

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Note: Several of the figures mentioned are located at the end of this article.

SUMMARY

DLL conflict problems can be tough to solve, but a large number of tools are available to help. There are also many Windows APIs that you can use to build custom debugging tools. Three such tools are discussed here and provided as samples. DllSpy lists all the DLLs loaded in the system and the processes that are using them. ProcessSpy enumerates the running processes and exposes the DLLs they are using, and ProcessXP displays the list of concurrent running sessions for Windows XP.

DLL Hell is nothing new. When you rely upon DLLs from external sources, you can get all kinds of problems, like missing entry points or incompatible versions of the library. .NET allows side-by-side execution of components, alleviating these versioning problems, but if you're not yet deploying to .NET, what do you do? Well, you can trace DLL dependencies using a variety of tools. But when you use standard tools to trace this information, you might not get all the information you're after. And many tools don't provide functionality you need, like automatic logging, trace analysis, console-only operation, and scriptability.

I'll take a look at some of the available tools for examining running processes, and describe three I've developed: DllSpy, ProcessSpy, and ProcessXP, which can be used in your own debugging and development efforts.

The Existing Tools

Depends.exe is one tool that comes with Visual C++. Perhaps the simplest tool, it lists the DLLs that an application or another DLL needs. (Look at <http://www.dependencywalker.com> for updated versions.) If you need the full path of the DLLs and executables, simply right-click on the DLL name in the list or the tree and select Full Path from the context menu, as shown in Figure 1.

While tools like Depends.exe are fine for static dependencies (those constructed during the link process), except with profiling in newer versions, they are useless if you are using COM objects instantiated by the runtime or dynamically loading a DLL to call a particular function via LoadLibrary and GetProcAddress. You don't know when or from which folder this type of DLL will be loaded.

One way to identify dynamically loaded DLLs is to find out which DLL is loaded by each process. The SysInternals Web site, at <http://www.sysinternals.com>, provides LISTDLLS.EXE, a console-mode tool, and a GUI tool called Process Explorer that does this and much more (see Figure 2).

In addition to listing the DLLs used by a process, Process Explorer lets you know which kernel objects are consumed. Since version 3.11, Process Explorer has allowed you to easily detect the new or unused objects between two snapshots.

Occasionally, a DLL is loaded for a short period of time and then released before you can detect it using Process Explorer. When this happens, you need another kind of tool, which I'll discuss in my next article.

To navigate through processes and DLLs, you first need to know which processes are using each loaded DLL. You can use my sample program, DllSpy, to discover them (see Figure 3). In DllSpy an upper pane lists every loaded DLL and a lower pane enumerates the processes that are using the selected DLL.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

My ProcessSpy tool exposes the opposite relationship between processes and DLLs (see Figure 4). It enumerates the running processes in the upper pane; the lower pane lists each DLL that is used by a selected process, focusing on the real load address rather than the preferred load address, and static rather than dynamic dependencies.

These tools are available in the code download (available at the link at the top of this article). If these don't meet your needs but you want something similar, you'll need to know how to get information about and enumerate running processes. Then you can list their modules, whether statically or dynamically loaded.

Getting the List of Running Processes

When you need to enumerate Win32® running processes, the three methods shown in Figure 5 are available. I won't discuss TOOLHELP32 since MSDN® provides a lot of sample code that uses its functions. Performance counters provide much more information than the processes list. They're invaluable if you need to fetch information from remote computers. If you always wanted to get a process list from another machine, you have your solution!

The process status API (PSAPI) is a useful tool that's available in the Microsoft Platform SDK. The CProcessList class in my sample file, Process.cpp, wraps PSAPI to get the process list. Once Refresh has been called, a process description can be retrieved from a process ID and easily enumerated using GetFirst and GetNext (see Figure 6). Refresh is implemented using EnumProcesses (found in PSAPI), as shown in Figure 7.

If you are still supporting 16-bit code for Windows®, such as the applications listed under ntvdm.exe by the TaskManager, enumerating processes is more involved. The Knowledge Base articles referenced at the beginning of this article can help you, as well as two Under the Hood columns by Matt Pietrek in the August 1998 and September 1998 issues of MSJ.

Getting Information About a Process

Once you have a list of the running processes, you'll need to get as much information as possible from each of them, based on their IDs returned by EnumProcesses, to build a useful tool. Using a process handle obtained by calling OpenProcess with PROCESS_QUERY_INFORMATION | PROCESS_VM_READ as a parameter, the AttachProcess method (in the CProcessList class in Process.cpp) creates the description shown in Figure 8.

A few details about AttachProcess require explanation. First, to avoid being statically linked with OS-specific DLLs such as PSAPI or NTDLL, it is worth writing classes that wrap each function needed from these DLLs. (See Wrappers.h and Wrappers.cpp in the sample code for details.) For example, you only need to define a CPSAPIWrapper object and call its GetModuleFileNameEx method; you don't need to link with the PSAPI library. In addition, you should call its IsValid method to check that these DLLs are usable on the system you are running. This allows your code to run on any Windows platform without raising a system error such as undefined links. But check the Windows version or the result from IsValid before using a particular feature. (See DIISpyApp::InitInstance in DIISpy.cpp for an example.)

Notice that the GetModuleFileNameEx function from PSAPI returns strange file names such as "\\SystemRoot\System32\smss.exe" or "\\??\C:\WINNT\system32\winlogon.exe." The TranslateFilename function (in Helpers.cpp) gives you file names that are easier to read. More on this later.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

When it's time to find the main window of a process, EnumWindows executes the callback as a parameter for each top-level window. In the callback function, GetWindowThreadProcessId retrieves the ID of the process that creates the corresponding window; if this window is visible, I stop the enumeration (see the GetMainWindow implementation). Note that GetWindowText can be used to get the title of a window from a different process.

Conversely, to get the name of the file name corresponding to the process that created a particular window, you'll run into problems using GetWindowModuleFileName. This function always returns the path name of the current running process.

