Native Debugging

Now it's time to look at the native side of things, and how the wrapper layer inside ntdll.dll communicates with the kernel. The advantage of having the DbgUi layer is that it allows better separation between Win32 and the NT Kernel, which has always been a part of NT design. NTDLL and NTOSKRNL are built together, so it's normal for them to have intricate knowledge of each others. They share the same structures, they need to have the same system call IDs, etc. In a perfect world, the NT Kernel should have to know nothing about Win32.

Additionally, it helps anyone that wants to write debugging capabilities inside a native application, or to write a fully-featured native-mode debugger. Without DbgUi, one would have to call the Nt*DebugObject APIs manually, and do some extensive pre/post processing in some cases. DbgUi simplifies all this work to a simple call, and provides a clean interface to do it. If the kernel changes internally, DbgUi will probably stay the same, only its internal code would be modified.

We start our exploration with the function responsible for creating and associating a Debug Object with the current Process. Unlike in the Win32 world, there is a clear distinction between creating a Debug Object, and actually attaching to a process.

```
NTSTATUS
NTAPI
DbgUiConnectToDbg(VOID)
{
    OBJECT_ATTRIBUTES ObjectAttributes;

    /* Don't connect twice */
    if (NtCurrentTeb()->DbgSsReserved[1]) return STATUS_SUCCESS;

    /* Setup the Attributes */
    InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, 0);

    /* Create the object */
    return ZwCreateDebugObject(&NtCurrentTeb()->DbgSsReserved[1],
                               DEBUG_OBJECT_ALL_ACCESS,
                               &ObjectAttributes,
                               TRUE);
}
```

As you can see, this is a trivial implementation, but it shows us two things. Firstly, a thread can only have one debug object associated to it, and secondly, the handle to this object is stored in the TEB's DbgSsReserved array field. Recall that in Win32, the first index, [0], is where the Thread Data was stored. We've now learnt that [1] is where the handle is stored.

Now let's see how attaching and detaching are done:

```
NTSTATUS
NTAPI
DbgUiDebugActiveProcess(IN HANDLE Process)
{
    NTSTATUS Status;

    /* Tell the kernel to start debugging */
    Status = NtDebugActiveProcess(Process, NtCurrentTeb()->DbgSsReserved[1]);
    if (NT_SUCCESS(Status))
    {
        /* Now break-in the process */
        Status = DbgUiIssueRemoteBreakin(Process);
        if (!NT_SUCCESS(Status))
        {
            /* We couldn't break-in, cancel debugging */
            DbgUiStopDebugging(Process);
        }
    }

    /* Return status */
    return Status;
}

NTSTATUS
NTAPI
DbgUiStopDebugging(IN HANDLE Process)
{
    /* Call the kernel to remove the debug object */
    return NtRemoveProcessDebug(Process, NtCurrentTeb()->DbgSsReserved[1]);
}
```

Again, these are very simple implementations. We can learn, however, that the kernel is not responsible for actually breaking inside the remote process, but that this is done by the native layer. This DbgUiIssueRemoteBreakin API is also used by Win32 when calling DebugBreakProcess, so let's look at it:

```
NTSTATUS
NTAPI
DbgUiIssueRemoteBreakin(IN HANDLE Process)
{
    HANDLE hThread;
    CLIENT_ID ClientId;
    NTSTATUS Status;

    /* Create the thread that will do the breakin */
    Status = RtlCreateUserThread(Process,
                                 NULL,
                                 FALSE,
                                 0,
                                 0,
                                 PAGE_SIZE,
```

```
                                    (PVOID)DbgUiRemoteBreakin,
                                    NULL,
                                    &hThread,
                                    &ClientId);

    /* Close the handle on success */
    if(NT_SUCCESS(Status)) NtClose(hThread);

    /* Return status */
    return Status;
}
```

