

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

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

SUMMARY

Building on his article published in the June issue, which demonstrated several ways to get process and DLL-related information from APIs such as PSAPI, NTDLL, and TOOLHELP32, the author presents some unusual ways to get system-oriented info that you can easily integrate in your own toolkit.

There are three tools included as samples: LoadLibrarySpy, which monitors an application and detects which DLLs are really loaded; WindowDump, which retrieves the content and a detailed description of any window; and FileUsage, which redirects console-mode applications to tell you which process is using any opened file.

You saw how to get the list of the running processes and the DLLs they have loaded using well-documented API functions in Part 1 of this article in the June 2002 issue. Now I'll present different ways, often undocumented, to get system-related information. First I'll dig into the Win32® debugging API and the traces provided by the Windows® Loader, to find out exactly how a DLL is loaded by a given process. In addition, I'll present several ways to detect the cause of a DLL's relocation with my reusable class CApplicationDebugger.

Next, I'll build two tools. LoadLibrarySpy detects DLL relocations. WindowDump steals the content and the detailed description of any window. Finally, before navigating into the Process Environment Block (PEB) internal structure, I'll show you how to control the output that is generated by console mode applications in order to fish for undocumented information.

Back into DLL Hell

As you saw in Part 1 of this article, it is easy enough to get the list of all statically or dynamically loaded DLLs. But for a dynamically loaded DLL, the picture is more complicated than it may appear. For example, My DIISpy and ProcessSpy tools rely on a snapshot taken at a certain point in time. Therefore, a new DLL could be quickly loaded and unloaded without being detected. The Win32 debugging API provides a solution to this problem: when you debug an application, you are notified whenever a DLL gets loaded and unloaded by the debuggee.

For my purposes, a full-featured real-world debugger is not required, but I do need to detect when a new DLL is loaded into a process address space. Therefore, I'll discuss the basis of the Win32 debugging API in addition to useful extensions you can get under Windows NT®, Windows 2000, and Windows XP.

To debug an application, you first need to launch it using CreateProcess with one of these special flags. DEBUG_PROCESS asks for events from the debuggee and every process the debuggee starts. DEBUG_ONLY_THIS_PROCESS asks for events from the debuggee only (not from its child processes).

With DEBUG_ONLY_THIS_PROCESS, no event from a process launched by the debuggee will be received by the debugger. A good example of an application for which this flag is useless is the Performance Monitor (perfmon.exe). This is a simple wrapper program whose role is to start another application, Microsoft® Management Console (MMC), and pass whatever parameter is needed to make it display the performance counters.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

During the life of the debuggee, Windows notifies the debugger of the events listed in Figure 1. These are described by a `DEBUG_EVENT` structure, as shown in Figure 2.

In order to receive these events, the debugger has to call `WaitForDebugEvent`. This function blocks until either one of the events listed in Figure 1 occurs for the debuggee, or the timeout given as second parameter elapses. When the debugger has handled an event, it calls `ContinueDebugEvent` to let the debuggee continue along its life. Note that all the debuggee threads are frozen when `WaitForDebugEvent` unblocks in the debugger and they are released during the `ContinueDebugEvent` call (see Figure 3).

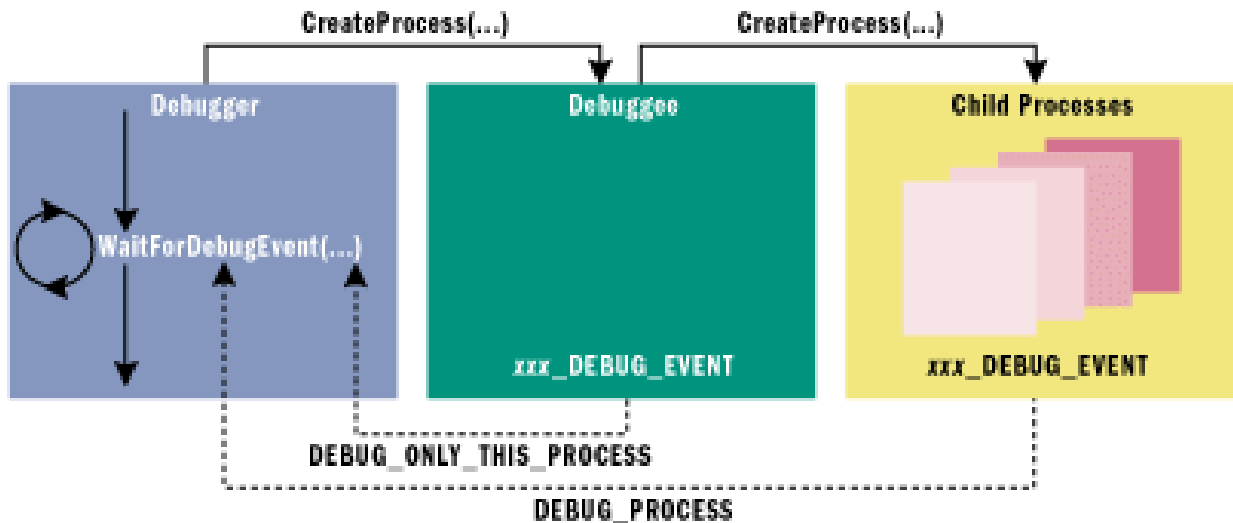


Figure 3 DebugEvent Flow

CApplicationDebugger

The thread that calls `CreateProcess` must be the thread that enters the debugging loop. Since the debugger blocks on `WaitForDebugEvent`, it's good to make all this code run in a dedicated thread separate from the main UI thread. This behavior is wrapped inside the `CApplicationDebugger` class whose declaration is listed in `ApplicationDebugger.h` (see the source at the link at the top of this article) and is inspired by Matt Pietrek's `LoadProf32` (found at `MSJJul95.exe`).

