

# WinDbg. From A to Z!

Everything you need to know about WinDbg.  
And nothing you don't.

By Robert Kuster

December 2007. All rights reserved.

[www.windbg.info](http://www.windbg.info)

Japanese Edition By Tsuyoshi Oguni, February 2009

日本語版 小國 健 (NTT DATA)

1

## WinDbgを選ぶ理由

なぜなら、WinDbgは...

- マイクロソフトのWindowsプロダクトチームが、Windowsの開発に利用している
- よく使われているVisual Studioのデバッガよりも高機能
- 拡張DLLによる、機能の拡張が可能
- Windows OS自身がおもつデバッグエンジンを利用

Windows XP以降では、dbgeng.dllとdbghelp.dllが「C:¥Windows¥System32」にインストールされている。

# 「WinDbg. From A to Z」の目的

- WinDbgのドキュメントは、入門者には分かりづらい
- 優れたドキュメントや使用例がないと、WinDbgの習得は非常に困難

インストールしたものの、すぐにあきらめてしまう人も多い。

- 「WinDbg. From A to Z!」は、WinDbgの簡単な導入マニュアルである。本資料を読めば、WinDbgとは何なのか、何ができるのかが分かる。

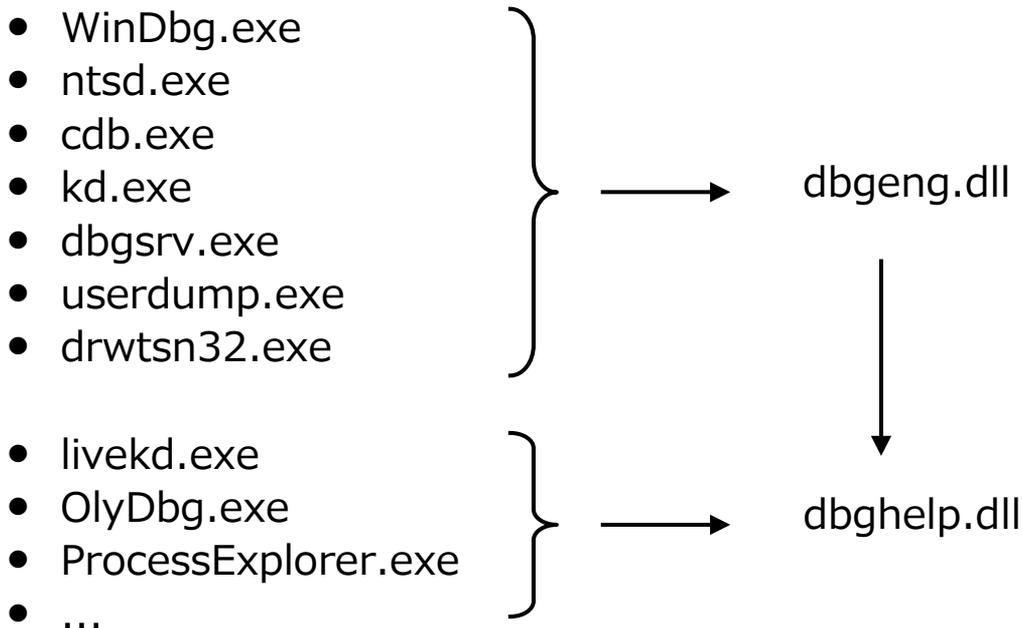
「WinDbg. From A to Z!」は、ユーザーモードでの使用例をもとに説明しているが、カーネルモードでの開発にも役立つ。ユーザーモードのデバッグでも、カーネルモードのデバッグでも、裏で動いているデバッグエンジンは変わらない。基本的に、カーネルモードでは、使用する拡張コマンドが違うに過ぎない。

## 目次 - ロードマップ

### → WinDbgの裏側

- WinDbgの使い方
- グローバルフラグ
- アプリケーション検証
- プロセスダンプ

# Windows XP用のデバッグツール



5

www.windbg.info

## デバッグヘルプライブラリ: dbghelp.dll

- MSDNにドキュメントあり
- Windows 2000以降に同梱されている
- 以下のサポートルーチンを含む
  - a) プロセスのダンプ取得(MiniDumpWriteDump, DbgHelpCreateUserDump, ...)
  - b) スタックトレースの取得(StackWalk64, ...)
  - c) シンボルの操作(SymFromAddr, Sym\*, ...)
  - d) 実行イメージからの情報取得(ImageNtHeader, FindDebugInfoFile, ...)

c)やd)の関数の多くは、まったく同じ宣言のものが、imagehlp.dllでもエクスポートされている。imagehlpの関数は、単にdbghelpの関数にフォワードされているものが多いが、逆アセンブルしてみると、同じソースから作られているものもあることが分かる（次のスライドの逆アセンブルを参照）。マイクロソフトのツールには、DbgHelp.dllを優先的に使っているものもあれば、Visual StudioやDependency Walkerのようにimagehlp.dllを使っているものもある。両方を使っているものもある。

6

www.windbg.info

# dbghelp!ImageNtHeaderとimagehlp!ImageNtHeaderの比較

```

Command
0:000> uf dbghelp!ImageNtHeader
dbghelp!RtlpImageNtHeader:
6d59a770 6a0c      push 0Ch
6d59a772 68e932586d push offset dbghelp!_paraFunctionEntryCache:'.vftable'+0xc (6d5832e8)
6d59a777 e8d0e00100 call dbghelp!_SEH_prolog (6d5b884c)
6d59a77c 33c0      xor eax,eax
6d59a77e 8b4d08    mov ecx,dword ptr [ebp+8]
6d59a781 85c9      test ecx,ecx
6d59a783 743c      je dbghelp!RtlpImageNtHeader+0x51 (6d59a7c1)

dbghelp!RtlpImageNtHeader+0x15:
6d59a785 83f9ff    cmp ecx,0FFFFFFFh
6d59a788 7437      je dbghelp!RtlpImageNtHeader+0x51 (6d59a7c1)

dbghelp!RtlpImageNtHeader+0x1a:
6d59a78a 2145fc    and dword ptr [ebp-4],eax
6d59a78d 6681394d5a cmp word ptr [ecx],5A4Dh
6d59a792 7529      jne dbghelp!RtlpImageNtHeader+0x4d (6d59a7bd)

dbghelp!RtlpImageNtHeader+0x24:
6d59a794 8b513c    mov edx,dword ptr [ecx+3Ch]
6d59a797 81fa000001010 cmp edx,10000000h
6d59a79d 731e      jae dbghelp!RtlpImageNtHeader+0x4d (6d59a7bd)

dbghelp!RtlpImageNtHeader+0x2f:
6d59a79f 8d040a    lea eax,[edx+ecx]
6d59a7a2 8945e4    mov dword ptr [ebp-1Ch],eax
6d59a7a5 813850450000 cmp word ptr [eax],4550h
6d59a7ab 7410      je dbghelp!RtlpImageNtHeader+0x4d (6d59a7bd)

dbghelp!RtlpImageNtHeader+0x3d:
6d59a7ad 33c0      xor eax,eax
6d59a7af 8945e4    mov dword ptr [ebp-1Ch],eax
6d59a7b5 eb09      jmp dbghelp!RtlpImageNtHeader+0x4d (6d59a7bd)

dbghelp!RtlpImageNtHeader+0x4d:
6d59a7bd 834dfcfff or dword ptr [ebp-4],0FFFFFFFh

dbghelp!RtlpImageNtHeader+0x51:
6d59a7c1 e8c1e00100 call dbghelp!_SEH_epilog (6d5b8887)
6d59a7c6 c20400    ret 4

dbghelp!ImageNtHeader:
6d59a7ce 8bff     mov edi,edi
6d59a7d0 55      push ebp
6d59a7d1 8bec    mov ebp,esp
6d59a7d3 5d      pop ebp
6d59a7d4 e997ffff jmp dbghelp!RtlpImageNtHeader (6d59a770)
    
```

```

Command
0:000> uf imagehlp!ImageNtHeader
imagehlp!RtlpImageNtHeader:
76c135d2 6a0c      push 0Ch
76c135d4 682036c176 push offset imagehlp!_string'+0x2c (76c13620)
76c135d9 e80edcffff call imagehlp!_SEH_prolog (76c111ec)
76c135de 33c0      xor eax,eax
76c135e0 8b4d08    mov ecx,dword ptr [ebp+8]
76c135e3 85c9      test ecx,ecx
76c135e5 7430      je imagehlp!RtlpImageNtHeader+0x51 (76c13617)

imagehlp!RtlpImageNtHeader+0x15:
76c135e7 83f9ff    cmp ecx,0FFFFFFFh
76c135ea 742b      je imagehlp!RtlpImageNtHeader+0x51 (76c13617)

imagehlp!RtlpImageNtHeader+0x1a:
76c135ec 2145fc    and dword ptr [ebp-4],eax
76c135ef 6681394d5a cmp word ptr [ecx],5A4Dh
76c135f4 751d      jne imagehlp!RtlpImageNtHeader+0x4d (76c13613)

imagehlp!RtlpImageNtHeader+0x24:
76c135f6 8b513c    mov edx,dword ptr [ecx+3Ch]
76c135f9 81fa000001010 cmp edx,10000000h
76c135ff 7312      jae imagehlp!RtlpImageNtHeader+0x4d (76c13613)

imagehlp!RtlpImageNtHeader+0x2f:
76c13601 8d040a    lea eax,[edx+ecx]
76c13604 8945e4    mov dword ptr [ebp-1Ch],eax
76c13607 813850450000 cmp word ptr [eax],4550h
76c1360d 8f857c3d0000 jne imagehlp!RtlpImageNtHeader+0x3d (76c1738f)

imagehlp!RtlpImageNtHeader+0x4d:
76c13613 834dfcfff or dword ptr [ebp-4],0FFFFFFFh

imagehlp!RtlpImageNtHeader+0x51:
76c13617 e80bdcffff call imagehlp!_SEH_epilog (76c11227)
76c1361c c20400    ret 4

imagehlp!RtlpImageNtHeader+0x3d:
76c1738f 33c0      xor eax,eax
76c17391 8945e4    mov dword ptr [ebp-1Ch],eax
76c17394 e97ac2ffff jmp imagehlp!RtlpImageNtHeader+0x4d (76c13613)

imagehlp!ImageNtHeader:
76c177ad 8bff     mov edi,edi
76c177af 55      push ebp
76c177b0 8bec    mov ebp,esp
76c177b2 5d      pop ebp
76c177b5 e91abefffff jmp imagehlp!RtlpImageNtHeader (76c135d2)
    
```

# ImageHlpの依存関係

Module	PI	Ordinal ^	Hint	Function	Entry Point
IMAGEHLP.DLL					
KERNEL32.DLL	C	N/A	N/A	SymGetLineNext	Not Bound
KERNEL32.DLL	C	N/A	N/A	SymGetLinePrev64	Not Bound
KERNEL32.DLL	C	N/A	N/A	SymGetLinePrev	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetModuleBase64	Not Bound
KERNEL32.DLL	C	N/A	N/A	SymGetModuleBase	Not Bound
VERSION.DLL	C	N/A	N/A	SymGetModuleInfo64	Not Bound
ADVAPI32.DLL	C	N/A	N/A	SymGetModuleInfo	Not Bound
RPCRT4.DLL	C	N/A	N/A	SymGetModuleInfoW64	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetModuleInfoW	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetOptions	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSearchPath	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymFromAddr64	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymFromAddr	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymFromName64	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymFromName	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymNext64	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymNext	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymPrev64	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetSymPrev	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetTypeFromName	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymGetTypeInfo	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymInitialize	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymLoadModule64	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymLoadModule	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymMatchFileName	Not Bound
MSVCRT.DLL	C	N/A	N/A	SymMatchString	Not Bound

# デバッガエンジンAPI: dbgeng.dll

- WinDbgのドキュメントに説明あり

dbgeng.dllのヘッダファイルとlibファイルは、WinDbgのインストール時にカスタムインストールを選び「SDK」コンポーネントを選択すると入手できる。

- Windows XP 以降に同梱されている
- COMインタフェースを使ってアクセス

→ IDebugAdvanced, IDebugControl, IDebugSystemObjects, ...

- デバッガが実現している動作はすべて、COMインタフェースで公開されている

事実1: WinDbgは、デバッグエンジン上のシェルに過ぎない。

事実2: このエンジンを利用して、独自に新ツールを開発することも可能。

9

www.windbg.info

## DbgEngの依存関係

PI	Ordinal ^	Hint	Function	Entry Point	
E	Ordinal	Hint ^	Function	Entry Point	
☐	1 (0x0001)	0 (0x0000)	DebugConnect	0x000C2A50	→ IDebugAdvanced, IDebugControl, IDebugSystemObject, ...
☐	2 (0x0002)	1 (0x0001)	DebugConnectWide	0x000C2B40	
☐	3 (0x0003)	2 (0x0002)	DebugCreate	0x000C2B80	

10

www.windbg.info

# デバッグシンボル

- 実行可能ファイルは、単なるバイトの羅列
- デバッガにシンボルを与えると...
  - 実行可能ファイルのアドレスと、ソースコードの行が関連付けられる
  - アプリケーションの内部の配置やデータを分析
- プログラムデータベース → PDBファイル
  - 最新のマイクロソフトデバッグ情報フォーマット  
COFFとCodeViewは非推奨
  - PDBの情報は、実行可能ファイルとは別のファイルに格納される
  - PDBのフォーマットは非公開
  - PDBを扱う特別なAPIあり: DbgHelp.dllやMsDiaXY.dll

# デバッグ情報の種類

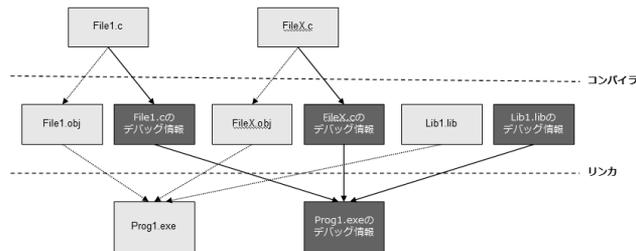
情報の種類	説明
パブリックな関数と変数	さまざまなコンパイル単位から参照される関数や変数
FPO情報	FPO最適化（フレームポインタ省略）を指定してコンパイルされた場合にスタックフレームを取り出すための追加情報
プライベートな関数と変数	ローカル変数や、関数のパラメータなどを含む、すべての関数や変数
ソースファイルと行情報	ソースファイルと行の情報
型情報	関数と変数の追加情報 変数: 型(int, string, ...) 関数: パラメータの個数と型, 呼び出し規約, 戻り値

リンカ:  
/pdbstripped

マイクロソフトのモジュール(kernel32.dll, user32.dll, ...)では、常にパブリックなシンボルが取り除かれている。

# デバッグ情報の生成

- ビルドの処理は、次の2段階から成る
  - 1) コンパイラ: マシン語の命令を生成し、.OBJファイルに格納
  - 2) リンカ: 関連する.OBJファイルや.LIBファイルを、最終的な実行可能ファイルにまとめる
- デバッグ情報の生成も、次の2段階から成る
  - 1) コンパイラ: ソースファイルごとに、デバッグ情報を生成
  - 2) リンカ: 関連するデバッグ情報を、実行可能ファイルに対する最終的なデバッグ情報としてまとめる



