Further Adventures in Debugging

Ryan Mangipana

(Reprinted from WindowsItPro Magazine)

How many times have you faced a problem where no error information was displayed on screen and related logs provided no data to help trace the failure? Here I'll introduce you to five awesome tips that intermediate problem solvers who are new to debugging can use as a starting point in learning to reverse-engineer this type of problem. You can apply these tips to any program that you're debugging. Although these tips are generic, I'll demonstrate them by using a real application that I support—Device Manager—which you're probably familiar with. I'll spare you the mind-numbing walk-through of the entire assembly-level debug of this particular problem and instead focus on sharing some basic debugging techniques that you'll need to be aware of as you begin crossing over into the intangible binary world of debugging.

Tip 1: Open a Process In The Debugger

When no information is output to the system about a problem, you can use the debugger (windbg.exe) as a powerful tool to identify what's going on in the process. (For more information about downloading and getting started using the debugger, see "Administrators' Intro to Debugging.") Before launching a process in the debugger, you'll need to obtain the command line that you will need to type into windbg to launch that process. You can find the command line by using Process Explorer; to obtain the command line, simply double-click the process, and you'll see the command line as displayed in Figure 1.

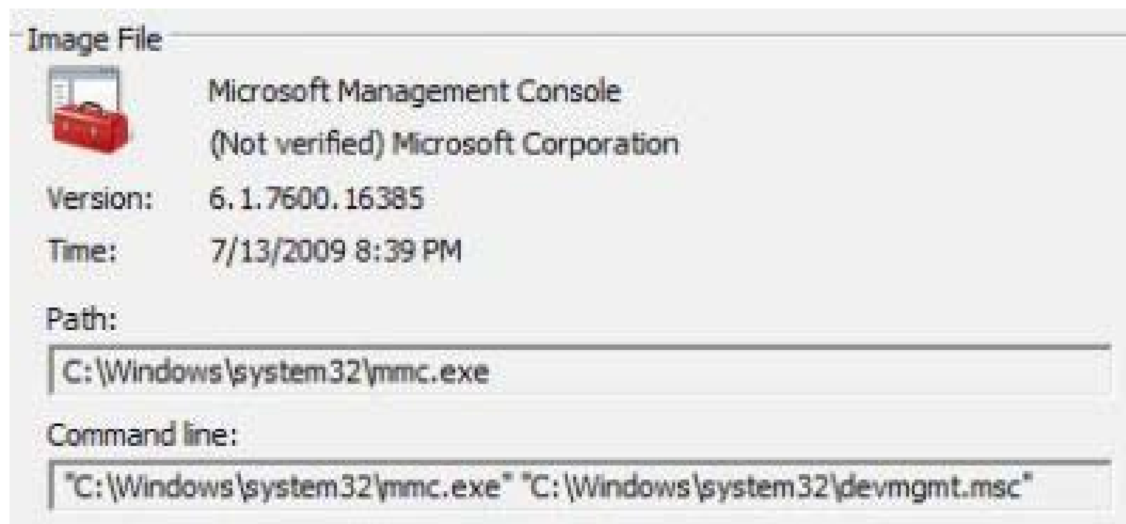

Figure 1
Command line to launch a process

After opening the Windows debugger from the Debugging Tools for Windows Start menu group, you can launch Device Manager by selecting Open Executable from the File menu, as Figure 2 shows.

