기술/번역문서	11. 8. 28. 작성

Windows Debugging Intrenals: A to Z

Written by ***Alex Ionescu***
Translated by ***6l4ck3y3***

Introduction

Windows 의 내부 매커니즘이 어떻게 유저 모드 디버깅을 제공하는지에 대하여 설명한 Alex Ionescu 의 3부작 시리즈를 번역하였습니다. Alex Ionescu 는 ReactOS 의 커널 개발자이면서『Windows Internals』의 저자입니다.

이 문서는 Dbgk 라는 NT 커널(ntoskrnl)의 객체와 디버그 이벤트를 주고받는 방법에 대해서 Win32 (kernel32) 관점부터 DbgUi 객체와 NT 시스템 라이브러리(ntdll)까지 설명합니다. 문서를 이해하기 위해서는 C 언어와 NT 커널 아키텍처(혹은 운영체제)에 대한 기초 지식이 필요합니다. 또한, 이 글은 디버깅이 무엇인지, 디버거를 어떻게 작성하는지 소개하는 글이 아닙니다. 경험 있는 디버거 작성자나 호기심이 강한 보안 전문가를 위한 레퍼런스입니다.

* 본 문서는 **KISA 대학정보보호동아리연합회(KUCIS)**와 **(주)잉카인터넷**의 지원을 받아 작성된 문서입니다.

Table of Content

1. Win32 Debugging

NT 계열의 Win32 시스템들은 처음 버전부터 프로세스의 디버깅을 제공했습니다. 그리고 이후 버전에서 몇몇 기능들과 디버깅 라이브러리들, 관련 심볼들, PE 정보가 추가되어 왔습니다. 그러나 상대적으로 바깥의 API 사용자에게는 변한 게 거의 없죠. Windows XP 에서 추가된 프로세스를 종료시키지 않으면서 디버깅을 멈추는 기능(DebugSetProcessKillOnExit) 같은 것을 제외하고 말이죠. 나중에 상세하게 설명하겠지만, XP 는 내부 코드에서 조금 개선되었습니다. 그러나 이러한 변화로 생긴 한 가지 큰 부작용은 LPC(Local Procedure Call) 포트와 csrss.exe 가 더 이상 쓰이지 않게 된 것입니다. 이것들은 커널 영역과 유저 영역 사이의 전달 문제로 디버깅이 힘들었던 바이너리를 디버깅할 수 있게 해주었던 모듈입니다.

프로세스를 디버깅하기 위한 기본적인 Win32 API 들은 간단합니다. 프로세스에 디버거를 연결할 때는 DebugActiveProcess API 를 호출합니다. 그리고 WaitForDebugEvent API 로 디버그 이벤트가 올 때까지 기다렸다가 이벤트를 처리하고, ContinueDebugEvent API 로 스레드를 다시 실행합니다. Windows XP 가 나오면서 세 개의 유용한 API 들이 더 추가됐습니다. 프로세스에서 디버거의 연결을 끊기 위해서는 DebugActiveProcessStop API 를 호출하고, 프로세스 실행을 유지하면서 디버거의 연결을 끊기 위해서는 DebugSetProcessKillOnExit API 를 호출합니다. 그리고 DebugBreakProcess 는 직접 원격 스레드를 생성할 필요없이 원격으로 DebugBreak 를 수행하기 위한 API 입니다. Windows XP Service Pack 1 에서 API 가 하나 더 추가 되었습니다. CheckRemoteDebuggerPresent API 는 IsDebuggerPresent 와 비슷하지만, PEB 를 읽지 않고 다른 프로세스가 디버거에 연결되었는지를 확인합니다.

NT 아키텍처 때문에 Windows 의 최근 버전(2003을 예제로 쓰겠지만, XP 에서도 잘 적용될 것입니다)에서 이러한 API 들은 그 자체로 동작하지는 못 합니다. 대신에 네이티브 함수를 호출하는 기존의 방식을 사용합니다. 그러면 Win32 호출자가 Win9x 와 기존의 Win32 API 정의에 호환되는 형식을 가질 수 있도록 결과가 처리됩니다. 아래의 코드를 보시죠.

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

    /* 디버거에 연결한다 */
    Status = DbgUiConnectToDbg();
    if (!NT_SUCCESS(Status))
    {
```



```
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* 프로세스 핸들을 구한다 */
    Handle = ProcessIdToHandle(dwProcessId);
    if (!Handle) return FALSE;

    /* 프로세스 디버깅을 시작한다 */
    Status = DbgUiDebugActiveProcess(Handle);
    NtClose(Handle);

    /* 디버깅이 동작하는지 검사한다 */
    if (!NT_SUCCESS(Status))
    {
        /* Fail */
        SetLastErrorByStatus(Status);
        return FALSE;
    }

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

위에서 볼 수 있듯이, 유저 모드 디버깅 객체에 초기 연결을 하는 작업이 전부입니다. 작업은 나중에 보게 될 ntdll 의 DbgUi 네이티브 API 를 통해서 이루어집니다. DbgUi 가 PID 대신 핸들을 사용하기 때문에, PID 는 우선 ProcessIdToHandle 를 통해 변환되어야 합니다.

```
HANDLE

WINAPI

ProcessIdToHandle(IN DWORD dwProcessId)

{

    NTSTATUS Status;

    OBJECT_ATTRIBUTES ObjectAttributes;

    HANDLE Handle;
```



```
    CLIENT_ID ClientId;


    /* 유효한 PID 가 아니면, PID 를 구한다 */
    if (dwProcessId == -1) dwProcessId = (DWORD)CsrGetProcessId();


    /* 프로세스의 핸들을 얻는다 */
    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;
    }


    /* 핸들을 반환한다 */
    return Handle;
}
```

네이티브 API 에 익숙하지 않다면, PID 를 가지고 OpenProcess 를 호출한 것과 같다고 이해하여도 좋습니다. 그래서 핸들을 얻을 수 있는 것이죠. DebugActiveProcess 로 돌아가서, DbgUiDebugActiveProcess 가 마지막으로 호출되는 것을 볼 수 있습니다. 이것도 네이티브 API 입니다. 연결이 이루어진 후에, PID 로 미리 얻은 핸들을 닫을 수 있습니다.

다른 API 함수들도 같은 방식입니다. XP 의 새로운 두 함수를 보시죠.


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

    /* 프로세스에 디버거를 연결하도록 요청한다 */
    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;
```

```
/* 디버그 객체를 구한다 */
Handle = DbgUiGetThreadDebugObject();
if (!Handle)
{
    /* Fail */
    SetLastErrorByStatus(STATUS_INVALID_HANDLE);
    return FALSE;
}

/*
 * kill-on-exit 상태를 설정한다
 * DebugObjectKillProcessOnExitInformation 에 디버그 객체의 정보가 있다
 */
State = KillOnExit;
Status = NtSetInformationDebugObject(Handle,
                                     DebugObjectKillProcessOnExitInformation,
                                     &State,
                                     sizeof(State),
                                     NULL);
if (!NT_SUCCESS(Status))
{
    /* Fail */
    SetLastError(Status);
    return FALSE;
}

/* Success */
return TRUE;
```

```
}
```

첫 번째 함수는 그다지 설명할 게 없습니다. 간단한 래퍼 함수입니다. 그러나, 두 번째 함수를 보시죠. 네이티브 API 에 친숙하다면, NtSetInformationXXX 형태의 API 와 유사하다는 것을 알 수 있을 것입니다. NtSetInformationXXX 형태의 함수는 특정 변수들을 다른 타입의 NT 객체들에 설정할 때 쓰입니다. 이러한 NT 객체에는 파일이나 프로세스, 스레드 같은 것들이 있습니다. 여기서 XP 의 흥미로우면서 새로운 것에 주목할 필요가 있습니다. 그것은 디버그 객체로 자기 자신 또한 디버깅하는 것입니다. 이제 함수를 보겠습니다.

첫 번째 API, DbgUiGetThreadDebugObject 는 또 다른 DbgUi API 함수입니다. 스레드와 관련된 디버그 객체에 대한 핸들을 반환합니다(저장되는 곳은 나중에 보겠습니다). 핸들을 얻으면 Dbgk(DbgUi 가 아닙니다)와 직접적으로 통신을 하는 네이티브 API(NtSetInformationDebugObject)를 호출합니다. 이 API 는 커널의 디버그 객체 구조체의 플래그(KillProcessOnExit)를 변경합니다. 이 플래그는 디버거가 종료될 때 커널이 확인힙니다.

이것과 비슷한 함수가 CheckRemoteDebuggerPresent 입니다. 프로세스의 정보를 얻기 위해 NT 의 같은 타입인 디버그 객체를 사용합니다.

```
BOOL

WINAPI

CheckRemoteDebuggerPresent(IN HANDLE hProcess,

                           OUT PBOOL pbDebuggerPresent)

{

    HANDLE DebugPort;

    NTSTATUS Status;


    /* pbDebuggerPresent 나 프로세스 핸들이 유효한지 확인한다 */

    if (!(pbDebuggerPresent) || !(hProcess))

    {

        /* Fail */

        SetLastError(ERROR_INVALID_PARAMETER);
```



```
        return FALSE;
    }

    /* 프로세스가 디버그 객체나 포트를 가지고 있는지 검사한다 */
    Status = NtQueryInformationProcess(hProcess,
                                       ProcessDebugPort,
                                       (PVOID)&DebugPort,
                                       sizeof(HANDLE),
                                       NULL);
    if (NT_SUCCESS(Status))
    {
        /* 디버깅 중이면 현재 상태를 반환한다 */
        *pbDebuggerPresent = (DebugPort) ? TRUE : FALSE;
        return TRUE;
    }

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

위에서 볼 수 있듯이, 이번에는 프로세스에 대하여 또 다른 NtQuery/SetInformationXXX API 가 쓰였습니다. NtCurrentPeb()->BeingDebugged 가 참인지 확인해서 디버깅을 탐지할 수 있지만, 이것과 다르게 커널에 질의를 하는 다른 방법이 있습니다. 커널은 디버깅 이벤트에서 유저 모드와 통신을 해야 하기 때문에, 이런 식의 방법이 필요합니다. XP 이전에는 LPC 포트를 통해서 가능했습니다. 그리고 지금은 디버그 객체를 통해서 가능합니다(하지만 같은 포인터를 공유하죠).

커널 모드의 EPROCESS 구조체의 디버그 포트(DebugPort) 정보 클래스를 통해 질의를 합니다. 만약 EPROCESS->DebugPort 가 무언가로 설정되어 있으면, CheckRemoteDebuggerPresent 는 프로세스가 디버깅중이라는 의미로 TRUE 를 반환합니다. 이 방법은 지역 프로세스에 사용될 수 있지만, 간단히 PEB 를

읽는 게 더 빠릅니다. 몇몇 애플리케이션에서 Peb->BeingDebugged 를 FALSE 로 설정하는 방식으로 안티-디버깅 프로그램을 우회하는 것을 볼 수 있습니다. 그러나 커널 자체는 디버그가 안 되기 때문에 DebugPort 를 NULL 로 설정하는 방법은 없습니다(그리고 유저 모드에서는 커널 구조체에 접근하지도 못합니다).

그걸 염두에 두고, Win32 디버깅 구조 전체의 핵심을 보겠습니다. WaitForDebugEvent 가 있습니다. 이 함수는 ContinueDebugEvent 나 DebugActiveProcessStop 보다 먼저 호출되어야 합니다. DbgUi 를 감싸는 Win32 의 하이레벨 내부 구조 때문이죠.

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

    /* 디버그 이벤트를 무한으로 대기할 지 검사한다 */
    if (dwMilliseconds == INFINITE)
    {
        /* NT 에서 무한 대기이면, 타이머 인자는 쓰이지 않는다 */
        Timeout = NULL;
    }
    else
    {
        /* 무한 대기가 아니면, 타이머를 NT 포맷으로 변환한다 */
        WaitTime.QuadPart = UInt32x32To64(-10000, dwMilliseconds);
        Timeout = &WaitTime;
```



```
}

/* 인터럽트를 받는동안 반복한다 */
do
{
    /* 네이티브 API 를 호출한다 */
    Status = DbgUiWaitStateChange(&WaitStateChange, Timeout);
} while ((Status == STATUS_ALERTED) || (Status == STATUS_USER_APC));

/* 디버그 이벤트를 대기하는 게 실패했는지 검사한다 */
if (!(NT_SUCCESS(Status)) || (Status != DBG_UNABLE_TO_PROVIDE_HANDLE))
{
    /* 오류 코드를 설정하고 빠져 나간다 */
    SetLastErrorByStatus(Status);
    return FALSE;
}

/* 대기 시간을 초과했는지 검사한다 */
if (Status == STATUS_TIMEOUT)
{
    /* 시간 초과 오류로 실패 */
    SetLastError(ERROR_SEM_TIMEOUT);
    return FALSE;
}

/* WaitStateChange 구조체를 DebugEvent 구조체로 변환한다 */
Status = DbgUiConvertStateChangeStructure(&WaitStateChange, lpDebugEvent);
if (!NT_SUCCESS(Status))
```

```
{
    /* 오류 코드를 설정하고 빠져 나간다 */
    SetLastErrorByStatus(Status);
    return FALSE;
}

/* 디버그 이벤트의 종류를 확인한다 */
switch (lpDebugEvent->dwDebugEventCode)
{
    /* 새로운 스레드가 생성되면 */
    case CREATE_THREAD_DEBUG_EVENT:

        /* 스레드 정보를 설정한다 */
        SaveThreadHandle(lpDebugEvent->dwProcessId,
                         lpDebugEvent->dwThreadId,
                         lpDebugEvent->u.CreateThread.hThread);
        break;

    /* 새로운 프로세스가 생성되면 */
    case CREATE_PROCESS_DEBUG_EVENT:

        /* 프로세스 정보를 설정한다 */
        SaveProcessHandle(lpDebugEvent->dwProcessId,
                          lpDebugEvent->u.CreateProcessInfo.hProcess);

        /* 그리고 스레드 정보도 설정한다 */
        SaveThreadHandle(lpDebugEvent->dwProcessId,
                         lpDebugEvent->dwThreadId,
```

```
                            lpDebugEvent->u.CreateThread.hThread);
            break;

        /* 프로세스가 종료되면 */
        case EXIT_PROCESS_DEBUG_EVENT:

            /* 프로세스 정보를 종료된 상태로 설정한다 */
            MarkProcessHandle(lpDebugEvent->dwProcessId);
            break;

        /* 스레드가 종료되면 */
        case EXIT_THREAD_DEBUG_EVENT:

            /* 스레드 정보를 종료된 상태로 설정한다 */
            MarkThreadHandle(lpDebugEvent->dwThreadId);
            break;

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

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

먼저, DbgUi API 를 보죠. 첫 번째, DbgUiWaitStateChange 는 WaitForDebugEvent 의 네이티브 버전입니다. 실제로 디버그 객체를 계속해서 기다리고, 이벤트와 관련된 구조체를 얻습니다. 그러나, DbgUi 는 나중에 살펴 볼 커널의 내부 구조체(DBGUI_WAIT_STATE_CHANGE)를 사용합니다. 그런데 Win32 는 Win9x

방식으로 정의된 다른 형태의 구조체를 사용해 왔습니다. 그래서 DbgUiConvertStateChange API 를 이용하여 Win32 형태로 변환해야 합니다. Win32 의 LPDEBUG_EVENT 구조체를 반환하는 것은 하위 호환성 때문입니다. DEBUG_EVENT 구조체에 대하여 자세한 내용은 부록(Appendix)에 있습니다.