`CApplicationDebugger` is a virtual class because you are supposed to derive from it and implement your own version of the overridables that are called when a particular debug event occurs. This class is used to build `LoadLibrarySpy` (see Figure 4), a tool that debugs an application and monitors which DLL is loaded and unloaded, whether this happens statically or not, and whether a loading address conflict occurs.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

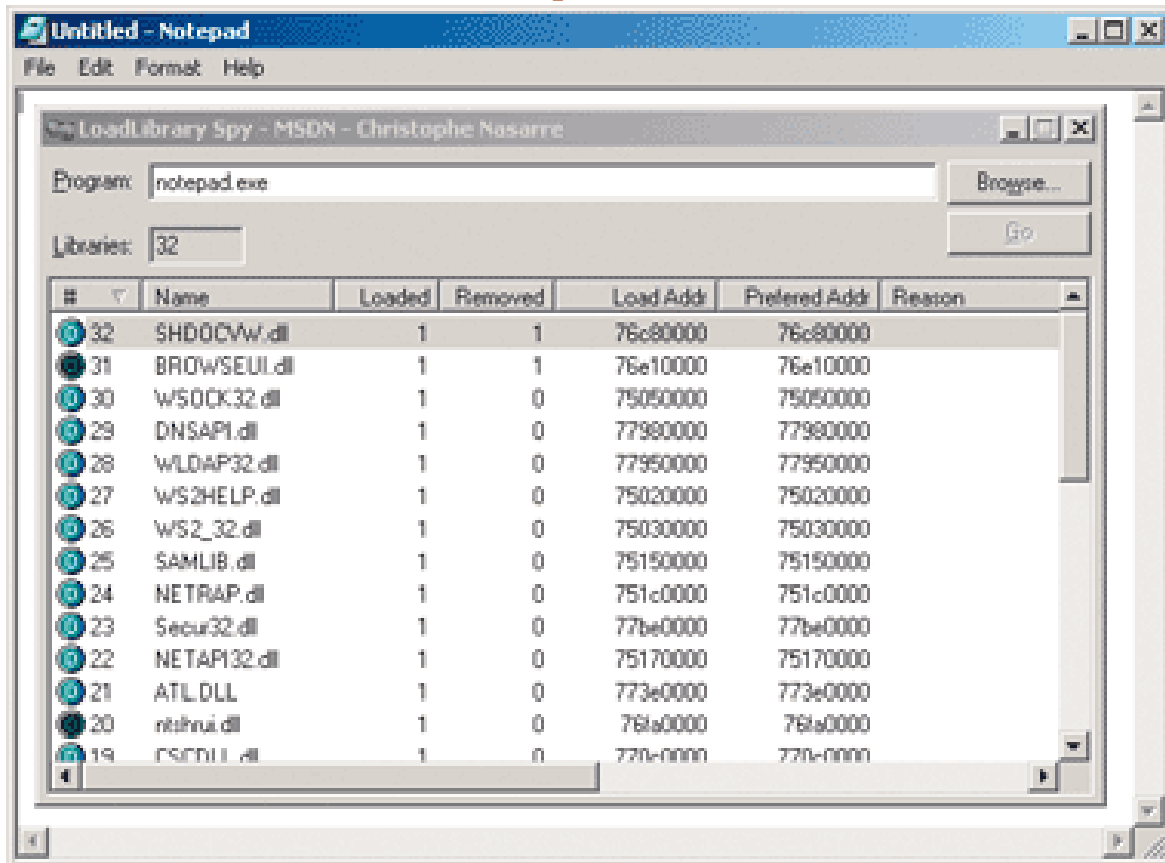


Figure 4 LoadLibrarySpy

The call to `CreateProcess` is done in `CApplicationDebugger::LoadTheProcess`, with `DEBUG_ONLY_THIS_PROCESS` as a parameter for simplicity. You can extend `CApplicationDebugger` to take into account the events coming from multiple debuggee processes if you really need to, which is useful for MMC snap-ins.

The `CLoadLibrarySpyDlg` class is responsible for the dialog itself, and also for the thread hosting the `CApplicationDebugger`-derived class that spies on the debuggee. The class `CModuleListCtrl` is responsible for displaying the `CModuleInfo*` attached behind each DLL; that is, behind each line. On a per-DLL basis, this class stores the details listed in Figure 5.

When a DLL is loaded, the `AddModule` method is then called by the dialog; when one is removed, the `RemoveModule` method is executed. Both will end up in the `UpdateModule` method, which updates either `m_nLoaded` or `m_nRemoved` of the `CModuleObject` corresponding to the DLL. If there is no such object, a new one is created for the DLL and then it is added to the list.

Don't get confused by `m_nLoaded` or `m_nRemoved`. If you call `LoadLibrary` on the same DLL four times in a row, the debugger will only receive `LOAD_DLL_DEBUG_EVENT` once, and `m_nLoaded` will be set to 1. If the debugger receives an `UNLOAD_DLL_DEBUG_EVENT` for a DLL, you can be sure that the DLL is no longer used by the process. Therefore, you will never receive this event for static DLLs, even though they might be dynamically loaded and unloaded using `LoadLibrary/FreeLibrary` after the process is started.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

Handling Debuggee Events

Once the debuggee process is started, the debugger thread has to wait for some events to happen. This is why it should be in a separate thread from the main UI, especially if the main window is a modal dialog!

To use CApplicationDebugger in CLoadLibrarySpyDlg efficiently, the GoThreadProc thread procedure first declares a CApplicationDebugger object, specifying the command line to be executed and whether the OutputDebugString or TRACE from the debuggee should be intercepted. Next, DebugProcess blocks, until either the debuggee ends (upon receiving an EXIT_PROCESS_DEBUG_EVENT or an unhandled second chance exception), or one of the overridables does not return DBG_CONTINUE.

The communication mechanism between the thread and the dialog is simple: the messages listed in Figure 6 are posted to the dialog by the debugger thread when a debuggee event occurs.

The first message is posted to the dialog when all statically linked DLLs are loaded; that is, from the OnProcessRunning overridable that is called when the first (fake) breakpoint is triggered by Windows to signal it to the debugger. The second message is posted by the OnUnloadDLLDebugEvent overridable event that is called when a DLL is unloaded in the debuggee.

The third message requires a longer explanation because, in order to create a CModuleInfo, the full path name of the DLL is required. None of the methods presented in the June 2002 installment succeeded in retrieving the file name of the DLL from its hModule or loading address. Even though the DLL is already mapped into the debuggee address space when the event is received by the debugger (since it is possible to browse into its PE header), Windows has not initialized the data structures needed by PSAPI at this point.

In fact, the LoadDll.lplmageName field of the corresponding LOAD_DLL_DEBUG_INFO union u from DEBUG_EVENT (see Figure 2) always points to a strange memory block with read/write/execute rights in the debuggee address space, as shown here:

```
typedef struct _LOAD_DLL_DEBUG_INFO
{
    HANDLE hFile;
    LPVOID lpBaseOfDll;
    DWORD dwDebugInfoFileOffset;
    DWORD nDebugInfoSize;
    LPVOID lpImageName;
    WORD fUnicode;
} LOAD_DLL_DEBUG_INFO, *LPLOAD_DLL_DEBUG_INFO;
```

This block contains the path name of the DLL that is just about to be loaded. Here is what MSDN® Online Help has to say about lplmageName:

Pointer to the filename associated with hFile. This member may be NULL, or it may contain the address of a string pointer in the address space of the process being debugged. That address may, in turn, either be NULL or point to the actual filename.