- コンパイラオプション: /Z7, /Zi, /ZI
- リンカオプション: /debug, /pdb, /pdbstripped

スタティックライブラリに関する注意: /Z7 オプションを使用すると、生成される.LIBファイルにデバッグ情報を格納できる。

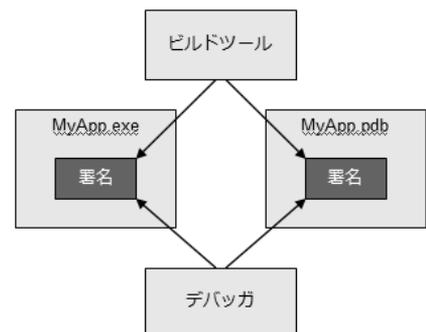
# デバッグ情報のマッチング

- ビルド時に、署名が、実行可能ファイルとPDBファイルに埋め込まれる

PDB 2.0 ファイル: タイムスタンプ

PDB 7.0 ファイル: ビルド時に生成されたGUID

- デバッガの突き合わせ処理では、この署名が同一のものである必要がある



- PDBファイルの検索アルゴリズム

1. モジュール (EXEやDLL) のフォルダを検索
2. PEファイル (NB10またはRSDSデバッグヘッダ) 内で指定された名前とパスを検索
3. 環境変数 `_NT_SYMBOL_PATH` や `_NT_ALT_SYMBOL_PATH` 内を検索

# コールスタック

## ● 有効なシンボルなし

```
002df350 ntdll!DbgBreakPoint
002df43c TestApplication+0x127eb
002df544 TestApplication+0x12862
002df550 MFC80UD!AfxDlgProc+0x3e
002df57c USER32!InternalCallWinProc+0x28
002df5f8 USER32!UserCallDlgProcCheckWow+0x102
002df648 USER32!DefDlgProcWorker+0xb2
002df668 USER32!DefDlgProcW+0x29
002df694 USER32!InternalCallWinProc+0x28
002df70c USER32!UserCallWinProcCheckWow+0x16a
002df744 USER32!CallWindowProcAorW+0xab
002df764 USER32!CallWindowProcW+0x1b
002df788 MFC80UD!CWnd::DefWindowProcW+0x32
002df7a4 MFC80UD!CWnd::Default+0x3b
002df7c8 MFC80UD!CDialog::HandleInitDialog+0xd3
002df900 MFC80UD!CWnd::OnWndMsg+0x817
002df920 MFC80UD!CWnd::WindowProc+0x30
002df99c MFC80UD!AfxCallWndProc+0xee
002df9bc MFC80UD!AfxWndProc+0xa4
002df9f8 MFC80UD!AfxWndProcBase+0x59
```

## ● 有効なシンボルあり

```
002df350 ntdll!DbgBreakPoint
002df43c TestApplication!CMyDlg::PreInit+0x3b [MyDlg.cpp @ 75]
002df544 TestApplication!CMyDlg::OnInitDialog+0x52 [MyDlg.cpp @ 91]
002df550 MFC80UD!AfxDlgProc+0x3e
002df57c USER32!InternalCallWinProc+0x28
002df5f8 USER32!UserCallDlgProcCheckWow+0x102
002df648 USER32!DefDlgProcWorker+0xb2
002df668 USER32!DefDlgProcW+0x29
002df694 USER32!InternalCallWinProc+0x28
002df70c USER32!UserCallWinProcCheckWow+0x16a
002df744 USER32!CallWindowProcAorW+0xab
002df764 USER32!CallWindowProcW+0x1b
002df788 MFC80UD!CWnd::DefWindowProcW+0x32
002df7a4 MFC80UD!CWnd::Default+0x3b
002df7c8 MFC80UD!CDialog::HandleInitDialog+0xd3
002df900 MFC80UD!CWnd::OnWndMsg+0x817
002df920 MFC80UD!CWnd::WindowProc+0x30
002df99c MFC80UD!AfxCallWndProc+0xee
002df9bc MFC80UD!AfxWndProc+0xa4
002df9f8 MFC80UD!AfxWndProcBase+0x59
```

# 侵入的アタッチと非侵入的アタッチ

## ● 侵入的アタッチ

- DebugActiveProcessが呼び出される
- break-inスレッドが生成される
- Windows XP以前では、デバッガ終了時やデタッチ時に、デバッグ対象アプリケーションが強制終了される
- 侵入的デバッガとしてアタッチできるのは、プロセス当たり一度にひとつだけ

## ● 非侵入的アタッチ

- OpenProcessが呼び出される
- break-inスレッドは生成されない
- プロセスに対し、デバッガとして真のアタッチをするわけではない
- デバッグ対象アプリケーションのスレッドはすべて凍結状態
- メモリの内容の変更や確認は可能
- ブレークポイントはセットできない
- アプリケーションのステップ実行はできない
- デバッグ対象アプリケーションを終了することなく、デバッガの終了やデタッチが可能
- ひとつのプロセスに、複数の非侵入的デバッガ（+ひとつの侵入的デバッガ）をアタッチできる
- こんな場合に便利...
  - Visual Studioなどの侵入的デバッガでアプリケーションをデバッグ中でも、WinDbgを非侵入的デバッガとしてアタッチし、追加の情報を取得できる。
  - デバッグ対象アプリケーションは完全に凍結しており、真のアタッチに必要なbreak-inスレッドが生成されることはない。

# 例外

- 言語に依存しない、システムが提供する機能

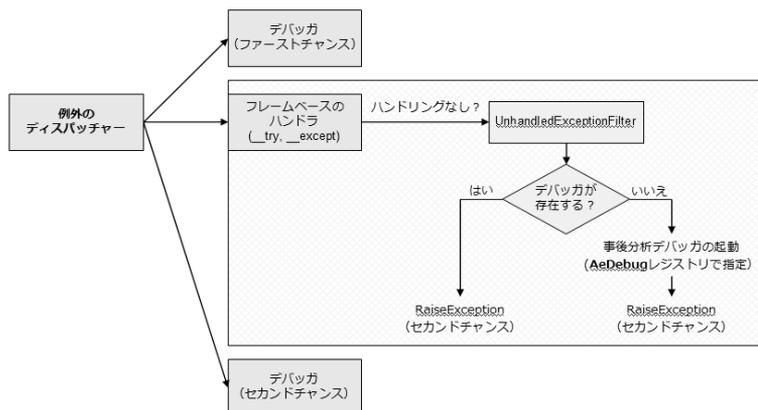
- 例外は、言語の拡張機能を使って処理

例: C++の `__try & __except` 構造

- 処理時間が重視される箇所で、`try-catch-except`を条件チェックに使わないこと

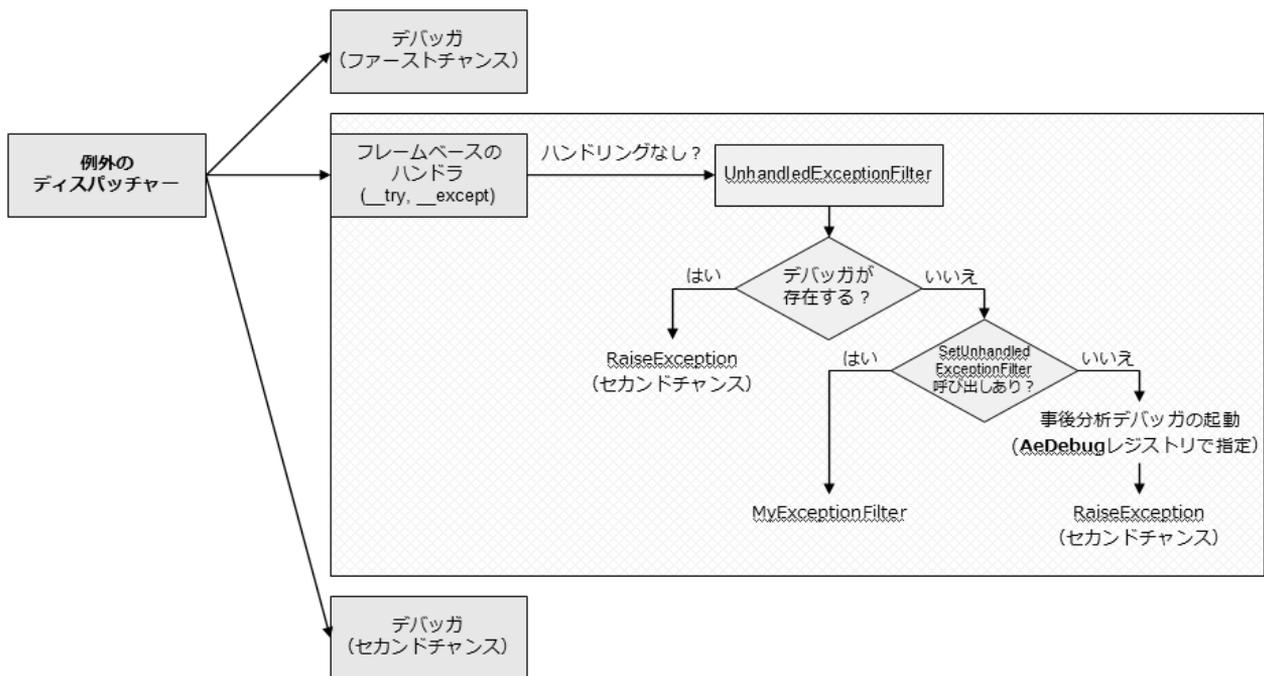
例外発生の際に、システムは例外レコードを生成し、全スタックフレームを通してフレームベースの例外ハンドラ(catch-except)を逆向きに検索し、それからようやくプログラムの実行を継続する。この処理には何百もの命令を要するため、性能が低下する。

# 例外のディスパッチ



- 1) システムはまず、プロセスのデバッガがあれば、そこに例外を通知しようとする。
- 2) プロセスがデバッグ中でない場合や、関連付けられているデバッガが例外をハンドリングしない (WinDbg → gN == 例外をハンドリングしないで実行する) 場合は、システムはフレームベースの例外ハンドラを探す。
- 3) フレームベースのハンドラが見つからない場合や、その例外をハンドリングする例外ハンドラがない場合は、UnhandledExceptionFilterがプロセスのデバッガに再度例外を通知しようとする。この例外は、セカンドチャンスまたはラストチャンスのお知らせとして知られている。
- 4) プロセスがデバッグ中でない場合や、関連付けられているデバッガが例外をハンドリングしない場合は、AeDebugで指定された事後分析デバッガが起動される。

# 例外のディスパッチとSetUnhandledExceptionFilter



# AeDebug? 事後分析デバッグ!

- 事後分析デバッガの指定と変更

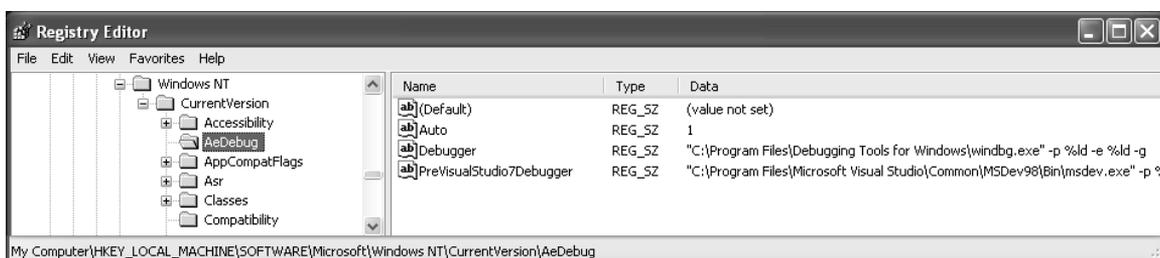
- WinDbg -I
- drwtsn32 -i

- 事後分析デバッガの設定

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug

AeDebugに指定したプログラムは、何でも実行される。

そのプログラムが本当にデバッガなのかどうかは、検証されない。



# 目次 - ロードマップ

✓ WinDbgの裏側

→ **WinDbgの使い方**

- グローバルフラグ
- アプリケーション検証
- プロセスダンプ

# WinDbgのコマンド

- 標準コマンド
  - プロセスのデバッグに使われる
  - 例: k, lm, g
- メタコマンド (ドットコマンド)
  - 通常、デバッガの制御に使われる
  - 例: .sympath, .cls, .lastevent, .detach, .if
- 拡張コマンド
  - 拡張DLL内に、エクスポート関数として実装されている
  - WinDbgが強力なデバッガである大きな理由
  - 拡張DLL一式がインストール済み: exts.dll, ntsdexts.dll, uext.dll, wow64exts.dll, kdexts.dll, ...
  - 独自に拡張DLLを開発することも可能
  - 例: !analyze, !address, !handle, !peb

# おもな拡張DLL

- !exts.help → 一般的な拡張コマンド
- !Uext.help → ユーザーモードの拡張コマンド (OS非依存)
- !Ntsdexts.help → ユーザーモードの拡張コマンド (OS依存)
- !Kdexts.help → カーネルモードの拡張コマンド
- !logexts.help → ログ取得拡張コマンド
- !clr10¥sos.help → マネージコードのデバッグ
- !wow64exts.help → Wow64デバッガ拡張
- ...