다음의 함수는 새로운 프로세스나 스레드를 생성/제거하는 스위치 역할을 합니다. 네 개의 API 중에서 SaveProcessHandle 과 SaveThreadHandle 은 각각의 핸들을 저장하고(새로운 프로세스는 반드시 하나의 연관 스레드를 가집니다. 그래서 스레드 핸들도 저장되죠), MarkProcessHandle 과 MarkThreadHandle 은 핸들의 플래그를 종료된 상태로 설정합니다. 이에 대해서 하이레벨의 프레임워크를 자세히 보도록 하죠.

```
VOID

WINAPI

SaveProcessHandle(IN DWORD dwProcessId,

                  IN HANDLE hProcess)

{

    PDBGSS_THREAD_DATA ThreadData;


    /* 스레드 구조체를 할당한다 */

    ThreadData = RtlAllocateHeap(RtlGetProcessHeap(),

                                 0,

                                 sizeof(DBGSS_THREAD_DATA));

    if (!ThreadData) return;


    /* 그리고 구조체에 정보를 채워넣는다 */

    ThreadData->ProcessHandle = hProcess;

    ThreadData->ProcessId = dwProcessId;

    ThreadData->ThreadId = 0;

    ThreadData->ThreadHandle = NULL;

    ThreadData->HandleMarked = FALSE;
```

```
    /* 다음의 ThreadData 를 연결한다 */
    ThreadData->Next = DbgSsGetThreadData();
    DbgSsSetThreadData(ThreadData);
}
```

SaveProcessHandle 은 스레드 정보(ThreadData) 구조체를 새로 할당합니다. 그리고 인자로 넘겨받은 프로세스 핸들과 ID 를 채워넣습니다. 마지막으로, 현재의 ThreadData 구조체를 Next 로 연결하고, 새로 할당된 ThreadData 를 현재의 ThreadData 구조체로 설정합니다(이렇게 DBGSS_THREAD_DATA 구조체의 원형 리스트가 생성됩니다). DBGSS_THREAD_DATA 구조체에 대해 한 번 보겠습니다.

```
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;
```

일반적인 형태의 구조체로, MarkProcess/ThreadHandle 에서 얘기했던 플래그(HandleMarked)와 프로세스/스레드의 핸들과 ID 를 담고 있습니다. DbgSsSet/GetThreadData 함수는 DBGSS_THREAD_DATA 구조체의 원형 배열(리스트)이 위치한 곳을 보여줍니다. DbgSsSet/GetThreadData 함수의 코드입니다.

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

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


DbgSsReserved 배열의 첫 번째 원소가 TEB 에 있음을 알 수 있습니다.

SaveThreadHandle 구현도 동일할 거라는 걸 추측할 수 있지만, 마저 보겠습니다.

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

    /* 스레드 구조체를 할당한다 */
    ThreadData = RtlAllocateHeap(RtlGetProcessHeap(),
                                 0,
                                 sizeof(DBGSS_THREAD_DATA));
    if (!ThreadData) return;

    /* 그리고 구조체에 정보를 채워넣는다 */
    ThreadData->ThreadHandle = hThread;
    ThreadData->ProcessId = dwProcessId;
    ThreadData->ThreadId = dwThreadId;
    ThreadData->ProcessHandle = NULL;
    ThreadData->HandleMarked = FALSE;

    /* 다음의 PDBGSS_THREAD_DATA 를 연결한다 */
    ThreadData->Next = DbgSsGetThreadData();
```

```
    DbgSsSetThreadData(ThreadData);
}
```

예상대로, 새로운 건 없습니다. MarkThread/Process 함수는 바로 다음에 나오는 바와 같습니다.

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

    /* 모든 스레드 정보에 대하여 반복한다 */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
        /* 스레드 ID 가 일치한지 검사한다 */
        if (ThreadData->ThreadId == dwThreadId)
        {
            /* 일치하면 플래그를 설정하고 빠져나간다 */
            ThreadData->HandleMarked = TRUE;
            break;
        }

        /* 다음의 스레드를 처리한다 */
        ThreadData = ThreadData->Next;
    }
}
```



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

    /* 모든 스레드 정보에 대하여 반복한다 */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
        /* 프로세스 ID 가 일치한지 검사한다 */
        if (ThreadData->ProcessId == dwProcessId)
        {
            /* 스레드 ID 를 확인한다 */
            if (!ThreadData->ThreadId)
            {
                /* 일치하면 플래그를 설정하고 빠져나간다 */
                ThreadData->HandleMarked = TRUE;
                break;
            }
        }

        /* 다음의 스레드를 처리한다 */
        ThreadData = ThreadData->Next;
    }
}
```

단지 일치하는 프로세스와 스레드의 ID 를 찾기 위해서 ThreadData 에 대한 분석이 필요할 뿐입니다. 지금까지 DBGSS_THREAD_DATA 를 살펴보았습니다.

ContinueDebugEvent 는 WaitForDebugEvent 가 호출된 후 스레드를 다시 시작하기 위해서 이용됩니다. ContinueDebugEvent API 를 보겠습니다.

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

    /* 클라이언트 ID 를 설정한다 */
    ClientId.UniqueProcess = (HANDLE)dwProcessId;
    ClientId.UniqueThread = (HANDLE)dwThreadId;

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

    /* 프로세스와 스레드의 핸들을 제거한다 */
    RemoveHandles(dwProcessId, dwThreadId);
```



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

이번에도 DbgUI API 가 있습니다. DbgUiContinue 가 모든 일을 다 하고, DbgUi 를 감싸는 하이레벨 구조면에서는 RemoveHandles 가 유일하게 호출되는 작업입니다. RemoveHandles 는 PID/TID 를 같이 파라미터로 넘겨받기 때문에 조금 더 복잡합니다. 그래서 조금 살펴보겠습니다.

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

    /* 모든 스레드 정보에 대하여 반복한다 */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
        /* 프로세스 ID 가 일치한지 검사한다 */
        if (ThreadData->ProcessId == dwProcessId)
        {
            /* 스레드 ID 도 일치하는지 검사한다 */
            if (ThreadData->ThreadId == dwThreadId)
            {
                /* 스레드 핸들이 있는지 검사한다 */
                if (ThreadData->ThreadHandle)
                {
                    /* 스레드 핸들을 닫는다 */
```



```
                CloseHandle(ThreadData->ThreadHandle);
            }

            /* 프로세스 핸들이 있는지 검사한다 */
            if (ThreadData->ProcessHandle)
            {
                /* 프로세스 핸들을 닫는다 */
                CloseHandle(ThreadData->ProcessHandle);
            }

            /* ThreadData 의 연결을 제거한다 */
            DbgSsSetThreadData(ThreadData->Next);

            /* 메모리 할당을 해제한다 */
            RtlFreeHeap(RtlGetProcessHeap(), 0, ThreadData);

            /* 다음의 ThreadData 를 처리한다 */
            ThreadData = DbgSsGetThreadData();
            continue;
        }
    }

    /* 다음의 ThreadData 를 처리한다 */
    ThreadData = ThreadData->Next;
  }
}
```

설명이 많이 필요하진 않습니다. 순환되는 리스트를 분석하기 때문에, 넘겨받은 PID/TID 와 일치하는 PID 와 TID 를 가진 ThreadData 를 찾아서 핸들이 프로세스/스레드와 연관된 핸들인지 검사합니다. 연관

되었다면, 핸들을 닫습니다.

그러므로, 이런 하이레벨 Win32 매커니즘의 용도는 분명합니다. 핸들을 ID 로 바꿀 수 있고, 끝내거나 진행할 때 핸들을 닫을 수도 있습니다. 이러한 핸들들은 Win32 가 아닌, Dbgk 에 의해 뒤에서 열립니다. 핸들이 닫히면, TEB 의 ThreadData 포인터를 리스트의 다음 구조체로 설정하여서 현재 ThreadData 의 연결을 끊습니다. 그러고나서 리스트를 해제합니다. 그리고 다음 구조체를 다시 분석합니다(왜냐면, 하나 이상의 ThreadData 가 이 PID/TID 와 관련있을 수 있기 때문이죠).

마지막으로, Win32 퍼즐의 마지막 한 조각이 남았습니다. 바로 detach 함수입니다. XP 에서 추가된 DebugActiveProcessStop 입니다. 코드를 보시죠.

```
BOOL

WINAPI

DebugActiveProcessStop(IN DWORD dwProcessId)

{

    NTSTATUS Status;

    HANDLE Handle;


    /* 프로세스 핸들을 얻는다 */

    Handle = ProcessIdToHandle(dwProcessId);

    if (!Handle) return FALSE;


    /* 모든 프로세스 핸들을 닫는다 */

    CloseAllProcessHandles(dwProcessId);


    /* 프로세스 디버깅을 멈춘다 */

    Status = DbgUiStopDebugging(Handle);

    NtClose(Handle);


    /* 프로세스 디버깅을 멈추는 것이 실패했는지 검사한다 */
```

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

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

정말로 이보다 더 이상 간단할 수는 없을 겁니다. 우선 PID 를 핸들로 변환합니다. 그러고나서 DbgUi 함수(DbgUiStopDebugging)를 호출합니다. 프로세스에서 DbgUi 의 연결을 끊기 위해서 PID 를 통해 얻은 프로세스의 핸들이 이용됩니다.

함수가 여기 하나 더 있습니다. CloseAllProcessHandles 인데, 이 함수는 DbgUi 윗 단의 Win32 하이레벨 디버깅 함수 중의 하나입니다. 이 루틴은 먼저 봤던 RemoveHandles 와 매우 흡사합니다. 그러나, 오직 프로세스 ID 만을 다룹니다. 그래서 코드가 더 간단하죠.

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

    /* 모든 스레드 정보에 대하여 반복한다 */
    ThreadData = DbgSsGetThreadData();
    while (ThreadData)
    {
```

```
        /* 프로세스 ID 가 일치한지 검사한다 */
        if (ThreadData->ProcessId == dwProcessId)
        {
            /* 스레드 핸들이 있는지 검사한다 */
            if (ThreadData->ThreadHandle)
            {
                /* 스레드 핸들을 닫는다 */
                CloseHandle(ThreadData->ThreadHandle);
            }

            /* 프로세스 핸들이 있는지 검사한다 */
            if (ThreadData->ProcessHandle)
            {
                /* 프로세스 핸들을 닫는다 */
                CloseHandle(ThreadData->ProcessHandle);
            }

            /* ThreadData 의 연결을 제거한다 */
            DbgSsSetThreadData(ThreadData->Next);

            /* 메모리 할당을 해제한다  */
            RtlFreeHeap(RtlGetProcessHeap(), 0, ThreadData);

            /* 다음의 ThreadData 를 처리한다 */
            ThreadData = DbgSsGetThreadData();
            continue;
        }
```

```
        /* 다음의 ThreadData 를 처리한다 */
        ThreadData = ThreadData->Next;
    }
}
```

Win32 API 에 대한 분석이 끝났습니다! 배운 것을 정리해 봅시다.

- 실제 디버깅 기능은 Dbgk 라고 불리는 커널 내부의 모듈에 존재합니다.
- Dbgk 는 DbgUi 네이티브 API 인터페이스를 통해 접근할 수 있습니다. DbgUi 네이티브 API 인터페이스는 NT 시스템 라이브러리인 ntdll 내부에 있습니다.
- Dbgk 는 NT 객체를 통해 디버깅 기능을 수행합니다. 이 객체는 디버그 객체라고 불리고, 특정 플래그를 수정하기 위해 NtSetInformation API 를 제공합니다.
- 디버그 객체는 DbgUiGetThreadObject 로 반환되는 스레드에 연결되어 있습니다. 그러나, 이 객체가 저장된 위치는 여전히 겉으로 보이지 않습니다.
- 프로세스가 디버깅 중인지를 검사하는 것은 NtQueryInformationProcess 와 DebugPort 정보 클래스로 알 수 있습니다. 이것은 루트킷이 없으면 속일 수가 없습니다.
- Dbgk 가 디버그 이벤트 동안 특정 핸들을 열기 때문에, Win32 는 관련된 ID 와 핸들을 필요로 합니다. 그리고 DBGSS_THREAD_DATA 라고 불리는 구조체의 순환 배열(리스트)을 이용합니다. DBGSS_THREAD_DATA 는 TEB 의 DbgSsReserved[0] 멤버에 저장되어 있습니다.

2. Native Debugging

이제 디버깅의 네이티브적인 측면, 그리고 ntdll.dll 내부의 래퍼 계층이 커널과 어떻게 소통을 하는지에 대하여 볼 차례입니다. DbgUi 계층을 두는 것은 NT 의 설계대로 NT 커널과 Win32 를 더 잘 구분지을 수 있는 이점이 있습니다. NTDLL 과 NTOSKRNL 은 함께 내장되어 있습니다. 그래서 보통 이것들은 좀 복잡하게 얽혀있습니다. NTDLL 과 NTOSKRNL 은 같은 구조체를 공유합니다. 같은 시스템 함수 ID 등을 가져야 하죠. 완벽한 설계라면, NT 커널은 Win32 에 대하여 아무 것도 알아서는 안 됩니다.

