

Windows 调试工具入门-7

驱动程序的源码调试

<http://www.DbgTech.net>

一、 准备

Windows 调试工具优于目前的其他内核调试器很重要的一点，就是能够非常方便的对自己编写的驱动程序进行源码调试。为了能够更好的说明，我们首先需要做一些准备工作，分别编写一个测试驱动程序和一个应用程序来使用驱动的功能。

1. 驱动程序

首先实现一个最简单的驱动程序，除了 `DriverEntry` 等框架代码之外，我们添加一个 `IRP_MJ_READ` 的 `Dispatch` 例程，当应用程序调用 `ReadFile` 时返回一个值递增的字节。另外，实现两个 `DeviceIoControl` Code，一个调用 `DbgPrint` 向调试器显示信息并返回，另一个访问非法指针造成崩溃。代码片断如下，完整的代码和编译出来的文件可以在附件中获取：

```
NTSTATUS DispatchRead (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    PIO_STACK_LOCATION pstIrpStack;
    PUCCHAR pbyUserBuffer;
    ULONG ulSize;
    BOOLEAN bIRtn = FALSE;
    static UCHAR s_byCounter = 0;

    pstIrpStack = IoGetCurrentIrpStackLocation( Irp);
    pbyUserBuffer = (PUCCHAR)Irp->UserBuffer;
    ulSize = pstIrpStack->Parameters.Read.Length;

    if ( ulSize == 1)
    {
        *pbyUserBuffer = s_byCounter++;
        bIRtn = TRUE;
    }

    if ( bIRtn)
```

```

    {
        Irp->IoStatus.Status = STATUS_SUCCESS;
        Irp->IoStatus.Information = 1L;
        IoCompleteRequest( Irp, 0 );
        return STATUS_SUCCESS;
    }
    else
    {
        Irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
        Irp->IoStatus.Information = 0L;
        IoCompleteRequest( Irp, 0 );
        return STATUS_UNSUCCESSFUL;
    }
}

NTSTATUS DispatchIoCtrl (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    NTSTATUS ntStatus = STATUS_UNSUCCESSFUL;
    PCHAR pucCrash = NULL;

    PIO_STACK_LOCATION irpStack = IoGetCurrentIrpStackLocation(Irp);
    //////////////////////////////////////
    switch(irpStack->Parameters.DeviceIoControl.IoControlCode)
    {
        case IOCTL_TEST_CTRL_CODE2:
            //崩溃
            *pucCrash = 1;
            break;
        case IOCTL_TEST_CTRL_CODE1:
            DbgPrint( "Received IOCTL_TEST_CTRL_CODE1\n");
            ntStatus = STATUS_SUCCESS;
            break;
        default:

```

```
        ntStatus = STATUS_SUCCESS;
    }

    //////////////////////////////////////

    //

    if(ntStatus == STATUS_SUCCESS)
        Irp->IoStatus.Information =
        irpStack->Parameters.DeviceIoControl.OutputBufferLength;
    else
        Irp->IoStatus.Information = 0;

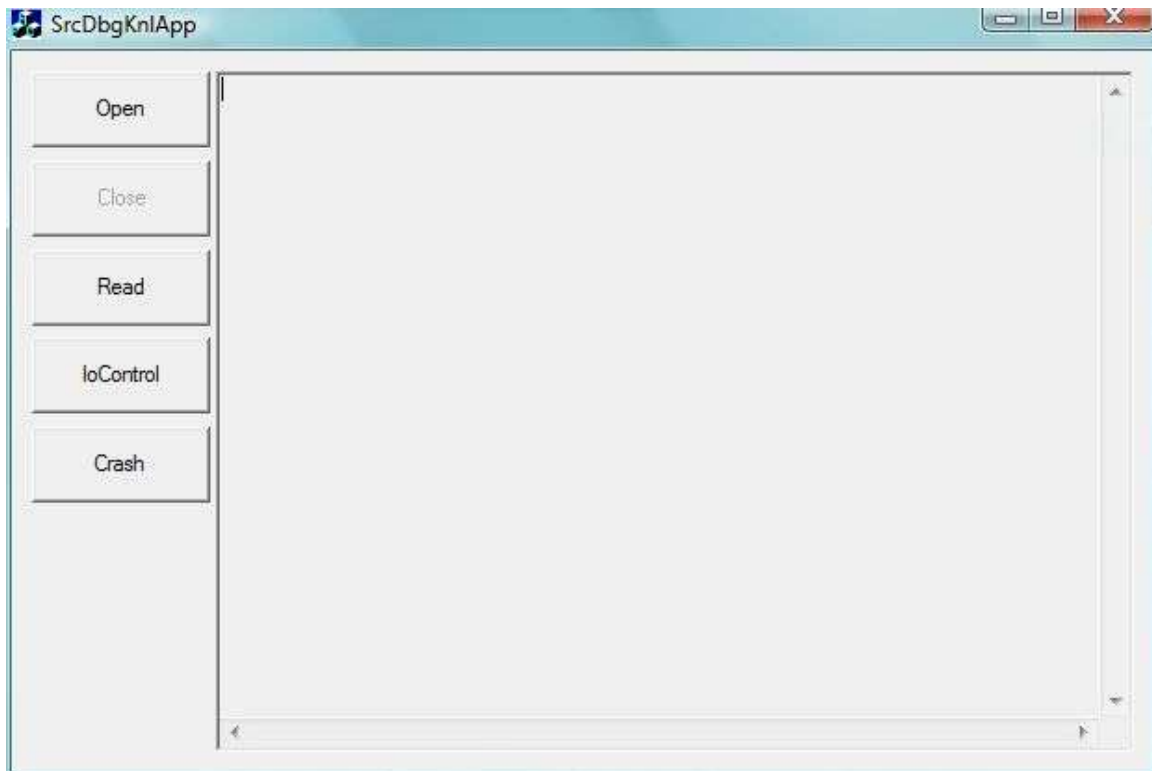
    Irp->IoStatus.Status = ntStatus;

    IoCompleteRequest( Irp, 0 );

    return ntStatus;
}
```

