

# Advanced Driver Debugging

# Goals

- Debugger overview
- Update on Windows Debuggers features
- Advanced debugging techniques
  - Focus on customizing and automating your debugger experience

# Outline

- Debugger overview
- Symbols
- !analyze
- Debugger Command Programs
- Debugger Extensions
- Remote Debugging

# Debugger Overview

# Documentation

- Read the documentation
  - Installed in the debugger root directory
- Reading the docs is key to using the debugger efficiently
  - Ever-increasing number of commands and command parameters
  - Very large number of debugging topics
  - No presentation or class can cover all the debugger features you care about
- Use the index for any topic you need more information on
  - Use search if the index does not list the topic
- New features are also listed on our web site

# Types of Debuggers

- Command line debuggers
  - kd.exe: kernel debugger
  - cdb.exe, ntsd.exe: user mode debugger
- WinDbg
  - GUI on top of kd.exe and cdb.exe
  - Identical extensions and command interface
  - Much more efficient for source debugging
  - Can be slower because of extra data displayed
- Dbgsvc.exe, kdsrv.exe, dbengprx.exe
  - Debugger protocol remoting tools
  - Discussed later as part of debugger remoting

# What Do the Debuggers Support?

- Processor architectures:
  - x86, Itanium, x64
- OS versions:
  - Windows NT 4 and later
    - No Win9x kernel debugging
  - Debugging the newest OS requires using the latest debugger
- Protocols
  - COM, 1394, EXDI, USB 2.0 in beta
- Can debug a Windows OS running inside the Virtual PC or VMware virtual machines

# Common Issues

- Local variable issues
  - Turn off compiler optimizations
    - `razzle no_opt`
- Breakpoint issues
  - Use the “bu” command to deal with unloading modules
- 1394 debugging won't connect
  - Disable 1394 host controller on the target
- COM port debugging won't connect
  - Disable legacy USB support in the BIOS
- If all else fails, you may have hit a debugger bug
  - Report it! We don't release until all known bugs are fixed.

# New Core Debugger Features

- New disassembly options
  - `ub`: disassemble before the current instruction, handy for seeing instructions leading up to a return address
  - `uf`: disassemble a function based on control flow
    - Convenient when you want to disassemble an entire function, displays source line over disassembly to show source/instruction mappings
    - Pulls functions that have been split up back into a single sequential form
- New symbol quoting
  - Resolve symbols with arbitrary text, no more problems with special characters confusing the evaluator
  - Specify wildcards in symbol names for easy matching
  - Specify function arguments in symbols for disambiguation of overloads

# More New Core Debugger Features

- p and t can now have command strings to execute when the step completes
  - Easy to build conditional stepping similar to conditional breakpoints
  - New gc command resumes previous execution in conditionals so that go/step/trace state is no longer lost
- .call command allows retargeting the current thread to make a specific call in the debuggee
  - Convenient way to cause arbitrary execution
  - Automatically builds call frames and handles return values
  - Can be dangerous
  - Currently user-mode x86 and x64 only

# More New Core Debugger Features

- `.fnret` displays source-formatted return values for a function
- `.printf` command provides printf-like formatted output
  - Much greater control over output
  - Naturally handles multiple output values
  - Extended formats: `%y` takes an address argument and displays the symbol for the address. More to come in the future
- Greater flexibility with `.shell` input and output
  - Can now spin off commands that the debugger doesn't wait for
  - Can now redirect output of a command to a shelled process for easy output filtering (redirect to Perl)

# More New Core Debugger Features

- More string commands
  - dpa and dpu indirect through memory and display the strings, like dps for string data. Great for looking at string tables
  - s-sa and s-su search for any readable strings in memory, like strings.exe
- New search options
  - Save search results and research them for refinement
  - User-mode search for only writable memory instead of all memory, much faster when searching for modifiable memory
  - Output raw search hits as a simple address for .foreach composition
- Symbol sorting options for dv and x
  - Sort by name or size instead of address, reverse sort order