If fUnicode is a nonzero value, the name string is Unicode; otherwise, it is ANSI. This member is strictly optional. Debuggers must be prepared to handle the case where lplmageName is NULL or

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

*lpImageName (in the address space of the process being debugged) is NULL. Specifically, the system will never provide an image name for a create process event, and it will not likely pass an image name for the first DLL event.

The system will also never provide this information in the case of debugging events that originate from a call to the DebugActiveProcess function.

The OnLoadDLLDebugEvent override method translates this explanation into plain C++ code that works 99 percent of the time. The rest of the time corresponds to the loading of ntdll.dll: this would appear to be the first DLL event described by the documentation. Even though the path name retrieval is postponed when the next debuggee event occurs (see OnDebugEvent for CLoadLibraryDebugger), it fails. In desperation, SearchPath is called to get the full path name from the simple module name, "system32," not a surprise for ntdll.dll. This API function uses the same algorithm as LoadLibrary to find a DLL in the file system. In theory, since it is called by the debugger, it is possible that the returned file might not be the one loaded by the debuggee—if a version of ntdll.dll exists in the debugger folder, for example. In practice, there is no chance that ntdll.dll will be patched and copied in a folder other than system32.

Preventing Leaks

Another under-documented side of the Win32 debugging API is the need to free the handles returned in the different XXX_DEBUG_EVENT structures. Matt Pietrek, in his November 1995 MSJ Under the Hood column, pointed out that handles returned in the XXX_DEBUG_EVENT structure to the debugger should be closed. In fact, almost every handle must be closed using CloseHandle. The only exception is the thread handle stored in CREATE_THREAD_DEBUG_EVENT, and that is closed by the system when the process ends. If you don't close the others, you create a system resources leak that could grow very fast, as shown in Figure 7. This type of garbage collection is automatically done for you by CApplicationDebugger::HandleDebugEvent.

Whatever cleanup method you use, the system inevitably leaks two handles every time you debug a process: a semaphore and a port, both unnamed. To convince you that CApplicationDebugger is not responsible for this leakage, let me point out that the same behavior can be observed for Visual Studio® 6.0 and Visual Studio .NET using either ProcessExplorer from <http://www.sysinternals.com> or DH.EXE from the Windows Resource Kit.

Now you have seen how to use the Win32 debugging API to get the exact list of the DLLs that have been loaded and removed from a process address space during its execution. Windows itself provides another way to get additional details about these DLLs.

The Windows Loader Knows Everything

In addition to the Win32 debugging API, Windows provides another great source of information about DLL loading address conflicts. Some global flags (or GFlags) set in the Registry under

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options

change the way that applications are handled by Windows. GFlags.exe (see Figure 8), an application that comes with the Microsoft Debugging Tools (see <http://www.microsoft.com/ddk/debugging/>), allows you to easily update the value of this Registry entry.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

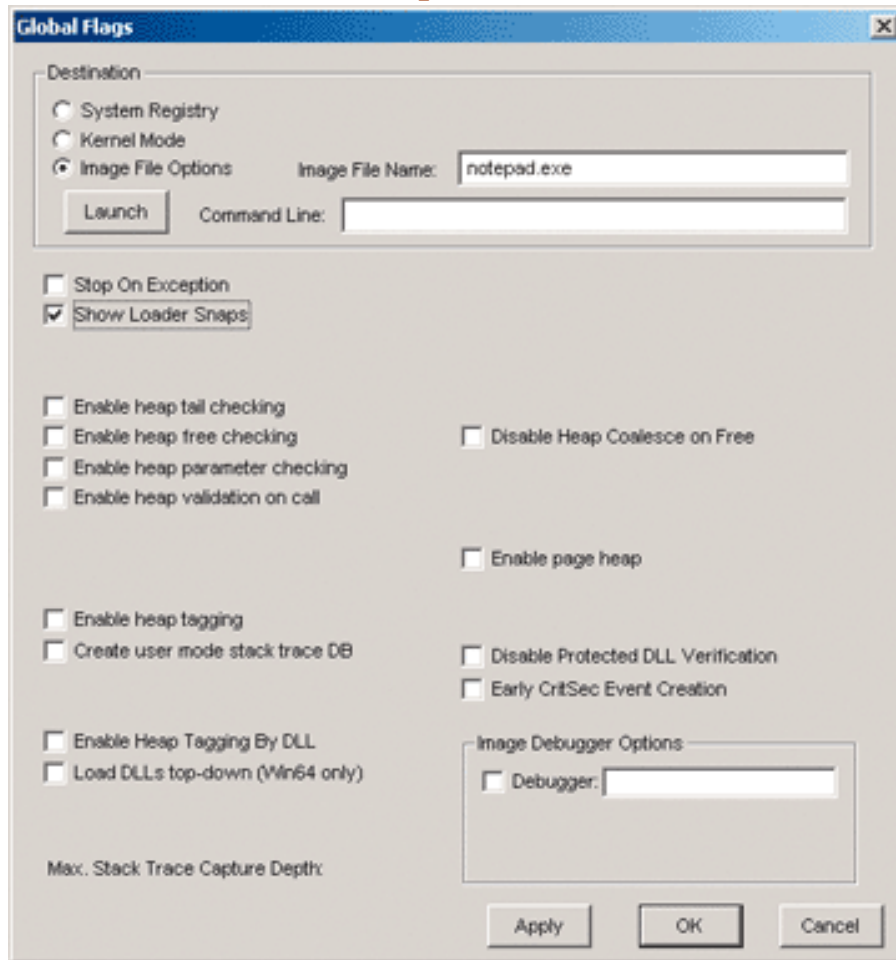


Figure 8 Global Flags

In the September 1999 Under the Hood column, Matt Pietrek explained how the FLG_SHOW_LDR_SNAPS value can be combined with these GFlags to make the Windows Loader generate some meaningful traces. If you want to catch these traces, you have two options. The first is to debug the application and then translate the OUTPUT_DEBUG_STRING_EVENT as CApplicationDebugger does. The other solution is easier: use a tool that catches them globally. If you want to know how to build your own, get DbgView from <http://www.sysinternals.com> or the "Inside Windows 2000, Third Edition" CD (Microsoft Press, 2000), which also displays kernel traces.