All it does is create a remote thread inside the process, and then return to the caller. Does that remote thread do anything magic? Let's see:

```
VOID
NTAPI
DbgUiRemoteBreakin(VOID)
{
    /* Make sure a debugger is enabled; if so, breakpoint */
    if (NtCurrentPeb()->BeingDebugged) DbgBreakPoint();

    /* Exit the thread */
    RtlExitUserThread(STATUS_SUCCESS);
}
```

Nothing special at all; the thread makes sure that the process is really being debugged, and then issues a breakpoint. And, because this API is exported, you can call it locally from your own process to issue a debug break (but note that you will kill your own thread). In our look at the Win32 Debugging implementation, we've noticed that the actual debug handle is never used, and that calls always go through DbgUi. Then the NtSetInformationDebugObject system call was called, a special DbgUi API was called before, to actually get the debug object associated with the thread. This API also has a counterpart, so let's see both in action:

```
HANDLE
NTAPI
DbgUiGetThreadDebugObject(VOID)
{
    /* Just return the handle from the TEB */
    return NtCurrentTeb()->DbgSsReserved[1];
}

VOID
NTAPI
DbgUiSetThreadDebugObject(HANDLE DebugObject)
{
    /* Just set the handle in the TEB */
    NtCurrentTeb()->DbgSsReserved[1] = DebugObject;
}
```

For those familiar with object-oriented programming, this will seem similar to the concept of accessor and mutator methods. Even though Win32 has perfect access to this handle and could simply read it on its own, the NT developers decided to make DbgUi much like a class, and make sure access to the handle goes through these public methods. This design allows the debug handle to be stored anywhere else if necessary, and only these two APIs will require changes, instead of multiple DLLs in Win32.

Now for a visit of the wait/continue functions, which under Win32 were simply wrappers:

```
NTSTATUS
NTAPI
DbgUiContinue(IN PCLIENT_ID ClientId,
              IN NTSTATUS ContinueStatus)
{
    /* Tell the kernel object to continue */
    return ZwDebugContinue(NtCurrentTeb()->DbgSsReserved[1],
                           ClientId,
                           ContinueStatus);
}

NTSTATUS
NTAPI
DbgUiWaitStateChange(OUT PDBGUI_WAIT_STATE_CHANGE DbgUiWaitStateCange,
                     IN PLARGE_INTEGER TimeOut OPTIONAL)
{
    /* Tell the kernel to wait */
    return NtWaitForDebugEvent(NtCurrentTeb()->DbgSsReserved[1],
                               TRUE,
                               TimeOut,
                               DbgUiWaitStateCange);
}
```

Not surprisingly, these functions are also wrappers in DbgUi. However, this is where things start to get interesting, since if you'll recall, DbgUi uses a completely different structure for debug events, called DBGUI_WAIT_STATE_CHANGE. There is one API that we have left to look at, which does the conversion, so first, let's look at the documentation for this structure:

```
//
// User-Mode Debug State Change Structure
//
typedef struct _DBGUI_WAIT_STATE_CHANGE
{
    DBG_STATE NewState;
    CLIENT_ID AppClientId;
    union
    {
        struct
        {
```

```
            HANDLE HandleToThread;
            DBGKM_CREATE_THREAD NewThread;
        } CreateThread;
        struct
        {
            HANDLE HandleToProcess;
            HANDLE HandleToThread;
            DBGKM_CREATE_PROCESS NewProcess;
        } CreateProcessInfo;
        DBGKM_EXIT_THREAD ExitThread;
        DBGKM_EXIT_PROCESS ExitProcess;
        DBGKM_EXCEPTION Exception;
        DBGKM_LOAD_DLL LoadDll;
        DBGKM_UNLOAD_DLL UnloadDll;
    } StateInfo;
} DBGUI_WAIT_STATE_CHANGE, *PDBGUI_WAIT_STATE_CHANGE;
```