# WinDbgにおけるシンボルの設定

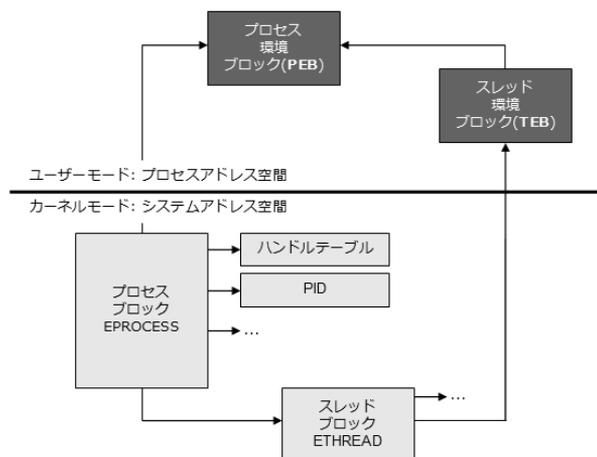
- **\_NT\_SYMBOL\_PATH**環境変数の設定は必須  
マイクロソフト用シンボルの設定例:  
`_NT_SYMBOL_PATH=srv*c:¥Symbols¥MsSymbols*http://msdl.microsoft.com/download/symbols;`  
この設定を行うと、WinDbgは、マイクロソフトのコンポーネント (kernel32など) 用のシンボルを、マイクロソフトのサーバーから必要に応じて自動的にダウンロードする。
- WinDbgのGUIからも、シンボルの設定が可能
  - (メニュー) File → Symbol File Path... (Ctrl+S)
- 便利なコマンド
  - .sympath → シンボルの検索パスの取得と設定
  - .sympath+ XY → シンボルの検索パスに、XYディレクトリを追加
  - !sym noisy → シンボルの検索情報を表示するようデバッガに指示
  - !d kernel32 → kernel32.dllのシンボルを読み込む
  - !d \* → すべてのモジュールのシンボルを読み込む
  - .reload → シンボル情報を読み直す
  - x kernel32!\* → kernel32の全シンボルを調べて一覧表示
  - x kernel32!\*LoadLibrary\* → LoadLibraryという文字列を含むkernel32の全シンボルを一覧表示
  - dt ntdll!\* → ntdll内の全変数を表示

# WinDbgにおけるソースの設定

- **\_NT\_SOURCE\_PATH**環境変数の設定は必須  
例: `_NT_SOURCE_PATH=C:¥Sources`
- WinDbgのGUIからも、ソースの設定が可能
  - (メニュー) File → Source File Path... (Ctrl+P)
- 便利なコマンド
  - `.srcpath` → ソースファイルの検索パスの取得と設定
  - `.srcpath+ XY` → ソースの検索パスに、XYディレクトリを追加

重要: WinDbgに、シンボルとソースを正しく設定すること。これが、WinDbgの入門者がミスしやすい、最初の重要なステップである。マイクロソフトのコンポーネント(kernel32.dll, ntdll.dll,...)のシンボルがなければ、以下のセクションで示すコマンドの大半が動作しないので注意。

# Windows NTのプロセスとスレッド



- すべてのWindowsプロセスは、カーネルモードのエグゼクティブプロセスブロック(EPROCESS)で表される。
- EPROCESSは、プロセスに関連のあるデータ構造を、数多く指し示している。たとえば、各プロセスは、エグゼクティブスレッドブロック(ETHREAD)で表されるスレッドをひとつ以上持っている。
- EPROCESSは、プロセスのアドレス空間にあるプロセス環境ブロック(PEB)を指す。
- ETHREADは、プロセスのアドレス空間にあるスレッド環境ブロック(TEB)を指す。

# PEBとTEB

- PEB=プロセス環境ブロック
  - イメージの基本情報 (ベースアドレス, バージョン番号, モジュール一覧)
  - プロセスヒープの情報
  - 環境変数
  - コマンドライン引数
  - DLL検索パス
  - PEBの表示方法: !peb, dt nt!\_PEB
- TEB=スレッド環境ブロック
  - スタック情報 (スタック下限, スタック上限)
  - TLS (スレッドローカルストレージ) 配列
  - TEBの表示方法: !teb, dt nt!\_TEB

事実: 多くのWinDbgのコマンド(!m, !dlls, !imgreloc, !tls, !gle)は、PEBやTEBから取得したデータを利用している。

## 例 - PEBの「ダンプ」

```
0:001> dt nt!_PEB -r @$peb // @$peb = プロセスのPEBのアドレス (疑似レジスタの文法を参照)
+0x000 InheritedAddressSpace : 0 ""
+0x001 ReadImageFileExecOptions : 0 ""
+0x002 BeingDebugged : 0x1 ""
...
+0x008 ImageBaseAddress : 0x00400000
+0x00c Ldr : 0x7d6a01e0 _PEB_LDR_DATA
+0x000 Length : 0x28
+0x004 Initialized : 0x1 ""
+0x008 SsHandle : (null)
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x2d1eb0 - 0x2da998 ]
+0x000 Flink : 0x002d1eb0 _LIST_ENTRY [ 0x2d1f08 - 0x7d6a01ec ]
+0x004 Blink : 0x002da998 _LIST_ENTRY [ 0x7d6a01ec - 0x2d9f38 ]
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x2d1eb8 - 0x2da9a0 ]
...
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x2d1f18 - 0x2da9a8 ]
...
+0x024 EntryInProgress : (null)
+0x010 ProcessParameters : 0x001c0000 _RTL_USER_PROCESS_PARAMETERS
+0x000 MaximumLength : 0x102c
+0x004 Length : 0x102c
+0x008 Flags : 0x4001
+0x00c DebugFlags : 0
...
+0x024 CurrentDirectory : _CURDIR
+0x000 DosPath : _UNICODE_STRING "D:¥Development¥Utils¥"
+0x008 Handle : 0x00000024
+0x030 DllPath : _UNICODE_STRING "C:¥WINDOWS¥system32;C:¥WINDOWS¥system;C:¥WINDOWS;..."
...
```

## プロセスやモジュールの情報を取得するためのWinDbgコマンド

コマンド	説明
!peb	プロセス環境ブロック(PEB)の情報を整形して表示
dt nt!_PEB Addr	PEBの全ダンプ
!m	ロードモジュールやアンロードモジュールの一覧
!mD	上に同じ。デバッガマークアップ言語で出力。
!m vm kernel32	kernel32の詳細を出力 (イメージやシンボルの情報を含む)
!mi kernel32	上に同じ。DLL拡張による実装。
!dlls	ロードモジュールの一覧。ローダーに関する情報 (エン트리ポイント, ロードカウント) を含む。
!dlls -c kernel32	上に同じ。ただしkernel32のみ。
!imgreloc	再配置情報の表示
!dh kernel32	kernel32のヘッダを表示

## 例 - モジュール情報

```

0:001>!dlls -c msvcrt
Dump dll containing 0x77ba0000:
0x002d40c0: C:\WINDOWS\system32\msvcrt.dll
      Base 0x77ba0000      EntryPoint 0x77baf78b      Size      0x0005a000
      Flags 0x80084006      LoadCount 0x00000007      TlsIndex  0x00000000
      LDRP_STATIC_LINK
      LDRP_IMAGE_DLL
      LDRP_ENTRY_PROCESSED
      LDRP_PROCESS_ATTACH_CALLED

0:001> !m vm msvcrt
start      end      module name
77ba0000   77bfa000 msvcrt (deferred)
      Image path: C:\WINDOWS\system32\msvcrt.dll
      Image name: msvcrt.dll
      Timestamp: Fri Mar 25 03:33:02 2005 (4243785E)
      CheckSum: 0006288A
      ImageSize: 0005A000
      File version: 7.0.3790.1830
      Product version: 6.1.8638.1830
      ...
      CompanyName: Microsoft Corporation
      ProductName: Microsoft® Windows® Operating System
      InternalName: msvcrt.dll
      OriginalFilename: msvcrt.dll
      ProductVersion: 7.0.3790.1830
      FileVersion: 7.0.3790.1830 (srv03_sp1_rtm.050324-1447)
      FileDescription: Windows NT CRT DLL
      LegalCopyright: © Microsoft Corporation. All rights reserved.

```

# スレッドの情報を取得するためのWinDbgコマンド

コマンド	説明
~	全スレッドのスレッド状態
~0	スレッド0のスレッド状態
~.	現在アクティブなスレッドのスレッド状態
~*	全スレッドのスレッド状態。追加情報あり（プライオリティ, 開始アドレス）。
~*k	全スレッドのコールスタック（~ と !uniqustack）
~<thread>s	カレントスレッドの設定
!gle	ラストエラーの取得
!runaway	→ 各スレッドの消費時間を表示 → どのスレッドが制御不能になっているのか、CPUタイムを大量に消費しているのかを調べる簡便な方法
!teb	スレッド環境ブロック(TEB)の情報を整形して表示
dt nt!_TEB Addr	TEBの全ダンプ

## 例 - スレッド

```

0:001> !runaway 7
User Mode Time
  Thread      Time
  0:d28       0 days 0:00:00.015
  1:2b0       0 days 0:00:00.000
Kernel Mode Time
  Thread      Time
  0:d28       0 days 0:00:00.093
  1:2b0       0 days 0:00:00.000
Elapsed Time
  Thread      Time
  0:d28       0 days 0:04:04.156
  1:2b0       0 days 0:03:53.328

0:000> ~*
. 0 Id: dac.d28 Suspend: 1 Teb: 7efdd000 Unfrozen
  Start: TestApp!ILT+1415(_wWinMainCRTStartup) (0041158c)
  Priority: 0 Priority class: 32 Affinity: 3
  1 Id: dac.2b0 Suspend: 1 Teb: 7efda000 Unfrozen
  Start: 00000001
  Priority: 0 Priority class: 32 Affinity: 3

0:000> !gle
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0
    
```

# WinDbgのウィンドウとメニュー

WinDbgのウィンドウは、ドッキングやフローティングが可能。

1) ドッキングウィンドウ = 複数のウィンドウ使用時におすすめ

- WinDbgの枠に合わせて伸び縮みする。
- 枠を変えると、それに合わせて位置やサイズが相対的に変わる。
- タブ化できる。タブ付きのウィンドウは前後に重なり合う。
- WinDbgは複数のドックを提供。マルチモニタ環境に重宝。
- Ctrl-Tabで、全ドック内の全ウィンドウが順次切り替わる。

2) フローティングウィンドウ（ドッキングなし）

- 常にWinDbgのウィンドウの前面に表示される。

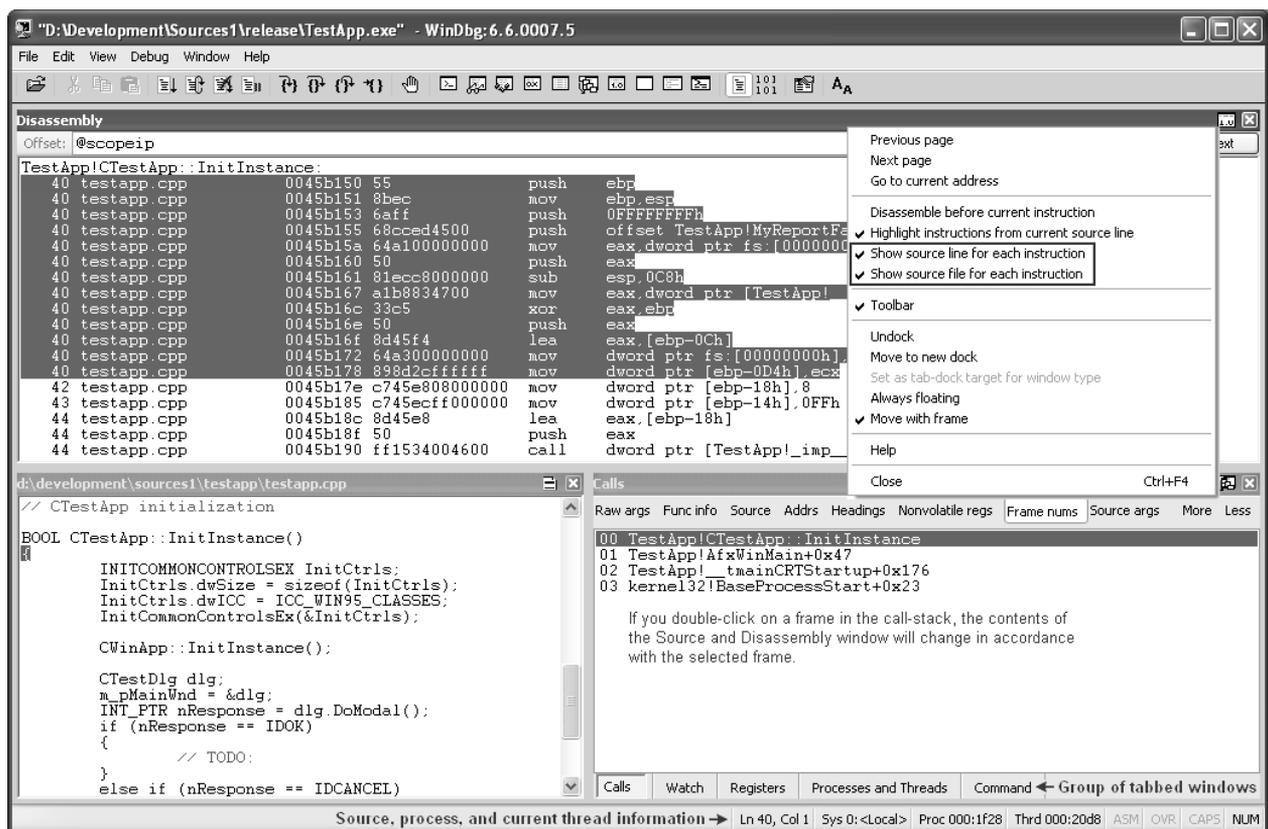
WinDbgの各ウィンドウは、それぞれ独自のメニューをもつ。

メニューの表示方法

- メニューボタン（[閉じる] ボタンの隣）を左クリック
- ウィンドウのタイトルバーを右クリック
- タブ付きウィンドウのタブを右クリック

各メニューを試してみるとよい。興味深い隠し機能があることが多い。

# WinDbgでのインスタンス実行例



# デバッガマークアップ言語(DML)

- DMLを使うと、タグ形式の中に指示や追加の非表示情報を埋め込み、デバッガに出力できる。
- デバッガのユーザーインターフェースが追加の情報を抜き出し、それに応じて動作する。
- DMLのおもな目的を次に示す。
  - 関連情報へのリンク
  - デバッガや拡張DLLがもつ機能の情報提示
  - デバッガや拡張DLLの出力の強化
- DMLはデバッグツールのバージョン6.6.0.7で導入された。

DMLコマンド	説明
.dml_start	各種DMLコマンドの開始。
.prefer_dml 1	グローバルな設定: すべてのDML強化コマンドがDMLを出力。
.help /D a*	.helpをDMLモードで実行。先頭にリンクを表示。
.chain /D	.chainをDMLモードで実行。拡張DLLを.extmatchコマンドにリンク。
その他の各種コマンドについては「..\¥Debugging Tools for Windows¥dml.doc」を参照。	

# WinDbgのDML

The screenshot shows the WinDbg 6.6.0007.5 command window. The command prompt shows the following sequence of commands and output:

```
0:001> .prefer_dml 1
DML versions of commands on by default
0:001> k
# ChildEBP RetAddr
00 0320ffc8 7c9507a8 ntdll!DbgBreakPoint
01 0320fff4 00000000 ntdll!DbgUiRemoteBreakin+0x2d
0:001> .dml_start
Analyze last event

Browse process information
Browse core commands
Browse extension chain
Browse dot commands
0:001> .help /D a*
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z All

. commands matching a*:
.abandon - abandon the current process
.allow_exec_cmds [0|1] - control execution commands
.allow_image_mapping [0|1] - control on-demand image file mapping
.apply_dbp [<options>] - add current data breakpoint state to a
register context
.asm [<options>] - set disassembly options
.asm- [<options>] - clear disassembly options
.attach <proc> - attach to <proc> at next execution

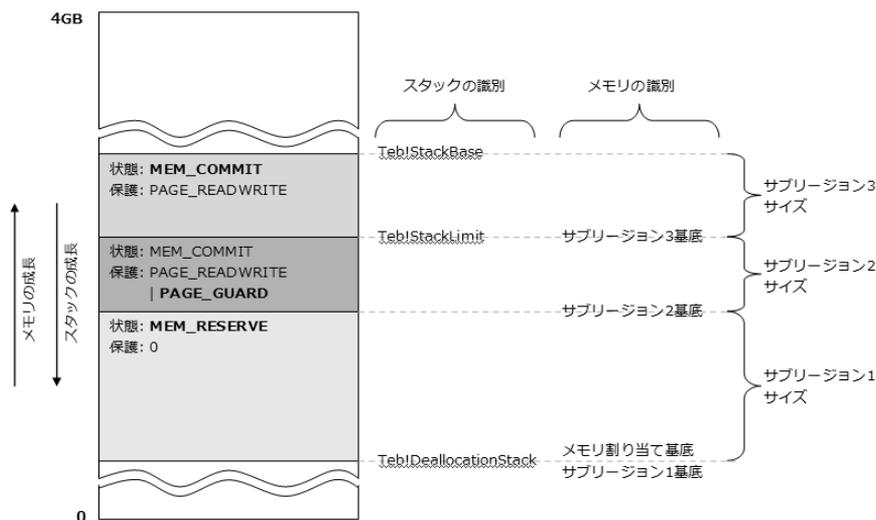
0:001>
```