The procedure for getting the main window of a process (for which you can use the GetWindowThreadProcessId API) is described in Jeff Prosise's Wicked Code column in the August 1999 issue of MSJ. You know how to get the full path name from a process ID using PSAPI. Taking the full path name and using GetWindowThreadProcessID will get you the file name of a process that created a particular window.

In AttachProcess, the call to OpenProcess that is needed to get most of the process information may return with an access denied error. In that case, I used the method presented by Keith Brown in his August 1999 Security Briefs column in MSJ for getting a process handle with high-level rights. (See the GetProcessHandleWithEnoughRights function in Helpers.cpp in the code for implementation details.)

One example where this behavior occurs is when a process is run as a scheduled task under another user account. Even the Windows Task Manager can't terminate such a process; it merely displays the dialog box shown in Figure 9.

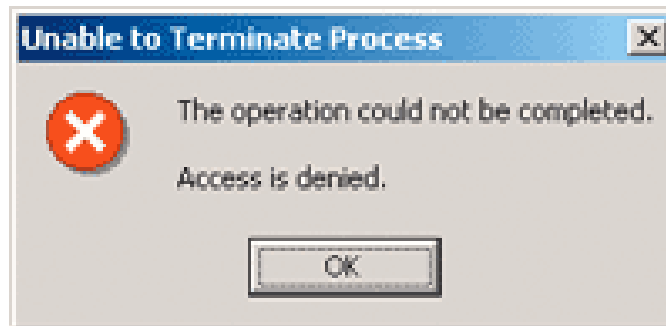


Figure 9 Can't Terminate Process

If you double-click on a process in my sample application ProcessSpy, it will be terminated. You should take a look at SlayProcess in the same project. This helper function uses GetProcessHandleWithEnoughRights to obtain a process handle, but with PROCESS_TERMINATE as an access right instead of PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, as is used by AttachProcess.

Finally, GetProcessOwner finds the user account (in \\Domain\User format) under which the process is running. It does this by using GetTokenInformation with TokenUser as a parameter and then LookupAccountSid to transform the returned user SID into a human-readable domain and user.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Sometimes, `OpenProcessToken` fails with an access denied error for some process such as `System`. Even `PULIST.EXE` from the Windows 2000 Resource Kit fails to display the owning user for the same processes. Only `ProcessExplorer` succeeds in finding the owner of such "secured" applications. You'll see later in this article how the Windows Terminal Services API can be used to get the owner of these processes under Windows XP.

Getting the Command Line for a Process

One method in the list in Figure 8, `GetCmdLine`, returns the command line of the process. Actually, it doesn't exactly return the command line, but rather the parameters received by the process at startup. Let's take an example. If you install `TweakUI` from the Microsoft® Power Toys (<http://www.microsoft.com/networkstation/downloads>), you get an additional `Command Prompt Here` entry in the context menu when you right-click on any folder in the Windows Explorer. And as if by magic, a command prompt appears with the folder as the current working directory.

But how do you know what parameter is used when `cmd.exe` is called? You could use `TLIST.EXE` from the Microsoft Debugging Tools (see <http://www.microsoft.com/ddk/debugging/>) to find out. This is a console-mode program that lists the running processes and information such as the command line, if called with a process ID as a parameter, as shown in Figure 10 in the case of a `C:` root.

The third line in Figure 10 shows the parameters `/k cd "C:\\"` used by the shell extension to call `cmd.exe`. If you specify the parameter `/k`, then `cmd.exe` carries out the specified command but it does not exit. This doesn't work if you are building your own tool because this invokes an application, whereas you need to call functions from an API.

The source code for the `TLIST` tool is available on the MSDN Web site at "TList: Task List Application Sample". Unfortunately, with that code you can only retrieve the ID, name, and main window for a process.

So you have three options for getting the command line of a process. The first is brutal: dig into `TLIST` at the assembly level. The result is the function `GetProcessCmdLine` in `Process.cpp`, which navigates inside the process address space to find its command line. A pointer to the command line (in Unicode) is stored in a memory block pointed to by a field of the `Process Environment Block (PEB)`, a kernel data structure that I'll discuss in my next article.

The second solution is to browse the Web and find someone who has already solved the problem! `GetCommandLine` tells you the command line, but only for the calling process. The best way around this is to execute this call within another process context using `CreateRemoteThread`, which Felix Kasza explains at <http://www.mvps.org/win32/processes/remthread.html>.

The last option is code reuse. Or, to be more accurate, output reuse. You could catch the output from `TLIST` so you only have to parse it in order to get the command line. I'll explain this method more fully in my next article.

Getting Process Information with WTS APIs

Windows XP has a new feature called `Fast User Switching`, which allows several users to be logged on at the same time on the same machine. Processes launched by one user can still run while another user is logged in. The magic behind this feature lies in the `Windows Terminal Services (WTS) APIs`. If you need a broad explanation of WTS, you should take a look at "Introducing the Terminal Services APIs for Windows NT Server and Windows 2000" in the October 1999 issue of `MSJ`.

Windows XP creates an instance of a `WTS session` for each logged-on user. A running process is

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

always associated with such a session. The Windows XP TaskManager allows you to list processes either for all sessions or for only your own session using the "Show processes for all users" checkbox (see Figure 11).