The fields should be pretty self-explanatory, so let's look at the DBG_STATE enumeration:

```
//
// Debug States
//
typedef enum _DBG_STATE
{
    DbgIdle,
    DbgReplyPending,
    DbgCreateThreadStateChange,
    DbgCreateProcessStateChange,
    DbgExitThreadStateChange,
    DbgExitProcessStateChange,
    DbgExceptionStateChange,
    DbgBreakpointStateChange,
    DbgSingleStepStateChange,
    DbgLoadDllStateChange,
    DbgUnloadDllStateChange
} DBG_STATE, *PDBG_STATE;
```

If you take a look at the Win32 DEBUG_EVENT structure and associated debug event types, you'll notice some differences which might be useful to you. For starters, Exceptions, Breakpoints and Single Step exceptions are handled differently. In the Win32 world, only two distinctions are made: RIP_EVENT for exceptions, and EXCEPTION_DEBUG_EVENT for a debug event. Although code can later figure out if this was a breakpoint or single step, this information comes directly in the native structure. You will also notice that OUTPUT_DEBUG_STRING event is missing. Here, it's DbgUi that's at a disadvantage, since the information is sent as an Exception, and post-processing is required (which we'll take a look at soon). There are also two more states that Win32 does not support, which is the Idle state and the Reply Pending state. These don't offer much information from the point of view of a debugger, so they are ignored.

Now let's take a look at the actual structures seen in the unions:

```
//
// Debug Message Structures
//
typedef struct _DBGKM_EXCEPTION
{
    EXCEPTION_RECORD ExceptionRecord;
    ULONG FirstChance;
} DBGKM_EXCEPTION, *PDBGKM_EXCEPTION;

typedef struct _DBGKM_CREATE_THREAD
{
    ULONG SubSystemKey;
    PVOID StartAddress;
} DBGKM_CREATE_THREAD, *PDBGKM_CREATE_THREAD;

typedef struct _DBGKM_CREATE_PROCESS
{
    ULONG SubSystemKey;
    HANDLE FileHandle;
    PVOID BaseOfImage;
    ULONG DebugInfoFileOffset;
    ULONG DebugInfoSize;
    DBGKM_CREATE_THREAD InitialThread;
} DBGKM_CREATE_PROCESS, *PDBGKM_CREATE_PROCESS;

typedef struct _DBGKM_EXIT_THREAD
{
    NTSTATUS ExitStatus;
} DBGKM_EXIT_THREAD, *PDBGKM_EXIT_THREAD;

typedef struct _DBGKM_EXIT_PROCESS
{
    NTSTATUS ExitStatus;
} DBGKM_EXIT_PROCESS, *PDBGKM_EXIT_PROCESS;

typedef struct _DBGKM_LOAD_DLL
{
    HANDLE FileHandle;
    PVOID BaseOfDll;
    ULONG DebugInfoFileOffset;
    ULONG DebugInfoSize;
    PVOID NamePointer;
} DBGKM_LOAD_DLL, *PDBGKM_LOAD_DLL;

typedef struct _DBGKM_UNLOAD_DLL
{
    PVOID BaseAddress;
} DBGKM_UNLOAD_DLL, *PDBGKM_UNLOAD_DLL;
```

If you're familiar with the DEBUG_EVENT structure, you should notice some subtle differences. First of all, no indication of the process name, which explains why MSDN documents this field being optional and not used by Win32. You will also notice the lack

of a pointer to the TEB in the thread structure. Finally, unlike new processes, Win32 does display the name of any new DLL loaded, but this also seems to be missing in the Load DLL structure; we'll see how this and other changes are dealt with soon. As far as extra information goes however, we have the "SubsystemKey" field. Because NT was designed to support multiple subsystems, this field is critical to identifying from which subsystem the new thread or process was created from. Windows 2003 SP1 adds support for debugging POSIX applications, and while I haven't looked at the POSIX debug APIs, I'm convinced they're built around the DbgUi implementation, and that this field is used differently by the POSIX library (much like Win32 ignores it).