A context menu is overlaid on the text "Browse core commands", with the following options:

- Open
- Open in New Window
- Copy Command

- リンクをクリックできる。
- リンクを右クリックすると、新しいウィンドウでコマンドを実行できる。

# メモリ: スタックの詳細



- MSDNより

- 新しいスレッドには、コミットメモリと予約メモリから構成される独自のスタック空間が与えられる。
- デフォルトでは、各スレッドは1MBの予約メモリと1ページのコミットメモリを使用。
- システムは、必要に応じて、予約スタックメモリから1ページずつコミットしていく。(MSDNのCreateThread > dwStacksize > 「Thread Stack Size」を参照)

## 例 - スレッドのスタックサイズ

```

0:000> !teb
TEB at 7ffdf000
    ExceptionList: 0012f784
    StackBase: 00130000
    StackLimit: 0012c000
    ...

0:000> dt ntdll!_TEB DeallocationStack 7ffdf000
+0xe0c DeallocationStack : 0x00030000

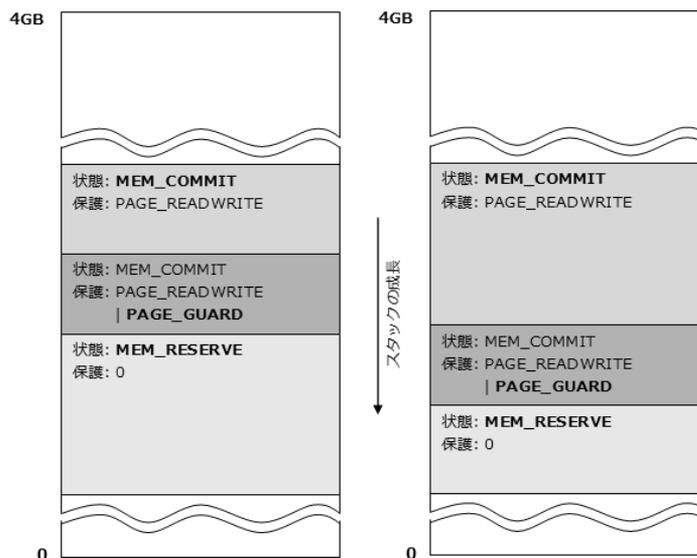
0:000> !address esp
AllocBase : SubRegionBase - SubRegionSize
00030000 : 0012c000 - 00004000
    Type 00020000 MEM_PRIVATE
    Protect 00000004 PAGE_READWRITE
    State 00001000 MEM_COMMIT
    Usage RegionUsageStack
    Pid.Tid e34.e78

0:000> ? 00130000 - 0012c000
Evaluate expression: 16384 = 00004000

0:000> ? 00130000 - 00030000
Evaluate expression: 1048576 = 00100000

0x004000 → 本スレッドは4ページ (16KB) のコミットメモリをもつ。
0x100000 → 本スレッドは256ページ (1MB) の予約メモリをもつ。
    
```

# メモリ：スタックの成長



- ESPレジスタは、スレッドの現在のスタックの位置を指す。
- プログラムがガードページ内のアドレスにアクセスしようとする時、システムは STATUS\_GUARD\_PAGE\_VIOLATION(0x80000001)例外を発生させる。ガードページは、ページのアクセスに対して、一度だけアラームを上げる。
- スタックが予約メモリの最後に達すると、STATUS\_STACK\_OVERFLOWが発生する。

39

www.windbg.info

## 例 - スタックの成長

```
0:000> !teb
TEB at 7ffdf000
ExceptionList: 0012f784
StackBase: 00130000
StackLimit: 0012c000
...
```

```
0:000> dt ntdll!_TEB DeallocationStack 7ffdf000
+0xe0c DeallocationStack : 0x00030000
```

```
0:000> ? 00130000 - 0012c000
Evaluate expression: 16384 = 00004000
```

**0x004000** →本スレッドは4ページ（16KB）のコミットメモリをもつ。  
**0x100000** →本スレッドは256ページ（1MB）の予約メモリをもつ。

```
0:000> !teb
TEB at 7ffdf000
ExceptionList: 0012f784
StackBase: 00130000
StackLimit: 00033000
...
```

```
0:000> ? 00130000 - 00033000
Evaluate expression: 1036288 = 000fd000
```

**0x0fd000** → 本スレッドのコミットメモリが253ページになった。  
さらにページを要求すると、システムはスタックオーバーフロー例外を投げる。

40

www.windbg.info

# コールスタック情報を取得するためのWinDbgコマンド

コマンド	説明
!uniqstack	カレントプロセスの全スレッドのコールスタックを表示
!findstack MySymbol 2	MySymbolを含む全コールスタックを探す
K	カレントスレッドのコールスタックを表示
kP	P == 各関数呼び出しの全パラメータ
Kf	f == 隣り合うフレームの距離を表示 (各フレームのスタック消費量のチェックに便利)
Kv	v == FOP情報と呼び出し規約を表示
Kb	b == 各関数に渡された最初の3つのパラメータを表示
kM	DML形式で出力。 フレーム番号に.frame/dvコマンドがリンクされているため、 そのフレームのローカル変数を表示できる。

## 例 - UniqStack

```
0:000> !uniqstack
Processing 2 threads, please wait
. 0 Id: dac.154c Suspend: 1 Teb: 7efdd000 Unfrozen
  Start: TestApp!ILT+1415(_wWinMainCRTStartup) (0041158c)
  Priority: 0 Priority class: 32 Affinity: 3

ChildEBP RetAddr
002df44c 00411eeb ntdll!DbgBreakPoint
002df52c 783c2100 TestApp!CMYDialog::OnBnClicked_ExecuteBreakPoint+0x2b [d:¥TestApp¥MyDialog.cpp @ 72]
002df570 783c2842 MFC80UD!_AfxDispatchCmdMsg+0xb0
002df5d4 7839d671 MFC80UD!CCmdTarget::OnCmdMsg+0x2e2
002df610 7836142d MFC80UD!CDialog::OnCmdMsg+0x21
...
002dfb8 0041371d TestApp!__tmainCRTStartup+0x289 [f:¥sp¥vctools¥crt_bld¥self_x86¥crt¥src¥crtexe.c @ 589]
002dff0 7d4e992a TestApp!wWinMainCRTStartup+0xd [f:¥sp¥vctools¥crt_bld¥self_x86¥crt¥src¥crtexe.c @ 414]
002dff0 00000000 kernel32!BaseProcessStart+0x28

. 1 Id: dac.127c Suspend: 1 Teb: 7efda000 Unfrozen
  Start: 00000001
  Priority: 0 Priority class: 32 Affinity: 3

ChildEBP RetAddr
0242f550 7d626c3f ntdll!NtQueryAttributesFile+0x12
...
0242ff08 7d62b958 ntdll!LdrpCallInitRoutine+0x14
0242ffbc 7d674613 ntdll!LdrShutdownThread+0xd2
0242ffc4 7d665017 ntdll!RtlExitUserThread+0xa
0242fff4 00000000 ntdll!DbgUiRemoteBreakin+0x41

Total threads: 2
```

# メモリ操作のためのWinDbgコマンド

コマンド	説明
d, dd, da, du, ...	メモリの表示 dd == ダブルワード値 da == ASCII文字を表示 du == UNICODE文字を表示
f	メモリを指定の値で埋める
!vprot MyAddr	アドレスMyAddrの仮想メモリ保護情報を表示
!address MyAddr	アドレスMyAddrが示すメモリの情報（タイプ、保護、使用状況, ...）を表示
!address -RegionUsageStack	プロセスの全スレッドのスタック領域を表示
dds	メモリの内容と、対応するシンボルを表示
ddp	メモリの内容と、参照先のメモリの内容を表示。 既知のシンボルと一致した場合は、そのシンボルも表示。

## 例 - プロセスのメモリ情報

```

0:000> !address
00000000 : 00000000 - 00010000
          Type      00000000
          Protect   00000001 PAGE_NOACCESS
          State     00010000 MEM_FREE
          Usage     RegionUsageFree
00010000 : 00010000 - 00001000
          Type      00020000 MEM_PRIVATE
          Protect   00000004 PAGE_READWRITE
          State     00001000 MEM_COMMIT
          Usage     RegionUsageEnvironmentBlock
...
----- Usage SUMMARY -----
TotSize (      KB)  Pct (Tots)  Pct (Busy)  Usage
1950000 ( 25920) : 01.24%   36.03%    : RegionUsageIsVAD
7b9b1000 ( 2025156) : 96.57%   00.00%    : RegionUsageFree
12e2000 ( 19336) : 00.92%   26.88%    : RegionUsageImage
110000 ( 1088) : 00.05%   01.51%    : RegionUsageStack
2000 ( 8) : 00.00%   00.01%    : RegionUsageTeb
2a0000 ( 2688) : 00.13%   03.74%    : RegionUsageHeap
1658000 ( 22880) : 01.09%   31.81%    : RegionUsagePageHeap
1000 ( 4) : 00.00%   00.01%    : RegionUsagePeb
1000 ( 4) : 00.00%   00.01%    : RegionUsageProcessParameters
1000 ( 4) : 00.00%   00.01%    : RegionUsageEnvironmentBlock
Tot: 7fff0000 (2097088 KB) Busy: 0463f000 (71932 KB)
...
----- State SUMMARY -----
TotSize (      KB)  Pct (Tots)  Usage
2efa000 ( 48104) : 02.29%    : MEM_COMMIT
7b9b1000 ( 2025156) : 96.57%    : MEM_FREE
1745000 ( 23828) : 01.14%    : MEM_RESERVE

```

# ヒープ情報取得のためのWinDbgコマンド

コマンド	説明
!heap -?	簡単なヘルプ
!heap -h	ヒープのインデックスや範囲 (=開始アドレス, 終了アドレス)を一覧表示
!heap -s 0	全ヒープの要約 = 予約メモリ, コミットメモリ, ...
!heap -flt s Size	指定のサイズ(Size)に一致する割り当てを表示
!heap -stat	ヒープハンドルを一覧表示 ヒープハンドル = HeapCreateやGetProcessHeapの戻り値
!heap -stat -h 0	メモリ割り当てサイズごとに使用状況の統計を表示 = 各割り当てサイズごとの、サイズ、ブロック数、合計メモリ
!heap -p	GFlagsの設定, ヒープハンドルの一覧
!heap -p -all	プロセスの全ヒープの全割り当ての詳細 = すべてのHeapAlloc呼び出しの一覧
!heap -p -a UserAddr	UserAddr (HeapAllocが返却したアドレス) を含むヒープ割り当ての詳細。取得できれば、バックトレースも表示。

45

www.windbg.info

## ヒープの構造

アプリケーションのページヒープが無効な場合の構造は次の通り。

デフォルトでは、ページヒープは無効。

- **\_HEAP構造**

- ntdll.dllで定義: dt ntdll!\_HEAP
- HeapCreateごとに、一意の\_HEAPが存在
- "!heap -p -all"により、プロセス内の全\_HEAP構造のアドレスが取得できる

- **\_HEAP\_ENTRY構造**

- ntdll.dllで定義: dt ntdll!\_HEAP\_ENTRY
- HeapAllocごとに、一意の\_HEAP\_ENTRYが存在
- "!heap -p -all"により、プロセス内の全ヒープエントリのアドレスが取得できる

46

www.windbg.info

# ページヒープの構造

アプリケーションのページヒープが有効な場合の構造は次の通り。  
ページヒープは、グローバルフラグ(gflags.exe)で有効にできる。

- **\_DPH\_HEAP\_ROOT**構造

- ntdll.dllで定義: dt ntdll!\_DPH\_HEAP\_ROOT
- HeapCreateごとに、一意の\_DPH\_HEAP\_ROOTが存在
- "!heap -p -all"により、プロセス内の全ヒープのルートのアドレスが取得できる
  - \_DPH\_HEAP\_ROOTのアドレスは、通常、ヒープハンドルの値 + 0x1000

- **\_DPH\_HEAP\_BLOCK**構造

- ntdll.dll で定義: dt ntdll!\_DPH\_HEAP\_BLOCK
- HeapAllocごとに、一意の\_DPH\_HEAP\_BLOCKが存在
- "!heap -p -all"により、プロセス内の全ヒープブロックのアドレスが取得できる



# HeapAllocの呼び出し元は？

- アプリケーションのスタックトレースとページヒープを有効化する
  - GFlagsを起動し、目的のイメージに対して「Create user mode stack trace database」と「Enable page heap」を選択
  - または、コマンドラインから次のように入力: gflags.exe /i <IMAGE.EXE> +ust +hpa
- アプリケーションを再起動し、WinDbgをアタッチ

WinDbgのコマンドラインにて

- **!heap -p -a <UserAddr>**
  - <UserAddr> = 割り当てアドレス (HeapAlloc, newなどの戻り値)
  - コールスタックを一覧表示。ソース情報なし。
- **dt ntdll!\_DPH\_HEAP\_BLOCK StackTrace <MyHeapBlockAddr>**
  - <MyHeapBlockAddr> = 事前に取得しておいたDPH\_HEAP\_BLOCKのアドレス
  - StackTrace = HeapAllocのコールスタックが格納されている、DPH\_HEAP\_BLOCKのメンバー
- **dds <StackTrace>**
  - <StackTrace> = 事前に取得しておいた値
  - コールスタックを一覧表示。ソース情報付き。

# 例 - HeapAllocの呼び出し元は？

```
// HeapAlloc( 0x00150000, 8, dwBytes = 0x00A00000 ) -->> 0x025F1000;

0:000> !heap -p -a 0x025F1000
address 025f1000 found in
_DPH_HEAP_ROOT @ 151000
in busy allocation ( DPH_HEAP_BLOCK: UserAddr UserSize - VirtAddr VirtSize)
                          15449c: 25f1000 a00000 - 25f0000 a02000
7c91b298 ntdll!RtlAllocateHeap+0x00000e64
0045b8b1 TestApp!CMyDlg::OnBnClicked_HeapAlloc+0x00000051
004016e0 TestApp!_AfxDispatchCmdMsg+0x00000043
004018ed TestApp!CCmdTarget::OnCmdMsg+0x00000118
00408f7f TestApp!CDialog::OnCmdMsg+0x0000001b
...

0:000> dt ntdll!_DPH_HEAP_BLOCK StackTrace 15449c
+0x024 StackTrace : 0x0238e328 _RTL_TRACE_BLOCK

0:000> dds 0x0238e328
0238e328 abcdaaaa
...
0238e334 00000001
0238e338 00a00000
0238e33c 00151000
0238e340 01b17b1c
0238e344 0238e348
0238e348 7c91b298 ntdll!RtlAllocateHeap+0xe64
0238e34c 0045b8b1 TestApp!CMyDlg::OnBnClicked_HeapAlloc+0x51 [d:%development%sources%TestApp%MyDlg.cpp @ 366]
0238e350 004016e0 TestApp!_AfxDispatchCmdMsg+0x43 [f:%sp%vctools%vc7%libs%ship%atlmfc%src%mfcc%cmdtarg.cpp @ 82]
0238e354 004018ed TestApp!CCmdTarget::OnCmdMsg+0x118 [f:%sp%vctools%vc7%libs%ship%atlmfc%src%mfcc%cmdtarg.cpp @ 381]
0238e358 00408f7f TestApp!CDialog::OnCmdMsg+0x1b [f:%sp%vctools%vc7%libs%ship%atlmfc%src%mfcc%dlgcore.cpp @ 85]
...
```