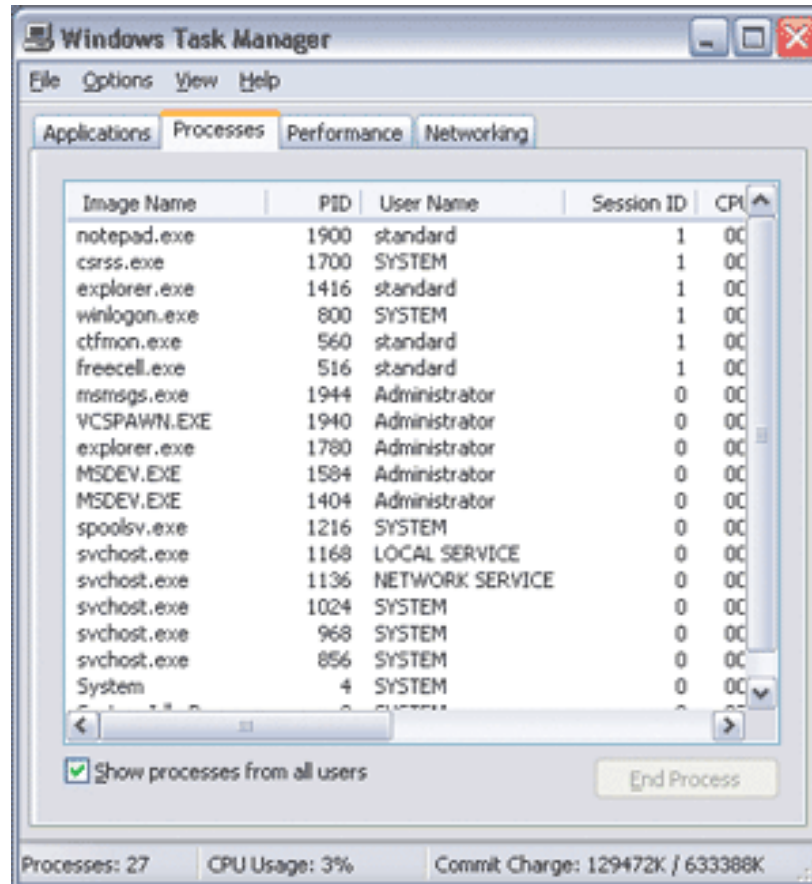


Figure 11 Listing Processes

If you need to learn the session ID under which a process is attached, call the `ProcessIdToSessionId` API exported by `kernel32.dll`. When given a process ID, it returns the corresponding session ID. It is interesting to note that this API is not exported by `wtsapi32.dll`, which implements all other Windows Terminal Services APIs, but by `kernel32.dll`. In fact, even when Windows Terminal Services are not enabled, the session ID is stored by Windows 2000 and Windows XP in the PEB.

Note that Windows NT® neither stores the session ID in its PEB nor exports `ProcessIdToSessionId` in `kernel32.dll`. When you call `ProcessIdToSessionId` on Windows 2000 without WTS enabled, you get 0 as session ID for all processes.

In addition to allowing you to list the open sessions, WTS has an API to enumerate running processes that is different from those in `PSAPI` and `TOOLHELP32`. I have written a class named `CWTSWrapper` to wrap the functions related to processes and sessions in WTS in order to avoid static linking with `wtsapi32.dll`. (See `\Common\wrappers.cpp` for implementation details.) Using `CWTSWrapper`, it is easy to build a console mode application such as `ProcessXP` (see Figure 12) that enumerates open sessions with the corresponding logged-on user and the running processes.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

As you can see in Figure 12, three sessions are open on the MACHINE computer. The first session has an ID of 0, is active (since this is the one where ProcessXP is running), and serves the logged-on user called Administrator. The second one has 1 as its ID, is disconnected, and serves the Standard user who has started Notepad and Freecell by hand. Finally, Player has opened the session 2 to launch WordPad, but it's now disconnected.

The source code that uses the CWTSWrapper class to generate this output can be found in the download for this article. Like the Registry, the WTS allows you to grab information from another machine. That's why WTS enumeration APIs take a server handle as the first parameter. WTS_CURRENT_SERVER_HANDLE is used for the current machine. The second parameter is reserved and should be set to 0. The third is the expected version and should be 1. The last two parameters contain the meaningful information on return. The last one is the count of sessions or processes, while the second to last points to an array of structures describing either a session or a process. Since the array has been allocated by WTS, you must remember to release it using WTSFreeMemory.

A session is described by a WTS_SESSION_INFO structure:

```
typedef struct _WTS_SESSION_INFO { DWORD SessionId; LPTSTR pWinStationName;
                                   WTS_CONNECTSTATE_CLASS State; }
WTS_SESSION_INFO, * PWTS_SESSION_INFO;
```

In addition to its SessionId, this structure provides the session name in the variable pWinStationName. The current session receives the name "console"; while the others have no name. The session state is WTSActive for the current session and WTSDisconnected for the others.

A process is described by a WTS_PROCESS_INFO:

```
typedef struct _WTS_PROCESS_INFO { DWORD SessionId; DWORD ProcessId; LPTSTR
                                   pProcessName; PSID pUserSid; }
WTS_PROCESS_INFO, * PWTS_PROCESS_INFO;
```

The SessionId is the same value as the one retrieved by ProcessIdToSessionId. The ProcessId contains the ID of the process. The pProcessName field is a pointer to the name of the process under a Name.EXE format.

The last pUserSid field points to the security identifier describing the user account under which the process is running. Using LookupAccountSid, you can get the name of the user from pUserSid. This information is already retrieved by the CProcess class in GetProcessOwner, but through the process token instead of through WTS. In some cases, it is not possible to get a process token, even though WTSEnumerateProcesses manages to provide it, which could be a reason to use this WTS API under Windows XP instead of PSAPI or TOOLHELP32.

Enumerating Loaded Modules

At any given time, it is possible to know the list of the DLLs loaded by a process using either PSAPI or TOOLHELP32. While researching this article, I found that the TOOLHELP32 implementation from an old Under The Hood column by Matt Pietrek, presented in Figure 13, misbehaves under Windows 2000 and Windows XP. Two reasons: one, the invalid usage of the if statement, and two, this code was published in MSJ in September 1998—long before Windows 2000.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

First, the following error handling code is not valid since `CreateToolhelp32Snapshot` does not return `NULL` when it fails:

```
if ( !hSnapshotModule ) continue;
```

Actually, in case of failure, `hSnapshotModule` receives `INVALID_HANDLE_VALUE` or `-1` as the value, and the `if` statement will not catch it. This is not a big deal. The interesting part is how the bug was found. When `ProcessSpy` was tested on Windows 2000, everything worked fine except that some processes were not returning an error even though the list was empty. Since the error handling code was wrong, the execution jumped to the loop and the `Module32First` call failed without any real error. If you use the `ModuleList` tool from this particular article on Windows 2000, you'll get incorrect results.