Now that we've seen the differences, the final API to look at is DbgUiConvertStateChangeStructure, which is responsible for doing these modifications and fixups:

```
NTSTATUS
NTAPI
DbgUiConvertStateChangeStructure(IN PDBGUI_WAIT_STATE_CHANGE WaitStateChange,
                                 OUT PVOID Win32DebugEvent)
{
    NTSTATUS Status;
    OBJECT_ATTRIBUTES ObjectAttributes;
    THREAD_BASIC_INFORMATION ThreadBasicInfo;
    LPDEBUG_EVENT DebugEvent = Win32DebugEvent;
    HANDLE ThreadHandle;

    /* Write common data */
    DebugEvent->dwProcessId = (DWORD)WaitStateChange->
                                     AppClientId.UniqueProcess;
    DebugEvent->dwThreadId = (DWORD)WaitStateChange->AppClientId.UniqueThread;

    /* Check what kind of even this is */
    switch (WaitStateChange->NewState)
    {
        /* New thread */
        case DbgCreateThreadStateChange:

            /* Setup Win32 code */
            DebugEvent->dwDebugEventCode = CREATE_THREAD_DEBUG_EVENT;

            /* Copy data over */
            DebugEvent->u.CreateThread.hThread =
                WaitStateChange->StateInfo.CreateThread.HandleToThread;
            DebugEvent->u.CreateThread.lpStartAddress =
                WaitStateChange->StateInfo.CreateThread.NewThread.StartAddress;

            /* Query the TEB */
            Status = NtQueryInformationThread(WaitStateChange->StateInfo.
                                              CreateThread.HandleToThread,
                                              ThreadBasicInformation,
                                              &ThreadBasicInfo,
                                              sizeof(ThreadBasicInfo),
```

```
                                        NULL);
    if (!NT_SUCCESS(Status))
    {
        /* Failed to get PEB address */
        DebugEvent->u.CreateThread.lpThreadLocalBase = NULL;
    }
    else
    {
        /* Write PEB Address */
        DebugEvent->u.CreateThread.lpThreadLocalBase =
            ThreadBasicInfo.TebBaseAddress;
    }
    break;

/* New process */
case DbgCreateProcessStateChange:

    /* Write Win32 debug code */
    DebugEvent->dwDebugEventCode = CREATE_PROCESS_DEBUG_EVENT;

    /* Copy data over */
    DebugEvent->u.CreateProcessInfo.hProcess =
        WaitStateChange->StateInfo.CreateProcessInfo.HandleToProcess;
    DebugEvent->u.CreateProcessInfo.hThread =
        WaitStateChange->StateInfo.CreateProcessInfo.HandleToThread;
    DebugEvent->u.CreateProcessInfo.hFile =
        WaitStateChange->StateInfo.CreateProcessInfo.NewProcess.
        FileHandle;
    DebugEvent->u.CreateProcessInfo.lpBaseOfImage =
        WaitStateChange->StateInfo.CreateProcessInfo.NewProcess.
        BaseOfImage;
    DebugEvent->u.CreateProcessInfo.dwDebugInfoFileOffset =
        WaitStateChange->StateInfo.CreateProcessInfo.NewProcess.
        DebugInfoFileOffset;
    DebugEvent->u.CreateProcessInfo.nDebugInfoSize =
        WaitStateChange->StateInfo.CreateProcessInfo.NewProcess.
        DebugInfoSize;
    DebugEvent->u.CreateProcessInfo.lpStartAddress =
        WaitStateChange->StateInfo.CreateProcessInfo.NewProcess.
        InitialThread.StartAddress;

    /* Query TEB address */
    Status = NtQueryInformationThread(WaitStateChange->StateInfo.
                                        CreateProcessInfo.HandleToThread,
                                        ThreadBasicInformation,
                                        &ThreadBasicInfo,
                                        sizeof(ThreadBasicInfo),
                                        NULL);
    if (!NT_SUCCESS(Status))
    {
        /* Failed to get PEB address */
        DebugEvent->u.CreateThread.lpThreadLocalBase = NULL;
    }
    else
    {
```

```
            /* Write PEB Address */
            DebugEvent->u.CreateThread.lpThreadLocalBase =
                ThreadBasicInfo.TebBaseAddress;
        }

        /* Clear image name */
        DebugEvent->u.CreateProcessInfo.lpImageName = NULL;
        DebugEvent->u.CreateProcessInfo.fUnicode = TRUE;
        break;

    /* Thread exited */
    case DbgExitThreadStateChange:

        /* Write the Win32 debug code and the exit status */
        DebugEvent->dwDebugEventCode = EXIT_THREAD_DEBUG_EVENT;
        DebugEvent->u.ExitThread.dwExitCode =
            WaitStateChange->StateInfo.ExitThread.ExitStatus;
        break;

    /* Process exited */
    case DbgExitProcessStateChange:

        /* Write the Win32 debug code and the exit status */
        DebugEvent->dwDebugEventCode = EXIT_PROCESS_DEBUG_EVENT;