# HeapCreateの呼び出し元は？

- アプリケーションのスタックトレースとページヒープを有効化する
  - GFlagsを起動し、目的のイメージに対して、「Create user mode stack trace database」と「Enable page heap」を選択
  - または、コマンドラインから次のように入力: gflags.exe /i <IMAGE.EXE> +ust +hpa
- アプリケーションを再起動し、WinDbgをアタッチ

WinDbgのコマンドラインにて

- **!heap -p -h <HeapHandle>**
  - <HeapHandle> = HeapCreateの戻り値
  - "!heap -stat"または"!heap -p"で、プロセスのヒープとそのハンドルが一覧表示できる
- **dt ntdll!\_DPH\_HEAP\_ROOT CreateStackTrace <MyHeapRootAddr>**
  - <MyHeapRootAddr> = 事前に取得しておいたDPH\_HEAP\_ROOTのアドレス
  - CreateStackTrace = HeapCreateのコールスタックが格納されている、DPH\_HEAP\_ROOTのメンバー
- **dds <CreateStackTrace>**
  - <CreateStackTrace> = 事前に取得しておいた値
  - コールスタックを一覧表示。ソース情報付き。

51

www.windbg.info

# 例 - HeapCreateの呼び出し元は？

```
// HeapCreate( 0x0000000A, 0, 0 ) -->> 0x03000000;

0:000> !heap -p -h 0x03000000
   _DPH_HEAP_ROOT @ 3001000
   Freed and decommitted blocks
   DPH_HEAP_BLOCK : VirtAddr VirtSize
   Busy allocations
   DPH_HEAP_BLOCK : UserAddr UserSize - VirtAddr VirtSize
   ...

0:000> dt ntdll!_DPH_HEAP_ROOT CreateStackTrace 3001000
+0x08c CreateStackTrace : 0x0238e328 _RTL_TRACE_BLOCK

0:000> dds 0x0238e328
0238e328 abcdaaaa
0238e32c 00000001
0238e330 00000010
0238e334 00000000
0238e338 00000000
0238e33c 00000000
0238e340 00000000
0238e344 0238e348
0238e348 7c93a874 ntdll!RtlCreateHeap+0x41
0238e34c 7c812bff kernel32!HeapCreate+0x55
0238e350 0045b841 TestApp!CMydlg::OnBnClicked_HeapCreate+0x31 [d:\development\sources\TestApp\MyDlg.cpp @ 345]
0238e354 0040b122 TestApp!_AfxDispatchCmdMsg+0x43 [f:\sp\vc7\tools\vc7\lib\ship\atlmfc\src\mfc\cmdtarg.cpp @ 82]
0238e358 0040b32f TestApp!CCmdTarget::OnCmdMsg+0x118 [f:\sp\vc7\tools\vc7\lib\ship\atlmfc\src\mfc\cmdtarg.cpp @ 381]
0238e35c 00408838 TestApp!CDialog::OnCmdMsg+0x1b [f:\sp\vc7\tools\vc7\lib\ship\atlmfc\src\mfc\dlgcore.cpp @ 85]
...
```

52

www.windbg.info

# ヒープ上のメモリリークの検出

- **!address -summary**
  - プロセスのメモリ使用状況を要約。**RegionUsageHeap**や**RegionUsagePageHeap**が常に増加している場合は、ヒープのメモリリークがあるかもしれない。以下のステップへ。
- アプリケーションのスタックトレースとページヒープを有効にする。
- アプリケーションを再起動し、WinDbgをアタッチ。

WinDbgのコマンドラインにて

- **!heap -stat -h 0**
  - メモリ割り当てサイズごとに使用状況の統計を表示。各割り当てサイズごとに、サイズ、ブロック数、合計メモリが表示される。
- **!heap -flt -s <size>**
  - <size> = HeapAllocで割り当てられたサイズ。事前に取得しておいた値。
- **!heap -p -a <UserAddr>**
  - <UserAddr> = 割り当てアドレス (HeapAlloc, newなどの戻り値)
  - コールスタックを一覧表示。ソース情報なし。ソース情報付きのコールスタックを表示するには、「HeapAllocの呼び出し元は？」のスライドを参照のこと。

## 例 - ヒープ上のメモリリークの検出

```
0:001> !heap -stat -h 0
Allocations statistics for
heap @ 00150000
group-by: TOTSIZE max-display: 20
size      #blocks  total      ( %) (percent of total busy bytes)
100000    101      - 10100000(99.99)  → 0x101 * 1MB 割り当てられている。メモリリークの発生が推測される。
928       2        - 1250      (0.00)
64        24       - e10      (0.00)
...

0:001> !heap -flt s 100000          → サイズが100000の割り当てについて表示
_DPH_HEAP_ROOT @ 151000
Freed and decommitted blocks
DPH_HEAP_BLOCK : VirtAddr VirtSize
Busy allocations
DPH_HEAP_BLOCK :   UserAddr  UserSize  - VirtAddr  VirtSize
024f0698 :          13831000  00100000  - 13830000  00102000
024f0620 :          13721000  00100000  - 13720000  00102000
... → ここに、サイズが100000のエントリが0x101個表示される。
      最初のエントリUserAddr=0x13831000について調査。

0:001> !heap -p -a 13831000
address 13831000 found in
_DPH_HEAP_ROOT @ 151000
in busy allocation ( DPH_HEAP_BLOCK:   UserAddr  UserSize  - VirtAddr  VirtSize)
                          24f0698:          13831000  100000  - 13830000  102000
7c91b298 ntdll!RtlAllocateHeap+0x00000e64
0045b74e TestApp!CMyDlg ::OnBnClicked_DoMemoryLeak+0x0000003e
0040b122 TestApp!_AfxDispatchCmdMsg+0x00000043
0040b32f TestApp!CCmdTarget ::OnCmdMsg+0x00000118
00408838 TestApp!CDialog ::OnCmdMsg+0x0000001b
...
```

# クリティカルセクション関連のコマンド

コマンド	説明
!locks	プロセス内のロックされているクリティカルセクションを一覧表示
!locks -v	プロセス内の全クリティカルセクションを表示
!cs [オプション] [クリティカルセクションのアドレス]	ひとつ以上のクリティカルセクション、あるいは全クリティカルセクションのツリーを表示  オプション: -l == ロックされたクリティカルセクションのみを表示 -s == 各クリティカルセクションの初期化スタックを表示 -o == 所有者のスタックを表示 -t == クリティカルセクションのツリーを表示 → EnterCnt, WaitCnt, ...
!avrf -cs	削除されたクリティカルセクションを表示 (DeleteCriticalSection API)

## 例 - クリティカルセクション

```

0:000> !cs -s -o 0x0012fe08
-----
Critical section      = 0x0012fe08 (+0x12FE08)
DebugInfo            = 0x031c4fe0
LOCKED
LockCount            = 0x0
OwningThread         = 0x00000c8c
...
OwningThread Stack =
  ChildEBP RetAddr  Args to Child
  0012f488 004badd9 0012f810 02854f10 00000000 ntdll!DbgBreakPoint
  0012f568 0054fd2c 00000000 004bb621 00681d70 TestApp!CMyDialog::OnBnClicked_EnterCs+0x39
  0012f594 00550365 0012fd78 0000001c 00000000 TestApp!_AfxDispatchCmdMsg+0x9c
  0012f5f0 005517f1 0000001c 00000000 00000000 TestApp!CCmdTarget::OnCmdMsg+0x285
  ...
Stack trace for DebugInfo (Initialization Stack)= 0x031c4fe0:
0x7c911a93: ntdll!RtlInitializeCriticalSectionAndSpinCount+0xc9
0x7c809eff: kernel32!InitializeCriticalSection+0xE
0x004c101d: TestApp!CCriticalSection::Init+0x3D
0x004c10a0: TestApp!CCriticalSection::CCriticalSection+0x40
...

0:000> !cs -t
Verifier package version >= 3.00
Tree root 02fd8fd0
Level   Node      CS      Debug   InitThr EnterThr WaitThr TryEnThr LeaveThr EnterCnt WaitCnt
-----
  0 02fd8fd0 0012fe08 031c4fe0   c8c    c8c     0       0       0         1       0
  1 02fa8fd0 006807f4 03148fe0   c8c     0       0       0       0         0       0
  2 02fa2fd0 00680f70 02850fe0   c8c    c8c     0       0     c8c     4848     0
...

```

# その他の便利なWinDbgコマンド

コマンド	説明
dt	ローカル変数、関数のパラメータ、グローバル変数、データ型の情報を表示
dt ntdll!*peb*	pebという文字を含むntdll.dllの全変数を一覧表示
dt ntdll!_PEB	PEBの型を表示
dt ntdll!_PEB 7efde000	アドレス7efde000のPEBを表示
dv	ローカル変数を表示
dv /t /i /V	ローカル変数を表示 /i == カテゴリで分類 (パラメータかローカル変数か) /V == アドレスと、フレームのベースレジスタからのオフセットを表示 (通常はEBP) /t == 型情報を表示

## 例 - dt と dv

```

0:000> dt TestApp!CMyDialog
+0x000 __VFN_table          : Ptr32
=00400000 classCObject      : CRuntimeClass
=00400000 classCCmdTarget   : CRuntimeClass
=00400000 _commandEntries  : [0] AFX_OLECMDMAP_ENTRY
=00400000 commandMap       : AFX_OLECMDMAP
=00400000 _dispatchEntries : [0] AFX_DISPATCHMAP_ENTRY
...
+0x004 m_dwRef              : Int4B
+0x008 m_pOuterUnknown     : Ptr32 IUnknown
+0x00c m_xInnerUnknown     : UInt4B
+0x010 m_xDispatch        : CCmdTarget::XDispatch
+0x014 m_bResultExpected   : Int4B
+0x018 m_xConnPtContainer  : CCmdTarget::XConnPtContainer
+0x01c m_pModuleState      : Ptr32 AFX_MODULE_STATE
=00400000 classCWnd        : CRuntimeClass
+0x020 m_hWnd              : Ptr32 HWND__
...
+0x064 m_lpDialogInit      : Ptr32 Void
+0x068 m_pParentWnd        : Ptr32 CWnd
+0x06c m_hWndTop           : Ptr32 HWND__
+0x070 m_pOccDialogInfo    : Ptr32 _AFX_OCC_DIALOG_INFO
+0x074 m_hIcon             : Ptr32 HICON__
+0x078 m_nn                : Int4B

0:000> dv /t /i /V
prv local 002df440 @ebp-0x08 class CMyDialog * this = 0x002dfe24
prv param 002df450 @ebp+0x08 int nn = 1
    
```

# WinDbgの疑似レジスタ

- デバッガが仮想的なレジスタを提供
- ドル記号(\$)で始まる

## 1) 自動疑似レジスタ

- デバッガが、便利な値を設定
- 例: \$ra, \$peb, \$teb, ...

## 2) ユーザー定義の疑似レジスタ

- 20個のユーザー定義レジスタ: \$t0, \$t1, ..., \$t19
- 中間データの格納には整数型の変数が使われる
- 型情報も保持できる
- r?は、型付きの結果を左辺値(lvalue)に割り当てる
  - r? \$t0=@\$peb->ProcessParameters
    - 型付きの値を\$t0に割り当て
    - \$t0の型が記憶されており、後の式で利用可能
  - ?? @\$t0->CommandLine

# 自動疑似レジスタ

コマンド	説明
\$ra	スタック上にあるリターンアドレス 実行コマンドで便利, 例: "g \$ra"
\$ip	命令ポインタ x86 = EIP, Itanium = IIP, x64 = RIP
\$exentry	カレントプロセスで最初に行われるエントリポイント
\$retreg	戻り値のレジスタ x86 = EAX, Itanium = ret0, x64 = rax
\$csp	コールスタックのポインタ x86 = ESP, Itanium = BSP, x64 = RSP
\$peb	プロセス環境ブロック(PEB)のアドレス
\$teb	カレントスレッドのスレッド環境ブロック(TEB)のアドレス
\$tpid	プロセスID(PID)
\$tid	スレッドID(TID)
\$ptrsize	ポインタのサイズ
\$pagesize	1ページ分のメモリのバイト数
...	WinDbgのヘルプの「Pseudo-Registry Syntax」を参照

# WinDbgの式

## 1) MASM式

- ? コマンドで評価
- 各シンボルはアドレスとして扱われる (シンボルの値は、そのシンボルのメモリアドレス → 値を取り出すにはpoiオペレータで逆参照する)
- `ソース:行番号`の式が利用可能(`myfile.c:43`)
- レジスタには、アットマークを付けても付けなくてもよい (eaxでも@eaxでも可)
- WinDbgのヘルプでは、ほとんどすべてにMASM式が使われている
- WinDbg 4.0より前のバージョンでは、これが唯一の式

## 2) C++式

- ?? コマンドで評価
- シンボルは適切なデータ型で解釈される
- `ソース:行番号`の式は利用不可
- レジスタにはアットマークが必要 (eaxは不可)

MASMの演算は、常にバイト単位で行われる。C++の演算は、C++型のルールに従う (ポインタ演算時のスケールリングを含む)。いずれも、内部では数値はULONG64値で処理される。

# さらに式について

## ● MASM

- シンボルの値は、そのシンボルのメモリアドレス
- 任意の数に対して、任意の演算子が使用可能
- 数字: 現在の基数にしたがって解釈される: n [8 | 10 | 16]  
接頭辞で基数を変更可能: 0x(16進), 0n(10進), 0t(8進), 0y(2進)

## ● C++

- 変数の値は、その変数もつ実際の値そのもの
- 演算子は、対応するデータ型でのみ使用可能
- 対応するC++データ型をもたないシンボルは、文法エラーになる
- データ構造は、実際の構造体として適切に使用すること。数値はもたない。
- 関数名やその他のエントリポイントはメモリアドレスであり、関数のポインタとして扱われる
- 数字: デフォルトは、常に10進数  
接頭辞で変更可能: 0x(16進), 0(8進)

## 例 - 変数の値

```
// -----  
void MyFunction() {  
    int nLocalVar = 7;  
    ... }  
// -----  
  
0:000> dd nLocal1 L1  
0012f830 00000007  
  
// MASM syntax  
// -----  
0:000> ? nLocalVar                // nLocalVarのアドレス (メモリ上の位置) を取得  
Evaluate expression: 1243184 = 0012f830  
  
0:000> ? dwo(nLocalVar)           // nLocalVarの値を取得 - 逆参照  
Evaluate expression: 7 = 00000007 // (dwo = ダブルワード, poi = ポインタサイズのデータ)  
  
0:000> ? poi(nLocalVar)  
Evaluate expression: 7 = 00000007  
  
// C++ syntax  
// -----  
0:000> ?? nLocalVar              // nLocalVarの値を取得  
int 7  
  
0:000> ?? &nLocalVar             // nLocalVarのアドレス (メモリ上の位置) を取得  
int * 0x0012f830
```