게다가, DbgUi 는 네이티브 애플리케이션 내부에서 디버깅 기능을 사용할 수 있게 해주거나, 완벽한 기능을 갖춘 네이티브 모드 디버거를 작성할 수 있도록 해줍니다. DbgUi 가 없다면, Nt*DebugObject API 를 직접 호출해야 합니다. 그리고 몇몇의 경우에서는 아주 많은 전후 처리가 필요합니다. DbgUi 는 이러한 작업을 간단한 함수 호출로 단순화시킵니다. 그리고 깔끔한 인터페이스를 제공하지요. 커널이 내부적으로 변경되더라도, DbgUi 는 아마 그대로 유지될 것입니다. 단지 내부 코드만 수정될 뿐이죠.

프로세스에 디버그 객체를 생성하거나 연관시키는 함수에 대해서 탐구를 시작하겠습니다. 디버그 객체를 생성하고 실제로 프로세스에 연결하는 것은 Win32 와 분명히 다릅니다.

```
NTSTATUS

NTAPI

DbgUiConnectToDbg(VOID)

{

    OBJECT_ATTRIBUTES ObjectAttributes;


    /* 디버그 객체에 중복으로 연결되는 것을 금지한다 */

    if (NtCurrentTeb()->DbgSsReserved[1]) return STATUS_SUCCESS;


    /* 속성들을 설정한다 */

    InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, 0);


    /* 디버그 객체를 생성한다 */

    return ZwCreateDebugObject(&NtCurrentTeb()->DbgSsReserved[1],
```



```
                                DEBUG_OBJECT_ALL_ACCESS,
                                &ObjectAttributes,
                                TRUE);
}
```

위에서 볼 수 있듯이, 아주 간단한 코드입니다. 하지만, 두 가지를 보고 넘어가야 합니다. 먼저, 스레드는 그 스레드와 연관된 오직 하나의 디버그 객체만을 가질 수 있습니다. 그리고 두 번째로, 이 객체에 대한 핸들은 TEB 의 DbgSsReserved 배열에 저장됩니다. Win32 계층에서 얘기했듯이, DbgSsReserved 배열의 첫 번째 인덱스는 스레드 정보가 저장되는 곳입니다. 그리고 두 번째 인덱스가 핸들이 저장되는 곳입니다.

이제 프로세스에 디버거를 어떻게 연결하고, 어떻게 연결을 끊는지 보시죠.

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

    /* 커널에게 디버깅의 시작을 알린다 */
    Status = NtDebugActiveProcess(Process, NtCurrentTeb()->DbgSsReserved[1]);
    if (NT_SUCCESS(Status))
    {
        /* 여기서 프로세스에 디버거를 연결한다 */
        Status = DbgUiIssueRemoteBreakin(Process);
        if (!NT_SUCCESS(Status))
        {
            /* 디버거를 연결할 수 없으면, 디버깅을 취소한다 */
            DbgUiStopDebugging(Process);
```



```
        }
    }

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

```
NTSTATUS
NTAPI
DbgUiStopDebugging(IN HANDLE Process)
{
    /* 커널에게 디버그 객체를 제거해줄 것을 요청한다 */
    return NtRemoveProcessDebug(Process, NtCurrentTeb()->DbgSsReserved[1]);
}
```

이번에도 아주 간단한 코드입니다. 그러나, 원격 프로세스 내부에서 디버거를 프로세스에 연결하는 작업을 커널이 직접 하는 게 아니라는 것을 알 수 있습니다. 이 작업은 네이티브 계층에서 이루어집니다. DbgUiIssueRemoteBreakin API 는 Win32 가 DebugBreakProcess API 를 호출할 때도 호출됩니다. 그러므로 DbgUiIssueRemoteBreakin 를 보겠습니다.

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

    /* 디버거를 연결하는 작업을 할 스레드를 생성한다 */
```



```
    Status = RtlCreateUserThread(Process,
                                 NULL,
                                 FALSE,
                                 0,
                                 0,
                                 PAGE_SIZE,
                                 (PVOID)DbgUiRemoteBreakin,
                                 NULL,
                                 &hThread,
                                 &ClientId);

    /* 작업이 성공하면 핸들을 닫는다 */
    if(NT_SUCCESS(Status)) NtClose(hThread);

    /* 상태 정보를 반환한다 */
    return Status;
}
```

프로세스 내에 원격 스레드를 생성하는 게 전부입니다. 그러고나서 호출자에게 반환합니다. 저 원격 스레드는 어떤 마법을 부리는 걸까요? 보시죠.

```
VOID
NTAPI
DbgUiRemoteBreakin(VOID)
{
    /* 디버거가 유효한지 확인한다. 유효하다면, 디버거를 붙인다 */
    if (NtCurrentPeb()->BeingDebugged) DbgBreakPoint();
```

```
    /* 스레드를 빠져나간다 */
    RtlExitUserThread(STATUS_SUCCESS);
}
```

특별한 것은 없습니다. 생성된 원격 스레드는 프로세스가 정말로 디버깅 중인지를 확인하고 디버거를 프로세스에 연결합니다. 그리고, 이 API 는 익스포트되어 있기 때문에, 디버거를 연결하기 위해서 해당 프로세스에서 지역적으로 호출할 수 있습니다(그러나 스레드가 종료될 거라는 걸 주의해야합니다). Win32 디버깅 코드에서, 실질적인 디버그 핸들은 전혀 쓰이지 않고 항상 DbgUi 를 통해 핸들이 호출되는 것을 보았었습니다. 그때 NtSetInformationDebugObject 시스템 함수가 호출됐었습니다. 스레드와 관련된 디버그 객체를 실제로 얻기 전에, 특별한 DbgUi API 가 호출됩니다. 이 API 는 다른 API 와 짝을 이룹니다. 둘 모두 보시죠.

```
HANDLE
NTAPI
DbgUiGetThreadDebugObject(VOID)
{
    /* TEB 에서 디버그 객체의 핸들을 반환한다 */
    return NtCurrentTeb()->DbgSsReserved[1];
}
```

```
VOID
NTAPI
DbgUiSetThreadDebugObject(HANDLE DebugObject)
{
    /* 디버그 객체의 핸들을 TEB 에 설정한다 */
    NtCurrentTeb()->DbgSsReserved[1] = DebugObject;
}
```

객체 지향 프로그래밍에 익숙하다면, 접근자/변경자 메소드와 유사하게 보일 것입니다. 비록 Win32 가 디버그 객체의 핸들에 완전히 접근할 수 있고 그 자체를 쉽게 간단히 읽을 수 있지만, NT 개발자들은 클

래스와 유사한 DbgUi 를 만들기로 결정했습니다. 그리고 public 메소드들을 통해서 핸들을 확인하도록 했습니다. 이 설계는 필요에 따라 어디에서든지 디버그 핸들이 저장될 수 있게 해줍니다. 그리고 오직 이 두 API 만이 Win32 의 여러 DLL 들 대신에 핸들을 변경시킬 수 있습니다.

이제 Win32 API 가 감싸던 대기(wait) 함수와 진행(continue) 함수를 보겠습니다.

```
NTSTATUS

NTAPI

DbgUiContinue(IN PCLIENT_ID ClientId,

              IN NTSTATUS ContinueStatus)

{

    /* 디버깅이 계속 될거라고 커널 객체에게 알린다 */

    return ZwDebugContinue(NtCurrentTeb()->DbgSsReserved[1],

                           ClientId,

                           ContinueStatus);

}
```

```
NTSTATUS
NTAPI

DbgUiWaitStateChange(OUT PDBGUI_WAIT_STATE_CHANGE DbgUiWaitStateCange,

                     IN PLARGE_INTEGER TimeOut OPTIONAL)

{

    /* 커널에게 기다리라고 알린다 */

    return NtWaitForDebugEvent(NtCurrentTeb()->DbgSsReserved[1],

                               TRUE,

                               TimeOut,

                               DbgUiWaitStateCange);

}
```

놀랍지 않겠지만, 이 함수들도 DbgUi 의 래퍼 함수들입니다. 하지만, 여기서 흥미로운 것을 알 수 있습니다. DbgUi 는 디버그 이벤트를 위해 DBGUI_WAIT_STATE_CHANGE 라는 완전히 색다른 구조체를 사용합니다. 변환 작업을 하는 API 하나가 있지만, 우선 DBGUI_WAIT_STATE_CHANGE 구조체에 대해 보겠습니다.

```
//
// 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;
```

필드들이 매우 알기 쉽게 구성되어 있습니다. DBG_STATE 열거형을 보시죠.

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

Win32 의 DEBUG_EVENT 구조체와 디버그 이벤트 종류들을 살펴 보면, 약간의 차이점이 보일 것입니다. 우선, 다른 예외들과 브레이크포인트/싱글스텝 예외들이 다르게 처리됩니다. Win32 에서는 두 가지만 구분이 됩니다. 예외에 대해서는 RipInfo, 디버그 이벤트에 대해서는 Exception 입니다. 발생된 예외가 브레이크포인트인지 싱글스텝인지는 코드를 통해 알 수 있지만, 이 정보는 네이티브 구조체 안에 직접 있

습니다. OUTPUT_DEBUG_STRING 이벤트가 빠져있는 게 보일 것입니다. 디버그 문자열 정보가 예외로서 전송되기 때문에 후처리가 필요한 것이 DbgUi 의 단점입니다. Win32 가 지원하지 않는 상태가 두 개 더 있습니다. 그것은 유휴(Idle) 상태와 응답 대기(Reply Pending) 상태입니다. 이것들은 디버거의 관점에서 많은 정보를 제공하지 않고, 무시됩니다.

이제 StateInfo 열거형 내부의 실질적인 구조체들을 보겠습니다.

```
//
// 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;
```

DEBUG_EVENT 구조체와 조금 다른 것을 볼 수 있습니다. 우선, DBGKM_CREATE_PROCESS 에서 프로세

스의 이름(lpImageName)을 가리키고 있지 않습니다. 이 필드가 선택적이고 Win32 에서 쓰이지 않는 이유를 MSDN 에서 설명하고 있습니다. 스레드 구조체에서 TEB 에 대한 포인터(lpThreadLocalBase)가 없는 것도 볼 수 있습니다. 마지막으로, 새로운 프로세스와는 다르게, Win32 에서는 새로 불러와진 DLL 의 이름(lpImageName)을 보여주지만 이것도 DBGKM_LOAD_DLL 구조체에서는 보이지 않습니다. 이러한 변화들이 어떻게 처리되는지를 곧이어 살펴보겠습니다.

추가된 정보로는 "SubsystemKey" 필드가 있습니다. NT 는 여러 서브 시스템들을 지원하도록 설계되었기 때문에, 이 필드는 새로운 스레드나 프로세스가 어떤 서브 시스템에서 생성되었는지를 식별하는데 결정적인 요소입니다. Windows 2003 SP1 는 추가적으로 POSIX 애플리케이션 디버깅을 지원합니다. 그리고 POSIX 디버그 API 를 보지는 못 했지만, DbgUi 코드를 이용할 거라고 확신합니다. SubsystemKey 필드는 POSIX 라이브러리를 통해 다르게 쓰일 것입니다(Win32 가 그것을 무시하듯이요).

지금까지 DEBUG_EVENT 구조체와 DBGUI_WAIT_STATE_CHANGE 구조체의 차이점을 봤습니다. 마지막으로 볼 API 는 DbgUiConvertStateChangeStructure 입니다. DbgUiConvertStateChangeStructure 는 위에서 살펴본 변경된 것들을 변환합니다.

```
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;


    /* DebugEvent 에 프로세스 ID 와 스레드 ID 를 채워넣는다 */

    DebugEvent->dwProcessId = (DWORD)WaitStateChange->

                                     AppClientId.UniqueProcess;

    DebugEvent->dwThreadId = (DWORD)WaitStateChange->AppClientId.UniqueThread;
```



```
/* 이벤트의 종류를 검사한다 */
switch (WaitStateChange->NewState)
{
    /* 새로운 스레드일 때 예외 처리 */
    case DbgCreateThreadStateChange:

        /* Win32 의 디버그 코드를 설정한다 */
        DebugEvent->dwDebugEventCode = CREATE_THREAD_DEBUG_EVENT;

        /* 데이터를 복사한다 */
        DebugEvent->u.CreateThread.hThread =
            WaitStateChange->StateInfo.CreateThread.HandleToThread;
        DebugEvent->u.CreateThread.lpStartAddress =
            WaitStateChange->StateInfo.CreateThread.NewThread.StartAddress;

        /* TEB 를 구한다 */
        Status = NtQueryInformationThread(WaitStateChange->StateInfo.
                                          CreateThread.HandleToThread,
                                          ThreadBasicInformation,
                                          &ThreadBasicInfo,
                                          sizeof(ThreadBasicInfo),
                                          NULL);
        if (!NT_SUCCESS(Status))
        {
            /* TEB 의 주소를 구하지 못했을 때 */
            DebugEvent->u.CreateThread.lpThreadLocalBase = NULL;
        }
```

```
        else
        {
            /* TEB 의 주소를 기록한다 */
            DebugEvent->u.CreateThread.lpThreadLocalBase =
                ThreadBasicInfo.TebBaseAddress;
        }
        break;

    /* 새로운 프로세스일 때 예외 처리 */
    case DbgCreateProcessStateChange:

        /* Win32 의 디버그 코드를 설정한다 */
        DebugEvent->dwDebugEventCode = CREATE_PROCESS_DEBUG_EVENT;

        /* 데이터를 복사한다 */
        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;

/* TEB 를 구한다 */
Status = NtQueryInformationThread(WaitStateChange->StateInfo.
                                  CreateProcessInfo.HandleToThread,
                                  ThreadBasicInformation,
                                  &ThreadBasicInfo,
                                  sizeof(ThreadBasicInfo),
                                  NULL);
if (!NT_SUCCESS(Status))
{
    /* TEB 의 주소를 구하지 못했을 때 */
    DebugEvent->u.CreateThread.lpThreadLocalBase = NULL;
}
else
{
    /* TEB 의 주소를 기록한다 */
    DebugEvent->u.CreateThread.lpThreadLocalBase =
        ThreadBasicInfo.TebBaseAddress;
}