        DebugEvent->u.ExitProcess.dwExitCode =
            WaitStateChange->StateInfo.ExitProcess.ExitStatus;
        break;

    /* Any sort of exception */
    case DbgExceptionStateChange:
    case DbgBreakpointStateChange:
    case DbgSingleStepStateChange:

        /* Check if this was a debug print */
        if (WaitStateChange->StateInfo.Exception.ExceptionRecord.
            ExceptionCode == DBG_PRINTEXCEPTION_C)
        {
            /* Set the Win32 code */
            DebugEvent->dwDebugEventCode = OUTPUT_DEBUG_STRING_EVENT;

            /* Copy debug string information */
            DebugEvent->u.DebugString.lpDebugStringData =
                (PVOID)WaitStateChange->
                       StateInfo.Exception.ExceptionRecord.
                       ExceptionInformation[1];
            DebugEvent->u.DebugString.nDebugStringLength =
                WaitStateChange->StateInfo.Exception.ExceptionRecord.
                ExceptionInformation[0];
            DebugEvent->u.DebugString.fUnicode = FALSE;
        }
        else if (WaitStateChange->StateInfo.Exception.ExceptionRecord.
                 ExceptionCode == DBG_RIPEXCEPTION)
        {
            /* Set the Win32 code */
            DebugEvent->dwDebugEventCode = RIP_EVENT;
```

```
            /* Set exception information */
            DebugEvent->u.RipInfo.dwType =
                WaitStateChange->StateInfo.Exception.ExceptionRecord.
                ExceptionInformation[1];
            DebugEvent->u.RipInfo.dwError =
                WaitStateChange->StateInfo.Exception.ExceptionRecord.
                ExceptionInformation[0];
        }
        else
        {
            /* Otherwise, this is a debug event, copy info over */
            DebugEvent->dwDebugEventCode = EXCEPTION_DEBUG_EVENT;
            DebugEvent->u.Exception.ExceptionRecord =
                WaitStateChange->StateInfo.Exception.ExceptionRecord;
            DebugEvent->u.Exception.dwFirstChance =
                WaitStateChange->StateInfo.Exception.FirstChance;
        }
        break;

    /* DLL Load */
    case DbgLoadDllStateChange :

        /* Set the Win32 debug code */
        DebugEvent->dwDebugEventCode = LOAD_DLL_DEBUG_EVENT;

        /* Copy the rest of the data */
        DebugEvent->u.LoadDll.lpBaseOfDll =
            WaitStateChange->StateInfo.LoadDll.BaseOfDll;
        DebugEvent->u.LoadDll.hFile =
            WaitStateChange->StateInfo.LoadDll.FileHandle;
        DebugEvent->u.LoadDll.dwDebugInfoFileOffset =
            WaitStateChange->StateInfo.LoadDll.DebugInfoFileOffset;
        DebugEvent->u.LoadDll.nDebugInfoSize =
            WaitStateChange->StateInfo.LoadDll.DebugInfoSize;

        /* Open the thread */
        InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, NULL);
        Status = NtOpenThread(&ThreadHandle,
                              THREAD_QUERY_INFORMATION,
                              &ObjectAttributes,
                              &WaitStateChange->AppClientId);
        if (NT_SUCCESS(Status))
        {
            /* Query thread information */
            Status = NtQueryInformationThread(ThreadHandle,
                                              ThreadBasicInformation,
                                              &ThreadBasicInfo,
                                              sizeof(ThreadBasicInfo),
                                              NULL);
            NtClose(ThreadHandle);
        }

        /* Check if we got thread information */
        if (NT_SUCCESS(Status))
```

```
            {
                /* Save the image name from the TIB */
                DebugEvent->u.LoadDll.lpImageName =
                    &((PTEB)ThreadBasicInfo.TebBaseAddress)->
                    Tib.ArbitraryUserPointer;
            }
            else
            {
                /* Otherwise, no name */
                DebugEvent->u.LoadDll.lpImageName = NULL;
            }

            /* It's Unicode */
            DebugEvent->u.LoadDll.fUnicode = TRUE;
            break;

        /* DLL Unload */
        case DbgUnloadDllStateChange:

            /* Set Win32 code and DLL Base */
            DebugEvent->dwDebugEventCode = UNLOAD_DLL_DEBUG_EVENT;
            DebugEvent->u.UnloadDll.lpBaseOfDll =
                WaitStateChange->StateInfo.UnloadDll.BaseAddress;
            break;

        /* Anything else, fail */
        default: return STATUS_UNSUCCESSFUL;
    }

    /* Return success */
    return STATUS_SUCCESS;
}
```