## 例 - MASM式 対 C++式

```
// -----  
// 以下の例では、次の値を返す:  
//   eax == ebx ? 0  
//   eax > ebx ? 1  
//   eax < ebx ? -1  
//  
// C++文法の次の点に注意:  
//   -> レジスタの値を表すには@が必要  
//   -> BOOLをintに変換するには明示的なキャストが必要  
//   -> 等価演算子は、二重のイコール記号  
// -----  
  
0:000> r eax = 4, ebx = 3  
  
0:000> ? 0*(eax = ebx) + 1*(eax > ebx) + -1*(eax < ebx)  
Evaluate expression: 1 = 00000001  
  
0:000> ?? 0*(int)(@eax == @ebx) + 1*(int)(@eax > @ebx) + -1*(int)(@eax < @ebx)  
int 1  
  
0:000> r eax = 3, ebx = 4  
  
0:000> ? 0*(eax = ebx) + 1*(eax > ebx) + -1*(eax < ebx)  
Evaluate expression: -1 = ffffffff  
  
0:000> ?? 0*(int)(@eax == @ebx) + 1*(int)(@eax > @ebx) + -1*(int)(@eax < @ebx)  
int -1
```

# MASMの一般的な数値演算子

## 単項演算子

演算子	説明
dwo, qwo, poi	dwo = 指定のアドレスからDWORD分を取得 qwo = 指定のアドレスからQWORD分を取得 poi = 指定のアドレスからポインタサイズ分を取得
wo, by	wo = 指定のアドレスから下位WORD分を取得 by = 指定のアドレスから下位BYTE分を取得

## 二項演算子

演算子	説明
= (または ==), !=	等しい, 等しくない
<, >, <=, >=	より小さい, より大きい, 以下, 以上
and (または &), xor (または ^), or (または  )	ビットごとの論理積, ビットごとの排他的論理和, ビットごとの論理和
+, -, *, /	加算, 減算, 乗算, 除算
<<, >>, >>>	左シフト, 右シフト, 算術右シフト

# MASMの非数値演算子

演算子	説明
\$iment(Address)	イメージのエントリポイント。Address=イメージのベースアドレス
\$scmp("String1", "String2")	-1, 0, 1のいずれかを返却。strcmpを参照。
\$sicmp("String1", "String2")	-1, 0, 1のいずれかを返却。stricmpを参照。
\$spat("String", "Pattern")	TRUE → StringがPatternにマッチする FALSE → StringがPatternにマッチしない Pattern = エイリアスまたは文字列の定数。メモリポインタは指定不可（すなわち、\$spatの引数に直接"poi(address)"を指定することはできない。最初にその結果をエイリアスに格納のこと）。Patternに、各種ワイルドカードを使用可能。
\$vvalid(Address, Length)	1 → 指定範囲のメモリは有効 0 → メモリは無効

シンボルの検索に無駄な時間をかけないために

- MASM

- レジスタに@を付けることを推奨。そうしないと、シンボルとして解釈されるかもしれない。

- C++

- ローカルなシンボルのための接頭辞: \$!MySymbol
- グローバルなシンボルのための接頭辞:  
<モジュール名>!MySymbol

## 例 - C++文法の構造体

```
// -----  
// 処理の高速化に有効: $!symName          ... ローカルシンボルの場合  
//          <ModuleName>!symName      ... グローバルシンボルの場合  
// -----  
0:000> ?? dlg.m_nn  
int 0  
0:000> ?? $!dlg.m_nn  
int 0  
0:000> ?? sizeof($!dlg.m_nn)  
unsigned int 0xac  
  
0:000> ?? ((MyModule!CMyDlg*) 0x12f878)->m_nn  
int 0  
  
0:000> ?? ((ntdll!_TEB*) 0x7ffdf000)->ClientId  
struct _CLIENT_ID  
+0x000 UniqueProcess : 0x000017d8      // → PID  
+0x004 UniqueThread : 0x00000ea8      // → TID  
  
0:000> ?? @$teb->ClientId                // C++の式評価器は、  
struct _CLIENT_ID                        // 疑似レジスタを適切な型にキャストする  
+0x000 UniqueProcess : 0x000017d8  
+0x004 UniqueThread : 0x00000ea8  
  
0:001> r? $t0 = @$peb->ProcessParameters // ユーザー定義の議事レジスタには  
0:001> ?? @$t0->CommandLine              // 型情報が保持される点に注意  
struct _UNICODE_STRING  
""D:¥Development¥Sources¥CrashMe¥release¥CrashMe.exe" "  
+0x000 Length : 0x6a  
+0x002 MaximumLength : 0x6c  
+0x004 Buffer : 0x00020724 ""D:¥Development¥Sources¥CrashMe¥release¥CrashMe.exe" "
```

## 例 - ポインタ演算

```
// -----  
// int myInt[2] = { 1,2 };  
// MASMの演算は常にバイト単位で行われる点に注意。  
// 一方、ポインタの計算はC++の方式で行われている。  
// -----  
  
// MASM 文法  
// -----  
0:000> ? myInt  
Evaluate expression: 1243256 = 0012f878  
0:000> ? dwo(myInt)  
Evaluate expression: 1 = 00000001  
  
0:000> ? myInt+4  
Evaluate expression: 1243260 = 0012f87c  
0:000> ? dwo(myInt+4)  
Evaluate expression: 2 = 00000002  
  
// C++ 文法  
// -----  
0:000> ?? (&myInt)  
int * 0x0012f878  
0:000> ?? myInt  
int [2] 0x0012f878  
1  
  
0:000> ?? (&myInt+1)  
int * 0x0012f87c  
0:000> ?? *(&myInt+1)  
int 2
```

69

www.windbg.info

## デフォルトの式評価器

- 次の場合は常にC++の評価器が使われる
  - ?? コマンド (C++式を評価)
  - watchウィンドウ
  - localsウィンドウ
- 上記以外のすべてのコマンドやデバッグ情報ウィンドウでは、デフォルトの式評価器が使われる
- .exprコマンドでデフォルトの評価器を変更可能
  - .expr → 現在の評価器を表示
  - .expr /q → 使用可能な評価器を表示
  - .expr /s c++ → デフォルトの式評価器としてC++を設定
  - .expr /s masm → デフォルトの式評価器としてMASMを設定

70

www.windbg.info

# 両評価器の混在

- ひとつのコマンド内で両方の式評価器を使用可能
- 両モードを混在させるには: @@(...)
  - 式の一部が、@@とそれに続く丸カッコに囲まれていた場合、その中は現在の式評価器とは逆の式評価器で評価される。
  - これにより、ひとつのコマンド内の異なるパラメータで、2つの異なる式評価器が使用可能。
  - この記号を入れ子で使うことも可能。記号が現れるたびに、式評価器が切り替わる。
- 式評価器の明示的な指定
  - @@c++(...)
  - @@masm(...)

## 例 - 式評価器の混在

```
// -----  
// 以下のコマンドは、デフォルトの式評価器をMASMに設定する。  
// そのため、Expression1とExpression3をMASM式として評価される。  
// 一方、Expression2はC++式として評価される。  
// -----  
0:000> .expr /s masm  
0:000> ? Expression1 + @@( Expression2 ) + Expression3  
  
0:000> ? `myFile.cpp:118` // myFile.cppの118行目のアドレスを取得  
Evaluate expression: 4570359 = 0045bcf7  
  
// -----  
// ソース:行番号の式は、C++式では使えないので、  
// C++式の中でMASM式を入れ子にするとよい。  
// → myFile.cppの118行目のアドレスを、nLocalVarに格納。  
// -----  
  
0:000> ?? nLocalVar = @@(`myFile.cpp:118`)  
int 4570359
```

# WinDbgのエイリアス

- 文字列を、自動的にほかの文字列に置き換える
- エイリアス名 + 対応する値から構成される

## 1) ユーザー命名エイリアス

- ユーザーが名前と値を設定 (いずれも大文字小文字が区別される)
- `as` または `aS` (エイリアスの設定)、`ad` (エイリアスの削除)、`al` (エイリアスの一覧表示) で操作

## 2) 固定名エイリアス

- ユーザーが値を設定。名前は `$u0`, `$u1`, ..., `$u9`
- `r` (登録) コマンド + "u"の前の. (ドット) で設定  
例: `r $.u0 = "dd esp+8;g"`

## 3) 自動エイリアス

- デバッガが名前と値を設定
- 自動疑似レジスタに類似。ただし、`${...}` といったエイリアス関連のトークンが使える点が異なる (疑似レジスタでは使えない)。
- 例: `$ntsym`, `$CurrentDumpFile`, `$CurrentDumpPath`, ...

# ユーザー命名エイリアスと固定名エイリアス

## 1) ユーザー命名エイリアス

- デフォルトでは、ユーザー命名エイリアスは、ほかの文字と離さなければならない。エイリアス名の最初と最後の文字は、次のいずれかの条件を満たすこと。
  - 行頭または行末
  - 前後が、空白かセミコロンかクエスチョンマーク
- ユーザー命名エイリアスがほかのテキストと隣接する場合は、`${ }` (エイリアスインタープリタ) で囲まなければならない
- 固定名エイリアスの定義内で使用可能
  - ユーザー命名エイリアスを、ほかのユーザー命名エイリアスの定義内で使う場合は、`as` コマンドまたは `aS` コマンドの前にセミコロンを付ける (そうしないと、行内でのエイリアスの置き換えは行われない)。解説: `as`, `aS`, `ad`, `al` で始まる行内のテキストは、エイリアスの置き換えが行われない。置き換えが必要な場合は、先頭にセミコロンを付けること。
- 固定名エイリアスより使いやすい
  - エイリアス定義の文法が単純
  - `al` (List Aliases) コマンドで一覧表示が可能

## 2) 固定名エイリアス

- ほかのテキストと隣接していても、自動的に置き換えられる
- 任意のエイリアス定義内で使用可能

# ユーザー命名エイリアスのコマンド

演算子	説明
as Name Equivalent as /ma Name Address as /mu Name Address ...	エイリアスの設定 Addressにあるnullで終わるASCII文字列にエイリアスを設定 Addressにあるnullで終わるUnicode文字列にエイリアスを設定
ad Name ad *	名前(Name)を指定してエイリアスを削除 すべてのエイリアスの削除
al	ユーザー命名エイリアスを一覧表示
<code>\${Alias}</code>  <code>\$/f:Alias</code>  <code>\$/n:Alias</code> <code>\$/d:Alias</code>	ほかのテキストに隣接している場合も、 <code>\${Alias}</code> をエイリアスの値に置き換える。エイリアスが未定義の場合は、 <code>\${Alias}</code> の置き換えは行われない。 上と同様だが、エイリアスが未定義の場合は、 <code>\$/f:Alias</code> が空の文字列に置き換えられる。 エイリアス名自体を返す。 評価: 1=エイリアス定義済み, 0=エイリアス未定義

## 例 - エイリアス

```
0:001> as Short kernel32!CreateRemoteThread // → ユーザー命名エイリアス
0:001> uf Short

0:001> r $.u0 = kernel32!CreateRemoteThread // → 固定名エイリアス
0:001> uf $u0

0:001> as DoInc r eax=eax+1; r ebx=ebx+1 // → エイリアスをコマンドのマクロとして使用
0:001> DoInc
0:001> DoInc

// -----
// エイリアスを使用するとその場で置き換えらる
// -----
0:001> r $.u2 = 2
0:001> r $.u1 = 1+$u2
0:001> r $.u2 = 6
0:001> ? $u1
Evaluate expression: 3 = 00000003

0:001> as two 2
0:001> r $.u1 = 1+ two // → twoの前に空白があるので注意
0:001> as two 6
0:001> ? $u1
Evaluate expression: 3 = 00000003

// -----
// エイリアスをほかのエイリアスの中で使う
// -----
0:001> as two 2
0:001> as xy1 two + 1 // → xy1 = two + 1
0:001> ;as xy2 two + 1 // → xy2 = 2 + 1 (エイリアスを置き換えるにはasの前にセミコロンが必要)
```

# デバッガコマンドプログラム

- 構成
  - デバッガコマンド
  - フロー制御トークン(.if, .for, .while, ...)
- 変数
  - ユーザー命名エイリアスまたは固定名エイリアスを、「ローカル変数」として使用
  - 疑似レジスタ(\$t0, ...)を、数値や型付きの変数として使用
- コメントは \$\$\${任意のテキスト} のように記述
- 中カッコ { } で、ステートメントのブロックを囲む
  - 各ブロックに入る際に、ブロック内の全エイリアスが評価される
  - 開きカッコの前に、フロー制御トークンが必要
  - エイリアスの評価のために単なるブロックを作るには、.block {...} を使用
  - ほかのテキストと隣接するユーザー命名エイリアスには、\${Alias} (エイリアスインタープリタ) を使用

## フロー制御トークン

- 繰り返しや条件分岐の作成に使用
- 各条件は式でなければならない (コマンドは許可されていない)

コマンド	説明
.block	何の動作も行わない。単にブロックの導入にのみ使われる。{ }だけではブロックを作成できないので注意。
.if, .else, .elseif	C言語のキーワードのif, else, else ifとほぼ同じ
.for, .while, .break, .continue	C言語のキーワードのfor, while, break, continueとほぼ同じ
.foreach	デバッガコマンド (文字列またはテキストファイル) の出力を解析し、解析で得た各項目を後続のデバッガコマンドの入力に使用する。

# コマンドプログラムの実行

## プログラムの実行方法

- 全ステートメントを、デバッガウィンドウに、ひとつの文字列として入力（コマンドはセミコロンで区切る）
- 全ステートメントをスクリプトファイルに保存し、**\$\$><**を使ってファイルを実行

**\$\$><**（スクリプトファイルの実行）：

- 指定のファイルを開く
- すべての改行をセミコロンに置き換える
- その結果を、ひとつのコマンドブロックとして実行

## 例 - デバッガコマンドプログラム

```
$$ -----
$$ WinDbgのヘルプ「Debugger Command Program Examples」から引用
$$ 説明はヘルプ内にあり。
$$ -----

$$ モジュールリストLIST_ENTRY を$t0に取得
r? $t0 = &@$peb->Ldr->InLoadOrderModuleList

$$ リスト内の全モジュールを列挙
.for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)$t0;
 (@$t1 != 0) & (@$t1 != @$t0);
 r? $t1 = (ntdll!_LDR_DATA_TABLE_ENTRY*)@$t1->InLoadOrderLinks.Flink)
{
    $$ ベースアドレスを$Baseに取得
    as /x ${/v:$Base} @c++(@$t1->DllBase)

    $$ フルネームを$Modに取得
    as /msu ${/v:$Mod} @c++(&@$t1->FullDllName)

    .block
    {
        .echo ${/v:$Mod} at ${/v:$Base}
    }

    ad ${/v:$Base}
    ad ${/v:$Mod}
}
}
```

