Windows内核调试器原理浅析

文章地址：http://www.ykzj.org/article.php?articleid=1847

SoBeIt

前段时间忽然对内核调试器实现原来发生了兴趣，于是简单分析了一下当前windows下主流内核调试器原理，并模仿原理自己也写了个极其简单的调试器:)

WinDBG

WinDBG和用户调试器一点很大不同是内核调试器在一台机器上启动，通过串口调试另一个相联系的以Debug方式启动的系统，这个系统可以是虚拟机上的系统，也可以是另一台机器上的系统(这只是微软推荐和实现的方法，其实象SoftICE这类内核调试器可以实现单机调试)。很多人认为主要功能都是在WinDBG里实现，事实上并不是那么一回事，windows已经把内核调试的机制集成进了内核，WinDBG、kd之类的内核调试器要做的仅仅是通过串行发送特定格式数据包来进行联系，比如中断系统、下断点、显示内存数据等等。然后把收到的数据包经过WinDBG处理显示出来。

在进一步介绍WinDBG之前，先介绍两个函数：KdpTrace、KdpStub，我在《windows异常处理流程》一文里简单提过这两个函数。现在再提一下，当异常发生于内核态下，会调用KiDebugRoutine两次，异常发生于用户态下，会调用KiDebugRoutine一次，而且第一次调用都是刚开始处理异常的时候。

当WinDBG未被加载时KiDebugRoutine为KdpStub，处理也很简单，主要是对由int 0x2d引起的异常如DbgPrint、DbgPrompt、加载卸载SYMBOLS(关于int 0x2d引起的异常将在后面详细介绍)等，把Context.Eip加1，跳过int 0x2d后面跟着的int 0x3指令。

[page]

真正实现了WinDBG功能的函数是KdpTrap，它负责处理所有STATUS_BREAKPOINT和STATUS_SINGLE_STEP(单步)异常。STATUS_BREAKPOINT的异常包括int 0x3、DbgPrint、DbgPrompt、加载卸载SYMBOLS。DbgPrint的处理最简单，KdpTrap直接向调试器发含有字符串的包。DbgPrompt因为是要输出并接收字符串，所以先将含有字符串的包发送出去，再陷入循环等待接收来自调试器的含有回复字符串的包。SYMBOLS的加载和卸载通过调用KdpReportSymbolsStateChange，int 0x3断点异常和int 0x1单步异常(这两个异常基本上是内核调试器处理得最多的异常)通过调用KdpReportExceptionStateChange，这两个函数很相似，都是通过调用KdpSendWaitContinue函数