# More New Core Debugger Features

- More flexible typed data handling
  - Temporary values (\$t0-\$t19) can now hold typed information
  - r? assigns a typed result to an lvalue
    - r? \$t0 = @\$peb->ProcessParameters
      - Assigns a typed value to \$t0.
    - ?? @\$t0->CommandLine
      - \$t0's type is remembered so it can be used in further typed expressions
- Many additions for command programs
  - Greatly expanded alias functionality to provide variable-like behavior
  - Many new pseudo-registers for programmability
    - \$fnsucc, \$ptrsize, \$pagesize, \$exr\_chance, \$exr\_code, \$exr\_param0-14, \$extret, \$scmp, \$sicmp, \$spat, \$vvalid

# New Core Extensions

- !address attempts to classify the given address
  - Intended to be fully general, kind of like !analyze for a specific address
  - Determines image vs. stack vs. pool, etc.
  - Handles both user and kernel mode
    - Not perfect, still being enhanced
- New iterator extensions
  - Use aliases to pass in lots of info for each step of the iteration
  - !for\_each\_process, !for\_each\_thread: run a given command for every process or thread
  - !for\_each\_processor: run a command for every processor in a kernel session
  - !for\_each\_module: run a command for every module

# New WinDbg Features

- Memory windows now auto-fit columns and can have explicit column control
- New memory window display formats for pointer and symbol (dps-like)
- Options for evaluating selected expressions in source and command output
- Scratch pad can now be associated with a file for automatic persistence
- Disassembly window can now display source line and file info for equating with source and checking optimizations

# More New WinDbg Features

- Process attach dialog can now display session and user name
- MRU list for remote connections
- Sound notification when idle
- Simple command prefix matching in the command history (like console F8)
- Several pre-built workspace “themes” to provide an initial window layout
  - Saves hassle of doing an initial window layout
  - Collected from submissions from real user workspaces
- Many others

# New Windows Server 2003 Features

- Kdbgctrl.exe
  - Tool to configure the kernel's debugging options
  - Run it on the target machine
  - Change behavior of DbgPrint, user mode int 3, DbgPrint buffer size, etc.

# Upcoming Longhorn Features

- Native assert mechanism
  - Native support for asserts (int 2c)
  - Debugger can manage asserts more effectively
- Kernel error report generation
  - API to create minidumps without crashing the system
  - Great for troubleshooting rare problems

# Symbols

# Symbol Server

- Structured storage for symbols and images
  - Optimized for fast and accurate symbol lookup
- Public Microsoft symbol server
  - <http://msdl.microsoft.com/download/symbols>
  - Symbols for all released MS OS binaries
    - If some files are missing, let us know
  - Core Microsoft OS images
    - Needed to debug minidump failures effectively
  - Can be used from WinDbg and Visual Studio

# Symbol Server and Minidumps

- Minidumps store the timestamp of images
  - Debugger uses the file name, timestamp and image size to map the image
  - Debugger looks for the symbol file name in the mapped image
  - If the wrong image is loaded by the debugger, the symbols will also be wrong
- Storing images and symbols in symbol server is the best way for the debugger to get the correct version of the image
  - Also simplifies archiving of driver versions

# IHV and ISV Symbols

- Symbols greatly help with the analysis of failures
  - Don't lose your symbols!
- Sharing symbols with Microsoft
  - You can submit symbols with driver submissions to WHQL
  - On-site vendors can host their own symbol server
  - Symbol data is stored securely
  - Sharing symbols is totally optional, but encouraged

# Building a Symbol Server

- You can build your own symbol server
- Take advantage of the Internet symbol server
  - Symbols downloaded from the internet symbol server are stored in a symbol server structure
- Augment with all of your symbols
  - Symstore stores symbol files into a symbol server tree
    - Symstore described in the debugger docs
  - Store all your symbols in the same location
    - Everyone can find them
    - No duplication
  - Store PDBs and images (.sys, .dll)
    - Debugging minidumps requires the images

# Source Server

- New infrastructure to automatically load the correct version of a source file
- Symbol file stores name and version of all source files used in building the image
  - Requires building with PDB files
  - Requires running post build scripts to append source indexing info to the PDB
  - Tools shipped with the debugger package
- Dependent on the source control system
  - Supporting additional source control systems only requires enhancing the tools and build scripts

