User Mode Debugging Internals

Introduction

The internal mechanisms of what allows user-mode debugging to work have rarely ever been fully explained. Even worse, these mechanisms have radically changed in Windows XP, when much of the support was re-written, as well as made more subsystem portable by including most of the routines in ntdll, as part of the Native API. This article will explain this functionality in three parts, starting from the Win32 (kernel32) viewpoint all the way down (or up) to the NT Kernel (ntoskrnl)'s module responsible for this support, called Dbgk.

The reader is expected to have some basic knowledge of C and general NT Kernel architecture and semantics. Also, this is not an introduction on what debugging is or how to write a debugger. It is meant as a reference for experienced debugger writers, or curious security experts.

Win32 Debugging

The Win32 subsystem of NT has allowed the debugging of processes ever since the first release, with later releases adding more features and debugging help libraries, related to symbols and other PE information. However, relatively few things have changed to the outside API user, except for the welcome addition of the ability to stop debugging a process, without killing it, which was added in Windows XP. This release of NT also contained several overhauls to the underlying implementation, which will be discussed in detail. However, one important side-effect of these changes was that LPC (and csrss.exe) were not used anymore, which allowed debugging of this binary to happen (previously, debugging this binary was impossible, since it was the one responsible for handling the kernel-to-user notifications).

The basic Win32 APIs for dealing with debugging a process were simple: DebugActiveProcess, to attach, WaitForDebugEvent, to wait for debug events to come through, so that your debugging can handle them, and ContinueDebugEvent, to resume thread execution. The release of Windows XP added three more useful APIs: DebugActiveProcessStop, which allows you to stop debugging a process (detach), DebugSetProcessKillOnExit, which allows you to continue running a process even after its' been detached, and DebugBreakProcess, which allows you to perform a remote DebugBreak without having to manually create a remote thread. In Windows XP Service Pack 1, one more API was added, CheckRemoteDebuggerPresent. Much like its IsDebuggerPresent counterpart, this API allows you to check for a connected debugger in another process, without having to read the PEB remotely.

Because of NT's architecture, these APIs, on recent versions of Windows (2003 will be used as an example, but the information applies to XP as well) do not much do much work themselves. Instead, they do the typical job of calling out the native functions required, and then process the output so that the Win32 caller can have it in a format that is compatible with Win9x and the original Win32 API definition. Let's look at these very simple implementations:

```
BOOL
WINAPI
DebugActiveProcess(IN DWORD dwProcessId)
{
    NTSTATUS Status;
    HANDLE Handle;

    /* Connect to the debugger */
    Status = DbgUiConnectToDbg();
    if (!NT_SUCCESS(Status))
    {
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Get the process handle */
    Handle = ProcessIdToHandle(dwProcessId);
    if (!Handle) return FALSE;

    /* Now debug the process */
    Status = DbgUiDebugActiveProcess(Handle);
    NtClose(Handle);

    /* Check if debugging worked */
    if (!NT_SUCCESS(Status))
    {
        /* Fail */
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Success */
    return TRUE;
}
```

As you can see, the only work that's being done here is to create the initial connection to the user-mode debugging component, which is done through the DbgUi Native API Set, located in ntdll, which we'll see later. Because DbgUi uses handles instead of PIDs, the PID must first be converted with a simple helper function:

```
HANDLE
WINAPI
ProcessIdToHandle(IN DWORD dwProcessId)
{
```

```
    NTSTATUS Status;
    OBJECT_ATTRIBUTES ObjectAttributes;
    HANDLE Handle;
    CLIENT_ID ClientId;

    /* If we don't have a PID, look it up */
    if (dwProcessId == -1) dwProcessId = (DWORD)CsrGetProcessId();

    /* Open a handle to the process */
    ClientId.UniqueProcess = (HANDLE)dwProcessId;
    InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, NULL);
    Status = NtOpenProcess(&Handle,
                           PROCESS_ALL_ACCESS,
                           &ObjectAttributes,
                           &ClientId);
    if (!NT_SUCCESS(Status))
    {
        /* Fail */
        SetLastErrorByStatus(Status);
        return 0;
    }

    /* Return the handle */
    return Handle;
}
```