In the LoadLibrarySpy tool, before starting the debuggee application, the GFlags value corresponding to the debuggee will be updated in PreLoadingProcess by CApplicationDebugger, and its former value will be restored in PostMortem. This means that, from the Windows Loader, the debugger gets a dedicated output that is easy to filter in OnOutputDebugStringDebugEvent because "LDR:" is used as a prefix.

The main benefit of such a Loader log is the information that's output just after the LDR: Automatic DLL Relocation message. It explains which DLL has an address conflict with another DLL. This is how the data for the Reason column of CModuleListCtrl is obtained. Unfortunately, the Windows 2000 Loader seems to suppress this specific output message. If you are used to loading a process to get

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

access to its resources, such as explorer.exe for system animations or icons, the 0x400000 loading address is usually already used by your application and the Loader will automatically carry out relocations. In this particular case, even in Windows NT 4.0, it does not emit a LDR: Automatic DLL Relocation for a dynamically loaded process.

Another solution is to enumerate every DLL loaded and compare each dedicated address space area (starting at hModule) to find ones which conflict (see CLoadLibraryDebugger::OnLoadDLLDebugEvent for implementation details). The loader provides another interesting notification prefixed by "LDR: Loading (DYNAMIC)" and followed by the full path name of the module. This seems to happen when a DLL is explicitly loaded using LoadLibrary.

Using these hints from the Windows Loader, LoadLibrarySpy provides a specific icon for each DLL, depending on its loading status, as shown in Figure 9.

The DLLs with square icons are loaded during the process initialization and are called statically loaded. The round ones are loaded afterward and are therefore dynamically loaded. The color of the icon tells you that there is either a loading address conflict (red) or none (blue).

The difference between those with a black background and the other dynamic DLLs is tricky: a DLL with a black icon has been loaded, either explicitly using LoadLibrary or implicitly using other API functions such as CoCreateInstance. A DLL with a non-black background has been loaded because it is needed by another DLL. For example, in Figure 4, BROWSEUI.dll has a black icon because it has been dynamically loaded. SHDOCVW.dll has no black background because it has been automatically loaded by Windows. The reason is simple: BROWSEUI.dll is statically linked to SHDOCVW.dll. So in order to load BROWSEUI, Windows has to load SHDOCVW too.

Another Way to "Steal" Information

Before leaving the Win32 debugging API, I want to take a shortcut through the exception-handling mechanism. When an exception occurs in the debuggee, the debugger is notified via EXCEPTION_DEBUG_EVENT, and the u.Exception.ExceptionRecord.ExceptionCode field contains the exception code. The exact list of possible exceptions is not available in one easy-to-read roadmap; it's scattered within WINNT.H and WINBASE.H. The GetExceptionDescription method of CApplicationDebugger transforms the exception code into a human-readable string.

There is another source that lists possible exceptions: Microsoft Visual C++® itself. When you debug an application, the Debug menu leads to an Exception dialog that allows you to select how you want the exceptions to be handled by the debugger, as shown in Figure 10.

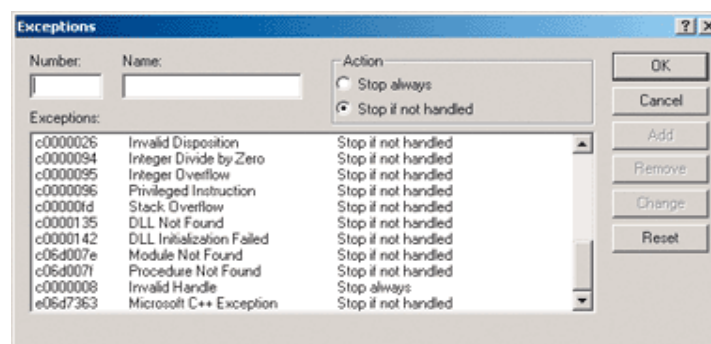


Figure 10 Exceptions Dialog

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

You may be surprised to find exception codes here that are not defined anywhere else. Instead of copying them by hand, it would be nice to "steal" the contents of the listbox. This is exactly the goal of WindowDump. It allows you to pick a window with the mouse (or by its handle value) and dump its content into an edit box. In addition to the content, it also gathers class and style descriptions, as shown in Figure 11.

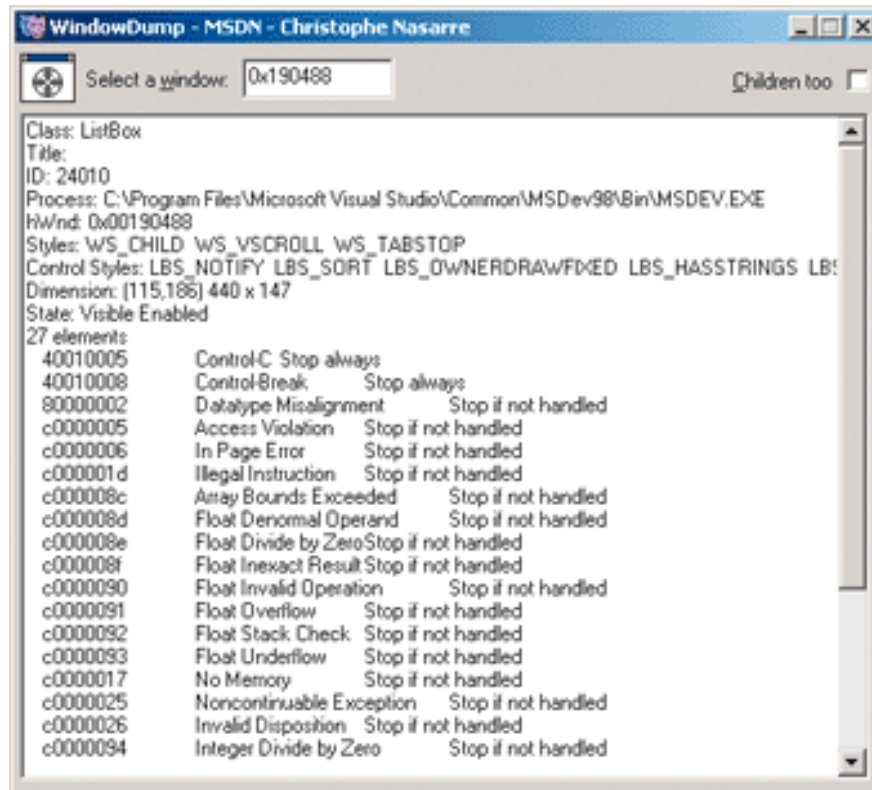


Figure 11 Exception Code in WindowDump

There is no magic behind WindowDump. The only interesting point is the fact that Windows usually allows GetWindowText and WM_GETTEXT to operate between different processes. This is not the case for common controls such as listviews or treeviews. Jeffrey Richter explained how to dump the content of a listview from another process in his September 1997 Win32 MSJ column, with the LV2Clip sample. Here are the window classes whose content is stolen by WindowDump : Edit, ScrollBar, ListBox, ComboBox, ListView, and TreeView. Along with the window content, you also get the description listed in Figure 12.