2. 应用程序

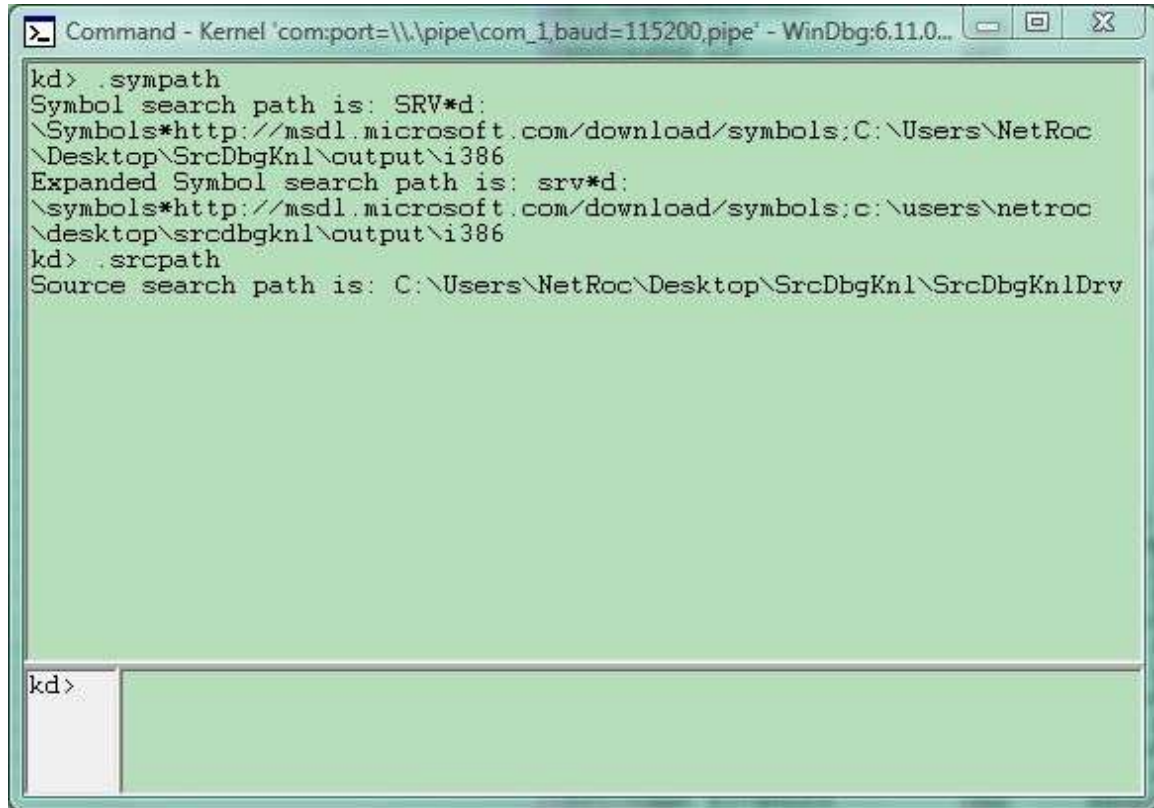
应用程序主要用来控制驱动设备对象，显示返回的结果。



二、 开始源码调试

1. 设置

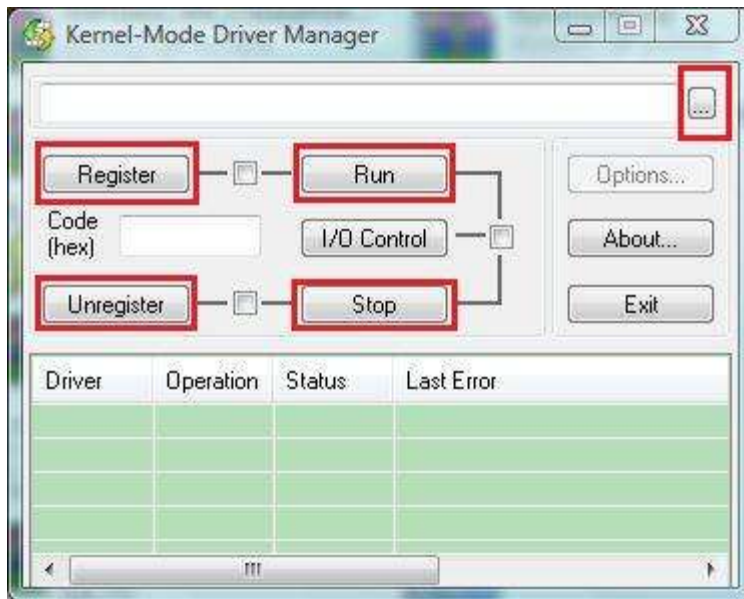
系列前面的几篇文章已经介绍过对符号路径和源码路径的设置。调试自己编写的驱动时，如果是主控机上编译，在目标机上运行，那么一般都不需要专门设置路径 WinDbg 就能找到正确的符号和源文件。如果驱动不是在主控机上编译的，或者编译之后移动了源码或符号文件就必须要进行设置。在这里我是这样设置的：



```
Command - Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.11.0...
kd> .sympath
Symbol search path is: SRV*d:
\Symbols*http://msdl.microsoft.com/download/symbols;C:\Users\NetRoc
\Desktop\SrcDbgKnl\output\i386
Expanded Symbol search path is: srv*d:
\symbols*http://msdl.microsoft.com/download/symbols;c:\users\netroc
\desktop\srcdbgknl\output\i386
kd> .srcpath
Source search path is: C:\Users\NetRoc\Desktop\SrcDbgKnl\SrcDbgKnlDrv
kd>
```