If you are not familiar with Native API, it is sufficient to say that this code is the simple equivalent of an OpenProcess on the PID, so that a handle can be obtained. Going back to DebugActiveProcess, the final call which does the work is DbgUiDebugActiveProcess, which is again located in the Native API. After the connection is made, we can close the handle that we had obtained from the PID previously. Other APIs function much in the same way. Let's take a look at two of the newer XP ones:

```
BOOL
WINAPI
DebugBreakProcess(IN HANDLE Process)
{
    NTSTATUS Status;

    /* Send the breakin request */
    Status = DbgUiIssueRemoteBreakin(Process);
    if(!NT_SUCCESS(Status))
    {
        /* Failure */
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Success */
    return TRUE;
}
```

```
BOOL
WINAPI
DebugSetProcessKillOnExit(IN BOOL KillOnExit)
{
    HANDLE Handle;
    NTSTATUS Status;
    ULONG State;

    /* Get the debug object */
    Handle = DbgUiGetThreadDebugObject();
    if (!Handle)
    {
        /* Fail */
        SetLastErrorByStatus(STATUS_INVALID_HANDLE);
        return FALSE;
    }

    /* Now set the kill-on-exit state */
    State = KillOnExit;
    Status = NtSetInformationDebugObject(Handle,
                                         DebugObjectKillProcessOnExitInformation,
                                           &State,
                                           sizeof(State),
                                           NULL);
    if (!NT_SUCCESS(Status))
    {
        /* Fail */
        SetLastError(Status);
        return FALSE;
    }

    /* Success */
    return TRUE;
}
```

The first hopefully requires no explanation, as it's a simple wrapper, but let's take a look at the second. If you're familiar with the Native API, you'll instantly recognize the familiar NtSetInformationXxx type of API, which is used for setting various settings on the different types of NT Objects, such as files, processes, threads, etc. The interesting to note here, which is new to XP, is that debugging itself is also now done with a Debug Object. The specifics of this object will however be discussed later. For now, let's look at the function. The first API, DbgUiGetThreadDebugObject is another call to DbgUi, which will return a handle to the Debug Object associated with our thread (we'll see where this is stored later). Once we have the handle, we call a Native API which directly communicates with Dbgk (and not DbgUi), which will simply change a flag in the kernel's Debug Object structure. This flag, as we'll see, will be read by the kernel when detaching.

A similar function to this one is the CheckRemoteDebuggerPresent, which uses the same type of NT semantics to obtain the information about the process:

```
BOOL
WINAPI
CheckRemoteDebuggerPresent(IN HANDLE hProcess,
                           OUT PBOOL pbDebuggerPresent)
{
    HANDLE DebugPort;
    NTSTATUS Status;

    /* Make sure we have an output and process*/
    if (!(pbDebuggerPresent) || !(hProcess))
    {
        /* Fail */
        SetLastError(ERROR_INVALID_PARAMETER);
        return FALSE;
    }

    /* Check if the process has a debug object/port */
    Status = NtQueryInformationProcess(hProcess,
                                       ProcessDebugPort,
                                       (PVOID)&DebugPort,
                                       sizeof(HANDLE),
                                       NULL);
    if (NT_SUCCESS(Status))
    {
        /* Return the current state */
        *pbDebuggerPresent = (DebugPort) ? TRUE : FALSE;
        return TRUE;
    }

    /* Otherwise, fail */
    SetLastErrorByStatus(Status);
    return FALSE;
}
```

As you can see, another NtQuery/SetInformationXxx API is being used, but this time for the process. Although you probably now that to detect debugging, one can simple check `if ((BOOL)NtCurrentPeb()->BeingDebugged);`, there exists another way to do this, and this is by querying the kernel. Since the kernel needs to communicate with user-mode on debugging events, it needs some sort of way of doing this. Before XP, this used to be done through an LPC port, and now, through a Debug Object (which shares the same pointer, however).

Since is located in the EPROCESS structure in kernel mode, we do a query, using the DebugPort information class. If EPROCESS->DebugPort is set to something, then this API will return TRUE, which means that the process is being debugged. This trick can also be used for the local process, but it's much faster to simply read the PEB. One can notice that although some applications like to set Peb->BeingDebugged to FALSE to trick anti-debugging programs, there is no way to set DebugPort to NULL, since the Kernel itself would not let you debug (and you also don't have access to kernel structures).