To understand what is happening in this code, a `GetLastErrorMessage` helper function is useful (see `Helpers.cpp`). It calls `GetLastError` and `FormatMessage` to get the corresponding failure reason in plain text. The reason will always be the same: `Access Denied`. With `PSAPI` functions, however, there is no access problem when getting the modules list of the same processes.

When the access problem does occur, it's due to a lack of privileges. The `SE_DEBUG_NAME` privilege needs to be enabled for the `TOOLHELP32` code to work perfectly. For more information on this problem, see `Win32 Q&A` in the March 1998 issue of `MSJ` and `Security Briefs` in the August 1999 issue.

Everything You Want to Know About a DLL

Both the `PSAPI` and `TOOLHELP32` solutions for getting the list of modules loaded in a process provide only the address where the DLL is mapped into the address space. The next step in this process is to get as complete a description as possible. My implementation does not offer a single `AttachModule` method as has been done in `CProcess`. Since some details can be really expensive to get, I chose to split the detail gathering into different functions. The cheapest are retrieved in the `CModule` constructor and the others are delayed (through a `Refresh` helper) until the corresponding accessor methods are called. Look in `Module.cpp` at the constructor of `CModule` and its `Refresh` method after the `Refresh/RefreshTOOLHELP32` methods of `CModuleList` for the implementation details. Figure 14 provides a list of these accessors and explains how each of them are implemented.

As I mentioned, there is a trick to getting the full path name of a module. For some reason, the path names returned by `GetModuleFilenameEx` or the `TOOLHELP32` module functions are very strange; they don't follow the Win32 standard. For example, `smss` is retrieved as `"\SystemRoot\System32\smss.exe"`; `"\SystemRoot"` must be replaced by the actual name of the Windows folder. For `winlogon`, you get `"\??\C:\WINNT\system32\winlogon.exe,"` which should be translated into `"C:\WINNT\system32\winlogon.exe."` The `\??\` prefix might be a leftover from the Windows NT namespace root, essential in kernel mode, even though it is rarely used at the Win32 programming level. I wrote a `TranslateFilename` helper function to convert these file names into more standard names. The implementation of `TranslateFilename` is in `Helpers.cpp` in the code download.

I collect the rest of the module description with my `Refresh` method, whose implementation is in `Module.cpp` and is summed up in Figure 15. Most of the description is extracted from the file itself, using the PE format knowledge provided by Matt Pietrek's various articles.

If you want more details from a PE file, you should take a closer look at Matt Pietrek's `PE_EXE` class and `PEDUMP` implementation found in the articles referenced in the download. This code is a goldmine of handy ideas!

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

An interesting use of `GetBaseAddress` is to compare its return value with the one from `GetModuleHandle`. The latter is the real address where the module is loaded in the process address space and the former is where the module expects to be loaded. This is perfect for finding loading conflicts and they make ideal targets for rebasing, as Figure 4 illustrates. In that figure you can see that WinRAR shell extensions need to be rebased from 0x400000.

When a process starts, the Windows loader automatically loads the static DLLs. These static links are quite easy to get programmatically using the PE format and the `MODULE_DEPENDENCY_LIST` class. There is no API that detects the difference between this kind of module and those dynamically loaded using `LoadLibrary` or `CoCreateInstance`. If a DLL is used by a process, but is not among its static links, it should be dynamically loaded.

In `ProcessSpy`, each module in the lower pane is prefixed by a small icon, a round D for dynamic and square S for Static. The color is also significant: red for a module with a base address that is different from its load address and blue otherwise (see Figure 4).

The rest of the module description is extracted from its resource version. Paul DiLascia provides `CModuleVersion` as a nice wrapper class to these resource version descriptions in his April 1998 MSJ C++ Q&A column. For each `VS_VERSION_INFO` detail, an accessor method has been added. It returns a `CString` reference, filled by a call to `CModuleVersion::GetFileVersion` with the corresponding string. `GetCompanyName` is a perfect example of such an accessor.

I would have to update Paul DiLascia's code for it to fit my needs since the `GetFileVersionInfo` method is supposed to receive a module name but not a true file name. To find the corresponding file name, `GetModuleHandle` is called. If it fails to find the module in the current process address space (this is rarely the case), it bails out. To solve this problem, when the given module name proves to be a real file name (using `GetFileAttributes`), it is used directly.

The Windows resource information usually provides more details than simply the name of the company. For example, it is easy to know if an application has been built in debug mode or if it is a private or a special build. You just have to take a look at the `dwFileFlags` field in the `VS_FIXEDFILEINFO` structure. As the MSDN Online documentation states, it contains a bitmask of the values listed in Figure 16. In the same version structure, the `dwFileType` field defines the file type (see Figure 17). These flags are used in `ProcessSpy` to update the version column with D for Debug and P for a patched version as Figure 4 shows for `dbgeng.dll` and `DBGHLP.dll`.