2. 加载驱动程序

这里我们不自己写程序加载，而是通过附件中的 KmdManager.exe 工具来进行。



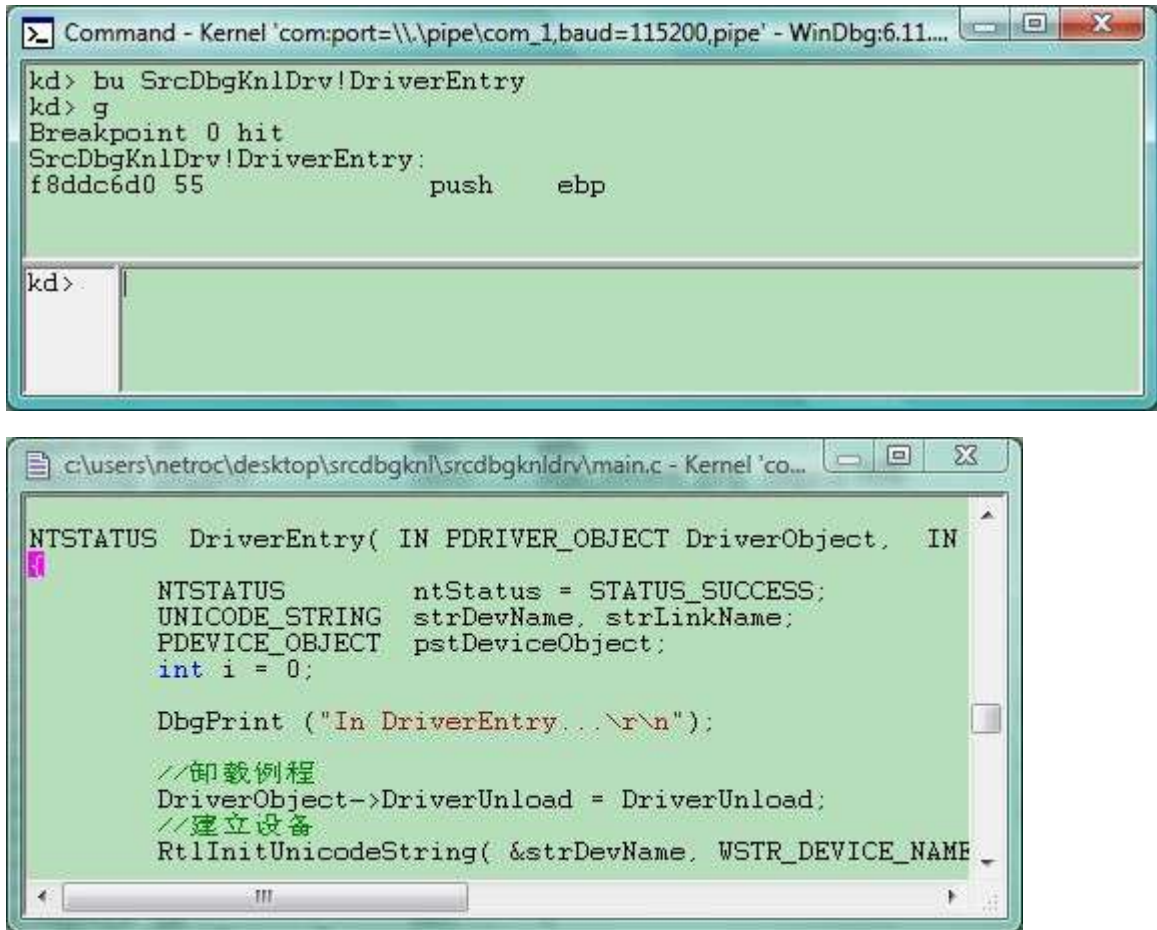
选择要加载的驱动程序之后,首先点击 **Register** 就能在注册表中注册该驱动程序的服务。然后点击 **Run** 加载并运行。按下 **Run** 之后,驱动的 **DriverEntry** 就会被调用,所以如果我们要想在驱动入口点中断下来的话,就要在这之前设置断点。**Stop** 和 **Unregister** 按钮用来停止和卸载驱动服务。注意操作时要按照 **Register->Run->Stop->Unregister** 的顺序,并且 **Stop** 之前要在应用层关闭所有已打开的句柄,否则可能只有重起机器才能卸载了。

3. 设置断点

初学者调试驱动时常常会问的一个问题就是,如何在 **DriverEntry** 运行之前中断下来。我们在实际的调试工作中也经常需要调试 **DriverEntry**。这需要用到以前介绍过的未定断点,因为一般来说 **DriverEntry** 运行之前,我们的驱动还没有被加载进来,这时在调试器是找不到这个驱动的任何符号的。