。KdpSendWaitContinue可以说是内核调试器功能的大管家，负责各个功能的分派。这个函数向内核调试器发送要发送的信息，比如当前所有寄存器状态，每次单步后我们都可以发现寄存器的信息被更新，就是内核调试器接受它发出的包含最新机器状态的包；还有SYMBOLS的状态，这样加载和卸载了SYMBOLS我们都能在内核调试器里看到相应的反应。然后KdpSendWaitContinue等待从内核调试器发来的包含命令的包，决定下一步该干什么。让我们来看看KdpSendWaitContinue都能干些什么：

```
case DbgKdReadVirtualMemoryApi:
  KdpReadVirtualMemory(&ManipulateState,&MessageData,ContextRecord);
  break;

case DbgKdReadVirtualMemory64Api:
  KdpReadVirtualMemory64(&ManipulateState,&MessageData,ContextRecord);
  break;

case DbgKdWriteVirtualMemoryApi:
  KdpWriteVirtualMemory(&ManipulateState,&MessageData,ContextRecord);
  break;

case DbgKdWriteVirtualMemory64Api:
  KdpWriteVirtualMemory64(&ManipulateState,&MessageData,ContextRecord);
  break;

case DbgKdReadPhysicalMemoryApi:
  KdpReadPhysicalMemory(&ManipulateState,&MessageData,ContextRecord);
  break;

case DbgKdWritePhysicalMemoryApi:
  KdpWritePhysicalMemory(&ManipulateState,&MessageData,ContextRecord);
  break;

case DbgKdGetContextApi:
  KdpGetContext(&ManipulateState,&MessageData,ContextRecord);
  break;
```

```
case DbgKdSetContextApi:
   KdpSetContext(&ManipulateState,&MessageData,ContextRecord);
   break;

case DbgKdWriteBreakPointApi:
   KdpWriteBreakpoint(&ManipulateState,&MessageData,ContextRecord);
   break;

case DbgKdRestoreBreakPointApi:
   KdpRestoreBreakpoin(&ManipulateState,&MessageData,ContextRecord);
   break;

case DbgKdReadControlSpaceApi:
   KdpReadControlSpace(&ManipulateState,&MessageData,ContextRecord);
   break;

case DbgKdWriteControlSpaceApi:
   KdpWriteControlSpace(&ManipulateState,&MessageData,ContextRecord);
   break;

case DbgKdReadIoSpaceApi:
   KdpReadIoSpace(&ManipulateState,&MessageData,ContextRecord);
   break;

case DbgKdWriteIoSpaceApi:
   KdpWriteIoSpace(&ManipulateState,&MessageData,ContextRecord);
   break;

case DbgKdContinueApi:
   if (NT_SUCCESS(ManipulateState.u.Continue.ContinueStatus) != FALSE) {
      return ContinueSuccess;
   } else {
```

```
            return ContinueError;
        }
        break;

    case DbgKdContinueApi2:
        if (NT_SUCCESS(ManipulateState.u.Continue2.ContinueStatus) != FALSE) {
            KdpGetStateChange(&ManipulateState,ContextRecord);
            return ContinueSuccess;
        } else {
            return ContinueError;
        }
        break;

    case DbgKdRebootApi:
        KdpReboot();
        break;

    case DbgKdReadMachineSpecificRegister:
        KdpReadMachineSpecificRegister(&ManipulateState,&MessageData,ContextRecord);
        break;

    case DbgKdWriteMachineSpecificRegister:
        KdpWriteMachineSpecificRegister(&ManipulateState,&MessageData,ContextRecord);
        break;

    case DbgKdSetSpecialCallApi:
        KdSetSpecialCall(&ManipulateState,ContextRecord);
        break;

    case DbgKdClearSpecialCallsApi:
        KdClearSpecialCalls();
        break;
```

```
case DbgKdSetInternalBreakPointApi:
    KdSetInternalBreakpoint(&ManipulateState);
    break;

case DbgKdGetInternalBreakPointApi:
    KdGetInternalBreakpoint(&ManipulateState);
    break;

case DbgKdGetVersionApi:
    KdpGetVersion(&ManipulateState);
    break;

case DbgKdCauseBugCheckApi:
    KdpCauseBugCheck(&ManipulateState);
    break;

case DbgKdPageInApi:
    KdpNotSupported(&ManipulateState);
    break;

case DbgKdWriteBreakPointExApi:
    Status = KdpWriteBreakPointEx(&ManipulateState,
                        &MessageData,
                        ContextRecord);
    if (Status) {
        ManipulateState.ApiNumber = DbgKdContinueApi;
        ManipulateState.u.Continue.ContinueStatus = Status;
        return ContinueError;
    }
    break;

case DbgKdRestoreBreakPointExApi:
    KdpRestoreBreakPointEx(&ManipulateState,&MessageData,ContextRecord);
```

```
        break;

    case DbgKdSwitchProcessor:
        KdPortRestore ();
        ContinueStatus = KeSwitchFrozenProcessor(ManipulateState.Processor);
        KdPortSave ();
        return ContinueStatus;

    case DbgKdSearchMemoryApi:
        KdpSearchMemory(&ManipulateState, &MessageData, ContextRecord);
        break;
```

读写内存、搜索内存、设置/恢复断点、继续执行、重启等等，WinDBG里的功能是不是都能实现了?呵呵。

[page]