# Symbol Enhancements

- symproxy: a symbol server request proxy
  - Similar to a web server proxy
  - Allows local caching to reduce load on primary server
  - Putting a proxy on a bridge server allows symsrv requests to cross network boundaries
- agestore: a new tool to manage local caches
  - Simple aging and cleanup of locally-stored files

**!analyze**

# What is !analyze?

- Debugger extension designed to find root cause of bugs
  - Automated analysis
  - Simplifies analysis of known problems
    - Understand various states of the OS
  - Provides good starting point to analyze complex problems
    - Extract commonly used debugging information
- Results of the analysis are
  - “Bucket ID”
    - Unique string representing the bug
  - An owner for the problem, extracted from triage.ini
  - In verbose mode
    - Detailed list of all the data found during the analysis

# !analyze Algorithm

- Multi-step algorithm
- Uses bugcheck, exception or verifier code as initial input
  - Can get directly from dump info and can also locate by stack scanning
  - Can run in hang analysis mode regardless of the current state
- Does stack analysis
- Uses additional data about known problems provided by developers
- Iterates on all the data above to determine the root cause

# Analysis Step One

- Use bugcheck or exception parameters to extract basic information
  - Each condition is processed by a separate routine that understands the meaning of each parameter
  - If specific follow-up or faulting code is found, report results
  - Save trap frame, context recording, faulting thread, etc.

# Analysis Step Two

- Use information in step one to get faulting stack
- Scan the stack for special functions such as Trap0E or UnhandledExceptionFilter to find alternate stack
- Analyze frames on the final stack to determine most likely culprit
  - Different weights are assigned to routines
    - Internal kernel routines have lowest weight
    - Device drivers have highest weight
    - Fine grain control provided by triage.ini
  - Highest weight frame found on the stack is treated as the culprit

# Symbols Make a Difference

## ● Without Symbols

f18e7968 nt!KeBugCheckEx+0x19  
f18e7980 nt!lopfCallDriver+0x18  
f18e7990 Fastfat!FatSingleAsync+0x74  
f18e7a5c Fastfat!FatCommonRead+0x88e  
f18e7acc Fastfat!FatFsdRead+0x136  
f18e7adc nt!lopfCallDriver+0x31  
**f18e7b0c SOMEDRV+0x61cb**  
f18e7b2c nt!IoPageRead+0x19  
f18e7b9c nt!MiDispatchFault+0x270  
f18e7bec nt!MmAccessFault+0x5b7  
f18e7bec nt!\_KiTrap0E+0xb8  
f18e7cc4 nt!CcMapData+0xef  
f18e7cf0 Fastfat!FatReadVolumeFile+0x38  
f18e7e78 Fastfat!FatMountVolume+0x1f7  
...

**BUCKET\_ID: 0x35\_SOMEDRV+61cb**

## ● With Symbols

f18e7968 nt!KeBugCheckEx+0x19  
f18e7980 nt!lopfCallDriver+0x18  
f18e7990 Fastfat!FatSingleAsync+0x74  
f18e7a5c Fastfat!FatCommonRead+0x88e  
f18e7acc Fastfat!FatFsdRead+0x136  
f18e7adc nt!lopfCallDriver+0x31  
**f18e7ae8 SOMEDRV!CSymIrp::IrpRead+0x4b**  
f18e7af8 nt!lopfCallDriver+0x31  
f18e7b0c nt!IoPageReadInternal+0xf2  
f18e7b2c nt!IoPageRead+0x19  
f18e7b9c nt!MiDispatchFault+0x270  
f18e7bec nt!MmAccessFault+0x5b7  
f18e7bec nt!\_KiTrap0E+0xb8  
f18e7cc4 nt!CcMapData+0xef  
f18e7cf0 Fastfat!FatReadVolumeFile+0x38  
f18e7e78 Fastfat!FatMountVolume+0x1f7  
...