首先在调试器命令窗口输入 [bu SrcDbgKn!Drv!DriverEntry](#) 命令, F5 运行起来。在目标机中用 **KmdManager.exe** 工具 **Register** 并 **Run**。这时会发现目标机中断到了调试器中。

如果前面的路径设置没有问题的话,WinDbg 中会自动弹出源码,可以看到目前中断在 **DriverEntry** 函数入口处。



从这时起,就可以使用各个 WinDbg 的调试窗口了,比如查看局部变量、查看内存等等,也可以打开各个源文件直接针对源码设置断点。

三、 查看驱动设备的信息

运行到 `DriverEntry` 时, `SrcDbgKnIDrv.sys` 已经被成功加载到系统中了。`DriverEntry` 运行完成之后就能够看到我们的驱动创建的设备了。

使用 `gu` 命令,执行到 `DriverEntry` 返回处。使用 `!drvobj` 和 `!devobj` 命令可以查看驱动程序对象和设备对象的信息。

```
kd> !drvobj SrcDbgKnIDrv
```

```
Driver object (8219f5f0) is for:
```

```
\Driver\SrcDbgKnIDrv
```

```
Driver Extension List: (id , addr)
```

```
Device Object list:
```

```
82138030
```

```
kd> !devobj SrcDbgKnIDrv
```

```
Device object (82138030) is for:
```

```
SrcDbgKnIDrv \Driver\SrcDbgKnIDrv DriverObject 8219f5f0
```

```
Current Irp 00000000 RefCount 0 Type 00000022 Flags 000000c0
```

```
Dacl e10361f4 DevExt 00000000 DevObjExt 821380e8
```

```
ExtensionFlags (0000000000)
```

```
Device queue is not busy.
```

如果我们的设备有附加到某个设备栈上的话，可以用 [!devstack](#) 扩展命令显示设备栈的信息。

```
kd> !devstack 82138030
```

```
!DevObj    !DrvObj          !DevExt    ObjectName
```

```
> 82138030  \Driver\SrcDbgKnIDrv00000000  SrcDbgKnIDrv
```

这里看到的输出说明 SrcDbgKnIDrv 没有附加到任何设备栈。

通过 [!devhandles](#) 命令可以查看设备被打开的句柄。目前在这个地方使用的话，由于没有句柄被打开，还看不到什么有用的信息，在后面进行演示。

四、 Dispatch 例程的调试

Dispatch 例程是驱动程序响应应用层请求的地方。通常来说，驱动程序和应用程序的各种交互就是通过 Dispatch 例程实现。因此，要跟踪应用层请求处理的情况，可以在这些 Dispatch 例程上设置断点。

我们的测试程序只处理了 IRP_MJ_READ 和 IRP_MJ_DEVICE_CONTROL 两个请求。来试着调试一下。

接着上面中断下来的调试会话，首先用 [bp SrcDbgKnIDrv!DispatchRead](#) 命令和 [bp SrcDbgKnIDrv!DispatchIoCtrl](#) 命令在程序的两个 Dispatch 例程上下断。F5 运行起来。在目标机打开 SrcDbgKnIApp 程序点击 Open，会打开一个 SrcDbgKnIDrv.sys 驱动设备的句柄。这时中断目标机，使用 [!devhandles](#) 扩展命令就能看到有用的信息了：

```
kd> !devhandles 82138030
```

```
Checking handle table for process 0x823b97c0
```

```
Handle table at e1002000 with 241 Entries in use
```

```
省略掉一部分……
```

```
Checking handle table for process 0x82164da0
```

```
Handle table at e1137000 with 34 Entries in use
```

```
PROCESS 82164da0  SessionId: 0  Cid: 05e4  Peb: 7ffd6000  ParentCid: 05d0
```



```
DirBase: 07480240 ObjectTable: e11c1cd0 HandleCount: 34.
```

```
Image: SrcDbgKnIAppD.exe
```

```
008c: Object: 821cb028 GrantedAccess: 0012019f
```

命令显示 EPROCESS 地址为 0x82164da0 的进程打开了一个 \\.\SrcDbgKnIDrv 设备的句柄。再使用 [!process 82164da0](#) 扩展命令可以看到，这个进程正是我们的 SrcDbgKnIApp。在调试驱动程序的时候，这是一种查看哪些应用程序在使用某个驱动设备对象的方法。