/* 이미지 이름을 지운다 */
DebugEvent->u.CreateProcessInfo.lpImageName = NULL;
```

```
            DebugEvent->u.CreateProcessInfo.fUnicode = TRUE;
            break;

        /* 스레드가 종료될 때 예외 처리 */
        case DbgExitThreadStateChange:

            /* Win32 의 디버그 코드를 설정하고 빠져나간다 */
            DebugEvent->dwDebugEventCode = EXIT_THREAD_DEBUG_EVENT;
            DebugEvent->u.ExitThread.dwExitCode =
                WaitStateChange->StateInfo.ExitThread.ExitStatus;
            break;

        /* 프로세스가 종료될 때 예외 처리 */
        case DbgExitProcessStateChange:

            /* Win32 의 디버그 코드를 설정하고 빠져나간다 */
            DebugEvent->dwDebugEventCode = EXIT_PROCESS_DEBUG_EVENT;
            DebugEvent->u.ExitProcess.dwExitCode =
                WaitStateChange->StateInfo.ExitProcess.ExitStatus;
            break;

        /* 나머지 예외 처리 */
        case DbgExceptionStateChange:
        case DbgBreakpointStateChange:
        case DbgSingleStepStateChange:

            /* 디버그 문자열 출력인지 검사한다 */
            if (WaitStateChange->StateInfo.Exception.ExceptionRecord.
```

```
        ExceptionCode == DBG_PRINTEXCEPTION_C)
    {
        /* Win32 의 디버그 코드를 설정한다 */
        DebugEvent->dwDebugEventCode = OUTPUT_DEBUG_STRING_EVENT;

        /* 디버그 문자열 정보를 복사한다 */
        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)
    {
        /* Win32 의 디버그 코드를 설정한다 */
        DebugEvent->dwDebugEventCode = RIP_EVENT;

        /* 예외 정보를 설정한다 */
        DebugEvent->u.RipInfo.dwType =
            WaitStateChange->StateInfo.Exception.ExceptionRecord.
            ExceptionInformation[1];
        DebugEvent->u.RipInfo.dwError =
            WaitStateChange->StateInfo.Exception.ExceptionRecord.
            ExceptionInformation[0];
```

```
        }
        else
        {
            /* 디버그 이벤트에 대해서 정보를 복사한다 */
            DebugEvent->dwDebugEventCode = EXCEPTION_DEBUG_EVENT;
            DebugEvent->u.Exception.ExceptionRecord =
                WaitStateChange->StateInfo.Exception.ExceptionRecord;
            DebugEvent->u.Exception.dwFirstChance =
                WaitStateChange->StateInfo.Exception.FirstChance;
        }
        break;

    /* DLL 이 불러와질 때 예외처리 */
    case DbgLoadDllStateChange :

        /* Win32 의 디버그 코드를 설정한다 */
        DebugEvent->dwDebugEventCode = LOAD_DLL_DEBUG_EVENT;

        /* 나머지 정보를 복사한다 */
        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;
```



```
/* 스레드를 연다 */
InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, NULL);
Status = NtOpenThread(&ThreadHandle,
                      THREAD_QUERY_INFORMATION,
                      &ObjectAttributes,
                      &WaitStateChange->AppClientId);
if (NT_SUCCESS(Status))
{
    /* 스레드 정보를 구한다 */
    Status = NtQueryInformationThread(ThreadHandle,
                                      ThreadBasicInformation,
                                      &ThreadBasicInfo,
                                      sizeof(ThreadBasicInfo),
                                      NULL);
    NtClose(ThreadHandle);
}

/* 스레드 정보가 구해졌는지 검사한다 */
if (NT_SUCCESS(Status))
{
    /* TIB 의 이미지 이름을 저장한다 */
    DebugEvent->u.LoadDll.lpImageName =
        &((PTEB)ThreadBasicInfo.TebBaseAddress)->
        Tib.ArbitraryUserPointer;
}
else
{
    /* 이미지 이름이 없을 경우 */
```

```
                DebugEvent->u.LoadDll.lpImageName = NULL;
            }

            /* 유니코드 플래그를 설정한다 */
            DebugEvent->u.LoadDll.fUnicode = TRUE;
            break;

        /* DLL 을 내려놓을 때 예외 처리 */
        case DbgUnloadDllStateChange:

            /* Win32 의 디버그 코드와 DLL 의 BaseAddress 를 설정한다 */
            DebugEvent->dwDebugEventCode = UNLOAD_DLL_DEBUG_EVENT;
            DebugEvent->u.UnloadDll.lpBaseOfDll =
                WaitStateChange->StateInfo.UnloadDll.BaseAddress;
            break;

        /* 예외 발생이 아닐 때 */
        default: return STATUS_UNSUCCESSFUL;
    }

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

변환되는 내용을 살펴보겠습니다. 먼저, TEB 포인터가 빠지고 그 대신에 ThreadBasicInformation 변수를 인자로 하여서 NtQueryInformationThread 가 호출됩니다. NtQueryInformationThread API 는 다른 여러 TEB 중에서 Win32 구조체에 저장된 특정 TEB 의 포인터를 반환합니다. 디버그 문자열에 대해서는, DbgUiConvertStateChangeStructure API 가 예외 코드를 분석해서 DBG_PRINTEXCEPTION_C 를 찾습니다. DBG_PRINTEXCEPTION_C 는 특정 예외 기록을 가지고 있어서, 분석되어서 디버그 문자열 출력으로 변환됩니다.

여태까지는 그런대로 잘 됐지만, 아마도 가장 힘겨운 분석은 DLL 이 로딩되는 부분일 것니다. 로드된 DLL 은 커널 메모리의 EPROCESS 나 ETHREAD 같은 구조체를 가지고 있지 않지만 ntdll 의 Ldr private 구조체에 위치합니다. 이 때문에 DLL 을 식별하는 유일한 수단이 메모리 매핑 파일의 메모리에 위치한 섹션 객체입니다. 실행 가능한 메모리 매핑 파일의 섹션을 생성하도록 커널이 요청 받으면, 커널은 임의 사용자 포인터(ArbitraryUserPointer)라는 TEB(또는 TIB)의 필드에 있는 파일의 이름을 저장합니다.

DbgUiConvertStateChangeStructure API 는 ArbitraryUserPointer 에 저장되어 있는 파일의 이름을 DEBUG_EVENT 의 이미지 이름 주소(lpImageName)멤버의 포인터로 설정합니다. 이 매커니즘은 첫 번째 NT 빌드 이후로 모두 적용되어 왔습니다. 그리고 비스타에서도 똑같은 걸로 알고 있습니다. 조금 어려울 수 있었나요?

다시 한 번, 이 주제에 대하여 마무리 지을 때가 되었습니다. ntdll 에서 디버그 객체를 처리하는 것에 대해 얘기할 게 이제 없습니다. Windows 디버깅 인터널 시리즈의 이번 파트에서 얘기했던 것들을 정리해 보겠습니다.

- DbgUi 는 커널과 Win32 혹은 커널과 다른 서브 시스템 사이의 별도의 계층을 제공합니다. DbgUi 는 완전히 독립적인 클래스이고, 심지어 핸들 자체를 노출시키는 대신 접근자/변경자 메소드를 가집니다.
- 한 스레드의 디버그 객체에 대한 핸들은 TEB 의 DbgSsReserved 배열 두 번째 필드에 저장됩니다.
- DbgUi 는 하나의 스레드가 단 하나의 디버그 객체만을 가지도록 하지만, 네이티브 시스템 함수를 사용하여서 원하는대로 디버그 객체를 가질 수 있습니다.
- 대개의 DbgUi API 는 NtXxxDebugObject 시스템 함수에 대한 단순한 래퍼입니다. 그리고 전달을 위해 TEB 핸들을 사용합니다.
- DbgUi 가 프로세스에 디버거를 연결하는 일을 합니다. 커널이 하지 않습니다.
- DbgUi 는 디버그 이벤트를 위해 커널이 이해하는 내부 구조체를 사용합니다. 몇 가지 방법으로, 이 구조체는 몇몇 이벤트에 대해 더 많은 정보를 제공합니다(서브 시스템 같은 것과 싱글스텝 예외인지 브레이크포인트 예외인지 같은 것), 하지만 한편, 어떤 정보는 제공되지 않습니다(해당 스레드의 TEB 의 포인터나 별도의 디버그 문자열 구조체 같은 것).
- TIB(TEB 에 포함된)의 ArbitraryPointer 멤버에는 로드된 DLL 의 이름이 디버그 이벤트동안 저장됩니다.

3. Kernel User-Mode Debugging Support

커널모드 내부에 마지막 퍼즐 조각이 남아 있습니다. 그 조각은 디버깅이 제대로 될 수 있도록 우리가 지금까지 보아온 이벤트와 구조체들을 제공합니다. Dbgk 는 KD(Kernel Debugger)에 의존하지 않으면서 Windows XP 부터 디버깅을 처리하기 위해 시스템 함수들을 가지는 별도의 객체입니다. Windows 의 이전 버전들은 Dbgk 가 없었습니다. 대신에 정적인 데이터 구조체에 의존하였습니다. 그 구조체가 메모리 내에서 분석되면 구조체는 Windows 의 로컬 프로시저 호출(LPC) 매커니즘을 통해 다양한 메시지를 전송하는데 사용되었습니다.

이런 시스템 함수들과 디버그 객체가 있어서 커널모드 드라이버가 유저모드 디버깅을 할 수 있습니다. NT 설계의 목적이 이게 아니었겠지만, 이 기능은 몇 가지 관심을 가질만 합니다 실제로 Nt* 함수는 익스포트되어 있진 않지만, 시스템 함수 ID 를 아는 드라이버를 통해 접근할 수 있습니다. 심지어 시스템 함수 번호가 OS 버전마다 바뀌더라도, 드라이버 내의 테이블을 유지하는 것이 상대적으로 쉽습니다. 이런 드라이버에 TDI 인터페이스를 추가하여서, 고속의 원격 디버거 드라이버를 개발할 수 있게 되었습니다. 원격 디버깅 드라이버는 유저모드 객체를 가지지 않고, 모든 단일 프로세스에 대하여 원격으로 디버깅을 제공합니다.

유저모드 디버깅의 내부 구현 중에서 살펴 봐야 할 첫 번째 객체는 DEBUG_OBJECT 입니다.

```
//
// Debug Object
//
typedef struct _DEBUG_OBJECT
{
    KEVENT EventsPresent;
    FAST_MUTEX Mutex;
    LIST_ENTRY EventList;
    union
    {
        ULONG Flags;
        struct
        {
            UCHAR DebuggerInactive:1;
            UCHAR KillProcessOnExit:1;
        };
    };
```



```
} DEBUG_OBJECT, *PDEBUG_OBJECT;
```

위에서 볼 수 있듯이, DEBUG_OBJECT 객체는 그 자체가 유저모드의 WaitForDebugEvent 에서 쓰이는 이벤트에 대한 아주 작은 래퍼 객체입니다. 여기에는 실제로 디버그 이벤트의 목록(EventList), 잠금장치(Mutex), 그리고 디버거가 연결되어 있는지 아닌지(DebuggerInactive), 디버거가 떨어질 때 프로세스가 종료될지 아닐지(KillProcessOnExit) 등의 디버깅 세션과 관련있는 특정 플래그들이 있습니다. 그리고 우리가 더 관심을 가져야 할 구조체는 DEBUG_EVENT(Win32 의 DEBUG_EVENT 와 다릅니다)입니다.

```
//
// Debug Event
//
typedef struct _DEBUG_EVENT
{
    LIST_ENTRY EventList;
    KEVENT ContinueEvent;
    CLIENT_ID ClientId;
    PEPROCESS Process;
    PETHREAD Thread;
    NTSTATUS Status;
    ULONG Flags;
    PETHREAD BackoutThread;
    DBGKM_MSG ApiMsg;
} DEBUG_EVENT, *PDEBUG_EVENT;
```

디버깅 이벤트와 관련된 모든 데이터를 가지는 구조체가 바로 이 구조체입니다. 호출자가 유저모드에서 WaitForDebugEvent 를 호출하기 전에 많은 이벤트가 대기될 수 있으므로, 디버그 이벤트들은 DEBUG_OBJECT 와 함께 연결되어야 합니다. 그 연결고리가 이벤트 목록(EventList)입니다.

다른 멤버 중에서 클라이언트 ID(ClientId)는 이벤트를 발생시킨 PID 와 TID 를 가지고 있습니다. 뿐만 아니라, 커널모드에 있는 각각의 프로세스(Process)와 스레드(Thread) 객체들에 대한 포인터들도 있습니다. DEBUG_EVENT 구조체에 있는 이벤트(ContinueEvent)는 디버거 메시지에 대한 응답이 가능하다는 것을 커널에게 알리기 위해 사용됩니다. 이 응답은 보통 Win32 의 ContinueDebugEvent 함수에서 발생합니다. ContinueDebugEvent 는 이벤트를 신호로 알립니다.

디버그 이벤트의 마지막 구조체(ApiMsg)는 실제로 전송되는 API 메시지입니다. 이 메시지는 유저모드에서 참조될 데이터를 가지고 있는 DBGUI_WAIT_STATE_CHANGE 의 커널모드의 형태입니다. 맞습니다. 커

널에는 디버그 이벤트를 나타내는 수단이 또 있는 것이죠. 그리고 그것도 네이티브 디버깅 인터페이스가 이해할 수 있도록 변환되어야 합니다.

