询封包中所需要的信息。所以第一次发包并不一定是下断的包。并且，仔细观察包中的数据，可以发现一个 ULONG 类型的数据：0x 80 57 26 4C. 是不是很像一个函数地址。在 windbg 中输入命令 u nt!NtCreateFile

```
kd> u nt!NtCreateFile
nt!NtCreateFile:
8057264c 8bff          mov     edi,edi
8057264e 55            push    ebp
8057264f 8bec          mov     ebp,esp
80572651 33c0          xor     eax,eax
80572653 50            push    eax
80572654 50            push    eax
80572655 50            push    eax
80572656 ff7530         push    dword ptr [ebp+30h]
```

可见，该数据正是 NtCreateFile 在被调试系统的内存地址。至于其他的数据，均是 windbg 根据 DBGKD_MANIPULATE_STATE64 结构来填写的。有兴趣可以自己分析。

第三个数据包大小是 1，其数据包的内容正是标识数据结束的结束符 0xAA。这也印证了我们上面的分析。

最后，我们在 windbg 中验证一下上述的函数调用流程，下如下的断点：

```
kd> bp nt!KdReceivePacket
```

应该算是比较低层的函数了。然后 g,再然后 Ctrl+Break，断下目标系统的运行，输入命令 kn 得到 windbg 的输出：

```
kd> kn
```

```
 # ChildEBP RetAddr
00 80551234 8067e0bd nt!KdReceivePacket
01 80551354 805335a4 nt!KdpReportExceptionStateChange+0x8a
02 80551374 8067f137 nt!KdpReport+0x60
03 805513a0 804fb243 nt!KdpTrap+0x108
04 8055176c 804e0ada nt!KiDispatchException+0x129
05 805517d4 804e1208 nt!CommonDispatchException+0x4d
06 805517d4 804e4b26 nt!KiTrap03+0xad
07 8055184c 804e48a2 nt!RtlpBreakWithStatusInstruction+0x1
08 8055184c 806f3742 nt!KeUpdateSystemTime+0x165
09 805518d0 804dd0d7 hal!HalProcessorIdle+0x2
0a 805518d4 00000000 nt!KiIdleLoop+0x10
```

这个也正好是我们上面所分析的流程结果。

原打算对内核调试机制进行一个比较细致的分析的，无奈使用 windbg 对它本身调试用的 API 下断无疑是太岁头上动土，结果就是一次一次的 fatal System Error。暂时还没有想到好的下断的地方。

至于开篇 xueTr 所发现的那个 inline hook,应该是这样子：

由于我们下的条件断点对于 windbg 来说是按照如下的方法来处理的：

- 调用 KdpWriteBreakpoint 对被调试系统的 NtCreateFile 的函数头部写入 INT3(0xCC)
- 目标系统运行，碰到 INT3 发生异常。
- Windbg 截获到异常的数据判断是否满足我们设定的条件，并将结果发送给被调试系统。
- 被调试系统根据 windbg 的结果来选择是断下还是继续运行。

所以按照上述的陈述，NtCreateFile 的头部被改写成了 0xCC。所以 xueTr 认为是 Inline hook，并且由于没有 jmp 等跳转指令。其 hook 完的地址没有发生变化。

另外，KdpWriteBreakpoint 函数调用 KdpAddBreakpoint 改写函数的头部来添加断点，具体可以参考 ReactOS 的源码，不再赘述。

6、总结

第五部分的分析只是从黑盒的角度做了下简单的分析，由于调试过程中下断位置没有选好，导致目标系统频繁的崩溃，暂时我还没有很好的办法。除非选用其他类型的调试器。可惜时间不允许，所以这部分暂时如此，待我请教完高人后再继续分析。^_^

我是边学边分析的，入行没几个月，所以错误在所难免，如有错误或者更好的想法，欢迎交流，共同进步。