接下来继续，F5 运行，点击 SrcDbgKnIApp 的 Read 按钮，可以看到中断到了 DispatchRead 例程上。这里就可以用单步、查看局部变量等等普通的方法调试我们的程序运行情况了。对于驱动程序来说，我们还可以做一些其他事情，例如查看 IRP 的信息，查看 IRQL 等。如：

```
kd> !irp @@(Irp)
```

```
Irp is active with 1 stacks 1 is current (= 0x8238d2d0)
```

```
No Mdl: No System Buffer: Thread 82191da8: Irp stack trace.
```

```
cmd flg cl Device File Completion-Context
```

```
>[ 3, 0] 0 0 82138030 821cb028 00000000-00000000
```

```
\Driver\SrcDbgKnIDrv
```

```
Args: 00000001 00000000 00000000 00000000
```

```
kd> !irql
```

```
nt!_KPRCB.DebuggerSavedIRQL not found, error : 0x4.
```

```
Saved IRQL not available prior to Windows Server 2003
```

!irql 命令只能对 Windows 2003 之后的系统使用，我的机器上是 Windows XP，因此这里提示了错误。

另外，也可以使用 `kd> !devobj @@(DeviceObject)` 这样的命令形式，直接用 C++ 表达式将局部变量作为命令参数来使用。

用 `bc*` 命令清除断点，F5 运行。每次点击 Read 都可以看到从驱动中读取出来的数据。

五、驱动异常和内核崩溃转储文件的源码调试

当驱动程序中发生异常时，如果有调试器附加上去，一般会中断到调试器中。如果没有附加调试器，系统通常会自动为我们生成内核 dump 文件。如果异常发生在我们的驱动中，那么也可以用 WinDbg 源码调试直观的看出问题所在。

在 SrcDbgKnIApp 程序中点击 Crash。由于现在是在调试会话中，可以在调试器命令窗口中看到异常的信息：

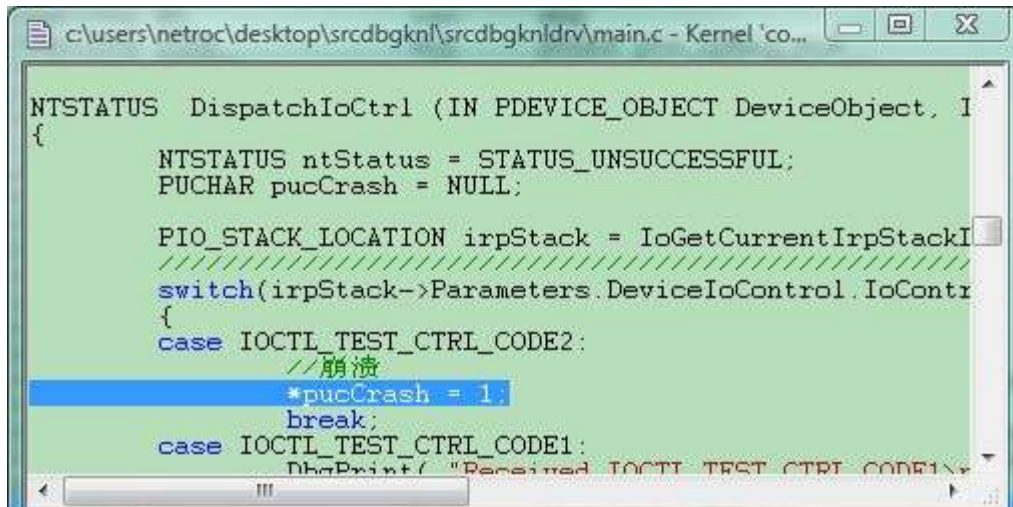
```
kd> g
```

```
Access violation - code c0000005 (!!! second chance !!!)
```

```
SrcDbgKnIDrv!DispatchIoCtrl+0x3d:
```

```
f8d615ad c60101 mov byte ptr [ecx],1
```