아래 구조체에서 보이듯이, 필드 대부분이 상수입니다. 그리고 커널은 여전히 내부적으로 DbgUi 구조체에서 보았던 DBGKM 구조체를 사용하고 있습니다. 하지만, DBG_STATE 상수를 사용하는 대신에, 커널은 API 메시지 번호라는 상수값을 사용합니다. 다음을 보시죠.

```
//
// Debug Message API Number
//
typedef enum _DBGKM_APINUMBER
{
    DbgKmExceptionApi = 0,
    DbgKmCreateThreadApi = 1,
    DbgKmCreateProcessApi = 2,
    DbgKmExitThreadApi = 3,
    DbgKmExitProcessApi = 4,
    DbgKmLoadDllApi = 5,
    DbgKmUnloadDllApi = 6,
    DbgKmErrorReportApi = 7,
    DbgKmMaxApiNumber = 8,
} DBGKM_APINUMBER;
```

이 API 번호들은 이름만으로 이해하기 쉽고 예전의 LPC 매커니즘에 대하여 여전히 호환성을 가지고 있습니다. 커널은 DbgUi 를 위해 API 번호들을 실질적인 디버그 상태로 변환합니다. 이제 디버그 메시지 구조체를 보겠습니다. 이 구조체는 이전 버전의 Windows 의 LPC 매커니즘의 메시지들과 일치합니다.

```
//
// LPC Debug Message
//
typedef struct _DBGKM_MSG
{
    PORT_MESSAGE h;
    DBGKM_APINUMBER ApiNumber;
    ULONG ReturnedStatus;
    union
```



```
    {
        DBGKM_EXCEPTION Exception;
        DBGKM_CREATE_THREAD CreateThread;
        DBGKM_CREATE_PROCESS CreateProcess;
        DBGKM_EXIT_THREAD ExitThread;
        DBGKM_EXIT_PROCESS ExitProcess;
        DBGKM_LOAD_DLL LoadDll;
        DBGKM_UNLOAD_DLL UnloadDll;
    };
} DBGKM_MSG, *PDBGKM_MSG;
```

참고로, DbgSs 를 보지 않을 것이므로, 구조체의 PORT_MESSAGE 부분은 무시하여도 됩니다(DbgSs 는 Windows XP 이전에서 DbgUi 를 감싸는 계층과 LPC 메시지들을 처리하던 객체입니다).

이제 우리는 커널 내부 구조체의 형태를 알고 있습니다. 지금부터 네이티브 API 함수 중 몇 가지를 보겠습니다. 이 네이티브 API 함수들은 이벤트나 세마포어 같은 다른 객체처럼 디버그 객체를 감싸고 있습니다.

첫 번째로 볼 시스템 함수는 파트 2의 DbgUI 에서 봤었던 NtCreateDebugObject 네이티브 API 입니다. NtCreateDebugObject 는 프로세스에 연결되어 대기할 디버그 객체의 핸들을 반환합니다. 코드는 정말 간단합니다.

```
NTSTATUS
NTAPI
NtCreateDebugObject(OUT PHANDLE DebugHandle,
                    IN ACCESS_MASK DesiredAccess,
                    IN POBJECT_ATTRIBUTES ObjectAttributes,
                    IN BOOLEAN KillProcessOnExit)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    PDEBUG_OBJECT DebugObject;
    HANDLE hDebug;
    NTSTATUS Status = STATUS_SUCCESS;
    PAGED_CODE();

    /* 유저모드에서 호출되었는지 검사한다 */
```



```
if (PreviousMode != KernelMode)
{
    /* 조사를 위해 SEH 로 진입한다 */
    _SEH_TRY
    {
        /* 핸들을 조사한다 */
        ProbeForWriteHandle(DebugHandle);
    }
    _SEH_HANDLE
    {
        /* 예외 코드를 구한다 */
        Status = _SEH_GetExceptionCode();
    } _SEH_END;
    if (!NT_SUCCESS(Status)) return Status;
}

/* 객체를 생성한다 */
Status = ObCreateObject(PreviousMode,
                        DbgkDebugObjectType,
                        ObjectAttributes,
                        PreviousMode,
                        NULL,
                        sizeof(DEBUG_OBJECT),
                        0,
                        0,
                        (PVOID*)&DebugObject);
if (NT_SUCCESS(Status))
{
    /* 디버그 객체의 뮤텍스를 초기화한다 */
    ExInitializeFastMutex(&DebugObject->Mutex);

    /* 상태 이벤트 리스트를 초기화한다 */
    InitializeListHead(&DebugObject->EventList);

    /* 디버그 객체의 대기(wait) 상태를 초기화한다 */
    KeInitializeEvent(&DebugObject->EventsPresent,
                      NotificationEvent,
                      FALSE);
```



```
        /* 플래그를 설정한다 */
        DebugObject->KillProcessOnExit = KillProcessOnExit;

        /* 객체를 삽입한다 */
        Status = ObInsertObject((PVOID)DebugObject,
                                NULL,
                                DesiredAccess,
                                0,
                                NULL,
                                &hDebug);
        if (NT_SUCCESS(Status))
        {
            _SEH_TRY
            {
                *DebugHandle = hDebug;
            }
            _SEH_HANDLE
            {
                Status = _SEH_GetExceptionCode();
            } _SEH_END;
        }
    }

    /* 상태 값을 반환한다 */
    DBGKTRACE(DBGK_OBJECT_DEBUG, "Handle: %p DebugObject: %p\n",
              hDebug, DebugObject);
    return Status;
}
```

유저모드에서 이 API 를 직접 쓸 때 흥미로운 것은 디버그 객체를 지정할 수 있다는 것입니다. 그래서 그 객체는 객체 디렉토리의 네임 스페이스에 삽입될 수 있습니다. 불행하게도, NtOpenDebugObject 같은 함수가 존재하지 않아서 객체의 이름을 조회할 수 없으나 내부에 저장될 수 있습니다. 대신 디버그 객체가 DebugObjects 와 같은 리스트에 삽입되어서 나중에 열거될 수 있습니다.

DbgUi 계층이 생략될 수 있다는 사실과 함께 이것은 디버그 객체 핸들이 TEB 안에 저장되지 않아도 됨을 말해줍니다. 그리고 디버거 제작자에게 동시에 여러 프로세스들을 디버깅하면서, 디버그 객체들 사

이를 원활하게 전환시키고, WaitForMultipleObjects 를 사용함으로써 사용자 WaitForDebugEvent 를 구현한 디버거를 제작할 수 있도록 해줍니다. WaitForMultipleObjects 는 여러 프로세스들로부터 디버깅 이벤트를 받을 수 있는 API 입니다.

프로세스들로부터 오는 메시지와 핸들을 처리하는 것이 조금 어려울 수 있지만, kernel32 가 TEB 에 스레드마다 데이터를 저장하는 것처럼 한다면, 프로세스마다 데이터를 추가적으로 저장할 수 있습니다. 최종 결과는 다른 디버거들이 아직 지원하지 못한 혁신적이고 강력한 디버거가 될 것입니다.

이제 디버거가 NtDebugActiveProcess 에 의해 실제로 프로세스에 연결되는 것을 보겠습니다.

```
NTSTATUS
NTAPI
NtDebugActiveProcess(IN HANDLE ProcessHandle,
                     IN HANDLE DebugHandle)
{
    PEPROCESS Process;
    PDEBUG_OBJECT DebugObject;
    KPROCESSOR_MODE PreviousMode = KeGetPreviousMode();
    PETHREAD LastThread;
    NTSTATUS Status;
    PAGED_CODE();
    DBGKTRACE(DBGK_PROCESS_DEBUG, "Process: %p Handle: %p\n",
              ProcessHandle, DebugHandle);

    /* 프로세스를 참조한다 */
    Status = ObReferenceObjectByHandle(ProcessHandle,
                                       PROCESS_SUSPEND_RESUME,
                                       PsProcessType,
                                       PreviousMode,
                                       (PVOID*)&Process,
                                       NULL);
    if (!NT_SUCCESS(Status)) return Status;

    /* 초기 시스템 프로세스에 대한 디버깅을 금지한다 */
    if (Process == PsInitialSystemProcess) return STATUS_ACCESS_DENIED;
```



```
/* 디버그 객체를 참조한다 */
Status = ObReferenceObjectByHandle(DebugHandle,
                                   DEBUG_OBJECT_ADD_REMOVE_PROCESS,
                                   DbgkDebugObjectType,
                                   PreviousMode,
                                   (PVOID*)&DebugObject,
                                   NULL);
if (!NT_SUCCESS(Status))
{
    /* 실패시, 프로세스 참조를 끝내고 빠져 나간다 */
    ObDereferenceObject(Process);
    return Status;
}

/* 프로세스의 런다운(Rundown) 보호를 취득한다 */
if (!ExAcquireRundownProtection(&Process->RundownProtect))
{
    /* 실패시, 프로세스와 디버그 객체의 참조를 끝내고 빠져 나간다 */
    ObDereferenceObject(Process);
    ObDereferenceObject(DebugObject);
    return STATUS_PROCESS_IS_TERMINATING;
}

/* 상태 변화가 없는 디버거에 대해서 가짜 생성 메시지를 전송한다 */
Status = DbgkpPostFakeProcessCreateMessages(Process,
                                            DebugObject,
                                            &LastThread);
Status = DbgkpSetProcessDebugObject(Process,
                                    DebugObject,
                                    Status,
                                    LastThread);

/* 런다운 보호를 해제한다 */
ExReleaseRundownProtection(&Process->RundownProtect);

/* 프로세스와 디버그 객체의 참조를 끝내고 상태 값을 반환한다 */
ObDereferenceObject(Process);
ObDereferenceObject(DebugObject);
```

```
    return Status;
}
```

이 API 도 간단합니다. 그리고 내부 루틴이 작업의 대부분을 수행합니다. 우선, 프로세스에 연결하면서 문제가 DLL 을 많이 로드하는 것 같이 프로세스가 이미 여러 스레드를 생성한 경우입니다. 시스템은 프로세스가 디버깅될 지를 예측할 수 없습니다. 그래서 디버그 이벤트들을 기다리지 않습니다. 대신에, Dbgk 모듈이 각각의 스레드와 모듈을 검사하고, 디버거에게 적합한 가짜 이벤트 메시지를 보냅니다. 예를 들면, 프로세스에 연결할 때, 대량의 DLL 로드 이벤트들이 생성되어서 디버거가 무슨 일이 일어나는지를 알 수 있습니다.

Dbgkp 함수의 내부로 가지 않고, DLL 로드 메시지가 전송되는 방법은 Peb->Ldr 을 통해 PEB 안의 로더 리스트(_PEB_LDR_DATA)를 순회하는 것입니다. 하드 코딩된 DLL 들은 최대 500 개입니다. 그래서 그 리스트는 무한으로 반복되지는 않습니다. LDR_DATA_TABLE_ENTRY 구조체에서 일치하는 DLL 이름을 찾는 대신에, MmGetFileNameForAddress 라는 내부 API 가 사용됩니다. 이 API 는 DLL 의 기본 주소를 구하기 위해 VAD(Virtual Address Descriptor)를 찾고, DLL 의 기본 주소와 관련있는 SECTION_OBJECT 를 구합니다. SECTION_OBJECT 에서 메모리 관리자는 FILE_OBJECT 를 찾을 수 있습니다. 그러고나서 DLL 의 전체 이름을 구하기 위해 ObQueryNameString 를 사용합니다. DLL 의 전체 이름은 유저모드가 받을 핸들을 구하는데 사용될 수 있습니다.

DBGKM_LOAD_DLL 구조체의 이름 포인터(NamePointer) 변수가 아직 채워지지 않은 것에 유의합니다. 그러나 이것은 쉽게 될 수 있습니다.

새로 생성된 스레드들을 순회하기 위해서, 모든 스레드를 순회하는 PsGetNextProcessThread API 가 사용됩니다. 첫 번째 스레드는 프로세스 생성 디버그 이벤트를 발생시키지만, 이후의 각 스레드는 스레드 생성 메시지를 발생시킵니다. 프로세스에 대한 이벤트 데이터는 EPROCESS 의 섹션 기본 주소(SectionBaseAddress)에서 얻어집니다. SectionBaseAddress 는 기본 주소의 포인터입니다. 스레드의 경우에는 ETHREAD 에 미리 저장되어 있는 시작 주소(StartAddress)만 반환됩니다.

마지막으로, DbgkpSetProcessDebugObject 는 프로세스의 객체에 대해서 복잡한 작업을 합니다. 첫 번째로, 초기 리스트가 DbgpPostFakeThreadMessages 에서 분석된 이후에 새로운 스레드들이 생성될 가능성이 있습니다. 따라서 이 루틴은 실제로 디버그 포트의 뮤텍스를 얻고나서 지나쳐버린 스레드들을 구하기 위해서 DbgpPostFakeThreadMessages 를 다시 호출합니다. 논리적으로, 같은 메시지가 중복으로 전송될 수 있지만, ETHREAD 플래그 중의 생성 SkipCreationMsg 를 DbgpPostFakeThreadMessages(정확히는, 내부에서 호출되는 DbgkpQueueMessage)가 메시지가 전송되기 전에 검사하여서 중복을 피합니다.

DbgkpSetProcessDebugObject 의 두 번째 작업은 이미 객체와 연관된 디버그 이벤트를 분석하는 것입니다. 이 작업은 가짜 메시지들을 모두 전송합니다. 그리고 모든 스레드에 대하여 경쟁 조건(Race Condition)이나 완전히 삽입되지 않은 스레드 등을 검사하는 런다운 보호(Rundown Protect)를 얻습니다. 마지막으로, PEB 가 수정되어서(DbgkpMarkProcessPeb 를 호출하여서) BeingDebugged 플래그가 활성화 됩니다. 일단 루틴이 수행되면, 디버그 객체는 대상 프로세스에 완전히 연결됩니다.