# 便利なブレークポイントコマンド

コマンド	説明
bl	ブレークポイントの一覧表示
bp	ブレークポイントの設定
bu	未解決のブレークポイントの設定。 ブレークポイントの実際の設定を、モジュールロード時まで保留する。
ba	メモリアクセス時に中断
bc	ブレークポイントの消去
be, bd	ブレークポイントの有効化, 無効化

## 例 - 単純なブレークポイントの設定

```
0:000> bu kernel32!LoadLibraryExW
0:000> bu kernel32!CreateProcessW
0:000> bu kernel32!CreateThread
0:000> ba r4 0012fe34          → アクセス時に中断(read or write); 4バイト監視
0:000> ba w2 0012fe38          → アクセス時に中断(write); 2バイト監視

0:000> bl
   0 e 7c801af1      0001 (0001) 0:**** kernel32!LoadLibraryExW
   1 e 7c802332      0001 (0001) 0:**** kernel32!CreateProcessW
   2 e 7c810637      0001 (0001) 0:**** kernel32!CreateThread
   3 e 0012fe34 r 4  0001 (0001) 0:****
   4 e 0012fe38 2 2  0001 (0001) 0:****

0:000> bd 0,2                  → 0番と2番のブレークポイントを無効化
0:000> bc 4                    → 4番のブレークポイントを消去

0:000> bl
   0 d 7c801af1      0001 (0001) 0:**** kernel32!LoadLibraryExW
   1 e 7c802332      0001 (0001) 0:**** kernel32!CreateProcessW
   2 d 7c810637      0001 (0001) 0:**** kernel32!CreateThread
   3 e 0012fe38 r 4  0001 (0001) 0:****
```

## 例 - より複雑なブレークポイント

- 指定したソースの指定した行で中断

```
0:000> bp `mod!source.c:12`
```

- 5回通過後からヒットするブレークポイント

```
0:000> bu kernel32!LoadLibraryExW 5
0:001> bl // → 3回通過後 (0002=残りカウント)
0 e 7c801af1 0002 (0005) 0:*** kernel32!LoadLibraryExW
```

- スレッド1から呼び出された場合にのみ中断

```
0:000> ~1 bu kernel32!LoadLibraryExW
0:001> bl
0 e 7c801af1 0001 (0001) 0:~001 kernel32!LoadLibraryExW
```

- パターンmyFunc\*にマッチする全シンボルで中断

```
0:000> bm /a mod!myFunc*
```

- xコマンドのシンボルパターンと同様

- メンバーのメソッドで中断

```
0:000> bp @@c++( MyClass::MyMethod )
```

- 同じメソッドがオーバーロードされているため複数のアドレスが存在する場合に便利

## 例 - コマンド付きブレークポイント

- WinMainの実行をスキップ

```
0:000> bu MyApp!WinMain "r eip = poi(@esp); r esp = @esp + 0x14; .echo WinSpy!WinMain entered; gc"
```

- 関数のエントリポイントに入った瞬間、スタックの先頭にリターンアドレスがセットされている。
  - r eip=poi(@esp) → EIP (命令ポインタ) にスタックのオフセット0の内容をセット
- WinMainのパラメータ4個×各4バイト (=0x10バイト) +リターンアドレス4バイト=0x14
  - r esp=@esp + 0x14 → ESPに0x14を加算し、スタックポインタを適切に巻き戻す

- スレッド1から呼び出された場合のみ中断

```
0:000> bu kernel32!LoadLibraryExW ";as /mu ${/v:MyAlias} poi(@esp+4); .if ( $spat(¥"${MyAlias}¥", ¥"*MYDLL*¥" ) != 0 ) { kn; } .else { gc }"
```

- LoadLibraryの最初のパラメータ (アドレスESP+4) は、当該DLL名へのポインタ。
- このポインタと、所定のワイルドカード付き文字列 (この例では \*MYDLL\*) を、MASMの\$spat演算子で比較。
- \$spatはエイリアスや定数は受け取れるものの、メモリのポインタは受け取れない。そこで、当該ポインタをあらかじめエイリアス(MyAlias)に格納しておく。
- このkernel32!LoadLibraryExWブレークポイントは、\$spatのパターンマッチングに成功した場合にのみヒットする。それ以外の場合は、アプリケーションは実行を継続する。

# 例外分析コマンド

コマンド	説明
.lastevent	ファーストチャンスかセカンドチャンスか
!analyze -v	現在の例外に関する詳細情報を表示
.exr -1	直近の例外を表示
.exr Addr	アドレスAddrの例外レコードを表示
!cppexr	C++の例外レコードを表示
g, gH	例外をハンドリングして実行
gN	例外をハンドリングせずに実行

## 例 - 例外

```
0:000> .lastevent
Last event: dac.154c: Stack overflow - code c00000fd (first chance)
debugger time: Wed Aug 29 16:04:15.367 2007 (GMT+2)

0:000> .exr -1
ExceptionAddress: 00413fb7 (TestApp!_chkstk+0x00000027)
ExceptionCode: c00000fd (Stack overflow)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000000
Parameter[1]: 001e2000

0:000> !analyze -v
FAULTING_IP:
TestApp!_chkstk+27 [F:¥SP¥vctools¥crt_bld¥SELF_X86¥crt¥src¥intel¥chkstk.asm @ 99]
00413fb7 8500 test dword ptr [eax],eax
...
```

# WinDbgによるリモートデバッグ

- ターゲットコンピュータ（サーバー）
  - dbgsrv.exe, dbgeng.dll, dbghelp.dllをリモートコンピュータにコピーする
  - 「dbgsrv.exe」に対するファイアウォールを無効化する
  - 実行 → dbgsrv.exe -t tcp:port=1025  
Windows Vista: 全プロセスが参照できるようdbgsrv.exeを管理者権限で開始
- ホストコンピュータ（クライアント）
  - 実行 → WinDbg.exe -premote tcp:server=ターゲットのIPアドレスまたは名前,port=1025
  - File（メニュー） → Attach to Process → ターゲットコンピュータのデバッグ対象プロセスを選択

## リモートデバッグのためのWinDbgコマンド

コマンド	説明
Cdb.exe -QR サーバー（IPアドレスまたは名前）	指定のネットワークサーバー上のすべてのデバッグサーバーを一覧表示
.detach	プロセスからデタッチ
.endpsrv	リモートコンピュータ上のdbgsrv.exeを終了。デバッグ中のプロセスは、事前にデタッチしていない場合、強制終了される。
.tlist	（リモート）システム上で実行中の全プロセスを一覧表示

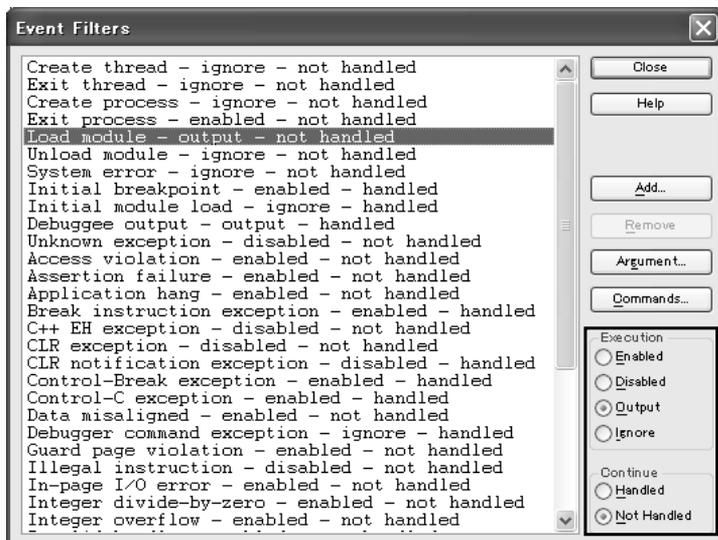
# イベントの監視

- デバッガエンジンが、対象アプリケーション内のイベントの監視機能と応答機能を提供
- イベントは大きく二種類に分けられる
  - 例外イベント  
ブレークポイント、アクセス違反、スタックオーバーフロー、0除算など。  
すべての一覧は、ヘルプの「Specific Exceptions」を参照。
  - 非例外イベント  
プロセスの生成、スレッドの生成、モジュールのロード、モジュールのアンロード。  
すべての一覧は、ヘルプの「DEBUG\_FILTER\_XXX」を参照。
- 次のコマンドで最後に発生したイベントを取得。デバッグ時には必ず何らかの値がセットされている。
  - コマンド: `.lastevent`

# WinDbgのイベントフィルタ

- 簡単なイベントフィルタを提供
- ターゲットにてイベント発生後の、デバッガエンジンの動作に影響
- 全イベントの一覧表示: `sx`
- 中断または実行状態
  - イベント発生時にデバッガがターゲットを中断するかどうかに影響
    - ファーストチャンスで中断(`sxe`)
    - セカンドチャンスで中断(`sxd`)
    - イベントに関するデバッガのメッセージを出力(`sxn`)
    - イベントを無視(`sxi`)
- ハンドリングまたは継続状態
  - ターゲット内の例外イベントをハンドリングするか(`gH`)しないか(`gN`)を決定

# イベントフィルタダイアログ



実行:

- Enabled - ファーストチャンスで中断(sxe)
- Disabled - セカンドチャンスで中断(sxd)
- Output - イベント発生でメッセージ出力(sxn)
- Ignore - イベントを無視(sxi)

継続:

- Handled - 実行再開時、イベントをハンドリングする
- Not-Handled - 実行再開時、イベントをハンドリングしない

# イベントの引数

- いくつかのフィルタは、マッチするイベントを制限するための引数をもつ
- 引数なし → 制限なし

イベント	マッチの基準
CreateProcess	生成プロセスの名前が、引数にマッチすること。
ExitProcess	終了プロセスの名前が、引数にマッチすること。
ロードモジュール	ロードモジュールの名前が、引数にマッチすること。
ターゲットの出力	ターゲットからのデバッグ出力が、引数にマッチすること。
アンロードモジュール	アンロードモジュールのベースアドレスが、引数と一致すること。

ワイルド  
カード文字  
列の文法

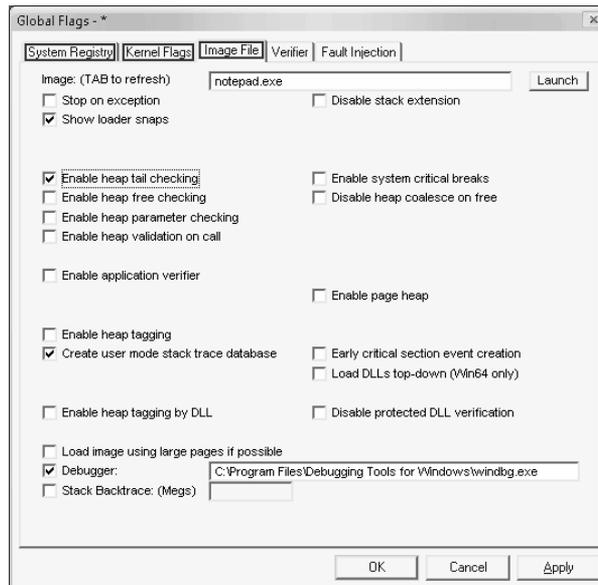
## 目次 - ロードマップ

- ✓ WinDbgの裏側
- ✓ WinDbgの使い方
- **グローバルフラグ**
- アプリケーション検証
- プロセスダンプ

## フラグ? Gフラグ? グローバルフラグ!

- GFlagsは、Windowsのレジストリの編集により、機能を有効化・無効化する
- GFlagsは、システム全体に対する設定も、イメージに特化した設定も可能
- イメージに特化した設定は、次の場所に格納される
  - HKLM¥SOFTWARE¥Microsoft¥Windows NT¥CurrentVersion¥**Image File Execution Options¥ImageFileName¥GlobalFlag**
- OSはこれらの設定を読み、その内容に応じて動作する
- GFlagsは、コマンドラインからも、ダイアログボックスからも設定できる
- WinDbg内でも !gflags により、グローバルフラグの設定や表示が可能
- GFlagsで次の機能を有効にできる
  - ヒープのチェック
  - ヒープのタグ付け
  - ローダーのスナップ表示
  - イメージに対するデバッガ指定（イメージ開始のたびに自動的にアタッチされる）
  - アプリケーション検証
  - その他

# GFlagsダイアログ



- System Registry: システム全体の設定。Windowsで実行中の全プロセスに影響する。設定内容は、設定を変更するまでずっと効果を発揮。設定内容を実際に有効にするには、Windowsの再起動が必要。
- Kernel Flags: 実行時の設定。システム全体に影響する。再起動しなくても、すぐに効果を発揮。システムのシャットダウンや再起動により、設定内容は失われる。
- Image File: コマンド完了後に起動した指定プログラムのインスタンスに影響。設定内容はレジストリに格納され、設定変更するまでずっと効果を発揮。

95  
www.windbg.info

## GFlags: 「Show loader snaps」有効化

### WinDbg の出力:

```
LDR: LdrLoadDll, loading samlib.dll from
C:\WINDOWS\system32;C:\WINDOWS\system32;C:\WINDOWS;.;C:\WINDOWS\System32\Wbem;C:\WINDOWS\system32\ktools
LDR: Loading (DYNAMIC, NON_REDIRECTED) C:\WINDOWS\system32\samlib.dll
ModLoad: 71bf0000 71c03000 C:\WINDOWS\system32\samlib.dll
LDR: samlib.dll bound to ntdll.dll
LDR: samlib.dll has correct binding to ntdll.dll
LDR: samlib.dll bound to ADVAPI32.dll
LDR: samlib.dll has correct binding to ADVAPI32.dll
LDR: samlib.dll bound to RPCRT4.dll
LDR: samlib.dll has correct binding to RPCRT4.dll
LDR: samlib.dll bound to KERNEL32.dll
LDR: samlib.dll has stale binding to KERNEL32.dll
LDR: samlib.dll bound to ntdll.dll via forwarder(s) from kernel32.dll
LDR: samlib.dll has correct binding to ntdll.dll
LDR: Stale Bind KERNEL32.dll from samlib.dll
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap
LDR: LdrGetProcedureAddress by NAME - RtlFreeHeap
LDR: LdrGetProcedureAddress by NAME - RtlGetLastWin32Error
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: samlib.dll bound to USER32.dll
LDR: samlib.dll has stale binding to USER32.dll
LDR: Stale Bind USER32.dll from samlib.dll
[d58,690] LDR: Real INIT LIST for process C:\Development\Sources\TestApp\Release\TestApp.exe pid 3416 0xd58
[d58,690] C:\WINDOWS\system32\samlib.dll init routine 003A0F30
[d58,690] LDR: samlib.dll loaded - Calling init routine at 003A0F30
```

96  
www.windbg.info

- ✓ WinDbgの裏側
- ✓ WinDbgの使い方
- ✓ グローバルフラグ
- ➔ **アプリケーション検証**
- プロセスダンプ

## アプリケーション検証の有効化

- アプリケーション検証
  - Windowsアプリケーションの実行時検証ツール
  - アプリケーションとOSのインタラクションを監視
  - プロファイルと追跡
    - マイクロソフトWin32 API（ヒープ、ハンドル、ロック、スレッド、DLLロード/アンロード、ほか多数）
    - 例外
    - カーネルオブジェクト
    - レジストリ
    - ファイルシステム
  - !avrfにより、この追跡情報を入手