源码窗口中可以看到崩溃处的代码：



```
c:\users\netroc\desktop\srcdbgknl\srcdbgknl\drv\main.c - Kernel 'co...
NTSTATUS DispatchIoCtrl (IN PDEVICE_OBJECT DeviceObject, I
{
    NTSTATUS ntStatus = STATUS_UNSUCCESSFUL;
    PCHAR pucCrash = NULL;

    PIO_STACK_LOCATION irpStack = IoGetCurrentIrpStackI
    ///////////////////////////////////////////////////
    switch(irpStack->Parameters.DeviceIoControl.IoContr
    {
    case IOCTL_TEST_CTRL_CODE2:
        //崩溃
        *pucCrash = 1;
        break;
    case IOCTL_TEST_CTRL_CODE1:
        DbgPrint( "Received IOCTL_TEST_CTRL_CODE1\n");
    }
```

pucCrash 的值为 NULL，造成了崩溃：

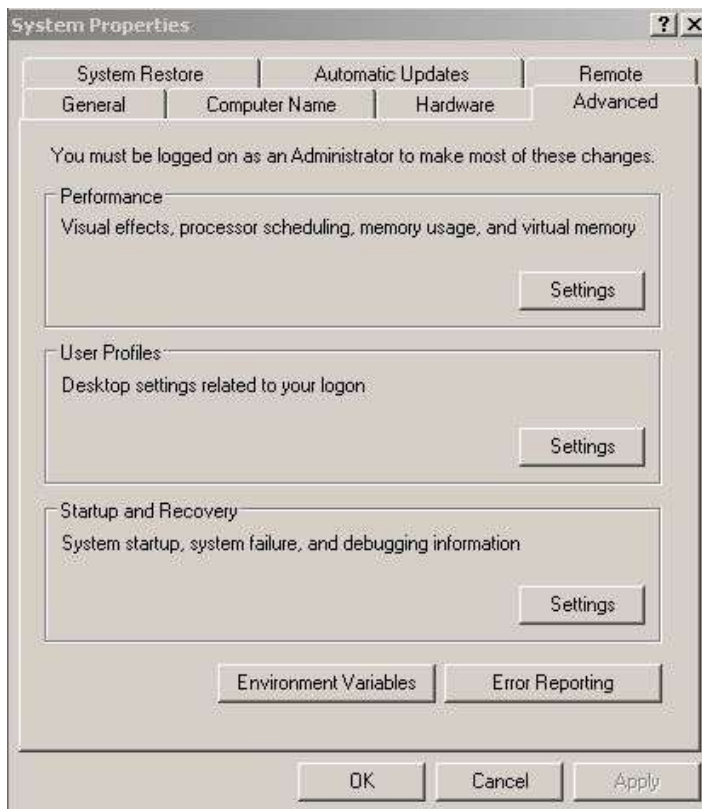
```
kd> ?? pucCrash
```

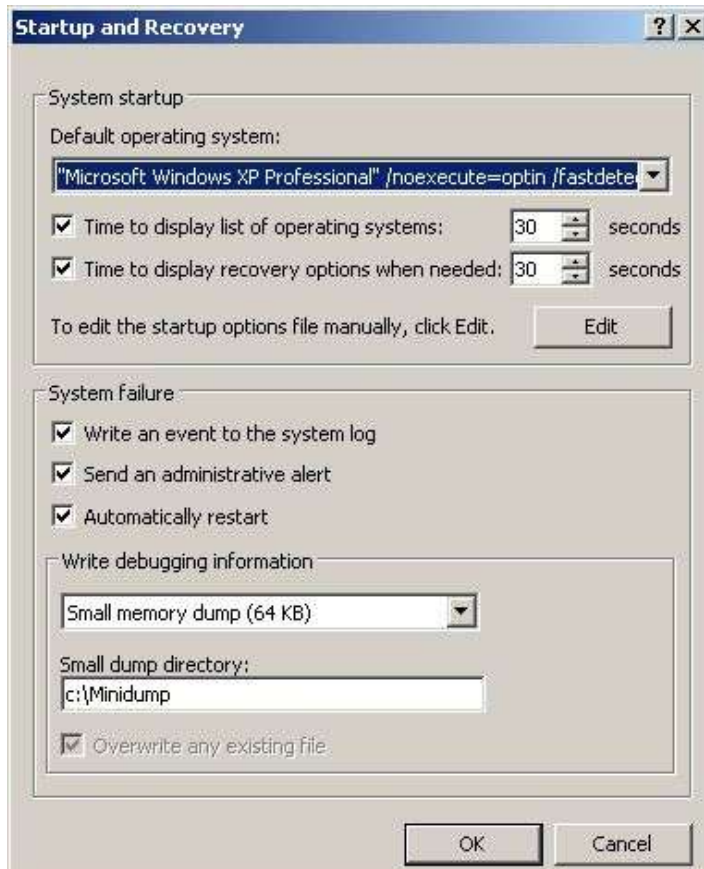
```
unsigned char * 0x00000000
```

```
""
```