Looking Ahead

In an upcoming issue of MSDN Magazine, I'll present unusual ways to get additional sources of information. What other means can be used when there is no API to help you? I haven't yet mentioned a great source of information: the shell. Under a module hides a file, and no one knows more about a file than Windows Explorer, as shown in Figure 18.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

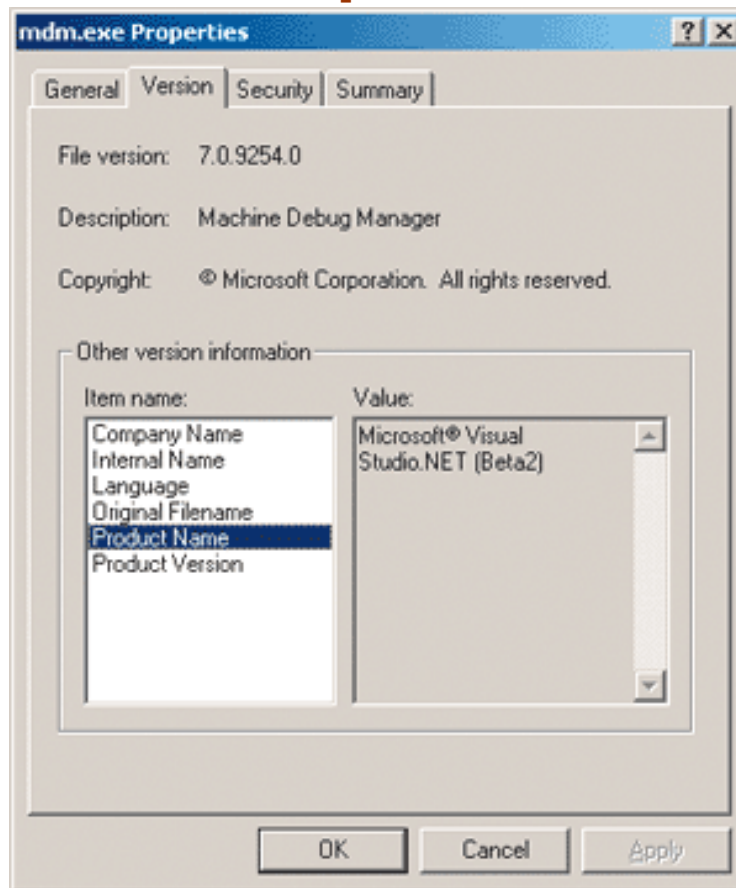


Figure 18 Using Explorer to Glean Info

It would be great to be able to show the Properties pages that Windows Explorer provides when you select Properties from a right mouse click. See the Knowledge Base article Q179377, or use ShellExecuteEx, as shown in this code:

```
SHELLEXECUTEINFO sei; ZeroMemory(&sei,sizeof(sei)); sei.cbSize = sizeof(sei);
                                                    sei.lpFile = szFilename; sei.lpVerb
= _T("properties"); sei.fMask = SEE_MASK_INVOKEIDLIST;
                                                    ShellExecuteEx(&sei);
```

This is exactly what you get when you double-click on any module in the ProcessSpy lower pane. Windows XP does not support more than one call to ShellExecuteEx. On the second execution, the thread freezes without any explanation.

As you can see, there are many ways to get information about loaded DLLs and active processes. The tools I've developed are a good place to start to learn about loaded DLLs and active processes, but you may need to develop custom tools for your own debugging scenarios. If so, you'll want to experiment with the various options I've described in this article and use the C++ classes I've provided. In my next article I will present new ways to get information from the system.

For more information, see the articles I mentioned. A list of their URLs is included with the code download at the link at the top of this article.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Figure 1 Dependency Walker

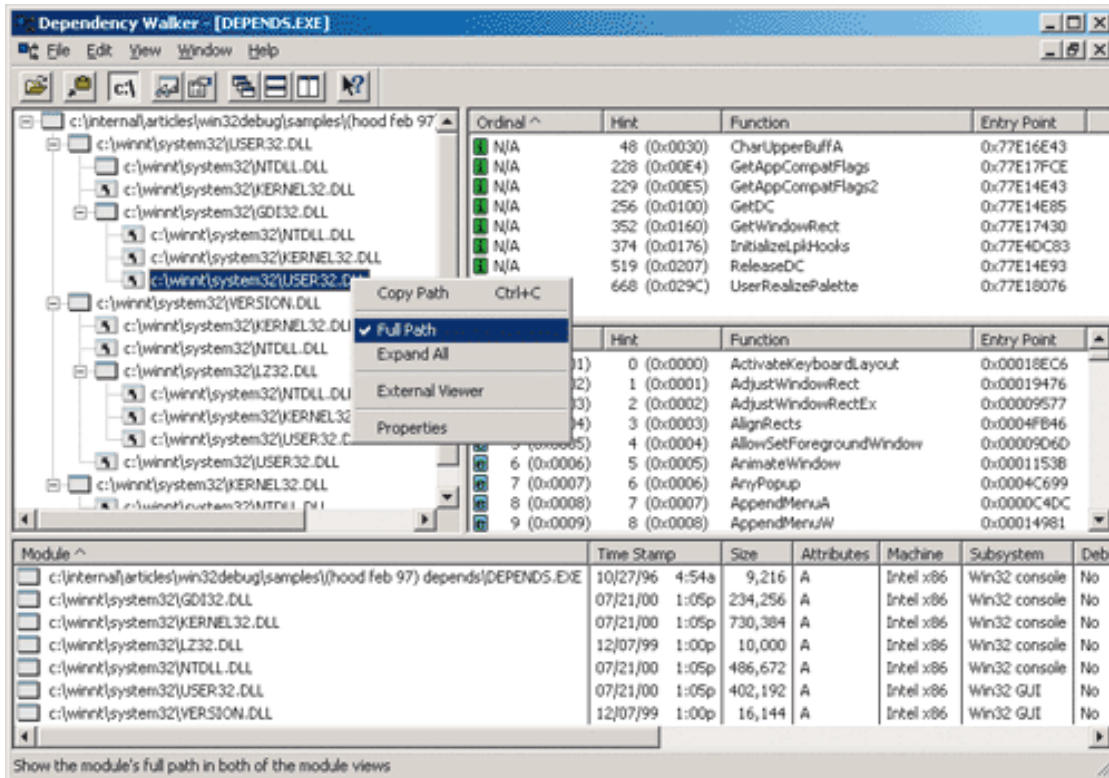
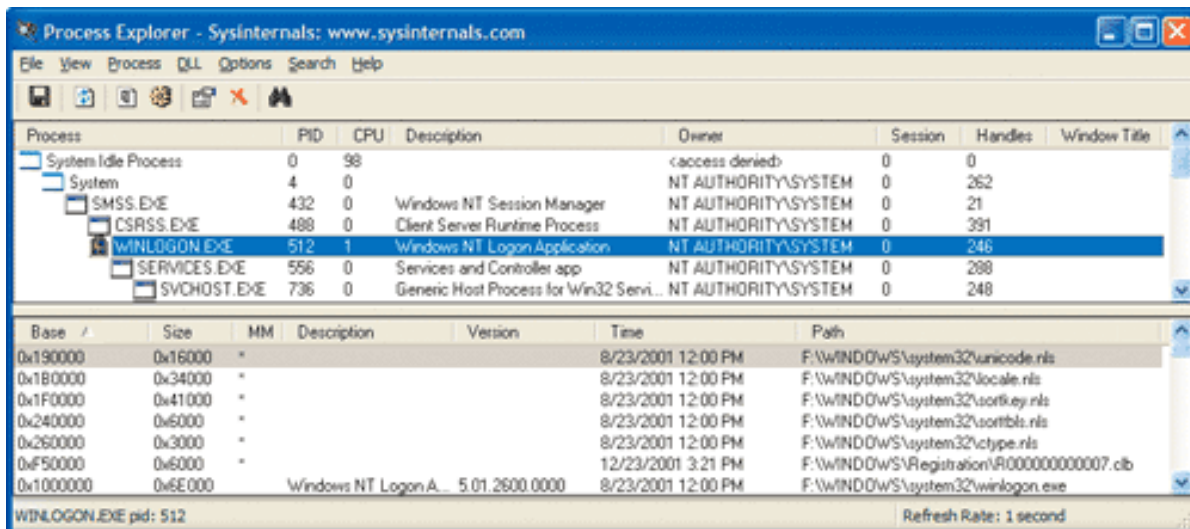