The last important point to make about the WindowsDump implementation involves the process ID. Starting from a window handle, it is not difficult to identify the thread and the process responsible for its creation using GetWindowThreadProcessId. If you want to know the module name afterwards, you could get tripped up by GetWindowModuleFileName. Contrary to the information given in the documentation, this API function fails under Windows NT, Windows 2000, or Windows XP. You have to dig into Knowledge Base article Q228469 to find that out.

In this case, you should use PSAPI and its GetModuleFileNameEx function. It takes a process and an hModule to return the corresponding path name. To find a process executable path name, 0 should

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

be given as hModule. Don't use 0x400000: some processes are loaded at different addresses such as winlogon and Task Manager at 0x1000000, ntvdm at f000000, and Microsoft Word 2000 at 0x30000000.

CreateRemoteThread, Command Line, and More

You have seen how the Win32 debugging API can be used to discover how and where DLLs are loaded by a process. Now let's dust off CreateRemoteThread, a function that allows you to make another process start one of your functions as a running thread in its context, as shown here:

```
HANDLE CreateRemoteThread(  
    HANDLE                hProcess  
    LPSECURITY_ATTRIBUTES lpThreadAttributes  
    DWORD                dwStackSize  
    LPTHREAD_START_ROUTINE lpStartAddress  
    LPVOID              lpParameter  
    DWORD                dwCreationFlags  
    LPDWORD             lpThreadId  
);
```

The lpStartAddress parameter is supposed to be the address of the thread procedure to be executed in another process context. The trick is that lpStartAddress must be an address in the other process address space, and that's why this function is so hard to use. If you don't want to recreate the assembly for your code and then copy it into the other process address space, there is an easier solution available to you.

If you compare a thread function and the LoadLibrary exported by kernel32.dll, you'll conclude that they share the same signature. Both take a 32-bit value as the parameter and return a 32-bit value. Based on this similarity, Jeffrey Richter explained in Programming Applications for Microsoft Windows (Microsoft Press, 1999) how to inject a DLL into another process address space with InjectLib and unload it with EjectLib (see Inject.cpp in the download for implementation details). The code he provided has been modified here to dynamically use PSAPI through the corresponding CPSAPIWrapper class, and OpenProcess has been replaced by GetProcessHandleWithEnoughRights.

But why is loading a DLL of such interest? If you're writing the DLL, this is an easy way to let your code run in the context of the other process. To be of real interest, a communication channel should be set between the calling process and the code in the remote DLL. A driver app, GrabInfo, and an injected DLL, GrabHook, are built as an example. Their goal is to get four parameters that are not supposed to be known from another process. These parameters are the command line (GetCommandLine), the environment strings (GetEnvironmentStrings), if it is debugged (IsDebuggerPresent), and the window station under which it is running (GetProcessWindowStation and GetUserObjectInformation).

Each function to be remotely called is wrapped in a helper exported by the DLL and ends up in the same generic function: ExecuteRemoteAction. This function needs the process ID of the target application and a command ID corresponding to the action to be executed (see the RA_XXX constants in GrabHook.cpp). Both are stored in the variables s_dwProcessID and s_Action, and the execution failure or success result is saved in s_bSuccess. These three variables need to be accessed by both processes. Since the variables also need to share the same value in both processes, they are declared in a shared section of the DLL, using #pragma data_seg and #pragma comment, as shown here:

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

```
#pragma data_seg(".shared")
    DWORD s_Action      = 0;
    DWORD s_dwProcessID = 0;
    BOOL  s_bSuccess    = FALSE;
#pragma data_seg()

#pragma comment(linker, "-section:.shared,rws")
```

What happens between the settings of `s_Action` and `s_dwProcessID` in the calling process and the assignment of the result to `s_bSuccess` by the called process? Figure 13 illustrates this type of remote execution.

Once the DLL is loaded into the other process's address space, the `DllMain` entry point is called with `DLL_PROCESS_ATTACH` as parameter. The `s_dwProcessID` is compared to the ID of the running process as a security check. A memory-mapped file is created with `GrabHook_SharedBuffer` as name, and a pointer to the corresponding shared memory is stored in `g_lpvMem`. This buffer is used to exchange a large amount of data between the DLL code running inside the remote process and the calling process itself. If you need to allocate a buffer whose size is only known by the code being executed remotely, you simply replace the memory-mapped file by a piece of memory that's allocated using `VirtualAlloc`, and whose pointer address is stored in a shared variable. Then, in the calling process, you read the buffer content using `ReadProcessMemory`. Deallocate it with `VirtualFreeEx` to avoid any leak.

The rest is simply a question of buffer copying and unloading the DLL from the remote process. The same technique using `CreateRemoteThread` is used since `FreeLibrary` has the same signature as a thread procedure. The only tricky part is to find the address where the DLL has been loaded. `Toolhelp32` has been replaced by `PSAPI` so that the code can also run under Windows NT.

This injection technique has a few limitations. For some processes, such as those running as another user, `CreateRemoteThread` fails. In that case, the kernel emits the following error trace:

```
SE: Warning, new thread does not have
    SET_THREAD_TOKEN for itself
SE: Check that thread 468.2ec isn't in
    some weird state at the kernel level
```

But this `SET_THREAD_TOKEN` does not look like any SE privilege. Instead, it looks like `THREAD_SET_THREAD_TOKEN`, which is a thread-specific access right. Since no security descriptor (NULL) is passed as a parameter to `CreateRemoteThread`, this seems to be the reason why the call fails.

You should not use MFC in your injected DLL because you may get some undesired results, such as a crash if the targeted process is `csrss.exe`, (the Win32 subsystem).

In the June installment I presented three other solutions for getting the command line of a remote process. I also referred briefly to another solution that I called output reuse. The idea is to get the output from a console application and to parse it afterward to get the needed description. For the command line example, `TLIST` is a good candidate.

A lot of other console mode tools unveil tons of undocumented pieces of information about how

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

Windows is working, especially the Resources Kit and Platform SDK Tools (see Figure 14). A discussion of how to get the output from a console application can be found in Knowledge Base article Q190351.