With that in mind, let's see how the gist of the entire Win32 debugging infrastructure, WaitForDebugEvent, is implemented. This needs to be shown before the much-simpler ContinueDebugEvent/DebugActiveProcessStop, because it introduces Win32's high-level internal structure that it uses to wrap around DbgUi.

```
BOOL
WINAPI
WaitForDebugEvent(IN LPDEBUG_EVENT lpDebugEvent,
                  IN DWORD dwMilliseconds)
{
    LARGE_INTEGER WaitTime;
    PLARGE_INTEGER Timeout;
    DBGUI_WAIT_STATE_CHANGE WaitStateChange;
    NTSTATUS Status;

    /* Check if this is an infinite wait */
    if (dwMilliseconds == INFINITE)
    {
        /* Under NT, this means no timer argument */
        Timeout = NULL;
    }
    else
    {
        /* Otherwise, convert the time to NT Format */
        WaitTime.QuadPart = UInt32x32To64(-10000, dwMilliseconds);
        Timeout = &WaitTime;
    }

    /* Loop while we keep getting interrupted */
    do
    {
        /* Call the native API */
        Status = DbgUiWaitStateChange(&WaitStateChange, Timeout);
    } while ((Status == STATUS_ALERTED) || (Status == STATUS_USER_APC));

    /* Check if the wait failed */
    if (!(NT_SUCCESS(Status)) || (Status != DBG_UNABLE_TO_PROVIDE_HANDLE))
    {
        /* Set the error code and quit */
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Check if we timed out */
    if (Status == STATUS_TIMEOUT)
    {
        /* Fail with a timeout error */
        SetLastError(ERROR_SEM_TIMEOUT);
        return FALSE;
    }

    /* Convert the structure */
    Status = DbgUiConvertStateChangeStructure(&WaitStateChange, lpDebugEvent);
```

```
    if (!NT_SUCCESS(Status))
    {
        /* Set the error code and quit */
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Check what kind of event this was */
    switch (lpDebugEvent->dwDebugEventCode)
    {
        /* New thread was created */
        case CREATE_THREAD_DEBUG_EVENT:

            /* Setup the thread data */
            SaveThreadHandle(lpDebugEvent->dwProcessId,
                             lpDebugEvent->dwThreadId,
                             lpDebugEvent->u.CreateThread.hThread);
            break;

        /* New process was created */
        case CREATE_PROCESS_DEBUG_EVENT:

            /* Setup the process data */
            SaveProcessHandle(lpDebugEvent->dwProcessId,
                              lpDebugEvent->u.CreateProcessInfo.hProcess);

            /* Setup the thread data */
            SaveThreadHandle(lpDebugEvent->dwProcessId,
                             lpDebugEvent->dwThreadId,
                             lpDebugEvent->u.CreateThread.hThread);
            break;

        /* Process was exited */
        case EXIT_PROCESS_DEBUG_EVENT:

            /* Mark the thread data as such */
            MarkProcessHandle(lpDebugEvent->dwProcessId);
            break;

        /* Thread was exited */
        case EXIT_THREAD_DEBUG_EVENT:

            /* Mark the thread data */
            MarkThreadHandle(lpDebugEvent->dwThreadId);
            break;

        /* Nothing to do for anything else */
        default:
            break;
    }

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

First, let's look at the DbgUi APIs present. The first, DbgUiWaitStateChange is the Native version of WaitForDebugEvent, and it's responsible for doing the actual wait on the Debug Object, and getting the structure associated with this event. However, DbgUi uses its own internal structures (which we'll show later) so that the Kernel can understand it, while Win32 has had much different structures defined in the Win9x ways. Therefore, one needs to convert this to the Win32 representation, and the DbgUiConvertStateChange API is what does this conversion, returning the LPDEBUG_EVENT Win32 structure that is backwards-compatible and documented on MSDN.

What follows after is a switch which is interested in the creation or deletion of a new process or thread. Four APIs are used: SaveProcessHandle and SaveThreadHandle, which save these respective handles (remember that a new process must have an associated thread, so the thread handle is saved as well), and MarkProcessHandle and MarkThreadHandle, which flag these handles as being exited. Let's look as this high-level framework in detail.

```
VOID
WINAPI
SaveProcessHandle(IN DWORD dwProcessId,
                  IN HANDLE hProcess)
{
    PDBGSS_THREAD_DATA ThreadData;

    /* Allocate a thread structure */
    ThreadData = RtlAllocateHeap(RtlGetProcessHeap(),
                                 0,
                                 sizeof(DBGSS_THREAD_DATA));
    if (!ThreadData) return;

    /* Fill it out */
    ThreadData->ProcessHandle = hProcess;
    ThreadData->ProcessId = dwProcessId;
    ThreadData->ThreadId = 0;
    ThreadData->ThreadHandle = NULL;
    ThreadData->HandleMarked = FALSE;

    /* Link it */
    ThreadData->Next = DbgSsGetThreadData();
    DbgSsSetThreadData(ThreadData);
}
```

This function allocates a new structure, DBGSS_THREAD_DATA, and simply fills it out with the Process handle and ID that was sent. Finally, it links it with the current DBGSS_THREAD_DATA structure, and set itself as the new current one (thus creating a circular list of DBGSS_THREAD_DATA structures). Let's take a look as this structure:

```
typedef struct _DBGSS_THREAD_DATA
{
    struct _DBGSS_THREAD_DATA *Next;
    HANDLE ThreadHandle;
    HANDLE ProcessHandle;
    DWORD ProcessId;
    DWORD ThreadId;
    BOOLEAN HandleMarked;
} DBGSS_THREAD_DATA, *PDBGSS_THREAD_DATA;
```

This generic structure thus allows storing process/thread handles and IDs, as well as the flag which we've talked about in regards to MarkProcess/ThreadHandle. We've also seen some DbgSsSet/GetThreadData functions, which will show us where this circular array of structures is located. Let's look at their implementations:

```
#define DbgSsSetThreadData(d) \
    NtCurrentTeb()->DbgSsReserved[0] = d

#define DbgSsGetThreadData() \
    ((PDBGSS_THREAD_DATA)NtCurrentTeb()->DbgSsReserved[0])
```

Easy enough, and now we know what the first element of the mysterious DbgSsReserved array in the TEB is. Although you can probably guess the SaveThreadHandle implementation yourself, let's look at it for completeness's sake:

```
VOID
WINAPI
SaveThreadHandle(IN DWORD dwProcessId,
                 IN DWORD dwThreadId,
                 IN HANDLE hThread)
{
    PDBGSS_THREAD_DATA ThreadData;

    /* Allocate a thread structure */
    ThreadData = RtlAllocateHeap(RtlGetProcessHeap(),
                                 0,
                                 sizeof(DBGSS_THREAD_DATA));
    if (!ThreadData) return;

    /* Fill it out */
    ThreadData->ThreadHandle = hThread;
    ThreadData->ProcessId = dwProcessId;
    ThreadData->ThreadId = dwThreadId;
    ThreadData->ProcessHandle = NULL;
    ThreadData->HandleMarked = FALSE;

    /* Link it */
    ThreadData->Next = DbgSsGetThreadData();
    DbgSsSetThreadData(ThreadData);
}
```

As expected, nothing new here. The MarkThread/Process functions as just as straight-forward:

```
VOID
WINAPI
MarkThreadHandle(IN DWORD dwThreadId)
{
    PDBGSS_THREAD_DATA ThreadData;

    /* Loop all thread data events */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
        /* Check if this one matches */
        if (ThreadData->ThreadId == dwThreadId)
        {
            /* Mark the structure and break out */
            ThreadData->HandleMarked = TRUE;
            break;
        }

        /* Move to the next one */
        ThreadData = ThreadData->Next;
    }
}
VOID
WINAPI
MarkProcessHandle(IN DWORD dwProcessId)
{
    PDBGSS_THREAD_DATA ThreadData;

    /* Loop all thread data events */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
        /* Check if this one matches */
        if (ThreadData->ProcessId == dwProcessId)
        {
            /* Make sure the thread ID is empty */
            if (!ThreadData->ThreadId)
            {
                /* Mark the structure and break out */
                ThreadData->HandleMarked = TRUE;
                break;
            }
        }

        /* Move to the next one */
        ThreadData = ThreadData->Next;
    }
}
```

Notice that the only less-than-trivial implementation detail is that the array needs to be parsed in order to find the matching Process and Thread ID.

Now that we've taken a look at these structures, let's see the associated ContinueDebugEvent API, which picks up after a WaitForDebugEvent API in order to resume the thread.

```
BOOL
WINAPI
ContinueDebugEvent(IN DWORD dwProcessId,
                   IN DWORD dwThreadId,
                   IN DWORD dwContinueStatus)
{
    CLIENT_ID ClientId;
    NTSTATUS Status;

    /* Set the Client ID */
    ClientId.UniqueProcess = (HANDLE)dwProcessId;
    ClientId.UniqueThread = (HANDLE)dwThreadId;

    /* Continue debugging */
    Status = DbgUiContinue(&ClientId, dwContinueStatus);
    if (!NT_SUCCESS(Status))
    {
        /* Fail */
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Remove the process/thread handles */
    RemoveHandles(dwProcessId, dwThreadId);

    /* Success */
    return TRUE;
}
```

Again, we're dealing with a DbgUI API, DbgUiContinue, which is going to do all the work for us. Our only job is to call RemoveHandles, which is part of the high-level structures that wrap DbgUi. This functions is slightly more complex then what we've seen, because we're given PID/TIDs, so we need to do some lookups:

```
VOID
WINAPI
RemoveHandles(IN DWORD dwProcessId,
              IN DWORD dwThreadId)
{
    PDBGSS_THREAD_DATA ThreadData;

    /* Loop all thread data events */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
        /* Check if this one matches */
        if (ThreadData->ProcessId == dwProcessId)
        {
```