每次内核调试器接管系统是通过调用在KiDispatchException里调用KiDebugRoutine(KdpTrace)，但我们知道要让系统执行到KiDispatchException必须是系统发生了异常。而内核调试器与被调试系统之间只是通过串口联系，串口只会发生中断，并不会让系统引发异常。那么是怎么让系统产生一个异常呢?答案就在KeUpdateSystemTime里，每当发生时钟中断后在HalpClockInterrupt做了一些底层处理后就会跳转到这个函数来更新系统时间(因为是跳转而不是调用，所以在WinDBG断下来后回溯堆栈是不会发现HalpClockInterrupt的地址的)，是系统中调用最频繁的几个函数之一。在KeUpdateSystemTime里会判断KdDebuggerEnable是否为TRUE，若为TRUE则调用KdPollBreakIn判断是否有来自内核调试器的包含中断信息的包，若有则调用DbgBreakPointWithStatus，执行一个int 0x3指令，在异常处理流程进入了KdpTrace后将根据处理不同向内核调试器发包并无限循环等待内核调试的回应。现在能理解为什么在WinDBG里中断系统后堆栈回溯可以依次发现KeUpdateSystemTime->RtlpBreakWithStatusInstruction，系统停在了int 0x3指令上(其实int 0x3已经执行过了，只不过Eip被减了1而已)，实际已经进入KiDispatchException->KdpTrap，将控制权交给了内核调试器。

系统与调试器交互的方法除了int 0x3外，还有DbgPrint、DbgPrompt、加载和卸载symbols，它们共同通过调用DebugService获得服务。

```
NTSTATUS DebugService(
    ULONG   ServiceClass,
```

```
    PVOID   Arg1,
    PVOID   Arg2
  )
{
  NTSTATUS    Status;

  __asm {
    mov     eax, ServiceClass
    mov     ecx, Arg1
    mov     edx, Arg2
    int     0x2d
    int     0x3
    mov     Status, eax
  }
  return Status;
}
```

ServiceClass可以是BEAKPOINT_PRINT(0x1)、BREAKPOINT_PROMPT(0x2)、BREAKPOINT_LOAD_SYMBOLS(0x3)、BREAKPOINT_UNLOAD_SYMBOLS(0x4)。为什么后面要跟个int 0x3，M$的说法是为了和int 0x3共享代码(我没弄明白啥意思-_-)，因为int 0x2d的陷阱处理程序是做些处理后跳到int 0x3的陷阱处理程序中继续处理。但事实上对这个int 0x3指令并没有任何处理，仅仅是把Eip加1跳过它。所以这个int 0x3可以换成任何字节。

[page]

int 0x2d和int 0x3生成的异常记录结(EXCEPTION_RECORD)ExceptionRecord.ExceptionCode都是STATUS_BREAKPOINT(0x80000003)，不同是int 0x2d产生的异常的ExceptionRecord.NumberParameters>0且ExceptionRecord.ExceptionInformation对应相应的ServiceClass比如BREAKPOINT_PRINT等。事实上，在内核调试器被挂接后，处理DbgPrint等发送字符给内核调试器不再是通过int 0x2d陷阱服务，而是直接发包。用M$的话说，这样更安全，因为不用调用KdEnterDebugger和KdExitDebugger。

最后说一下被调试系统和内核调试器之间的通信。被调试系统和内核调试器之间通过串口发数据包进行通信，Com1的IO端口地址为0x3f8，Com2的IO端口地址为0x2f8。在被调试系统准备要向内核调试器发包之前先会调用KdEnterDebugger暂停其它处理器的运行并获取Com端口自旋锁(当然，这都是对

多处理器而言的)，并设置端口标志为保存状态。发包结束后调用KdExitDebugger恢复。每个包就象网络上的数据包一样，包含包头和具体内容。包头的格式如下：

```
typedef struct _KD_PACKET {
   ULONG PacketLeader;
  USHORT PacketType;
   USHORT ByteCount;
   ULONG PacketId;
   ULONG Checksum;
} KD_PACKET, *PKD_PACKET;
```

PacketLeader是四个相同字节的标识符标识发来的包，一般的包是0x30303030，控制包是0x69696969，中断被调试系统的包是0x62626262。每次读一个字节，连续读4次来识别出包。中断系统的包很特殊，包里数据只有0x62626262。包标识符后是包的大小、类型、包ID、检测码等，包头后面就是跟具体的数据。这点和网络上传输的包很相似。还有一些相似的地方比如每发一个包给调试器都会收到一个ACK答复包，以确定调试器是否收到。若收到的是一个RESEND包或者很长时间没收到回应，则会再发一次。对于向调试器发送输出字符串、报告SYMBOL情况等的包都是一接收到ACK包就立刻返回，系统恢复执行，系统的表现就是会卡那么短短一下。只有报告状态的包才会等待内核调试器的每个控制包并完成对应功能，直到发来的包包含继续执行的命令为止。无论发包还是收包，都会在包的末尾加一个0xaa，表示结束。

