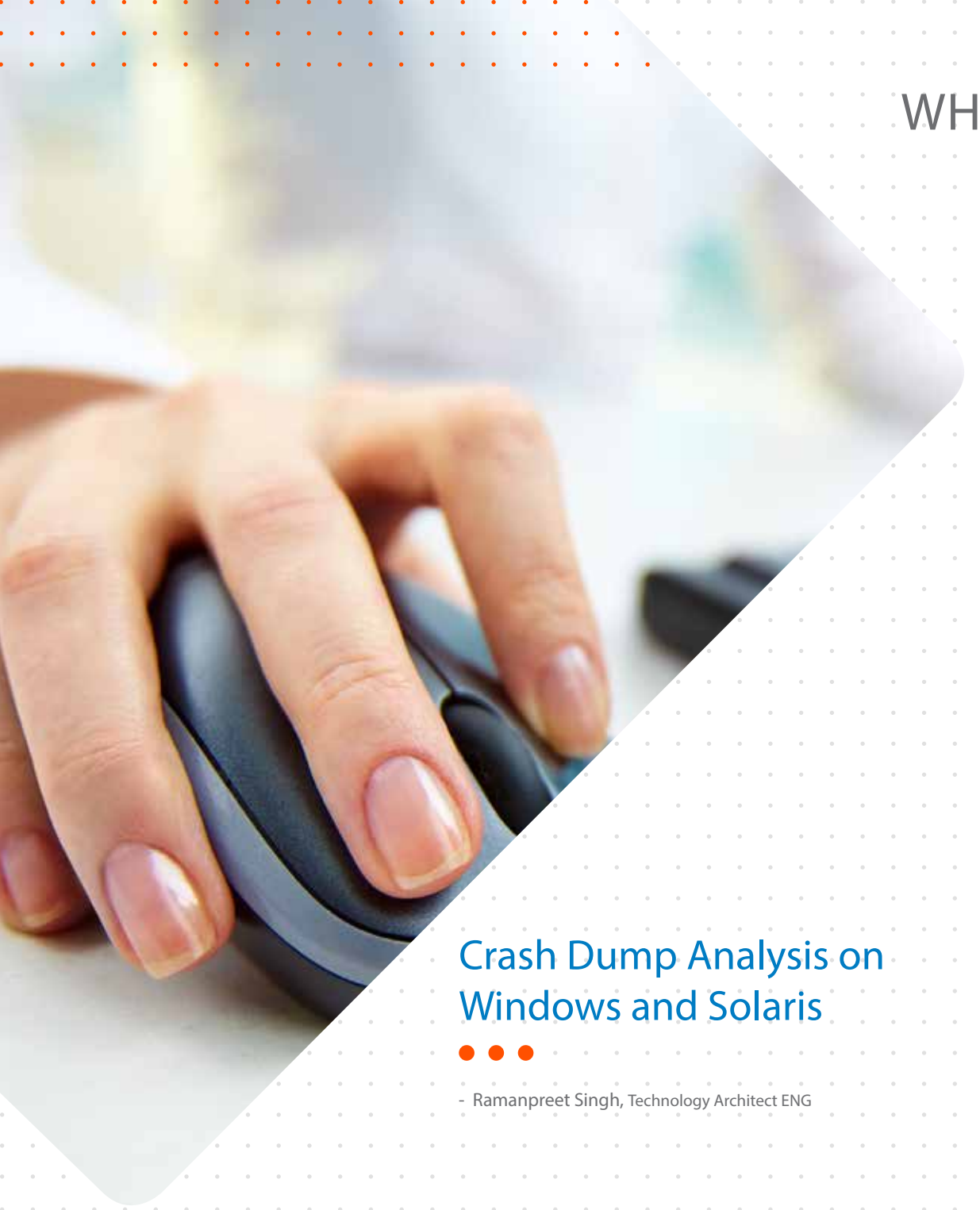
WHITE PAPER



Crash Dump Analysis on Windows and Solaris

- Ramanpreet Singh, Technology Architect ENG

Infosys
be more

Introduction

This document is targeted for developers working on assignments / bugs which require crash / core dump analysis on Windows / Solaris.