Figure 2 Process Explorer



Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Figure 3 DIISpy

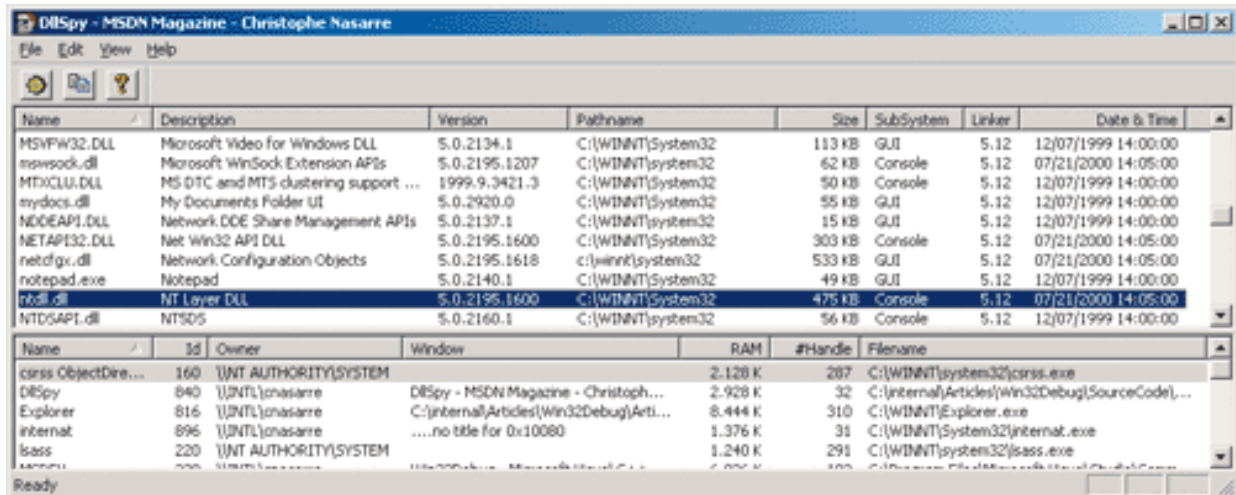


Figure 4 ProcessSpy

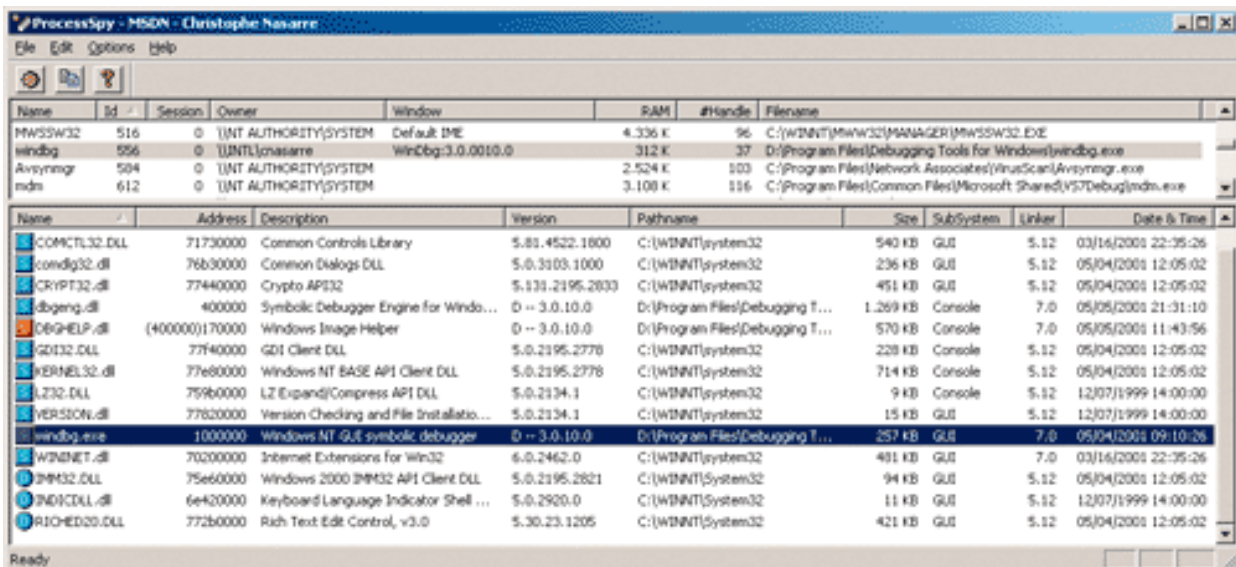


Figure 5 Methods for Enumerating Running Processes

Method	Platform	Note
PSAPI	Windows NT, Windows 2000, Windows XP	Gets information about process, driver, module, memory, and working set
Performance counters	Windows NT, Windows 2000, Windows XP	Provides more information than the process list and can be used from a remote computer
TOOLHELP32	Windows 9x, Windows 2000, Windows XP	Gets information about process, thread, module, and heap

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Figure 6 Using CProcessList to List Running Processes

```
// get one process after the other
CProcess* pProcess = NULL;
POSITION Pos = 0;
for (
    pProcess = ProcessList.GetFirst(Pos);
    (pProcess != NULL);
    pProcess = ProcessList.GetNext(Pos)
)
{
    if (pProcess != NULL)
    {
        // do what you want with the process information
    }
}
```

Figure 7 Refreshing the Processes List

```
void CProcessList::Refresh()
{
    // don't forget to reset and free the current list
    DefaultReset();

    // store the current process list
    DWORD aProcesses[MAX_PROCESS];
    DWORD cbNeeded = 0;

    // get a snapshot of the processes
    if (!g_PSAPI.EnumProcesses(aProcesses, sizeof(aProcesses), &cbNeeded))
        return;

    // Calculate how many process IDs were returned
    DWORD cProcesses = cbNeeded / sizeof(DWORD);

    // attach a CProcess object to each process ID
    DWORD dwProcessID;
    CProcess* pProcess;
    for (
        DWORD dwCurrentProcess = 0;
        dwCurrentProcess < cProcesses;
        dwCurrentProcess++
    )
    {
        dwProcessID = aProcesses[dwCurrentProcess];

        // add the process definition into the map
        pProcess = new CProcess(TRUE);
        if (pProcess != NULL)
        {
            // fill in the process information for the current process ID
            if (!pProcess->AttachProcess(aProcesses[dwCurrentProcess]))
                delete pProcess;
            else
                // store into the map
                m_ProcessMap[(LPVOID)dwProcessID] = pProcess;
        }