Let's take a look at the interesting fixups. First of all, the lack of a TEB pointer is easily fixed by calling NtQueryInformationThread with the ThreadBasicInformation type, which returns, among other things, a pointer to the TEB, which is then saved in the Win32 structure. As for Debug Strings, the API analyzes the exception code and looks for DBG_PRINTEXCEPTION_C, which has a specific exception record that is parsed and converted into a debug string output.

So far so good, but perhaps the nastiest hack is present in the code for DLL loading. Because a loaded DLL doesn't have a structure like EPROCESS or ETHREAD in kernel memory, but in ntdll's private Ldr structures, the only thing that identifies it is a Section Object in memory for its memory mapped file. When the kernel gets a request to create a section for an executable memory mapped file, it saves the name of the file in a field inside the TEB (or TIB, rather) called ArbitraryUserPointer.

This function then knows that a string is located there, and sets it as the pointer for the debug event's lpImageName member. This hack has been in NT every since the first builds, and as far as I know, it's still there in Vista. Could it be that hard to solve?

Once again, we come to an end in our discussion, since there isn't much left in ntdll that deals with the Debug Object. Here's an overview of what was discussed in this part of the series:

DbgUi provides a level of separation between the kernel and Win32 or other subsystems. It's written as a fully independent class, even having accessor and mutator methods instead of exposing its handles.

The handle to a thread's Debug Object is stored in the second field of the DbgSsReserved array in the TEB.

DbgUi allows a thread to have a single DebugObject, but using the native system calls allows you to do as many as you want.

Most DbgUi APIs are simple wrappers around the NtXxxDebugObject system calls, and use the TEB handle to communicate.

DbgUi is responsible for breaking into the attached process, not the kernel.

DbgUi uses its own structure for debug events, which the kernel understands. In some ways, this structure provides more information about some events (such as the subsystem and whether this was a single step or a breakpoint exception), but in others, some information is missing (such as a pointer to the thread's TEB or a separate debug string structure).

The TIB (located inside the TEB)'s ArbitraryPointer member contains the name of the loaded DLL during a Debug Event.