**BUCKET\_ID: POOL\_CORRUPTION\_Foo.sys**

# Analysis Step Three

- If stack does not yield an interesting frame, analyze raw stack data
  - Iterate on all stack values using the same weight algorithm
  - The 'dps' command will show that output
- This finds code that corrupts the stack

# Analysis Step Four

- Check for presence of memory or pool corrupting drivers
- Check for corrupted code streams
  - Bad RAM
- Check for other possible problems, such as invalid call sequences
  - Possible CPU problem

# Analysis Step Five

- Generate final bucket ID and follow-up based on all gathered information
  - Determine which fields need to be embedded in the bucket ID
- !analyze assigns ownership of failure

# !analyze Enhancements

- !analyze is continually being updated to refine analysis and add new analysis information
  - New bugchecks and exceptions understood
  - Other teams are providing app- and domain-specific analysis
  - Lots of work being done for automatic hang analysis
- If you have suggestions for automatable analysis send them to pfat @ microsoft.com

# Debugger Command Programs

# What Is A Debugger Command Program?

- A sequence of debugger commands using the control flow commands for program structure and aliases as variables
- In-between a command script and an extension
  - Built from normal debugger commands, like a script
  - Includes iteration and control flow, like an extension
- Much simpler than an extension
  - The price is less flexibility
- Not a true programming environment/scripting model
  - Aliases are not true variables and can be complicated to use in complex programs
  - No execution control allowed in a command program

# Control Flow Commands

- Basic flow
  - Looping: `.do`, `.for`, `.while`, `.break`, `.continue`
  - Collection looping: `.foreach`
    - Loops for each space-delimited token in a string, command output or a file
  - Conditionals: `.if`, `.elsif`, `.else`
- Error handling
  - `.catch`
  - `.leave`
- Scoping
  - `.block`
    - Critical for adding a layer of alias expansion
- One catch: everything has to be on one line
  - Special-case with `$><` to make script files work reasonably

# Alias Enhancements

- as now has many options for how to set the replacement text
  - /c: from collected command output
  - /e: from an environment variable
  - /f: from a file
  - /ma, /mu, /msa, /msu: from a string in memory
  - /x: from a numeric expression result
- Alias names can now be surrounded by `${<name>}` to allow replacement when embedded within other text
- Alias text replacement provides a form of value substitution that can work like a variable
  - Can sometimes be complex to get substitution when you want
    - And not when you don't want

# Extended Alias Replacement Rules

- Previously aliases were expanded once when command processing started
  - Does not allow for loops to get updated values
- Aliases are now expanded for every block
  - Each layer of { } causes a new alias expansion pass
  - Passes can be added arbitrarily with .block
  - Aliases can be block-specific, to give a scope
- Aliases replacement with `${/<flag>:<name>}` gives extended control
  - /v is the most important and protects text from being replaced, useful for protecting alias names
  - Several others

# More Command Program Commands

- aS command takes text just to a semi-colon
- \$\$ comments extend just to a semi-colon, for embedding
- \$>< collects lines in a file into a single blob for processing
- \$\$< and \$\$>< are forms of \$< and \$>< which take a file name up to the new semi-colon
- Im and s now have flags to produce results as simple text for consumption with .foreach
  - For example, you could use .foreach and s-[1] to write a command which executes arbitrary operations on every search hit
    - `.foreach ($addr { s-[1]d @esp l80 7ffda000 }) { .printf "Hit at 0x%p\n", $addr }`

# Command Program Example

```
$$ Get module list LIST_ENTRY in $t0.
```

```
r? $t0 = &@$peb->Ldr->InLoadOrderModuleList
```

```
$$ Iterate over all modules in list.
```

```
.for (r? $t1 = *(ntdll!_LDR_DATA_TABLE_ENTRY**)@$t0;
```

```
    (@$t1 != 0) & (@$t1 != @$t0);
```

```
    r? $t1 = (ntdll!_LDR_DATA_TABLE_ENTRY*)@$t1->InLoadOrderLinks.Flink)
```

```
{
```

```
    $$ Get base address in $Base.
```

```
    as /x ${/v:$Base} @ @c++(@$t1->DllBase)
```

```
    $$ Get full name into $Mod.
```

```
    as /msu ${/v:$Mod} @ @c++(&@$t1->FullDllName)
```

```
    .block
```

```
{
```

```
    .echo ${$Mod} at ${$Base}
```

```
}
```