    }

    // a second iteration is needed to know the process children
    SetChildrenList();
}
```

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Figure 8 Process Details

Method	Description
GetName	Uses GetModuleBaseName with NULL as parameter but without the ".EXE"
GetFileName	Uses GetModuleFileNameEx with NULL as parameter
GetMainWindowHandle GetMainWindowTitle	See the GetMainWindowHandle discussion
GetParentProcessID	Uses NtQueryInformationProcess with ProcessBasicInformation
GetKERNELHandleCount	Uses NtQueryInformationProcess with ProcessHandleCount
GetUSERHandleCount	Uses GetGuiResources with GR_USEROBJECTS
GetGDIHandleCount	Uses GetGuiResources with GR_GDIOBJECTS
GetWorkingSet	Uses GetProcessMemoryInfo
GetCmdLine	See GetProcessCmdLine discussion
GetOwner	See GetProcessOwner details
GetSessionID	ProcessIdToSessionId (see Fast User Switching discussion)
GetModuleList	CModuleList is a wrapper class around EnumProcessModules and GetModuleFileNameEx
GetChildrenCount and children list	There is no API (even undocumented) to get the list of processes spawned by a process. But since the parent of each process is known, it is easy to add a process to the child list of its parent. (See the implementation of SetChildrenList)

Figure 10 TLIST Detailed Output for a Running Process

```
C:\>tlist 632
632 CMD.EXE           C:\WINNT\System32\cmd.exe - tlist 632
  CWD:                C:\
  CmdLine:            C:\WINNT\System32\cmd.exe /k cd "C:\"
  VirtualSize:       13408 KB   PeakVirtualSize:    13412 KB
  WorkingSetSize:    948 KB    PeakWorkingSetSize: 952 KB
  NumberOfThreads:  1
    968 Win32StartAddr:0x4ad1a420 LastErr:0x000000cb State:Waiting
  5.0.2195.1600 shp  0x4ad00000  cmd.exe
  5.0.2195.1600 shp  0x77f80000  ntdll.dll
  5.0.2195.1600 shp  0x77e80000  KERNEL32.dll
  5.0.2195.1600 shp  0x77e10000  USER32.dll
  5.0.2195.1340 shp  0x77f40000  GDI32.DLL
  5.0.2195.1600 shp  0x77db0000  ADVAPI32.dll
  5.0.2195.1615 shp  0x77d40000  RPCRT4.DLL
    6.1.8637.0 shp  0x78000000  MSVCRT.dll
```

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Figure 12 ProcessXP Output

3 open sessions

ID	State	Window	Station
0	(WTSActive)	Console	[Administrator]
1	(WTSDisconnected)		[standard]
2	(WTSDisconnected)		[Player]