The core dump file is created whenever there is abnormal termination of a process which could be due to unexpected behavior of application etc. For example: process trying to write to an invalid address can result in termination of the process thereby generating the core dump. The core dump file contains the snapshot of memory, register contents and other debugging information which can provide valuable information to developers to find out the root cause of the issue which caused the program to terminate unexpectedly. The core file can provide information like the last function which was being executed when program got terminated, contents of memory address associated with the process, register contents, stack trace (containing complete sequence of calls / functions being executed), parameter validation (values of arguments passed to functions), disassembly of the code etc. with which one can find the condition under which program terminated and accordingly fix the issue.

Crash Dump Analysis on Windows

As mentioned in the introduction section, dump captures the vital information of the program state for later use. It contains the information of the process which is no more in active state and additionally this dump can be opened on the different machine and not necessarily on the machine on which process was running. This enables the developers to analyze the dump in their own environment without the need to obtain the access to customer's production environment where crash has occurred.

Types of exceptions

The exceptions can be categorized as –

1. First Chance exception
2. Second Chance exception

The first chance exceptions do not generally correspond to problem in code and second chance exceptions are responsible for causing crash in the program.

Actually, when debugger is attached to an application, the debugger gets the first chance to see an exception (before the application could see exception). The first pass is called first chance exception wherein debugger maybe configured to pass on the exception to the application.

If the application here gracefully handles the exception – then program would continue as normal. Otherwise the unhandled exception would again be sent to the debugger and this pass is called second chance exception. Therefore, first chance exception or handled exceptions are most likely not the cause of worry whereas second chance exceptions or unhandled exceptions can crash the program.

Typically, the crash dump file is created on windows platform with the extension ".dmp". Various tools are available to

analyze the dump on windows. As part of this paper, we will focus on WinDbg and Debug Diag.

Loading symbols

The initial step is to resolve the symbols of the dependent libraries. For this, we need to specify the path containing the PDB files corresponding to windows and application libraries.

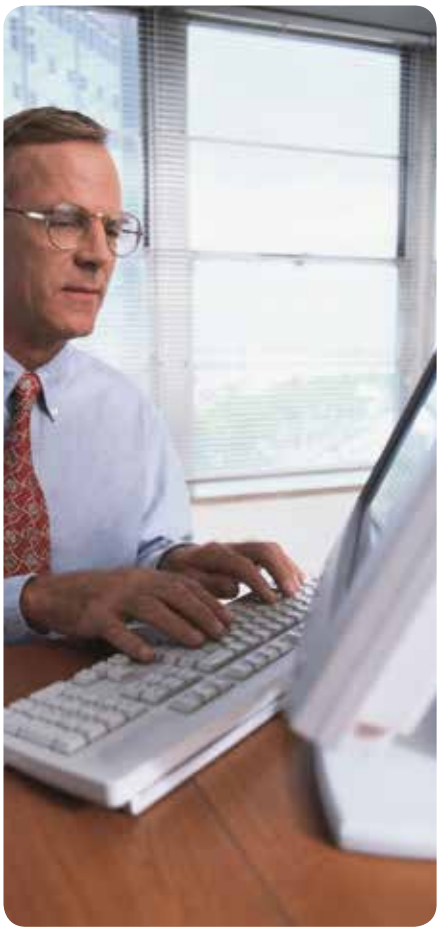
Online Microsoft symbols server can be used for resolving the symbols of windows libraries – to do so specify the below path in the symbol path:

SRV*<Local-Path>*http://msdl.microsoft.com/download/symbols

This will resolve the symbols of Windows libraries at runtime (when analyzing the dump) and also download the required symbols to the <Local-Path> location. At a later stage – when reopening the same dump file – we can also directly specify the <Local-Path> in symbol path – thereby resolving the Windows symbols in offline mode.

And for dependent application libraries, we can store all the required PDBs at some location and append the path of that location in symbol path.

In case of WinDbg, symbol path can be added using .sympath command or from the Menu File -> Symbol File Path. And in case of Debug Diag symbol path can be added using Menu option Tools ->Options and Settings.



Resolving issues pertaining to loading Symbols

You may experience issues in loading the symbols even if the PDBs correspond to the same version of the product.

We can enable the verbose mode in WinDbg to see why symbols cannot be loaded.

Use the following command to enable the verbose mode:

'!sym noisy'

Then reload the symbols using below command and watch the error displayed:

.reload /f

In case you get the mismatched PDBs error and you are sure that PDBs are correct and correspond to the same version of the product that generated the crash dump – then you may opt to forcefully load the symbols from PDBs.