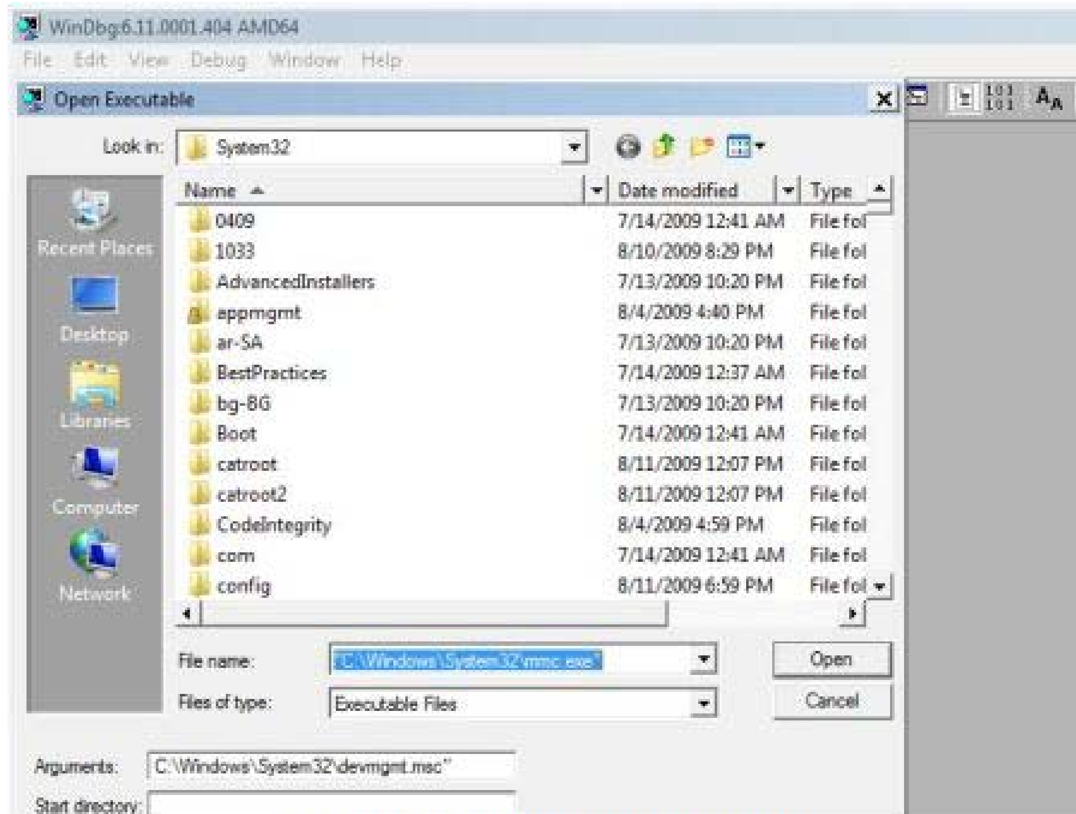

Figure 2: Launching Device Manager

Enter the command line that you'd normally use to start the process.

Tip 2: Find Out As Much As You Can Before Debugging

Before jumping into the debugger, get some basic information about the code you want to study. Determining where to start debugging often begins outside the debugger. You need a way to determine the names of functions related to your problem. For example, if your application is reporting an error stating it was unable to open a registry key, your goal is to identify the function that's used to open registry keys. So how do you know what functions are used for different tasks? Although the function names provide some clues, you can use MSDN to research what calls are available. For example, a quick MSDN search on "registry functions" would locate the MSDN documentation listing these functions at http://msdn.microsoft.com/en-us/library/ms724875(VS.85).aspx. You'd see that RegOpenKeyEx is the function used to open registry keys.

You can use the free Dependency Walker tool (depends.exe) to obtain information about relevant functions. Dependency Walker displays what DLLs a binary uses and the function names that the binary uses from the DLL. Obtaining this information is easy; launch depends.exe, then open the binary file that you're troubleshooting by using the open command from the File menu. Dependency

Walker will then display the names of the functions that this application calls when it executes. This information is important to your debugging because it lets you identify interesting calls that may be related to the problem. For example, if your application is popping up a message stating that the network connection attempt failed, you'd search Dependency Walker's output for function names that appear related to opening network connections. You can then use the debugger to investigate these calls as they're made.

As an example, let's use Dependency Walker to open devmgr.dll. This is the binary comprising the code that mmc.exe uses to create the Device Manager snap-in. As you can see in Figure 3, Dependency Walker shows that devmgr.dll imports various functions related to device enumeration from setupapi.dll.