[page]

现在我们用几个例子来看看调试流程。

记得我以前问过jiurl为什么WinDBG的单步那么慢(相对softICE)，他居然说没觉得慢?*$&$^$^(&(&(我ft。。。现在可以理解为什么WinDBG的单步和从操作系统正常执行中断下来为什么那么慢了。单步慢是因为每单步一次除了必要的处理外，还得从串行收发包，怎么能不慢。中断系统慢是因为只有等到时钟中断发生执行到KeUpdateSystemTime后被调试系统才会接受来自WinDBG的中断包。现在我们研究一下为什么在KiDispatchException里不能下断点却可以用单步跟踪KiDispatchException的原因。如果在KiDispatchException中某处下了断点，执行到断点时系统发生异常又重新回到KiDispatchException处,再执行到int 0x3，如此往复造成了死循环，无法不能恢复原来被断点int 0x3所修改的代码。但对于int 0x1，因为它的引起是因为EFLAG寄存中TF位被置位,并且每次都自动被复位，所以系统可以被继续执行而不会死循环。现在我们知道了内部机制，我们就可以调用KdXXX函数实现一个类似WinDBG之类的内核调试器，甚至可以替换KiDebugRoutine(KdpTrap)为自己的函数来自己实现一个功能更强大

的调试器，呵呵。

[page]

SoftICE

SoftICE的原理和WinDBG完全不一样。它通过替换正常系统中的中断处理程序来获得系统的控制权，也正因为这样它才能够实现单机调试。它的功能实现方法很底层，很少依赖与windows给的接口函数，大部分功能的实现都是靠IO端口读写等来完成的。

SoftICE替换了IDT表中以下的中断(陷阱)处理程序：

0x1： 单步陷阱处理程序
0x2： NMI不可屏蔽中断
0x3： 调试陷阱处理程序
0x6： 无效操作码陷阱处理程序
0xb： 段不存在陷阱处理程序
0xc： 堆栈错误陷阱处理程序
0xd： 一般保护性错误陷阱处理程序
0xe： 页面错误陷阱处理程序
0x2d： 调试服务陷阱处理程序
0x2e： 系统服务陷阱处理程序
0x31： 8042键盘控制器中断处理程序
0x33： 串口2(Com2)中断处理程序
0x34： 串口1(Com1)中断处理程序
0x37： 并口中断处理程序
0x3c： PS/2鼠标中断处理程序
0x41： 未使用

(这是在PIC系统上更换的中断。如果是APIC系统的话更换的中断号有不同，但同样是更换这些中断处理程序)

其中关键是替换了0x3 调试陷阱处理程序和0x31 i8042键盘中断处理驱动程序(键盘是由i8042芯片控制的)，SoftICE从这两个地方获取系统的控制权。

启动softICE服务后SoftICE除了更换了IDT里的处理程序，还有几点重要的，一是HOOK了i8042prt.sys里的READ_PORT_UCHAR函数，因为在对0x60端口读后，会改变0x64端口对应控制寄存器的状态。所以在SoftICE的键盘中断控制程序读了0x60端口后并返回控制权给正常的键盘中断控制程序后，不要让它再读一次。还有就是把物理内存前1MB的地址空间通过调用MmMapIoSpace映射到虚拟的地址空间里，里面包括显存物理地址，以后重画屏幕就通过修改映射到虚拟地址空间的这段显存内容就行了。

如果显示模式是彩色模式，那么显存起始地址是0xb8000，CRT索引寄存器端口0x3d4，CRT数据寄存器端口0x3d5。如果显示模式是单色模式，那么显存起始地址是0xb0000，CRT索引寄存器端口0x3b4，CRT数据寄存器端口0x3b5。首先写索引寄存器选择要进行设置的显示控制内部寄存器之一(r0-r17)，然后将参数写到其数据寄存器端口。