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

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

```
}
```

# Command Program Example

```
$$ Get process list LIST_ENTRY in $t0.
```

```
r $t0 = nt!PsActiveProcessHead
```

```
$$ Iterate over all processes in list.
```

```
.for (r $t1 = poi(@$t0);
```

```
    (@$t1 != 0) & (@$t1 != @$t0);
```

```
    r $t1 = poi(@$t1))
```

```
{
```

```
    r? $t2 = #CONTAINING_RECORD(@$t1, nt!_EPROCESS, ActiveProcessLinks);
```

```
    as /x ${/v:$Proc} @$t2
```

```
$$ Get image name into $ImageName.
```

```
as /ma ${/v:$ImageName} @ @c++(&@$t2->ImageFileName[0])
```

```
.block
```

```
{
```

```
    .echo ${$ImageName} at ${$Proc}
```

```
}
```

```
ad ${/v:$ImageName}
```

```
ad ${/v:$Proc}
```

```
}
```

# Debugger Command Program Advice

- When in doubt, double-check your alias usage
  - Make sure you're deleting aliases so that you don't have problems with stale values causing replacement
- Develop the program as small units and test each one before composing
  - Debugger software engineering
- If a component doesn't work, try its commands by hand
- Patience!
- At a certain level of complexity write a debugger extension instead

# Debugger Extensions

# What Can They Do?

- Dump or analyze complex data structures
  - !process, !devnode, !poolval
  - Leverages the type information from the PDB file
- Simplify routine tasks
  - !analyze
- Automate repetitive steps
  - Regularly check state of certain objects
- Fully control the state of the target
  - Can write a mini-debugger using the extension APIs

# When Should You Write One?

- Any task that is repetitive can be sped up by developing an extension
  - Allows other people (testers) to help with basic debugging
  - Can help identify common problems quickly
- To dump internal data structures in a custom readable format
- Avoid writing extensions when:
  - Code is still fluctuating a lot
    - Extension must match the code being debugged

# Debugger Extension Execution Model

- Debugger extensions are routines exported from a DLL
- Calling syntax is
  - !<module>.<routine> [<params>]
- Debugger captures the input
- Debugger loads the <module> and calls the <routine> passing <parameters> as input
  - Direct function call into the DLL
- Extension routine runs on the main debugger thread
- Debugger code takes back control when the extension returns

# Extension APIs

- Old APIs in `wdbgexts.h`
  - For compatibility with old extensions
  - Both 32- and 64-bit versions of routines
- New APIs in `dbgeng.h`
  - Large number of APIs
  - Provide the full power of the debugger
  - All 64 bit APIs
    - Can debug both 32 bit and 64 bit targets
  - Documentation is partial
    - We are actively improving it
- Upcoming C++ extension framework in the next debugger package
  - Greatly simplifies extension development

# KnownStructOutput

- Different kind of extension entry point that allows an extension to participate in typed data formatting
  - Affects dt, dv, windbg, etc.
- Extension provides names of types it understands
- When the debugger is formatting typed data it will ask the extension to fill a text buffer with a formatted representation of the type
- Similar to built-in handling of `LARGE_INTEGER`, `UNICODE_STRING` and so on

# Samples

- Numerous samples in the debugger package
  - Must manually install the SDK component
- Building the samples requires the latest Windows DDK
  - Overlay the headers and libs from the debugger on top of the DDK, or follow the readme.txt instructions if you want to keep the debugger files separate
  - The extension will run on any OS the debugger runs on

# Extensions Must Be Careful

- Debugger Extensions run “in-proc”
  - Debugger has a global exception handler around extensions to recover from AVs
  - Heap corruption in an extension will cause the debugger to crash
- Do all target data access through the debugger APIs
  - Don't use standard Win32 APIs to try to extract data from the debug target. The APIs will run locally on the host and won't be portable with crash dump files.
- Extensions must handle Ctrl-Break themselves
  - Debugger cannot stop extension code