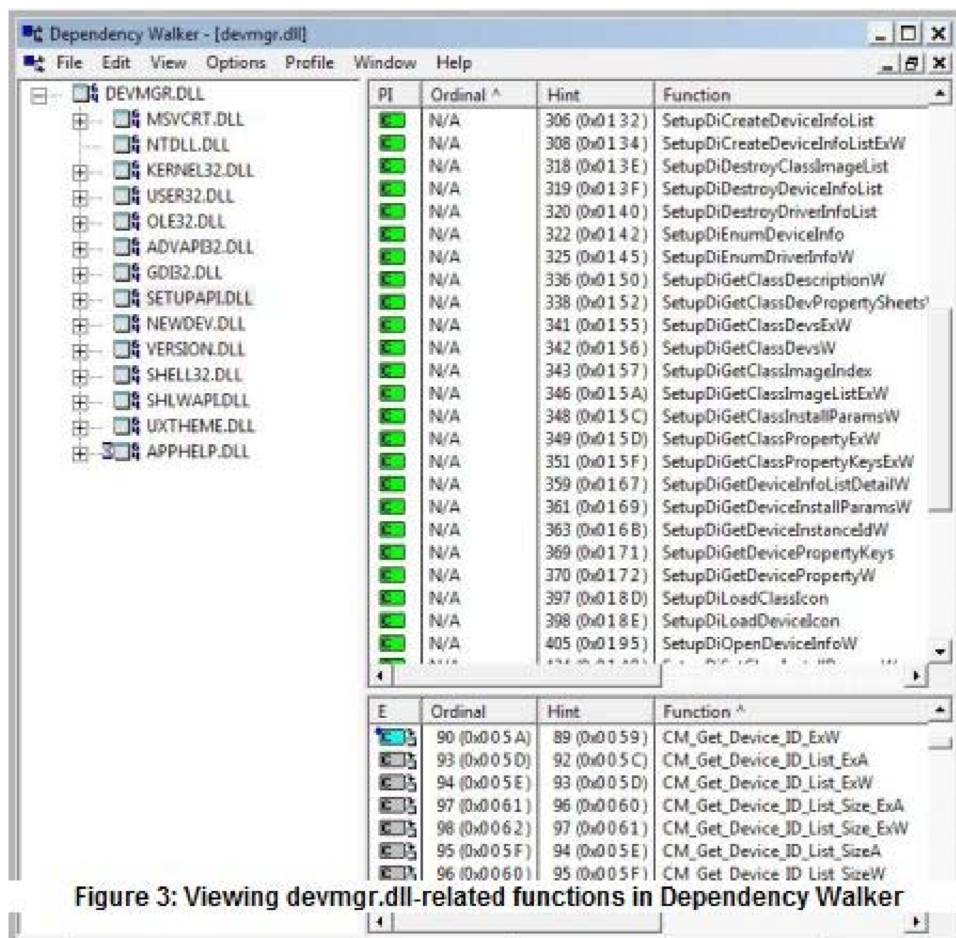

Figure 3: Viewing devmgr.dll-related functions in Dependency Walker

In case you're wondering how I determined that devmgr.dll is the DLL used to create Device Manager, devmgmt.msc is actually an XML file that lists devmgr.dll in the text. You can use Notepad to open it, as Figure 4 shows.

Figure 4
Devmgmt.msc contents showing devmgr.dll

Tip 3: Set Breakpoints

Once you start a process in the debugger, the debugger will stop at an initial breakpoint during process initialization. However, this breakpoint is not usually the best place to start debugging. Execution of a program typically consists of many different assembly instructions and function calls. However, only a small number of these may be related to the problem at hand. You need a way to get the debugger to allow the program to run until the functions that you've identified as relevant (by using depends.exe) are encountered. To accomplish this, you set breakpoints.

You can set a breakpoint against a function by using the bp (set breakpoint) command. Then you use the g (go) command to resume execution of the threads in the process so that they can continue running until something causes the debugger to break-in again. Here are the commands and output:

```
0:000> bp setupapi!CM_Get_Device_ID_List_ExW
0:000> g
Breakpoint 0 hit
```

When this breakpoint is hit, you'll be at the beginning of the function call that you're interested in. In tips 4 and 5, we'll review some commands you can run once you get to these locations.