All the grunt work has been wrapped in the CConsoleAppDriver class from which you should derive. You just need to indicate the command line to be executed through StartApp and, for each generated line, your OnNewLine override method will be called. If there is an error, simply set m_bParsingOk to FALSE. If you have caught the piece of information you need and you want to stop the parsing, you return FALSE.

The trick is to create a pipe dedicated to receiving the output of the application instead of letting Windows do the work. The link between this pipe and the new process is done in the PROCESS_INFORMATION structure passed to CreateProcess. Its hStdOutput field contains the end of the pipe that the class reads in order to get the process output (see ConsoleAppDriver.cpp for implementation details).

For a given DLL, DllSpy lists the processes that are using it. My tool provides the same feature for any given file. As you can see in Figure 14, OH is a console tool that lists for each process which kernel objects it is consuming. The major drawback of this console application is the incomplete naming convention. This line

```
000000A0 csrss.exe File 03bc \WINNT\system32\ega.cpi
```

means that CSRSS is using the file \WINNT\system32\ega.cpi. But, as you can see, there is no drive specification and the same kind of problem exists for registry keys whose names are not really user-friendly for Win32.

```
\REGISTRY\USER\S-1-5-21-1021013165-1664506389-1469997231-1938\Control  
Panel\International
```

The previous line should be translated into:

```
HKCU\Control Panel\International
```

Instead of using OH, you should get the HANDLE tool from <http://www.sysinternals.com>. This provides much better formatted output. The FileUsage tool, shown in Figure 15, takes advantage of the classes introduced in the June installment and implements a class derived from CConsoleAppDriver to parse the HANDLE output in its OnNewLine method (see FileInfo.cpp for parsing details). For both, GFlags from the Resource Kit must be used to enable the "Maintain a list of objects for each type" flag.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

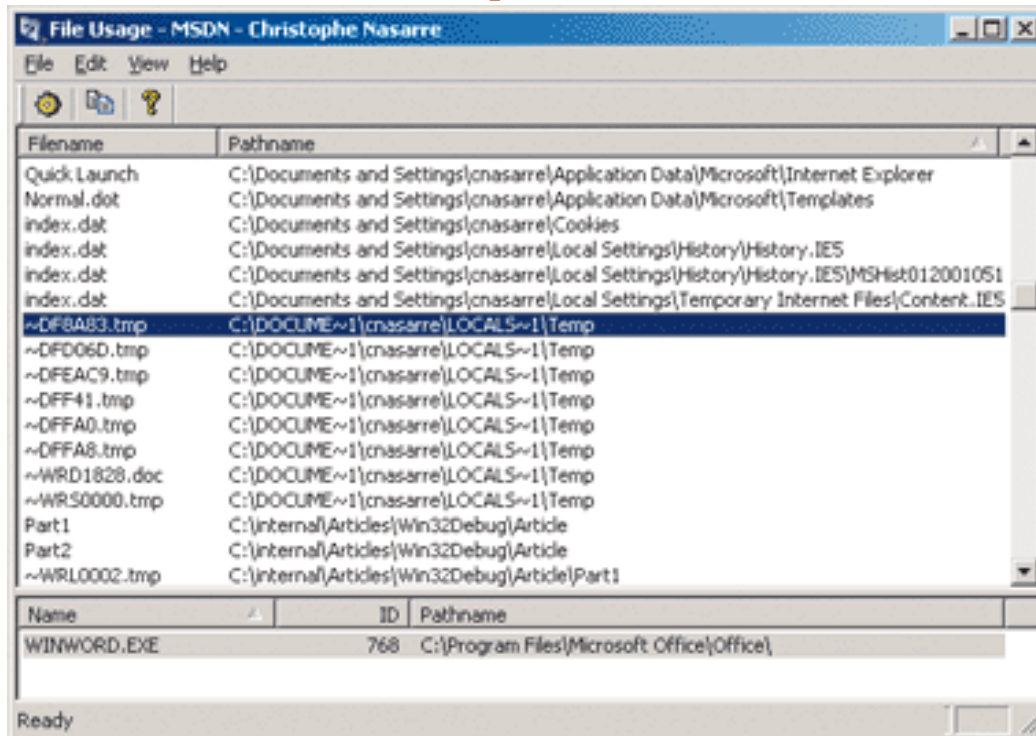


Figure 15 FileUsage Tool

Documentation

At the Win32 level, one of the more interesting structures used by Windows to manage processes, threads, and modules is the PEB. This structure is not documented in any file of the Platform SDK or the DDK. However, if you download the Debugging Tools for Windows, you get WinDbg. Using WinDbg you can dig into a PEB and much more.

In addition to being a kernel-mode debugger, WinDbg comes with invaluable extension DLLs. `kdex2x86` implements a `strct` command that allows you to discover the C-style definition of several interesting but undocumented data structures including PEB, EPROCESS, KPROCESS, and KTHREAD. Make sure to follow a simple rule: always use the extension for the Windows version you need to work on. Otherwise, these internal structures may be inconsistent with the real ones that your application tries to access. To use this command, you simply need to pretend to debug any process (such as Notepad); you are free to call `kdex2x86.strct` in the command line.

For a given process, it is not too difficult to access the content of its PEB using `ReadProcessMemory` and a process handle with `PROCESS_VM_READ` as desired access. This is because its PEB is always at address `0x7ffdf000` (or known using `NtQueryInformationProcess`). Unfortunately, the last three structures are located in the kernel and, therefore, are not accessible from a simple Win32-based application. If you want to cross this new frontier, you should first take a look at James Finnegan's March 1998 MSJ article, "Pop Open a Privileged Set of APIs with Windows NT Kernel Mode Drivers".

With the dissected PEB for Windows 2000 shown in Figure 16, the two opaque data structures used in the source code of the cryptic `GetProcessCmdLine` helper function from my June 2002 article are becoming quite similar.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

```
typedef struct
{
    DWORD Filler[4];
    DWORD InfoBlockAddress;
} __PEB;
```

```
typedef struct
{
    DWORD Filler[17];
    DWORD wszCmdLineAddress;
} __INFOBLOCK;
```

The command line is supposed to be stored in a memory block pointed to by a field in an __INFOBLOCK field, found using the field at a four DWORD offset from the beginning of the PEB. If you refer to Figure 16, you can see that a four DWORD offset (0x10) leads to the ProcessParameter pointer.

The next step is to find the definition of the _RTL_USER_PROCESS_PARAMETERS structure pointed to, and once again, the struct extension command gives you the answer, as shown in Figure 17.