이제 보게 될 루틴은 NtRemoveProcessDebug 의 코드입니다. NtRemoveProcessDebug 는 활동중인 디버기 프로세스에서 디버거의 연결을 차단합니다.

```
NTSTATUS
NTAPI
NtRemoveProcessDebug(IN HANDLE ProcessHandle,
                     IN HANDLE DebugHandle)
{
    PEPROCESS Process;
    PDEBUG_OBJECT DebugObject;
    KPROCESSOR_MODE PreviousMode = KeGetPreviousMode();
    NTSTATUS Status;
    PAGED_CODE();
    DBGKTRACE(DBGK_PROCESS_DEBUG, "Process: %p Handle: %p\n",
              ProcessHandle, DebugHandle);

    /* 프로세스를 참조한다 */
    Status = ObReferenceObjectByHandle(ProcessHandle,
                                       PROCESS_SUSPEND_RESUME,
                                       PsProcessType,
                                       PreviousMode,
                                       (PVOID*)&Process,
                                       NULL);
    if (!NT_SUCCESS(Status)) return Status;

    /* 디버그 객체를 참조한다 */
    Status = ObReferenceObjectByHandle(DebugHandle,
                                       DEBUG_OBJECT_ADD_REMOVE_PROCESS,
                                       DbgkDebugObjectType,
                                       PreviousMode,
                                       (PVOID*)&DebugObject,
```

```
                                        NULL);
    if (!NT_SUCCESS(Status))
    {
        /* 실패시, 프로세스 참조를 끝내고 빠져 나간다 */
        ObDereferenceObject(Process);
        return Status;
    }

    /* 디버그 객체를 제거한다 */
    Status = DbgkClearProcessDebugObject(Process, DebugObject);

    /* 프로세스와 디버그 객체의 참조를 끝내고 상태 값을 반환한다 */
    ObDereferenceObject(Process);
    ObDereferenceObject(DebugObject);
    return Status;
}
```

작업의 요점은 DbgkClearProcessDebugObject 가 수행되는 부분입니다. 이 루틴은 모든 관련 디버그 이벤트들을 순회하면서 대기 중인 스레드를 깨웁니다. 그리고 그 스레드에게 STATS_DEBUGGER_INACTIVE 를 보고합니다. 역시 PEB 를 디버거가 없는 상태로 변경합니다.

대부분의 NT 객체들처럼, Dbgk 는 디버그 객체의 인터페이스를 제공하고, 디버그 객체의 내용을 조회하거나 설정합니다. 지금은 설정 루틴만 보겠습니다. 이 루틴은 디버거의 연결을 끊으면서 프로세스를 종료해도 될지 안 될지의 단일 플래그(KillProcessOnExit)를 지원합니다. NtSetInformationDebugObject 시스템 함수를 통한 이 방법은 Win32 의 DebugSetProcessKillOnExit API 로 매핑됩니다.

```
NTSTATUS
NTAPI
NtSetInformationDebugObject(IN HANDLE DebugHandle,
                            IN DEBUGOBJECTINFOCLASS DebugObjectInformationClass,
                            IN PVOID DebugInformation,
                            IN ULONG DebugInformationLength,
                            OUT PULONG ReturnLength OPTIONAL)
{
    PDEBUG_OBJECT DebugObject;
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
```



```
NTSTATUS Status = STATUS_SUCCESS;
PDEBUG_OBJECT_KILL_PROCESS_ON_EXIT_INFORMATION DebugInfo = DebugInformation;
PAGED_CODE();

/* 버퍼와 파라미터를 검사한다 */
Status = DefaultSetInfoBufferCheck(DebugObjectInformationClass,
                                   DbgkpDebugObjectInfoClass,
                                   sizeof(DbgkpDebugObjectInfoClass) /
                                   sizeof(DbgkpDebugObjectInfoClass[0]),
                                   DebugInformation,
                                   DebugInformationLength,
                                   PreviousMode);
/* 호출자가 길이 반환을 원했는지를 검사한다 */
if (ReturnLength)
{
    /* 조사를 위해 SEH 로 진입한다 */
    _SEH_TRY
    {
        /* 유저모드로 길이를 반환한다 */
        ProbeForWriteUlong(ReturnLength);
        *ReturnLength = sizeof(*DebugInfo);
    }
    _SEH_EXCEPT(_SEH_ExSystemExceptionFilter)
    {
        /* SEH 예외 코드를 구한다 */
        Status = _SEH_GetExceptionCode();
    }
    _SEH_END;
}
if (!NT_SUCCESS(Status)) return Status;

/* 객체를 참조한다 */
Status = ObReferenceObjectByHandle(DebugHandle,
                                   DEBUG_OBJECT_WAIT_STATE_CHANGE,
                                   DbgkDebugObjectType,
                                   PreviousMode,
                                   (PVOID*)&DebugObject,
                                   NULL);
```

```
    if (NT_SUCCESS(Status))
    {
        /* 디버그 객체의 뮤텍스를 구한다 */
        ExAcquireFastMutex(&DebugObject->Mutex);

        /* 적절한 플래그를 설정한다 */
        if (DebugInfo->KillProcessOnExit)
        {
            /* 프로세스 종료를 허락한다 */
            DebugObject->KillProcessOnExit = TRUE;
        }
        else
        {
            /* 못하게 한다 */
            DebugObject->KillProcessOnExit = FALSE;
        }

        /* 뮤텍스를 해제한다 */
        ExReleaseFastMutex(&DebugObject->Mutex);

        /* 객체를 해제한다 */
        ObDereferenceObject(DebugObject);
    }

    /* 상태 값을 반환한다 */
    return Status;
}
```

마지막으로, 디버그 객체에서 제공하는 마지막 기능은 양방향 채널로 구현된 대기(wait) 함수와 진행(continue) 함수입니다. 이 함수들 덕분에 디버거는 이벤트를 받을 수 있고, 대상의 상태나 내부 데이터를 수정하고, 다시 실행시킬 수도 있습니다. 이 함수들은 고유의 동기화 문제들 때문에 많이 뒤얽혀있습니다. 먼저, NtWaitForDebugEvent 를 보시죠.

```
NTSTATUS
NTAPI
NtWaitForDebugEvent(IN HANDLE DebugHandle,
                    IN BOOLEAN Alertable,
```



```
                    IN PLARGE_INTEGER Timeout OPTIONAL,
                    OUT PDBGUI_WAIT_STATE_CHANGE StateChange)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    LARGE_INTEGER SafeTimeOut;
    PEPROCESS Process;
    LARGE_INTEGER StartTime;
    PETHREAD Thread;
    BOOLEAN GotEvent;
    LARGE_INTEGER NewTime;
    PDEBUG_OBJECT DebugObject;
    DBGUI_WAIT_STATE_CHANGE WaitStateChange;
    NTSTATUS Status = STATUS_SUCCESS;
    PDEBUG_EVENT DebugEvent, DebugEvent2;
    PLIST_ENTRY ListHead, NextEntry, NextEntry2;
    PAGED_CODE();
    DBGKTRACE(DBGK_OBJECT_DEBUG, "Handle: %p\n", DebugHandle);

    /* WaitStateChange 구조체를 초기화한다 */
    RtlZeroMemory(&WaitStateChange, sizeof(WaitStateChange));

    /* SEH 안에서 조사를 보호한다 */
    _SEH_TRY
    {
        /* 시간이 초과되었는지 검사한다 */
        if (Timeout)
        {
            /* 유저모드에서 호출되었는지 검사한다 */
            if (PreviousMode != KernelMode)
            {
                /* 조사한다 */
                ProbeForReadLargeInteger(Timeout);
            }

            /* 복사본을 만든다 */
            SafeTimeOut = *Timeout;
            Timeout = &SafeTimeOut;
```

```
            /* 현재의 시각을 조회한다 */
            KeQuerySystemTime(&StartTime);
        }

        /* 유저모드에서 호출되었는지 검사한다 */
        if (PreviousMode != KernelMode)
        {
            /* StateChange 구조체를 조사한다 */
            ProbeForWrite(StateChange, sizeof(*StateChange), sizeof(ULONG));
        }
    }
    _SEH_HANDLE
    {
        /* SEH 예외 코드를 구한다 */
        Status = _SEH_GetExceptionCode();
    }
    _SEH_END;
    if (!NT_SUCCESS(Status)) return Status;

    /* 디버그 객체를 구한다 */
    Status = ObReferenceObjectByHandle(DebugHandle,
                                       DEBUG_OBJECT_WAIT_STATE_CHANGE,
                                       DbgkDebugObjectType,
                                       PreviousMode,
                                       (PVOID*)&DebugObject,
                                       NULL);
    if (!NT_SUCCESS(Status)) return Status;

    /* 프로세스와 스레드를 초기화한다 */
    Process = NULL;
    Thread = NULL;

    /* 계속 대기한다 */
    while (TRUE)
    {
        /* 주어진 디버그 객체에 대하여 대기한다 */
        Status = KeWaitForSingleObject(&DebugObject->EventsPresent,
                                       Executive,
```

```
                                PreviousMode,
                                Alertable,
                                Timeout);
    if (!NT_SUCCESS(Status) ||
        (Status == STATUS_TIMEOUT) ||
        (Status == STATUS_ALERTED) ||
        (Status == STATUS_USER_APC))
    {
        /* 대기 상태에서 빠져 나간다 */
        break;
    }

    /* 객체를 잠근다 */
    GotEvent = FALSE;
    ExAcquireFastMutex(&DebugObject->Mutex);

    /* 디버거가 연결되었는지 검사한다 */
    if (DebugObject->DebuggerInactive)
    {
        /* 연결되지 않을 경우 */
        Status = STATUS_DEBUGGER_INACTIVE;
    }
    else
    {
        /* 이벤트를 순회한다 */
        ListHead = &DebugObject->EventList;
        NextEntry =  ListHead->Flink;
        while (ListHead != NextEntry)
        {
            /* 디버그 이벤트를 가져온다 */
            DebugEvent = CONTAINING_RECORD(NextEntry,
                                           DEBUG_EVENT,
                                           EventList);
            DBGKTRACE(DBGK_PROCESS_DEBUG, "DebugEvent: %p Flags: %lx\n",
                      DebugEvent, DebugEvent->Flags);

            /* 플래그를 검사한다 */
            if (!(DebugEvent->Flags &
```

```
            (DEBUG_EVENT_FLAGS_USED | DEBUG_EVENT_FLAGS_INACTIVE)))
        {
            /* 이벤트를 가져왔을 경우 */
            GotEvent = TRUE;

            /* 리스트를 내부적으로 순회한다 */
            NextEntry2 = DebugObject->EventList.Flink;
            while (NextEntry2 != NextEntry)
            {
                /* 디버그 이벤트를 가져온다 */
                DebugEvent2 = CONTAINING_RECORD(NextEntry2,
                                                DEBUG_EVENT,
                                                EventList);

                /* 프로세스 ID 가 일치하는지 확인한다 */
                if (DebugEvent2->ClientId.UniqueProcess ==
                    DebugEvent->ClientId.UniqueProcess)
                {
                    /* 일치하면 순회를 그만둔다 */
                    DebugEvent->Flags |= DEBUG_EVENT_FLAGS_USED;
                    DebugEvent->BackoutThread = NULL;
                    GotEvent = FALSE;
                    break;
                }

                /* 다음 이벤트를 처리한다 */
                NextEntry2 = NextEntry2->Flink;
            }

            /* 남은 이벤트가 있는지 확인한다 */
            if (GotEvent) break;
        }

        /* 다음 이벤트를 처리한다 */
        NextEntry = NextEntry->Flink;
    }

    /* 이벤트를 가져왔는지 검사한다 */
```

```
        if (GotEvent)
        {
            /* 프로세스와 스레드를 저장하고 참조한다 */
            Process = DebugEvent->Process;
            Thread = DebugEvent->Thread;
            ObReferenceObject(Process);
            ObReferenceObject(Thread);

            /* 유저모드 구조체(WaitStateChange)로 변환한다 */
            DbgkpConvertKernelToUserStateChange(&WaitStateChange,
                                                DebugEvent);

            /* 플래그를 설정한다 */
            DebugEvent->Flags |= DEBUG_EVENT_FLAGS_INACTIVE;
        }
        else
        {
            /* 이벤트 리스트를 정리한다 */
            KeClearEvent(&DebugObject->EventsPresent);
        }

        /* Set success */
        Status = STATUS_SUCCESS;
    }

    /* 뮤텍스를 해제한다 */
    ExReleaseFastMutex(&DebugObject->Mutex);
    if (!NT_SUCCESS(Status)) break;

    /* 이벤트를 가져왔는지 검사한다 */
    if (!GotEvent)
    {
        /* 다시 대기할 수 있는지 확인한다 */
        if (SafeTimeOut.QuadPart < 0)
        {
            /* 현재 시각을 구한다 */
            KeQuerySystemTime(&NewTime);
```

```
            /* 남은 시간을 계산한다 */
            SafeTimeOut.QuadPart += (NewTime.QuadPart - StartTime.QuadPart);
            StartTime = NewTime;

            /* 시간이 초과되었는지 검사한다 */
            if (SafeTimeOut.QuadPart > 0)
            {
                /* 시간 초과로 대기 상태에서 빠져 나온다 */
                Status = STATUS_TIMEOUT;
                break;
            }
        }
    }
    else
    {
        /* 핸들을 구하고 객체의 참조를 끝낸다 */
        DbgkpOpenHandles(&WaitStateChange, Process, Thread);
        ObDereferenceObject(Process);
        ObDereferenceObject(Thread);
        break;
    }
}

/* 객체의 참조를 끝낸다 */
ObDereferenceObject(DebugObject);

/* SEH 쓰기 보호 */
_SEH_TRY
{
    /* WaitStateChange 구조체를 반환한다 */
    RtlCopyMemory(StateChange,
                  &WaitStateChange,
                  sizeof(DBGUI_WAIT_STATE_CHANGE));
}
_SEH_EXCEPT(_SEH_ExSystemExceptionFilter)
{
    /* SEH 오류 코드를 구한다 */
    Status = _SEH_GetExceptionCode();
```

```
    }
    _SEH_END;

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

처음에는 디버그 객체의 EventsPresent 이벤트에 대하여 대기합니다. 대기 상태가 되면, 디버그 객체는 잠겨지고, NtWaitForDebugEvent 는 객체가 잠겨지기 전에 비활성화 상태(DebuggerInactive)가 아니었는지를 검사합니다. 이것이 확인되면, 현재의 디버그 이벤트를 분석하여서, 이미 처리된 디버그 이벤트인지(DEBUG_EVENT_FLAGS_USED)와 비활성화 상태가 되지 않았는지(DEBUG_EVENT_FLAGS_INACTIVE)를 확인합니다. 이런 플래그들이 검사되면, 리스트를 다시 분석하여서 동일한 프로세스에 다른 이벤트가 없는지를 검사합니다. 만약 어떠한 이벤트도 발견되지 않으면, 이벤트는 비활성화 상태가 되어서 아무것도 전송하지 않습니다. 이것은 같은 프로세스에서 이벤트가 중복되는 것을 막기 위해서입니다.

디버그 이벤트를 얻으면, 프로세스와 스레드 정보가 참조되어서 디버그 이벤트는 DbgUi 의 WaitStateChange 구조체로 변환됩니다. 그리고 이벤트가 처리되었다고 플래그가 설정됩니다. 디버그 객체의 잠금이 풀린 후, 디버그 이벤트가 있으면 DbgkOpenHandles 가 호출됩니다. DbgkOpenHandles 는 DbgUi 의 WaitStateChange 에서 요구되는 핸들들을 엽니다. 그리고 프로세스와 스레드의 핸들은 제거됩니다. 대기 상태가 끝나면, WaitStateChange 구조체가 호출자에게 다시 복사됩니다.

마지막 API 인 NtDebugContinue 는 디버그 이벤트 처리 후에 프로세스를 다시 실행시킵니다. 그리고 디버그 이벤트 전송을 제거하고 대상 프로세스를 깨웁니다. 코드는 다음과 같습니다.