[page]

i8042键盘控制器中断控制驱动程序在每按下一个键和弹起一个键都会被触发。SoftICE在HOOK了正常的键盘中断控制程序获得系统控制权后，首先从0x60端口读出按下键的扫描码然后向0x20端口发送通用EOI(0x20)表示中断已结束，如果没有按下激活热键(ctrl+d)，则返回正常键盘中断处理程序。如果是按下热键则会判断控制台(就是那个等待输入命令的显示代码的黑色屏幕)是否被激活，未被激活的话则先激活。然后设置IRQ1键盘中断的优先级为最高，同时设置两个8259A中断控制器里的中断屏蔽寄存器(向0x21和0xa1发中断掩码，要屏蔽哪个中断就把哪一位设为1)，只允许IRQ1(键盘中断)、IRQ2(中断控制器2级联中断，因为PS/2鼠标中断是归8259A-2中断控制器管的，只有开放IRQ2才能响应来自8259A-2管理的中断)、IRQ12(PS/2鼠标中断，如果有的话)，使系统这时只响应这3个中断。新的键盘和鼠标中断处理程序会建立一个缓冲区，保存一定数量的输入扫描信息。当前面的工作都完成后会进入一段循环代码，负责处理键盘和鼠标输入的扫描码缓冲区，同时不断地更新显存的映射地址缓冲区重画屏幕(这段循环代码和WinDBG里循环等待从串口发来的包的原理是一样的，都是在后台循环等待用户的命令)。这段循环代码是在激活控制台的例程里调用的，也就是说当控制台已被激活的话正常流程不会再次进入这段循环代码的(废话，再进入系统不就死循环了)。当有一个新的键按下时，都会重新调用一遍键盘中断处理程序，因为控制台已激活，所以它只是简单地更新键盘输入缓冲区内容然后iret返回。它并不会返回正常的键盘中断处理程序，因为那样会交出控制权(想证明这点也很简单，在SoftICE里断正常的键盘中断处理程序，然后g，1秒后在这里断下，这是我们可以F10，如果SoftICE会把控制权交给正常的键盘中断处理程序的话，在这里早就发生死循环了)。鼠标中断驱动也是一样。这个时候实际iret返回到的还是那段循环代码里面，所以被调试的代码并不会被执行，除非按下了F10之类的键，它会指示退出循环返回最开始时的中断处理程序，然后再iret返回最开始中断的地方。当然，因为设置了EFLAG里的TF位，执行了一个指令又会通过单步的处理程序进入那段循环的代码。

而处理int 0x3也差不多，若没有激活控制台则先激活并屏蔽除了键盘、鼠标及8259A-2中断控制器外的所有中断，然后进入那段循环代码。

作为对比同样来看一下在SoftICE里处理int 0x3和单步的过程。当执行到int 0x3时，激活控制台并屏蔽中断，然后将int 0x3指令前后范围的指令反汇编并写入显存映射地址空间，并把最新的寄存器值也写进去，最后在后台循环等待键盘输入命令。当命令是F10时，设置好EFLAG的TF位，清除8259A中断控制器里的中断屏蔽寄存器，开放所有中断，将控制台清除，从循环代码中返回新键盘(或int 0x3)中断处理程序，然后再返回到正常键盘(或int 0x3)中断处理程序，由这里iret到被中断代码处执行。执行了一个指令后因为发生单步异常又进入后台循环代码。

SoftICE里的单步比WinDBG要快得多的原因很简单，SoftICE只需要把反汇编出来的代码和数据经过简单处理再写入显存映射地址缓冲区里刷新屏幕就可以继续执行了，省略了串行的发包收包，怎么会不快。而中断系统更快，按下键中断就会发生，根本不用象WinDBG等时钟中断才能把系统断下来。

[page]

后记：

好象说得很简单，其实一个内核调试器实现起来极其复杂，没说得再详细，一是因为题目就叫“浅析”，就是类似于科普的东西；二是水平和时间有限(主要原因^^)；三是真要详细写起来就不是这几千字能说得明白的东西了。还有，反汇编ntice.sys真是一项艰巨的任务，比分析漏洞要复杂N倍，刚开始没着门道时真看得我头昏眼花。在此特别感谢Syser的作者，牛人就是牛人，在我对SoftICE工作原理的认识还处于混沌状态时，几句话点醒了我^^。因为水平有限难免有很多错漏，还忘高手指出:)

QQ:27324838
Email:27324838