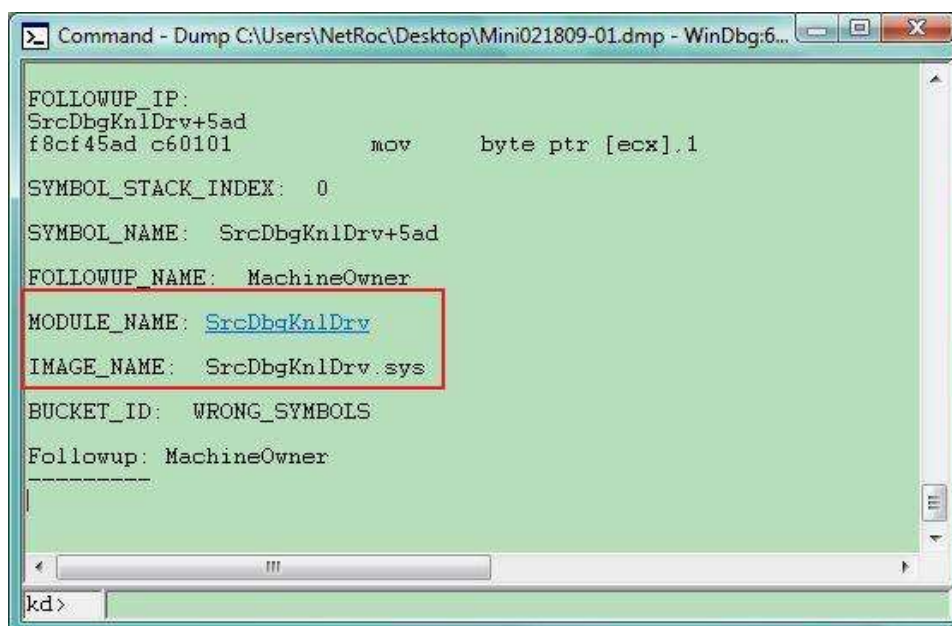
如果是实际工作时进行调试的话，这时可以继续分析崩溃的原因。

再来尝试一下没有附加调试器的情况。重起目标机，在我的电脑->属性->高级中设置一下内核 dump 的类型和生成的路径：





再使用刚才的方法，加载驱动程序后，用 `SrcDbgKnIApp` 让目标机崩溃。发生蓝屏之后重启，在设置的路径下就可以找到 `dump` 文件了。这里我们生成的是 `minidump`。将它拷贝到主控机上来，通过 `WinDbg` 菜单打开。如果驱动不是在本机上编译的话，因为还没有设置各种路径，现在看到的只有汇编代码。我们使用一下 [!analyze -v](#) 命令能够看到没有符号时对异常的分析结果，可以看到出问题的模块是 `SrcDbgKnIDrv.sys`：



接下来设置好可执行映像路径、符号路径和源码路径,使用`.reload`命令重新加载符号,会自动弹出源码窗口,调试器中也会更新显示符号信息。如果再次使用`!analyze -v`,能够看到非常详细的分析报告。这时还可以查看异常的每个堆栈帧中的源码和符号化信息。对于调试 Dump 文件中的 BUG 来说,使用源码可以大大提高调试效率。

关于内核模式 Dump 文件的调试这里就不再介绍了。Windows 调试工具的帮助文档中有相关技术较详细的资料。