```
NTSTATUS
NTAPI
NtDebugContinue(IN HANDLE DebugHandle,
                IN PCLIENT_ID AppClientId,
                IN NTSTATUS ContinueStatus)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    PDEBUG_OBJECT DebugObject;
    NTSTATUS Status = STATUS_SUCCESS;
    PDEBUG_EVENT DebugEvent = NULL, DebugEventToWake = NULL;
    PLIST_ENTRY ListHead, NextEntry;
    BOOLEAN NeedsWake = FALSE;
```



```
    CLIENT_ID ClientId;
    PAGED_CODE();
    DBGKTRACE(DBGK_OBJECT_DEBUG, "Handle: %p Status: %p\n",
              DebugHandle, ContinueStatus);

    /* 유저모드에서 호출되었는지 검사한다 */
    if (PreviousMode != KernelMode)
    {
        /* 조사를 위해 SEH 로 진입한다 */
        _SEH_TRY
        {
            /* 핸들을 조사한다 */
            ProbeForRead(AppClientId, sizeof(CLIENT_ID), sizeof(ULONG));
            ClientId = *AppClientId;
            AppClientId = &ClientId;
        }
        _SEH_HANDLE
        {
            /* SEF 예외 코드를 구한다 */
            Status = _SEH_GetExceptionCode();
        } _SEH_END;
        if (!NT_SUCCESS(Status)) return Status;
    }

    /* 상태 값이 유효한지 확인한다 */
    if ((ContinueStatus != DBG_CONTINUE) &&
        (ContinueStatus != DBG_EXCEPTION_HANDLED) &&
        (ContinueStatus != DBG_EXCEPTION_NOT_HANDLED) &&
        (ContinueStatus != DBG_TERMINATE_THREAD) &&
        (ContinueStatus != DBG_TERMINATE_PROCESS))
    {
        /* 유효하지 않은 상태 값일 경우 */
        Status = STATUS_INVALID_PARAMETER;
    }
    else
    {
        /* 디버그 객체를 가져온다 */
        Status = ObReferenceObjectByHandle(DebugHandle,
```

```
                                    DEBUG_OBJECT_WAIT_STATE_CHANGE,
                                    DbgkDebugObjectType,
                                    PreviousMode,
                                    (PVOID*)&DebugObject,
                                    NULL);
    if (NT_SUCCESS(Status))
    {
        /* 뮤텍스를 얻는다 */
        ExAcquireFastMutex(&DebugObject->Mutex);

        /* 이벤트 리스트를 순회한다 */
        ListHead = &DebugObject->EventList;
        NextEntry = ListHead->Flink;
        while (ListHead != NextEntry)
        {
            /* 현재의 디버그 이벤트를 가져온다 */
            DebugEvent = CONTAINING_RECORD(NextEntry,
                                           DEBUG_EVENT,
                                           EventList);

            /* 프로세스 ID 를 비교한다 */
            if (DebugEvent->ClientId.UniqueProcess ==
                AppClientId->UniqueProcess)
            {
                /* 중복 여부를 검사한다 */
                if (NeedsWake)
                {
                    /* 프로세스를 깨운 후에 리스트 순회를 빠져 나간다 */
                    DebugEvent->Flags &= ~DEBUG_EVENT_FLAGS_INACTIVE;
                    KeSetEvent(&DebugEvent->ContinueEvent,
                               IO_NO_INCREMENT,
                               FALSE);
                    break;
                }

                /* 스레드 ID 와 플래그를 비교한다 */
                if ((DebugEvent->ClientId.UniqueThread ==
                    AppClientId->UniqueThread) &&
```

```
                    (DebugEvent->Flags & DEBUG_EVENT_FLAGS_INACTIVE))
                {
                    /* 리스트에서 이벤트를 제거한다 */
                    RemoveEntryList(NextEntry);

                    /* 깨울 스레드를 기억한다 */
                    NeedsWake = TRUE;
                    DebugEventToWake = DebugEvent;
                }
            }

            /* 다음 이벤트를 처리한다 */
            NextEntry = NextEntry->Flink;
        }

        /* 뮤텍스를 해제한다 */
        ExReleaseFastMutex(&DebugObject->Mutex);

        /* 객체 참조를 끝낸다 */
        ObDereferenceObject(DebugObject);

        /* 대기할지 검사한다 */
        if (NeedsWake)
        {
            /* 진행(Continue) 상태를 설정한다 */
            DebugEvent->ApiMsg.ReturnedStatus = Status;
            DebugEvent->Status = STATUS_SUCCESS;

            /* 대상을 깨운다 */
            DbgkpWakeTarget(DebugEvent);
        }
        else
        {
            /* Fail */
            Status = STATUS_INVALID_PARAMETER;
        }
    }
}
```

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

먼저, 몇몇 인식 가능한 상태 코드에 대해서 진행(Continue) 상태인지를 검사합니다. 그러고나서, 각 디버그 이벤트를 순회합니다. 만약 프로세스와 스레드의 ID 가 일치하다면, 디버그 이벤트는 리스트에서 제거됩니다. 그리고 이벤트가 발견됐었다는 것이 기억됩니다. 동일 프로세스에 대해서 이벤트가 추가적으로 발견되면, 비활성화 플래그(DEBUG_EVENT_FLAGS_INACTIVE)가 제거됩니다. 이 플래그는 동일한 프로세스에서 중복된 이벤트들이 같이 전송되는 것을 막기 위해서 대기(wait) 루틴에서 다시 확인됩니다. 모든 디버그 이벤트들이 분석되면, 대상 프로세스가 깨어나게 됩니다. 원래 메시지가 가짜 스레드 생성 메시지였으며, 스레드를 다시 시작하고 런다운 보호를 해제합니다. 그러고나서 디버그 이벤트를 해제하든지, 누가 대기중인지(생성된 방법에 따라서)를 알려주든지 합니다.

Windows XP 와 이상의 버전에서의 유저모드 디버깅의 마지막 내부 구현까지 설명을 마쳤습니다. 이번 섹션은 커널모드에 초점을 맞추었습니다. 이번 섹션에서 배운 것을 정리해보겠습니다.

- Dbgk 는 디버깅 기능을 지원하는 코드를 처리하는 커널의 객체입니다.
- Dbgk 는 DEBUG_OBJECT 라는 NT 객체를 통해 노출되고 그것에 접근하기 위한 여러 시스템 함수들을 제공합니다.
- 유저모드 애플리케이션을 위한 커널모드 디버거를 만들 수 있습니다.
- DbgUi 계층을 생략하고 직접 시스템 함수를 사용함으로써 동시에 여러 애플리케이션을 디버깅할 수 있는 디버거를 만들 수 있습니다.
- 커널은 WaitStateChange 구조체를 가집니다. 이 구조체는 DEBUG_EVENT 구조체 안에 캡슐화되어 있습니다.
- 커널은 여전히 LPC 기반의 DbgSs 디버깅을 지원합니다.
- 커널이 이벤트 구조체에 있는 핸들들을 모두 열고, 유저모드가 그것들을 닫습니다.
- 커널은 동일한 프로세스에서 동시에 오직 하나의 이벤트만 전송되는 것처럼 작성되었습니다.
- 커널은 로드된 DLL 들의 목록을 구하기 위해서 PEB 로더 데이터를 분석해야 하고, 500번까지 반복되도록 하드 코딩된 제한이 있습니다.

이번 섹션이 디버깅 인터널 시리즈의 마지막입니다. 이번 분석을 통해서 즐거웠기를 바라고, 더 좋은, 더 유연한, 더 강력한 디버거를 작성할 수 있기를 바랍니다. 그래서 사용자와 개발자들에게 더 많은 능력을 제공할 수 있기를 바랍니다.

Appendix

A. DEBUG_EVENT

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO {
            EXCEPTION_RECORD       ExceptionRecord;
            DWORD                  dwFirstChance;
        } Exception;
        CREATE_THREAD_DEBUG_INFO {
            HANDLE                 hThread;
            LPVOID                 lpThreadLocalBase;
            LPTHREAD_START_ROUTINE lpStartAddress;
        } CreateThread;
        CREATE_PROCESS_DEBUG_INFO {
            HANDLE                 hFile;
            HANDLE                 hProcess;
            HANDLE                 hThread;
            LPVOID                 lpBaseOfImage;
            DWORD                  dwDebugInfoFileOffset;
            DWORD                  nDebugInfoSize;
            LPVOID                 lpThreadLocalBase;
            LPTHREAD_START_ROUTINE lpStartAddress;
            LPVOID                 lpImageName;
            WORD                   fUnicode;
        } CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO {
            DWORD                  dwExitCode;
        } ExitThread;
        EXIT_PROCESS_DEBUG_INFO {
            DWORD                  dwExitCode;
        } ExitProcess;
        LOAD_DLL_DEBUG_INFO {
```

```
            HANDLE                  hFile;
            LPVOID                  lpBaseOfDll;
            DWORD                   dwDebugInfoFileOffset;
            DWORD                   nDebugInfoSize;
            LPVOID                  lpImageName;
            WORD                    fUnicode;
        } LoadDll;
        UNLOAD_DLL_DEBUG_INFO {
            LPVOID                  lpBaseOfDll;
        } UnloadDll;
        OUTPUT_DEBUG_STRING_INFO {
            LPSTR                   lpDebugStringData;
            WORD                    fUnicode;
            WORD                    nDebugStringLength;
        } DebugString;
        RIP_INFO {
            DWORD                   dwError;
            DWORD                   dwType;
        } RipInfo;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

B. PEB (Process Environment Block)

```
nt!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar                // 프로세스가 현재 디버깅 중인지를 가리킨다.
                                                  // IsDebuggerPresent 로 검사한다.
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void           // 디버그될 때 Flags/ForceFlags 가 설정된다.
   +0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
   +0x028 EnvironmentUpdateCount : Uint4B
   +0x02c KernelCallbackTable : Ptr32 Void
   +0x030 SystemReserved   : [1] Uint4B
   +0x034 AtlThunkSListPtr32 : Uint4B
   +0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
   +0x03c TlsExpansionCounter : Uint4B
   +0x040 TlsBitmap        : Ptr32 Void
   +0x044 TlsBitmapBits    : [2] Uint4B
   +0x04c ReadOnlySharedMemoryBase : Ptr32 Void
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
   +0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
   +0x058 AnsiCodePageData : Ptr32 Void
   +0x05c OemCodePageData  : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag     : Uint4B              // 프로세스가 디버깅 중이면 0x70 이다.
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER
   +0x078 HeapSegmentReserve : Uint4B
   +0x07c HeapSegmentCommit : Uint4B
   +0x080 HeapDeCommitTotalFreeThreshold : Uint4B
   +0x084 HeapDeCommitFreeBlockThreshold : Uint4B
```

```
+0x088 NumberOfHeaps     : Uint4B
+0x08c MaximumNumberOfHeaps : Uint4B
+0x090 ProcessHeaps      : Ptr32 Ptr32 Void
+0x094 GdiSharedHandleTable : Ptr32 Void
+0x098 ProcessStarterHelper : Ptr32 Void
+0x09c GdiDCAttributeList : Uint4B
+0x0a0 LoaderLock        : Ptr32 Void
+0x0a4 OSMajorVersion    : Uint4B
+0x0a8 OSMinorVersion    : Uint4B
+0x0ac OSBuildNumber     : Uint2B
+0x0ae OSCSDVersion      : Uint2B
+0x0b0 OSPlatformId      : Uint4B
+0x0b4 ImageSubsystem    : Uint4B
+0x0b8 ImageSubsystemMajorVersion : Uint4B
+0x0bc ImageSubsystemMinorVersion : Uint4B
+0x0c0 ImageProcessAffinityMask : Uint4B
+0x0c4 GdiHandleBuffer   : [34] Uint4B
+0x14c PostProcessInitRoutine : Ptr32     void
+0x150 TlsExpansionBitmap : Ptr32 Void
+0x154 TlsExpansionBitmapBits : [32] Uint4B
+0x1d4 SessionId         : Uint4B
+0x1d8 AppCompatFlags    : _ULARGE_INTEGER
+0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER
+0x1e8 pShimData         : Ptr32 Void
+0x1ec AppCompatInfo     : Ptr32 Void
+0x1f0 CSDVersion        : _UNICODE_STRING
+0x1f8 ActivationContextData : Ptr32 Void
+0x1fc ProcessAssemblyStorageMap : Ptr32 Void
+0x200 SystemDefaultActivationContextData : Ptr32 Void
+0x204 SystemAssemblyStorageMap : Ptr32 Void
+0x208 MinimumStackCommit : Uint4B
```

C. EPROCESS

```
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF      // 경쟁 조건 등에서 프로세스를 보호한다.
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void          // 프로세스가 디버그될 때 LPC 포트를 가리킨다.
                                                 // CheckRemoteDebuggerPresent 나
                                                 // NtQueryInformationProcess 로 검사한다.
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
   +0x11c VadRoot          : Ptr32 Void
   +0x120 VadHint          : Ptr32 Void
   +0x124 CloneRoot        : Ptr32 Void
   +0x128 NumberOfPrivatePages : Uint4B
   +0x12c NumberOfLockedPages : Uint4B
   +0x130 Win32Process     : Ptr32 Void
   +0x134 Job              : Ptr32 _EJOB
   +0x138 SectionObject    : Ptr32 Void
   +0x13c SectionBaseAddress : Ptr32 Void
   +0x140 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
```