As you can see in the previous output, the debugger informed us that we've hit breakpoint zero. You can list the breakpoints by using the bl (breakpoint list) command. We have only one breakpoint, which is numbered as zero.

```
0:000> bl
0 e 770edf2d  0001 (0001)  0:**** setupapi!CM_Get_Device_ID_List_ExW
```

So how can you search for the names of the functions that you might want to set breakpoints against? The x (examine symbols) command can use the symbol information to obtain functions and other data matching a wildcard pattern. The example in Figure 5 lists all symbol data matching the wildcard pattern *Devices* from the devmgr module. You can then set breakpoints against any of these functions.

```
0:000> x devmgr!*Devices*
72af71a9 devmgr!CMachine::CreateClassesAndDevices = <no type information>
```

```
72aef942 devmgr!CClass::GetNumberOfDevices = <no type information>
72af0810 devmgr!ViewDevicesMenuItems = <no type information>
72af65fd devmgr!CMachine::DestroyClassesAndDevices = <no type information>
```

Figure 5
Using the x debugger command

If devmgr.dll hasn't yet been loaded into the process, this command will fail. In such situations, you'll need to instruct the debugger to halt when a specific module is loaded. The following command will cause the debugger to break-in when setupapi.dll is loaded:

```
0:000> sxe ld:setupapi
0:000> g
ModLoad: 770e0000 771e8000   c:\windows\system32\setupapi.dll
```

Tip 4: Identify Call Flow

Once you've hit your breakpoint, you can find out what called the function and what the function calls (i.e., call flow) by examining the stack using the kC (display stack back trace) command. In the example in Figure 6, I ran the kC command after hitting a breakpoint that I had set against setupapi!PNP_GetDeviceList. Stacks grow upward. This means that the top-listed function was the last one called. The output in Figure 6 shows the stack after hitting a breakpoint set against setupapi!PNP_GetDeviceList. Devmgr.dll has called into setupapi.dll to enumerate the list of devices.

```
0:000> kC
setupapi!PNP_GetDeviceList
setupapi!CM_Get_Device_ID_List_ExW
setupapi!SetupDiGetClassDevsExW
devmgr!CMachine::DiGetClassDevs
devmgr!CMachine::CreateClassesAndDevices
```

Figure 6
kC command output

To identify the calls a function makes by watching and logging its execution, you can use one of the most powerful commands in the Windows debugger: wt (watch trace). You can run this command from the beginning of a function call; doing so will output the calls made by this function to the screen. In the example in Figure 7, I used the -l2 parameter to limit the output depth to two levels. In this example, setupapi!PNP_GetDeviceList called setupapi!NdrClientCall2, which then called rpcrt4!NdrClientCall2.

```
0:000> wt -l2
Tracing setupapi!PNP_GetDeviceList to return address 770edf88
   10      0 [  0]    setupapi!PNP_GetDeviceList
    1      0 [  1]            setupapi!NdrClientCall2
```

```
    3     0 [  1]                   rpcrt4!NdrClientCall2
<Omitting lengthy output>
```

Figure 7
wt command output

Tip 5: Identify Whether A Function Call Returned An Error

Once you hit a breakpoint that you set for a function, how do you identify whether these functions have returned an error code? You use the gu (go up) command to return from the function, then use the r command to examine the return value. Figure 8 shows the commands' output.

```
:000> bp setupapi!PNP_GetDeviceList
0:000> g
Breakpoint 0 hit
0:000> gu
0:000> r $retreg
$retreg=0000001d
```

Figure 8
Using the gu and r commands

The gu command resumes execution until the current function returns. In this case, the gu command runs the PNP_GetDeviceList function, then stops breaks-in immediately when the function is done. The r (register) command outputs the contents of registers. $retreg represents the return register, which can be used to identify whether a function has finished successfully or returned an error. We received an error 0x1d from PNP_GetDeviceList(). I located the return value for PNP_GetDeviceList documented at msdn.microsoft.com/en-us/library/cc239018(PROT.10).aspx: An error occurred during an attempt to read the registry.

Final Steps

The device manager issue was resolved by using the p (step) command to trace through the execution of the function. The debug trace session showed that setupapi!PNP_GetDeviceList had made an RPC call directed to interface 8d9f4e40-a03d-11ce-8f69-08003e30051b. With a little help from Process Monitor, I found that this RPC call was answered by the function umpnpmgr.dll!PNP_GetDeviceList(), which was running in the services.exe process. This call had failed with NAME_NOT_FOUND because of registry corruption. I rebooted using the Last Known Good registry configuration. Problem solved!