# Extension Typed Data Handling Example

```
// Initialize type read from the address
if (InitTypeRead(Address, _EXCEPTION_RECORD) != 0) {
    // Use %p to print pointer values
    dprintf("Error in reading _EXCEPTION_RECORD at %p", Address);
} else {
    // Read and dump the fields
    dprintf("_EXCEPTION_RECORD @ %p\n", Address);
    dprintf("\tExceptionCode      : %lx\n",
        (ULONG) ReadField(ExceptionCode));
    dprintf("\tExceptionAddress      : %p\n",
        ReadField(ExceptionAddress));
    dprintf("\tExceptionInformation[1] : %l64lx\n",
        ReadField(ExceptionInformation[1]));
}
```

# Extension Known Struct Example

```
if (Flag == DEBUG_KNOWN_STRUCT_GET_SINGLE_LINE_OUTPUT)
{
    if (lstrcmp(StructName, "_SYSTEMTIME"))
    {
        SYSTEMTIME Data;
        ULONG Ret;

        if (ReadMemory(Address, &Data, sizeof(Data), &Ret) && Ret == sizeof(Data))
        {
            Hr = StringCbPrintf(Buffer, *BufferSize, " { %02ld:%02ld:%02ld %02ld/%02ld/%04ld }",
                Data.wHour,
                Data.wMinute,
                Data.wSecond,
                Data.wMonth,
                Data.wDay,
                Data.wYear);
        }
        else
        {
            Hr = E_INVALIDARG;
        }
    }
    else
    {
        Hr = E_INVALIDARG;
    }
}
```

# Remote Debugging

# Remote Debugging Definitions

- Target
  - Machine running the code being debugged
- Host
  - Machine running the core debugger, extensions
  - Host is where symbols are loaded
- Remote(s)
  - Machine(s) connecting to the host
  - Source code can be loaded on a remote
- Process server \ KD connection server
  - Remotely send data between the host and target
- Proxy \ repeater
  - Machine running a proxy tool, sitting between a remote and a host, or a host and target

# Remote Debugging Connections

- For both user and kernel mode
  - To connect from remote to host
    - -server on the host ; -remote on the remote
  - To go through internet proxies using a repeater
    - Dbengprx.exe
- For user mode
  - Target and host are generally the same machine
  - To split the host and target
    - Run dbgsrv.exe on the target machine
    - Connect the host to the target using –premode option
- For kernel mode
  - Target and host are physically connected (COM)
  - Run kdsrv.exe on a local host
  - The host debugger connects remotely to kdsrv

# Securing A Remote Session

- Secure connections can be established using SSL
- A debugger session can also be secured by using `–secure`
  - Disables the remote user's ability to tamper with the host machine
    - Launch new processes
    - Change symbol or debugger extension paths
    - `.shell` commands, etc
- Detailed info in the docs

# Other Remote Debugging Issues

- Remote.exe
  - Not related to debugger remoting
  - Cmd.exe remoting tool
- Ntsd -d
  - Not a remote debugger.
    - It's an option to debug user mode applications over the KD debugger port
    - '-d' only does input\output redirection
  - Causes lots of problems with symbols
  - More details in the docs if you need to use it

# Call to Action

- Get the latest debuggers from <http://www.microsoft.com/whdc/ddk/debugging>
- Read the debugger docs
- Use !analyze
- Try and script/program/extend the debugger to customize your experience

# Additional Resources

- Debugger issues and requests
  - windbgfb @ microsoft.com
- !analyze questions
  - pfat @ microsoft.com
- Web Resources
  - Windows Debuggers:  
<http://www.microsoft.com/whdc/ddk/debugging>

***Microsoft***<sup>®</sup>

*Your potential. Our passion.*<sup>™</sup>

© 2005 Microsoft Corporation. All rights reserved.

This presentation is for informational purposes only. Microsoft makes no warranties, express or implied, in this summary.