The 17 DWORD-long filler leads to a 0x44 offset in _RTL_USER_PROCESS_PARAMETERS and jumps directly into the pointer part of the _UNICODE_STRING CommandLine field without using the Length field.

```
+040 struct    _UNICODE_STRING CommandLine
+040     uint16   Length
+042     uint16   MaximumLength
+044     uint16   *Buffer
```

In addition to listing each field of a PEB, WinDbg allows you to decipher some of them easily, using the !peb command.

As Figure 18 shows, if you load a process (here, oh.exe from the Resource Kit), the !peb command lists the values of some interesting fields, such as ProcessParameters.CommandLine. With both commands, you are well armed to dig even further into the inner recesses of Windows NT, Windows 2000, and Windows XP.

Figure 1 Events Received by the Debugger

Event Value	Description
CREATE_PROCESS_DEBUG_EVENT	This is the first event received by the debugger, even before LOAD_DLL_DEBUG_EVENT for statically linked DLLs.
EXIT_PROCESS_DEBUG_EVENT	This is the last event received by the debugger. It means the debuggee has reached the end of its life.
EXCEPTION_DEBUG_EVENT	An exception occurs. Its description is in u.Exception. It is received before any catch when the dw-FirstChance flag is set. If there is no catch, a second event is received before the debuggee is terminated.
CREATE_THREAD_DEBUG_EVENT	A new thread is created. Its description is in u.CreateThread.
EXIT_THREAD_DEBUG_EVENT	The description of an exiting thread is set in the u.ExitThread member.
LOAD_DLL_DEBUG_EVENT	When a DLL is mapped in the debuggee address space, either statically linked or dynamically loaded, this event is received by the debugger.
UNLOAD_DLL_DEBUG_EVENT	Unlike the previous event, this occurs only when a DLL is dynamically unloaded. This means it cannot be used to detect when each statically loaded DLL is unloaded at the end of the process life.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

Event Value	Description
OUTPUT_DEBUG_STRING_EVENT	Each time the debuggee calls OutputDebugString, the debugger receives this event with the string in u.DebugString.lpDebug.StringData, but in the debuggee address space.
RIP_EVENT	According to the documentation, this event is received when a RIP-de-bugging event (system debugging error) occurs, but I have never seen this in practice.

Figure 2 DEBUG_EVENT

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union
    {
        EXCEPTION_DEBUG_INFO      Exception;
        CREATE_THREAD_DEBUG_INFO   CreateThread;
        CREATE_PROCESS_DEBUG_INFO  CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO     ExitThread;
        EXIT_PROCESS_DEBUG_INFO    ExitProcess;
        LOAD_DLL_DEBUG_INFO        LoadDll;
        UNLOAD_DLL_DEBUG_INFO      UnloadDll;
        OUTPUT_DEBUG_STRING_INFO   DebugString;
        RIP_INFO                   RipInfo;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

Figure 5 DLL Details

Type	Member	Description
CString	m_szName	Module name
DWORD	m_LoadAddress	hModule
DWORD	m_PreferredLoadAddress	Supposed loading address (at link time)
CString	m_szReason	Gets real info
BOOL	m_bDynamic	TRUE if loaded through LoadLibrary
BOOL	m_bAfterStartup	TRUE if loaded after the process starts
DWORD	m_nLoaded	Number of times it has been loaded
DWORD	m_nRemoved	Number of times it has been unloaded
CString	m_szFullPath	Full path name of the DLL
DWORD	m_Position	Loading position, starting from 1

Figure 6 Debugger Thread Messages

ID	wParam	lParam	Description
UM_INITPROCESS	0	0	The statically linked DLLs have all been loaded. The icon changes from static to dynamic after this event.
UM_FREELIBRARY	0	CModuleInfo*	The DLL has been unloaded. The corresponding line is updated in CModuleListCtrl.
UM_LOADLIBRARY	0	CModuleInfo*	The DLL has been loaded. The corresponding line is added or updated in CModuleListCtrl.

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

Figure 7 XXX_DEBUG_EVENT Handles

Event	Handles to Close
LOAD_DLL_DEBUG_EVENT	u.LoadDll.hFile
CREATE_PROCESS_DEBUG_EVENT	u.CreateProcessInfo.hFile u.CreateProcessInfo.hProcess u.CreateProcessInfo.hThread

Figure 9 Icons for Loading Status


Icon	Description
F	Statically linked, and loaded at its preferred address
E	Statically linked, but not loaded at its preferred address
D	Loaded after the process is initialized at its preferred address
	Loaded after the process is initialized, but not at its preferred address
B	Dynamically loaded at its preferred address
A	Dynamically loaded, but not at its preferred address

Figure 12 WindowDump Information

Detail	How to Get It
ClassName	GetClassName
Title	GetWindowText
ID	GetWindowLong(GWL_ID)
Process	GetWindowThreadProcessId and GetFullModuleName
Handle	During enumeration
Styles	GetWindowLong(GWL_STYLE)
Dimension	GetWindowRect
State	IsWindowVisible and IsWindowEnabled

Figure 14 Useful Console Mode Tools in Resource Kit

Program	Description
dh	Lists memory, heap, stack, and kernel objects consumption and thread information
dhcmp	Used with DH to help find leaks
dmdiag	Provides information about your hard drives, mount points, partitions, and devices
drivers	Lists which drivers are loaded and what they consume
memsnag	Dumps running processes memory; handles consumption into a file
oh	Lists the kernel objects used by one or all processes
pstat	Lists running processes with their ID, user/kernel time, working set, page faults, committed memory, priority, and threads (with their state)
pulist	Lists running processes with their ID and owner
sclist	Lists services with their state and description
showpriv	Displays the trustees assigned to a privilege (user right)

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

Program	Description
uptime	Gives statistics about boot and shutdown events
vadump	Creates a listing containing information about the memory usage of a specified process with per-DLL usage (Virtual Address Dump)
where	Looks for file; /e provides the exe type
whoami	Lists user, group, and privilege for the current user