```
            /* Make sure the thread ID matches too */
            if (ThreadData->ThreadId == dwThreadId)
            {
                /* Check if we have a thread handle */
                if (ThreadData->ThreadHandle)
                {
                    /* Close it */
                    CloseHandle(ThreadData->ThreadHandle);
                }

                /* Check if we have a process handle */
                if (ThreadData->ProcessHandle)
                {
                    /* Close it */
                    CloseHandle(ThreadData->ProcessHandle);
                }

                /* Unlink the thread data */
                DbgSsSetThreadData(ThreadData->Next);

                /* Free it*/
                RtlFreeHeap(RtlGetProcessHeap(), 0, ThreadData);

                /* Move to the next structure */
                ThreadData = DbgSsGetThreadData();
                continue;
            }
        }

        /* Move to the next one */
        ThreadData = ThreadData->Next;
    }
}
```

Not much explaining is required. As we parse the circular buffer, we try to locate a structure which matches the PID and TID that we were given. Once it's been located, we check if a handle is associated with the thread and the process. If it is, then we can now close the handle.

Therefore, the use of this high-level Win32 mechanism is now apparent: it's how we can associate handles to IDs, and close them when cleaning up or continuing. This is because these handles were not opened by Win32, but behind its back by Dbgk. Once the handles are closed, we unlink this structure by changing the TEB pointer to the next structure in the array, and we then free our own Array. We then resume parsing from the next structure on (because more than one such structure could be associated with this PID/TID).

Finally, one last piece of the Win32 puzzle is missing in our analysis, and this is the detach function, which was added in XP. Let's take a look at its trivial implementation:

```
BOOL
WINAPI
DebugActiveProcessStop(IN DWORD dwProcessId)
{
    NTSTATUS Status;
    HANDLE Handle;

    /* Get the process handle */
    Handle = ProcessIdToHandle(dwProcessId);
    if (!Handle) return FALSE;

    /* Close all the process handles */
    CloseAllProcessHandles(dwProcessId);

    /* Now stop debgging the process */
    Status = DbgUiStopDebugging(Handle);
    NtClose(Handle);

    /* Check for failure */
    if (!NT_SUCCESS(Status))
    {
        /* Fail */
        SetLastError(ERROR_ACCESS_DENIED);
        return FALSE;
    }

    /* Success */
    return TRUE;
}
```

It couldn't really get any simpler. Just like for attaching, we first convert the PID to a handle, and then use a DbgUi call (DbgUiStopDebugging) with this process handle in order to detach ourselves from the process. There's one more call being made here, which is CloseAllProcessHandles. This is part of Win32's high-level debugging on top of DbgUi, which we've seen just earlier. This routine is very similar to RemoveHandles, but it only deals with a Process ID, so the implementation is simpler:

```
VOID
WINAPI
CloseAllProcessHandles(IN DWORD dwProcessId)
{
    PDBGSS_THREAD_DATA ThreadData;

    /* Loop all thread data events */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
        /* Check if this one matches */
        if (ThreadData->ProcessId == dwProcessId)
        {
            /* Check if we have a thread handle */
            if (ThreadData->ThreadHandle)
```

```
            {
                /* Close it */
                CloseHandle(ThreadData->ThreadHandle);
            }

            /* Check if we have a process handle */
            if (ThreadData->ProcessHandle)
            {
                /* Close it */
                CloseHandle(ThreadData->ProcessHandle);
            }

            /* Unlink the thread data */
            DbgSsSetThreadData(ThreadData->Next);

            /* Free it*/
            RtlFreeHeap(RtlGetProcessHeap(), 0, ThreadData);

            /* Move to the next structure */
            ThreadData = DbgSsGetThreadData();
            continue;
        }

        /* Move to the next one */
        ThreadData = ThreadData->Next;
    }
}
```

And this completes our analysis of the Win32 APIs! Let's take a look at what we've learnt:

The actual debugging functionality is present in a module called Dbgk inside the Kernel.

It's accessible through the DbgUi Native API interface, located inside the NT System Library, ntdll.

Dbgk implements debugging functionality through an NT Object, called a Debug Object, which also provides an NtSetInformation API in order to modify certain flags.

The Debug Object associated to a thread can be retrieved with DbgUiGetThreadObject, but we have not yet shown where this is stored.

Checking if a process is being debugged can be done by using NtQueryInformationProcess and using the DebugPort information class. This cannot be cheated without a rootkit.

Because Dbgk opens certain handles during Debug Events, Win32 needs a way to associated IDs and handles, and uses a circular array of structures called DBGSS_THREAD_DATA to store this in the TEB's DbgSsReserved[0] member.