```
+0x144 WorkingSetWatch  : Ptr32 _PAGEFAULT_HISTORY
+0x148 Win32WindowStation : Ptr32 Void
+0x14c InheritedFromUniqueProcessId : Ptr32 Void
+0x150 LdtInformation   : Ptr32 Void
+0x154 VadFreeHint      : Ptr32 Void
+0x158 VdmObjects       : Ptr32 Void
+0x15c DeviceMap        : Ptr32 Void
+0x160 PhysicalVadList  : _LIST_ENTRY
+0x168 PageDirectoryPte : _HARDWARE_PTE
+0x168 Filler           : Uint8B
+0x170 Session          : Ptr32 Void
+0x174 ImageFileName    : [16] UChar
+0x184 JobLinks         : _LIST_ENTRY
+0x18c LockedPagesList  : Ptr32 Void
+0x190 ThreadListHead   : _LIST_ENTRY
+0x198 SecurityPort     : Ptr32 Void
+0x19c PaeTop           : Ptr32 Void
+0x1a0 ActiveThreads    : Uint4B
+0x1a4 GrantedAccess    : Uint4B
+0x1a8 DefaultHardErrorProcessing : Uint4B
+0x1ac LastThreadExitStatus : Int4B
+0x1b0 Peb              : Ptr32 _PEB
+0x1b4 PrefetchTrace    : _EX_FAST_REF
+0x1b8 ReadOperationCount : _LARGE_INTEGER
+0x1c0 WriteOperationCount : _LARGE_INTEGER
+0x1c8 OtherOperationCount : _LARGE_INTEGER
+0x1d0 ReadTransferCount : _LARGE_INTEGER
+0x1d8 WriteTransferCount : _LARGE_INTEGER
+0x1e0 OtherTransferCount : _LARGE_INTEGER
+0x1e8 CommitChargeLimit : Uint4B
+0x1ec CommitChargePeak : Uint4B
+0x1f0 AweInfo          : Ptr32 Void
+0x1f4 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
+0x1f8 Vm               : _MMSUPPORT
+0x238 LastFaultCount   : Uint4B
+0x23c ModifiedPageCount : Uint4B
+0x240 NumberOfVads     : Uint4B
+0x244 JobStatus        : Uint4B
```

```
+0x248 Flags            : Uint4B
+0x248 CreateReported   : Pos 0, 1 Bit
+0x248 NoDebugInherit   : Pos 1, 1 Bit        // 프로세스가 디버깅 중이면 FALSE 이고,
                                              // NtQueryInformationProcess 로 검사한다.
+0x248 ProcessExiting   : Pos 2, 1 Bit
+0x248 ProcessDelete    : Pos 3, 1 Bit
+0x248 Wow64SplitPages  : Pos 4, 1 Bit
+0x248 VmDeleted        : Pos 5, 1 Bit
+0x248 OutswapEnabled   : Pos 6, 1 Bit
+0x248 Outswapped       : Pos 7, 1 Bit
+0x248 ForkFailed       : Pos 8, 1 Bit
+0x248 HasPhysicalVad   : Pos 9, 1 Bit
+0x248 AddressSpaceInitialized : Pos 10, 2 Bits
+0x248 SetTimerResolution : Pos 12, 1 Bit
+0x248 BreakOnTermination : Pos 13, 1 Bit
+0x248 SessionCreationUnderway : Pos 14, 1 Bit
+0x248 WriteWatch       : Pos 15, 1 Bit
+0x248 ProcessInSession : Pos 16, 1 Bit
+0x248 OverrideAddressSpace : Pos 17, 1 Bit
+0x248 HasAddressSpace  : Pos 18, 1 Bit
+0x248 LaunchPrefetched : Pos 19, 1 Bit
+0x248 InjectInpageErrors : Pos 20, 1 Bit
+0x248 VmTopDown        : Pos 21, 1 Bit
+0x248 Unused3          : Pos 22, 1 Bit
+0x248 Unused4          : Pos 23, 1 Bit
+0x248 VdmAllowed       : Pos 24, 1 Bit
+0x248 Unused           : Pos 25, 5 Bits
+0x248 Unused1          : Pos 30, 1 Bit
+0x248 Unused2          : Pos 31, 1 Bit
+0x24c ExitStatus       : Int4B
+0x250 NextPageColor    : Uint2B
+0x252 SubSystemMinorVersion : UChar
+0x253 SubSystemMajorVersion : UChar
+0x252 SubSystemVersion : Uint2B
+0x254 PriorityClass    : UChar
+0x255 WorkingSetAcquiredUnsafe : UChar
+0x258 Cookie           : Uint4B
```

D. TEB (Thread Environment Block)

```
nt!_TEB
   +0x000 NtTib            : _NT_TIB             // Thread Information Block
      +0x000 ExceptionList    : Ptr32             // SEH 체인(리스트). FS:0 로 접근한다.
      +0x004 StackBase        : Ptr32
      +0x008 StackLimit       : Ptr32
      +0x00c SubSystemTib     : Ptr32
      +0x010 FiberData        : Ptr32
      +0x010 Version          : Uint4B
      +0x014 ArbitraryUserPointer : Ptr32         // 로드된 DLL 의 이름 포인터가 임시 저장된다.
      +0x018 Self             : Ptr32             // TEB 에 대한 포인터. FS:18h 로 접근한다.
   +0x01c EnvironmentPointer : Ptr32
   +0x020 ClientId         : _CLIENT_ID
      +0x000 UniqueProcess    : Ptr32             // PID
      +0x004 UniqueThread     : Ptr32             // TID
   +0x028 ActiveRpcHandle  : Ptr32
   +0x02c ThreadLocalStoragePointer : Ptr32
   +0x030 ProcessEnvironmentBlock : Ptr32         // PEB 에 대한 포인터
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32
   +0x040 Win32ThreadInfo  : Ptr32
   +0x044 User32Reserved   : Uint4B
   +0x0ac UserReserved     : Uint4B
   +0x0c0 WOW32Reserved    : Ptr32
   +0x0c4 CurrentLocale    : Uint4B
   +0x0c8 FpSoftwareStatusRegister : Uint4B
   +0x0cc SystemReserved1  : Ptr32
   +0x1a4 ExceptionCode    : Int4B
   +0x1a8 ActivationContextStack : _ACTIVATION_CONTEXT_STACK
      +0x000 Flags            : Uint4B
      +0x004 NextCookieSequenceNumber : Uint4B
      +0x008 ActiveFrame      : Ptr32
      +0x00c FrameListCache   : _LIST_ENTRY
         +0x000 Flink            : Ptr32
         +0x004 Blink            : Ptr32
   +0x1bc SpareBytes1      : UChar
   +0x1d4 GdiTebBatch      : _GDI_TEB_BATCH
```

```
   +0x000 Offset           : Uint4B
   +0x004 HDC              : Uint4B
   +0x008 Buffer           : Uint4B
+0x6b4 RealClientId     : _CLIENT_ID
   +0x000 UniqueProcess    : Ptr32
   +0x004 UniqueThread     : Ptr32
+0x6bc GdiCachedProcessHandle : Ptr32
+0x6c0 GdiClientPID     : Uint4B
+0x6c4 GdiClientTID     : Uint4B
+0x6c8 GdiThreadLocalInfo : Ptr32
+0x6cc Win32ClientInfo  : Uint4B
+0x7c4 glDispatchTable  : Ptr32
+0xb68 glReserved1      : Uint4B
+0xbdc glReserved2      : Ptr32
+0xbe0 glSectionInfo    : Ptr32
+0xbe4 glSection        : Ptr32
+0xbe8 glTable          : Ptr32
+0xbec glCurrentRC      : Ptr32
+0xbf0 glContext        : Ptr32
+0xbf4 LastStatusValue  : Uint4B
+0xbf8 StaticUnicodeString : _UNICODE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32
+0xc00 StaticUnicodeBuffer : Uint2B
+0xe0c DeallocationStack : Ptr32
+0xe10 TlsSlots         : Ptr32
+0xf10 TlsLinks         : _LIST_ENTRY
   +0x000 Flink            : Ptr32
   +0x004 Blink            : Ptr32
+0xf18 Vdm              : Ptr32
+0xf1c ReservedForNtRpc : Ptr32
+0xf20 DbgSsReserved    : Ptr32              // [0]에 ThreadData 포인터가 저장된다.
                                             // [1]에 디버그 객체의 핸들이 저장된다.
+0xf28 HardErrorsAreDisabled : Uint4B
+0xf2c Instrumentation  : Ptr32
+0xf6c WinSockData      : Ptr32
+0xf70 GdiBatchCount    : Uint4B
```

```
+0xf74 InDbgPrint       : UChar
+0xf75 FreeStackOnTermination : UChar
+0xf76 HasFiberData     : UChar
+0xf77 IdealProcessor   : UChar
+0xf78 Spare3           : Uint4B
+0xf7c ReservedForPerf  : Ptr32
+0xf80 ReservedForOle   : Ptr32
+0xf84 WaitingOnLoaderLock : Uint4B
+0xf88 Wx86Thread       : _Wx86ThreadState
   +0x000 CallBx86Eip      : Ptr32
   +0x004 DeallocationCpu  : Ptr32
   +0x008 UseKnownWx86Dll  : UChar
   +0x009 OleStubInvoked   : Char
+0xf94 TlsExpansionSlots : Ptr32
+0xf98 ImpersonationLocale : Uint4B
+0xf9c IsImpersonating  : Uint4B
+0xfa0 NlsCache         : Ptr32
+0xfa4 pShimData        : Ptr32
+0xfa8 HeapVirtualAffinity : Uint4B
+0xfac CurrentTransactionHandle : Ptr32
+0xfb0 ActiveFrame      : Ptr32
+0xfb4 SafeThunkCall    : UChar
+0xfb5 BooleanSpare     : UChar
```

E. ETHREAD

```
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
   +0x1dc KeyedWaitValue   : Ptr32 Void
   +0x1e0 ActiveTimerListLock : Uint4B
   +0x1e4 ActiveTimerListHead : _LIST_ENTRY
   +0x1ec Cid              : _CLIENT_ID
   +0x1f4 LpcReplySemaphore : _KSEMAPHORE
   +0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
   +0x208 LpcReplyMessage  : Ptr32 Void
   +0x208 LpcWaitingOnPort : Ptr32 Void
   +0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
   +0x210 IrpList          : _LIST_ENTRY
   +0x218 TopLevelIrp      : Uint4B
   +0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
   +0x220 ThreadsProcess   : Ptr32 _EPROCESS
   +0x224 StartAddress     : Ptr32 Void
   +0x228 Win32StartAddress : Ptr32 Void
   +0x228 LpcReceivedMessageId : Uint4B
   +0x22c ThreadListEntry  : _LIST_ENTRY
   +0x234 RundownProtect   : _EX_RUNDOWN_REF      // 경쟁 조건 등에서 스레드를 보호한다.
   +0x238 ThreadLock       : _EX_PUSH_LOCK
   +0x23c LpcReplyMessageId : Uint4B
   +0x240 ReadClusterSize  : Uint4B
   +0x244 GrantedAccess    : Uint4B
   +0x248 CrossThreadFlags : Uint4B
```

```
+0x248 Terminated       : Pos 0, 1 Bit
+0x248 DeadThread       : Pos 1, 1 Bit
+0x248 HideFromDebugger : Pos 2, 1 Bit
+0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
+0x248 SystemThread     : Pos 4, 1 Bit
+0x248 HardErrorsAreDisabled : Pos 5, 1 Bit
+0x248 BreakOnTermination : Pos 6, 1 Bit
+0x248 SkipCreationMsg  : Pos 7, 1 Bit        // 동일한 스레드 생성 디버그 이벤트가 전송되
                                              // 는 것을 피하기 위한 플래그
+0x248 SkipTerminationMsg : Pos 8, 1 Bit
+0x24c SameThreadPassiveFlags : Uint4B
+0x24c ActiveExWorker   : Pos 0, 1 Bit
+0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
+0x24c MemoryMaker      : Pos 2, 1 Bit
+0x250 SameThreadApcFlags : Uint4B
+0x250 LpcReceivedMsgIdValid : Pos 0, 1 Bit
+0x250 LpcExitThreadCalled : Pos 1, 1 Bit
+0x250 AddressSpaceOwner : Pos 2, 1 Bit
+0x254 ForwardClusterOnly : UChar
+0x255 DisablePageFaultClustering : UChar
```

Reference

"Windows Debugging Internals: A to Z" 의 원문은 아래의 주소에서 읽을 수 있습니다.

- Windows User Mode Debugging Internals
  (http://www.openrce.org/articles/full_view/24)
- Windows Native Debugging Internals
  (http://www.openrce.org/articles/full_view/25)
- Kernel User-Mode Debugging Support (Dbgk)
  (http://www.openrce.org/articles/full_view/26)

다음의 목록은 Windows 의 커널과 디버깅에 관한 참고 도서들과 자료들입니다.

- 『Windows Internals 제5판』 *마크 러시노비치, 데이비드 솔로몬, 알렉스 이오네스쿠 저*
- 『Debugging Applications for Microsoft .NET and Microsoft Windows』 *John Robbins 저*
- 『파이썬 해킹 프로그래밍』 *저스틴 지이츠 저*
- MSDN :: Debugging Functions
  (http://msdn.microsoft.com/en-us/library/ms679303(v=vs.85).aspx)
- dbgui.c File Reference
  (http://doxygen.reactos.org/dd/ddc/dbgui_8c.html)
- dbgkobj.c File Reference
  (http://doxygen.reactos.org/da/dd4/dbgkobj_8c.html)
- PEB, TEB, EPROCESS, KPROCESS, ETHREAD, KTHREAD
  (http://anster.egloos.com/2128538)
- Ntdebugging :: LPC (Local procedure calls) Part 1 architecture
  (http://blogs.msdn.com/b/ntdebugging/archive/2007/07/26/lpc-local-procedure-calls-part-1-architecture.aspx)
- An Anti-Reverse Engineering Guide
  (http://www.codeproject.com/KB/security/AntiReverseEngineering.aspx)