30 running processes

0	0		?
0	4	System	\\NT AUTHORITY\SYSTEM
0	388	smss.exe	\\NT AUTHORITY\SYSTEM
0	600	csrss.exe	\\NT AUTHORITY\SYSTEM
0	632	winlogon.exe	\\NT AUTHORITY\SYSTEM
0	676	services.exe	\\NT AUTHORITY\SYSTEM
0	688	lsass.exe	\\NT AUTHORITY\SYSTEM
0	856	svchost.exe	\\NT AUTHORITY\SYSTEM
0	968	svchost.exe	\\NT AUTHORITY\SYSTEM
0	1160	svchost.exe	\\NT AUTHORITY\NETWORK SERVICE
0	1192	svchost.exe	\\NT AUTHORITY\LOCAL SERVICE
0	1252	spoolsv.exe	\\NT AUTHORITY\SYSTEM
0	1888	explorer.exe	\\MACHINE\Administrator
0	2004	messaging.exe	\\MACHINE\Administrator
0	104	svchost.exe	\\NT AUTHORITY\SYSTEM
1	1496	csrss.exe	\\NT AUTHORITY\SYSTEM
1	1172	winlogon.exe	\\NT AUTHORITY\SYSTEM
1	1640	explorer.exe	\\MACHINE\standard
1	1900	ctfmon.exe	\\MACHINE\standard
1	352	notepad.exe	\\MACHINE\standard
1	1896	freecell.exe	\\MACHINE\standard
2	416	csrss.exe	\\NT AUTHORITY\SYSTEM
2	268	winlogon.exe	\\NT AUTHORITY\SYSTEM
2	1784	explorer.exe	\\MACHINE\Player
0	1820	msiexec.exe	\\NT AUTHORITY\SYSTEM
2	1544	ctfmon.exe	\\MACHINE\Player
2	1632	messaging.exe	\\MACHINE\Player
2	1268	wordpad.exe	\\MACHINE\Player
0	1696	wuauclt.exe	\\MACHINE\Administrator
0	1996	ProcessXP.exe	\\MACHINE\Administrator

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Figure 13 Module Walking Using TOOLHELP32

```
//
// Enumerate the module list for this process. Start by taking
// another ToolHelp32 snapshot, this time of the process's module list
//
HANDLE hSnapshotModule;
hSnapshotModule = pfnCreateToolhelp32Snapshot( TH32CS_SNAPMODULE,
                                              procEntry.th32ProcessID );

if ( !hSnapshotModule )
    continue;

// Iterate through each module in the snapshot
MODULEENTRY32 modEntry = { sizeof(MODULEENTRY32) };
BOOL fModWalkContinue;

for ( fModWalkContinue = pfnModule32First(hSnapshotModule,&modEntry);
      fModWalkContinue;
      fModWalkContinue = pfnModule32Next(hSnapshotModule,&modEntry) )
{
    // Hack! Cheezy way to figure out if this is EXE module itself
    // If so, we don't want to add it to the module list
    if ( 0 == strcmp( modEntry.szExePath, procEntry.szExeFile ) )
        continue;

    // Determine if this is a DLL we've already seen
    PModuleInstance pModInst = modList.Lookup(modEntry.hModule,
                                              modEntry.szExePath );

    // If we haven't see it, add it to the list
    if ( !pModInst )
        pModInst = modList.Add( modEntry.hModule, modEntry.szExePath );

    // Add this process to the list of processes using the DLL
    pModInst->AddProcessReference( procEntry.th32ProcessID );
}

CloseHandle( hSnapshotModule ); // Done with module list snapshot
```

Figure 14 Accessor Methods

Accessor	Notes
HMODULE GetModuleHandle	Address where the DLL is mapped
CString& GetFullPathName	From either TOOLHELP32::Module32xxx Or PSAPI::GetModuleFilenameEx
CString& GetModuleName	Same as GetFullPathName
CString& GetPathName	Same as GetFullPathName

Figure 15 Accessor Details

Accessor	Notes
DWORD GetBaseAddress	Uses PE_EXE::GetImageBase to get the preferred loading address
void GetFileTime(FILETIME& ft)	Uses GetFileTime API from KERNEL32.DLL to know when it has been created, modified, and last accessed
CString& GetFileTime	Get the same info as the previous one but in text format using GetFileDateAsString/GetFileTimeAsString helper functions

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities

Christophe Nasarre

Accessor	Notes
DWORD GetFileSize	Uses PE_EXE::GetFileSize to get the size of the file in bytes
CString& GetSubSystem	Uses PE_EXE::GetSubSystem to know the module subsystem among the IMAGE_SUBSYSTEM_xxx values from winnt.h; in the last version of this file, IMAGE_SUBSYSTEM_XBOX could be found
void GetLinkTime(FILETIME& ft)	Uses PE_EXE::GetTimeStamp to discover when the module has been linked
CString& GetLinkTime	Gets the same info as the previous one but in text format using GetFileDateAsString/GetFileTimeAsString helper functions
WORD GetLinkVersion	Uses PE_EXE::GetLinkerVersion to get the version of linker used to build the module

Figure 16 File Types According to Version Information

Flag	Description
VS_FF_DEBUG	Contains debugging information or is compiled with debugging features enabled
VS_FF_INFOINFERRED	Version structure was created dynamically; therefore, some of the members in this structure may be empty or incorrect. This flag should never be set in a file's VS_VERSIONINFO data
VS_FF_PATCHED	Has been modified and is not identical to the original shipping file of the same version number
VS_FF_PRERELEASE	A development version, not a commercially released product
VS_FF_PRIVATEBUILD	Not built using standard release procedures. If this flag is set, the StringFileInfo structure should contain a PrivateBuild entry
VS_FF_SPECIALBUILD	Built by the original company using standard release procedures but is a variation of the normal file of the same version number. If this flag is set, the StringFileInfo structure should contain a SpecialBuild entry

Figure 17 Flags in the dwFileType Field

Flag	Description
VFT_UNKNOWN	Unknown to the system
VFT_APP	Contains an application
VFT_DLL	Contains a DLL
VFT_DRV	Contains a device driver. If dwFileType is VFT_DRV, dwFileSubtype contains a more specific description of the driver
VFT_FONT	Contains a font. If dwFileType is VFT_FONT, dwFileSubtype contains a more specific description of the font file
VFT_VXD	Contains a virtual device
VFT_STATIC_LIB	Contains a static-link library