Figure 16 PEB Structure using WinDbg and kdex2x86

```
0:000> !kdex2x86.strct PEB
Loaded kdex2x86 extension DLL
struct  _PEB (sizeof=488)
+000 byte      InheritedAddressSpace
+001 byte      ReadImageFileExecOptions
+002 byte      BeingDebugged
+003 byte      SpareBool
+004 void      *Mutant
+008 void      *ImageBaseAddress
+00c struct    _PEB_LDR_DATA *Ldr
+010 struct    _RTL_USER_PROCESS_PARAMETERS *ProcessParameters
+014 void      *SubSystemData
+018 void      *ProcessHeap
+01c void      *FastPebLock
+020 void      *FastPebLockRoutine
+024 void      *FastPebUnlockRoutine
+028 uint32    EnvironmentUpdateCount
+02c void      *KernelCallbackTable
+030 uint32    SystemReserved[2]
+038 struct    _PEB_FREE_BLOCK *FreeList
+03c uint32    TlsExpansionCounter
+040 void      *TlsBitmap
+044 uint32    TlsBitmapBits[2]
+04c void      *ReadOnlySharedMemoryBase
+050 void      *ReadOnlySharedMemoryHeap
+054 void      **ReadOnlyStaticServerData
+058 void      *AnsiCodePageData
+05c void      *OemCodePageData
+060 void      *UnicodeCaseTableData
+064 uint32    NumberOfProcessors
+068 uint32    NtGlobalFlag
+070 union     _LARGE_INTEGER CriticalSectionTimeout
+070 uint32    LowPart
+074 int32     HighPart
+070 struct    __unnamed3 u
+070 uint32    LowPart
+074 int32     HighPart
+070 int64     QuadPart
+078 uint32    HeapSegmentReserve
+07c uint32    HeapSegmentCommit
+080 uint32    HeapDeCommitTotalFreeThreshold
+084 uint32    HeapDeCommitFreeBlockThreshold
+088 uint32    NumberOfHeaps
+08c uint32    MaximumNumberOfHeaps
+090 void      **ProcessHeaps
+094 void      *GdiSharedHandleTable
+098 void      *ProcessStarterHelper
```

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

```
+09c uint32  GdiDCAttributeList
+0a0 void    *LoaderLock
+0a4 uint32  OSMajorVersion
+0a8 uint32  OSMinorVersion
+0ac uint16  OSBuildNumber
+0ae uint16  OSCSDVersion
+0b0 uint32  OSPlatformId
+0b4 uint32  ImageSubsystem
+0b8 uint32  ImageSubsystemMajorVersion
+0bc uint32  ImageSubsystemMinorVersion
+0c0 uint32  ImageProcessAffinityMask
+0c4 uint32  GdiHandleBuffer[34]
+14c function *PostProcessInitRoutine
+150 void    *TlsExpansionBitmap
+154 uint32  TlsExpansionBitmapBits[32]
+1d4 uint32  SessionId
+1d8 void    *AppCompatInfo
+1dc struct  _UNICODE_STRING CSDVersion
+1dc uint16  Length
+1de uint16  MaximumLength
+1e0 uint16  *Buffer
```

Figure 17 Inner Structure of `_RTL_USER_PROCESS`

```
0:000> !kdex2x86.strct _RTL_USER_PROCESS_PARAMETERS
struct  _RTL_USER_PROCESS_PARAMETERS (sizeof=656)
+000 uint32  MaximumLength
+004 uint32  Length
+008 uint32  Flags
+00c uint32  DebugFlags
+010 void    *ConsoleHandle
+014 uint32  ConsoleFlags
+018 void    *StandardInput
+01c void    *StandardOutput
+020 void    *StandardError
+024 struct  _CURDIR CurrentDirectory
+024 struct  _UNICODE_STRING DosPath
+024 uint16  Length
+026 uint16  MaximumLength
+028 uint16  *Buffer
+02c void    *Handle
+030 struct  _UNICODE_STRING DllPath
+030 uint16  Length
+032 uint16  MaximumLength
+034 uint16  *Buffer
+038 struct  _UNICODE_STRING ImagePathName
+038 uint16  Length
+03a uint16  MaximumLength
+03c uint16  *Buffer
+040 struct  _UNICODE_STRING CommandLine
+040 uint16  Length
+042 uint16  MaximumLength
+044 uint16  *Buffer
+048 void    *Environment
+04c uint32  StartingX
+050 uint32  StartingY
+054 uint32  CountX
```

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

```
+058 uint32    County
+05c uint32    CountCharsX
+060 uint32    CountCharsY
+064 uint32    FillAttribute
+068 uint32    WindowFlags
+06c uint32    ShowWindowFlags
+070 struct    _UNICODE_STRING WindowTitle
+070 uint16    Length
+072 uint16    MaximumLength
+074 uint16    *Buffer
+078 struct    _UNICODE_STRING DesktopInfo
+078 uint16    Length
+07a uint16    MaximumLength
+07c uint16    *Buffer
+080 struct    _UNICODE_STRING ShellInfo
+080 uint16    Length
+082 uint16    MaximumLength
+084 uint16    *Buffer
+088 struct    _UNICODE_STRING RuntimeData
+088 uint16    Length
+08a uint16    MaximumLength
+08c uint16    *Buffer
+090 struct    _RTL_DRIVE_LETTER_CURDIR CurrentDirectoroes[32]
    uint16    Flags
    uint16    Length
    uint32    TimeStamp
    struct    _STRING DosPath
    uint16    Length
    uint16    MaximumLength
    char      *Buffer
```

Figure 18 Using !peb to Decipher Fields

```
0:000> !peb
PEB at 7FFDF000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:           Yes
    ImageBaseAddress:         01000000
    Ldr.Initialized: Yes
    Ldr.InInitializationOrderModuleList: 271f78 . 272290
    Ldr.InLoadOrderModuleList: 271ee0 . 272368
    Ldr.InMemoryOrderModuleList: 271ee8 . 272370
    01000000 C:\Program Files\Resource Pro Kit\oh.exe
    77F80000 C:\WINNT\System32\ntdll.dll
    78000000 C:\WINNT\system32\MSVCRT.dll
    77E80000 C:\WINNT\system32\KERNEL32.dll
    77DB0000 C:\WINNT\system32\ADVAPI32.dll
    77D40000 C:\WINNT\system32\RPCRT4.DLL
    SubSystemData:           0
    ProcessHeap:             270000
    ProcessParameters: 20000
    WindowTitle: 'C:\Program Files\Resource Pro Kit\oh.exe'
    ImageFile: 'C:\Program Files\Resource Pro Kit\oh.exe'
    CommandLine: '"C:\Program Files\Resource Pro Kit\oh.exe" '
    DllPath: 'C:\Program Files\Resource Pro
```

Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities Part 2

Christophe Nasarre

```
Kit;. ;C:\WINNT\System32;C:\WINNT\system;C:\WINNT;C:\WINNT\system32;C:\WINNT;
C:\
WINNT\System32\Wbem;C:\Program Files\Resource Pro
Kit\;c:\ntddk\bin;C:\Program Files\Microsoft Visual
Studio\VC98\Bin;C:\Program Files\Microsoft Platform SDK\Bin\;C:\Program
Files\Microsoft Platform SDK\Bin\WinNT'
Environment: 0x10000
```