注意: アプリケーション検証は、裏側で、対象アプリケーションに多くのDLL(verifier.dll, vrfcore.dll, vfbasics.dll, vfcompat.dll, ほか多数)を注入する。より正確に言うと、選択したテスト項目にしたがって、該当イメージのレジストリキーを設定し、Windowsローダーがこのレジストリキーを読み込み、アプリケーションの開始時にそのアドレス空間に所定のDLLをロードする。

# アプリケーション検証の種類

## GFlagsアプリケーション検証

- 対象プロセスに、verifier.dllのみを注入
- verifier.dllはWindows XPにインストール済み
- アプリケーション検証オプションの機能制限版
- GFlagsのこのオプションは旧式なため、将来削除されるかもしれない

## アプリケーション検証

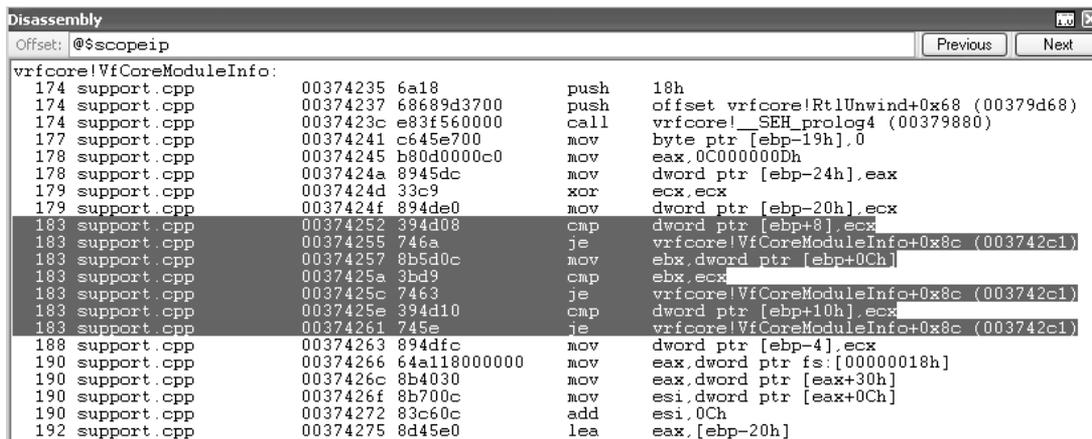
- マイクロソフトのウェブサイトから無料でダウンロードし、インストールできる
- vrfcore.dll, vfbasics.dll, vfcompat.dllなども、Windows¥System32に追加インストールされる
- GFlagsより多くのテストオプションが利用でき、!avr拡張の全機能が使える

# アプリケーション検証のシンボル

アプリケーション検証は、全シンボル情報を含むPDBとともにインストールされる

- 逆アセンブリに含まれるソース情報に注目
- 私の知る限り、全シンボル情報付きで配布されたマイクロソフトのモジュールはこれだけ
- WinDbgは、サーバーに公開されているシンボルではなく、これらのシンボルを使う必要がある。そうしないと、!avr拡張は機能しない。

```
.reload /f @"C:¥Windows¥System32¥verifier.pdb"
```



```
Disassembly
Offset: @$scope!p
vrfcore!VfCoreModuleInfo:
174 support.cpp 00374235 6a18 push 18h
174 support.cpp 00374237 68689d3700 push offset vrfcore!RtlUnwind+0x68 (00379d68)
174 support.cpp 0037423c e83f560000 call vrfcore!__SEH_prolog4 (00379880)
177 support.cpp 00374241 c645e700 mov byte ptr [ebp-19h],0
178 support.cpp 00374245 b80d0000c0 mov eax,0C000000Dh
178 support.cpp 0037424a 8945dc mov dword ptr [ebp-24h],eax
179 support.cpp 0037424d 33c9 xor ecx,ecx
179 support.cpp 0037424f 894de0 mov dword ptr [ebp-20h],ecx
183 support.cpp 00374252 394d08 cmp dword ptr [ebp+8],ecx
183 support.cpp 00374255 746a je vrfcore!VfCoreModuleInfo+0x8c (003742c1)
183 support.cpp 00374257 8b5d0c mov ebx,dword ptr [ebp+0Ch]
183 support.cpp 0037425a 3bd9 cmp ebx,ecx
183 support.cpp 0037425c 7463 je vrfcore!VfCoreModuleInfo+0x8c (003742c1)
183 support.cpp 0037425e 394d10 cmp dword ptr [ebp+10h],ecx
183 support.cpp 00374261 745e je vrfcore!VfCoreModuleInfo+0x8c (003742c1)
188 support.cpp 00374263 894dfc mov dword ptr [ebp-4],ecx
190 support.cpp 00374266 64a118000000 mov eax,dword ptr fs:[00000018h]
190 support.cpp 0037426c 8b4030 mov eax,dword ptr [eax+30h]
190 support.cpp 0037426f 8b700c mov esi,dword ptr [eax+0Ch]
190 support.cpp 00374272 83c60c add esi,0Ch
192 support.cpp 00374275 8d45e0 lea eax,[ebp-20h]
```

# 一般的な !avrf パラメータ

コマンド	説明
!avrf	現在のアプリケーション検証オプションを表示。アプリケーション検証停止が発生した場合は、停止の状況が表示される。
!avrf -cs	クリティカルセクションの削除(*)ログを表示 * DeleteCriticalSection API. ~CCriticalSectionはこれを暗黙的に呼び出す。
!avrf -hp 5	ヒープの操作ログ（直近の5つのエントリ）を表示 * HeapAlloc, HeapFree, delete
!avrf -dlls	DLLのロード/アンロードのログを表示
!avrf -ex	例外のログを表示
!avrf -cnt	グローバルカウンタを一覧表示(WaitForSingleObject呼び出し, CreateEvent呼び出し, ヒープ割り当て呼び出し, ...)
!avrf -threads	対象プロセスのスレッドの情報を表示。子スレッドについては、スタックサイズや親が指定したCreateThreadフラグも表示。
!avrf -trm	すべての終了(*)スレッドや中断されたスレッドのログを表示 * TerminateThread API

## 例 - !avrf

```
// アプリケーションが以下の命令を実行した直後:  
// HeapAlloc( 0x00140000, 8, dwBytes =0x00A00000 ) -->> 0x033D1000;  
  
0:000> !avrf -hp 1  
Verifier package version >= 3.00  
Dumping last 1 entries from tracker @ 01690fd8 with 1291 valid entries ...  
-----  
HeapAlloc: 33D1000 A00000 0 0  
004019cf: TestApp!CMyDialog::OnBnClicked_HeapAlloc+0x4F  
0041a0c1: TestApp!_AfxDispatchCmdMsg+0x3D  
0041a2a6: TestApp!CCmdTarget::OnCmdMsg+0x10A  
0041a76c: TestApp!CDialog::OnCmdMsg+0x1B  
0041d05c: TestApp!CWnd::OnCommand+0x51  
0041d92b: TestApp!CWnd::OnWndMsg+0x2F  
0041b2eb: TestApp!CWnd::WindowProc+0x22  
...  
  
0:000> !avrf -threads  
=====
```

**Thread ID = 0xDE4**  
Parent thread ID = 0xE3C  
Start address = 0x004a7d82: TestApp!ILT+11645(?ThreadProcYGKPAXZ)  
Parameter = 0x0061833c  
=====

**Thread ID = 0xE3C**  
**Initial thread**  
=====

Number of threads displayed: 0x2

- ✓ WinDbgの裏側
- ✓ WinDbgの使い方
- ✓ グローバルフラグ
- ✓ アプリケーション検証
- プロセスダンプ

## プロセスダンプ

- プロセスのダンプ
  - 非侵襲的アタッチに類似
  - プロセスのある瞬間のスナップショット
  - 含んでいるコンテンツや情報によって、サイズが異なる
- ダンプを使って
  - メモリやプロセスの内部構造の観察が可能
  - ブレークポイントの設定や、プログラムのステップ実行はできない
- ダンプをダンプ
  - 情報を多く含むダンプを、情報の少ないダンプに縮小できる
  - 動作中のプロセスをダンプするように、.dumpコマンドを使ってダンプをダンプできる

# ダンプの種類

## 1) カーネルモードダンプ

種類: 完全メモリダンプ、カーネルメモリダンプ、最小メモリダンプ

## 2) フルユーザーモードダンプ

- WinDbgの".dump /f"コマンドや、Windows 2000のDr.Watsonで生成される
- プロセスのメモリ空間全体、プログラム自身の実行イメージ、ハンドルテーブルを含む
- 従来は広く使われたが、マイクロソフトはサポートを止めつつある

## 3) ミニダンプ

- .dump /m??
- 最新のダンプフォーマット
- ダンプに含まれる情報を、細かい単位で制御できる (MSDNのMINIDUMP\_TYPEを参照)
- その名前に反して、最大のミニダンプファイルには、フルユーザーモードダンプよりも多くの情報が含まれる。たとえば、.dump /mfや.dump /maは、".dump /f"よりも大きく、より完全なダンプファイルを生成する。

# ダンプの種類の見分け方

```
Dump C:\my.dmp - WinDbg:6.6.0007.5
File Edit View Debug Window Help
Command
Loading Dump File [C:\my.dmp]
User Mini Dump File with Full Memory: Only application data is available

Windows XP Version 2600 (Service Pack 2) MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Sat Dec 1 14:10:50.000 2007 (GMT+1)
System Uptime: 0 days 0:16:07.306
Process Uptime: 0 days 0:01:16.000
Symbol search path is: srv*D:\Development\Symbols\MsSymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
.....
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
eax=7ffde000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c901230 esp=00bbffcc ebp=00bbfff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c901230 cc          int          3

0:001>
```

目的のダンプをWinDbgでロード。ミニダンプであれば「User Mini Dump File」と表示され、古いスタイルのクラッシュダンプであれば「User Dump File」と表示される。

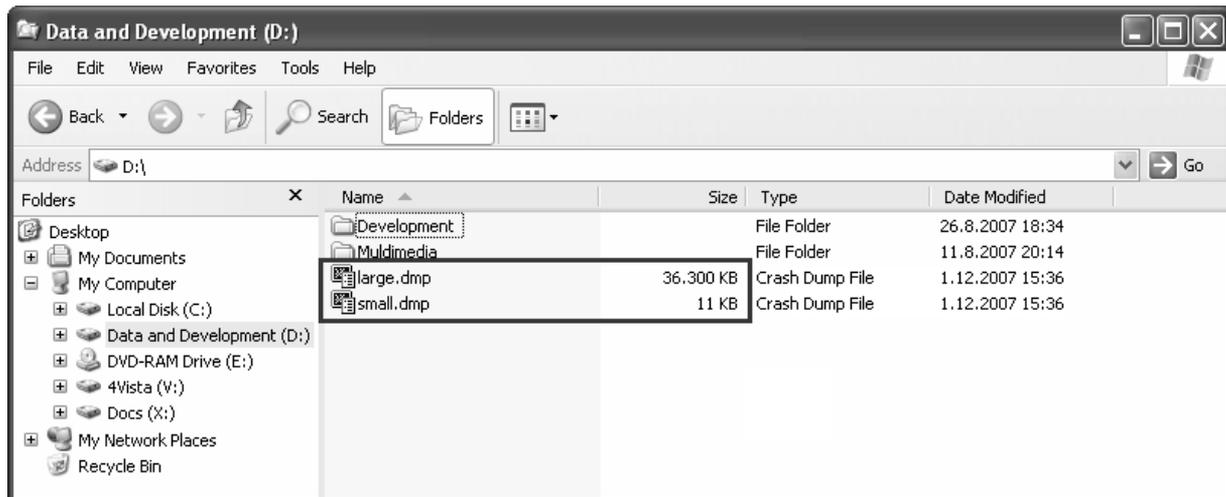
# 例 - ".dump"コマンド

```
0:000> .dump /ma d:%large.dump
Creating d:%large.dump - mini user dump
Dump successfully written
```

→ すべての情報: 全メモリ、コードセクション、PEBとTEB、  
ハンドルデータ、スレッド時間の情報、アンロードモジュールの一覧など

```
0:000> .dump /m d:%small.dump
Creating d:%small.dump - mini user dump
Dump successfully written
```

→ 基本情報のみ: モジュール情報 (署名) ,  
スレッドとスタックの情報



107  
www.windbg.info

# 最適なツールの選択

シナリオ/選択肢	ADPlus	Dr. Watson	CDBと WinDbg	UserDump
アプリケーションのクラッシュ (事後分析デバッグ)	○	○	○	○
アプリケーションのハング (無応答だが、クラッシュしているわけではない)	○	×	○	○
アプリケーションで例外発生	○	○	○	○
アプリケーションが正常に実行中	×	×	○	○
アプリケーションが起動時に失敗 (依存しているDLLがない)	×	×	○	○
既存のダンプファイルの縮小	×	×	○	×
同じイメージ名のすべての実行中アプリケーション に対して、ダンプを一度に取得	×	×	×	○
ダンプファイルに含める情報を制御	×	× <sup>1</sup>	○	× <sup>2</sup>

1: 常に基本情報のみの小さなミニダンプ(MiniDumpNormal)を生成。サイズは通常20KB未満。

2: 常に、全メモリ情報の入ったミニダンプを生成する。サイズは通常20~200MB。

108  
www.windbg.info

## 宿題

- WinDbgのドキュメントを読むこと
  - メモリリーク、ハンドル、デッドロック、条件付きブレークポイントなどについて、すべて説明してある。
- アセンブリ言語を学ぶこと
  - デバッグの技術が大幅に向上する。
  - デバッグが必要な状況においては、WinDbgのアセンブリが最良の友となるだろう。

## ご質問、ご提案は？



- WinDbgに関するご質問はありませんか。
- WinDbgラボやセミナーに興味をお持ちいただけましたか。
- 「WinDbg. From Ato Z!」に改善すべき点はありましたか。
- それとも「スゴイ！このプレゼンは実に役に立つ」と言ってもらえるでしょうか。
- お気軽にメールをお寄せください: [mailrkuster@windbg.info](mailto:mailrkuster@windbg.info)  
実際のメールアドレスは先頭の"mail"を除く(スパム防止用)
- 日本語のメールはこちらまで: [mailogunity@nttdata.co.jp](mailto:mailogunity@nttdata.co.jp)  
実際のメールアドレスは先頭の"mail"を除く(スパム防止用)

# 参考文献

- WinDbgのドキュメント, MSDN
- Common WinDbg Commands (Thematically Grouped)  
<http://software.rkuster.com/windbg/printcmd.htm>
- Matching Debug Information  
<http://www.debuginfo.com/articles/debuginfomatch.html>
- Generating Debug Information with Visual C++  
<http://www.debuginfo.com/articles/gendebuginfo.html>
- Microsoft Windows Internals, Fourth Edition  
M.E. Russinovich, D.A. Solomon, ISBN 0-7356-1917-4
- Advanced Kernel Debugging  
Andre Vachon, PowerPoint, WinHec 2004
- アプリケーション検証のドキュメント