Use the following command to forcefully load the symbols from mismatched PDBs:

.reload /i

Now, the reason for mismatch PDBs might be the difference in timestamp between the PDBs and the executable that caused the crash.

The forceful loading of symbols can prove useful if PDBs actually correspond to the executable and the debugger has somehow considered them to be mismatched. Otherwise, forceful loading can lead to wrong code lines in the stack trace / debugger output.

Analyzing the crash dump

The developers need to analyze the crash dump to find the root cause of the crash and identify the fix accordingly. The stack trace (the call stack at the time of crash), disassembly and registers values can be useful in analyzing the crash dump.

In case of windbg, use the following command to display the stack trace/call stack of the thread that crashed:

!analyze –v

This will display the stack trace i.e. sequence of the calls / functions corresponding to the thread that crashed.

To view stack of all the threads, following command can be used.

!uniqstack

To display disassembly – following command can be used.

uf <function-name>

The above command will display the disassembly for the function specified and this disassembly can be compared with the source code. This will help in locating the source code line that caused the crash since WinDbg will tell assembly language statement at which crash has occurred and it can be compared with source code accordingly.

And to dump the register values, 'r' command can be used.

In case of Debug Diag tool, the complete analysis report can be generated in a single go. The report depicts all the errors and exceptions along with their Thread IDs that caused those exceptions. In addition to this, it displays call stack for all the threads. This tool can be quite helpful in analyzing the hang issues wherein it can depict which threads are waiting and which threads are holding onto some critical section etc.

Crash Dump Analysis on Solaris

Dbx can be used to extract debugging information from core dump file on Solaris. The developer can use a set of dbx commands to fetch the information like stack trace, memory contents, register contents, disassembly etc. as explained in above section.

The dbx instructions can also be optionally used for specifying virtual paths. These instructions are used for loading the symbols as we did earlier for Windows.

In case of Windows, we simply placed all the required symbols (PDBs) at some location and added that path in symbol path and in that case debugger would look directly at symbol path for resolving any symbols. But in case of Solaris, debugger looks for libraries at actual path i.e. same path where libraries (system/application) were placed on the machine where crash occurred.

Since generally core is analyzed on different machine than where the crash occurred – we can specify the virtual paths on the debug machine using dbx instructions or other option is (if possible) create the same physical paths on the debug machine.

To create virtual paths, pathmap instructions can be used in dbxrc file as follows:

pathmap /lib $PWD/libs/usr/lib

In the above instruction, debugger will virtually map /lib folder to current folder/libs/usr/lib. Hence, this eliminates the need to create actual physical paths.

Now, open the core file using the following command:

dbx –s dbxrc

The following steps can be helpful in debugging the core:

a) Parameter Validation
b) Viewing Memory contents
c) Viewing Register Contents
d) Viewing Disassembly

Parameter Validation

The stack trace displays the sequence of functions/calls executed that led to crash. Along with the functions, the actual arguments passed to the calls are also shown. The argument values can be analyzed to see if they are different from the expected values and in that case it can be compared with the source code to detect the flaw in the code.

Sometimes, some argument might be containing NULL which could be the culprit. So the solution for this would be to see why it is pointing to NULL and fix the issue accordingly.

Also good programming practice requires adding NULL checks – this would at least avoid crash situations.

Viewing Memory contents

Though sometimes crash issue can be simply debugged by parameter validation, but it could not be that easy all the time. We may need to view memory contents at particular address to see what the actual value of the variable was.

By fetching the values of various variables, it may help to find out if any variable's actual value is different from expected one which could point to cause of the issue.

The 'examine' command can be used to view the contents at a particular memory address:

examine <memory-address>/<format>.
The <format> can be c (character), s (String), X (hexadecimal) etc. depending on the type of the variable stored in memory.

Viewing Register contents

Viewing register contents could also help debugging the core. To view the register contents, first we need to set the frame for which we need to view the register values. It may also point to the erroneous condition which caused the crash.

The register contents can be viewed simply by using the command 'regs'.

Viewing Disassembly

Viewing disassembly could help in debugging the core by mapping the disassembly with the source code and finding out which source code line actually caused the crash.

The disassembly can be viewed by using the following command:

dis <memory address of the function>

In the above command, function name can also be used instead of its memory address.

Now apart from analyzing the core dumps, hang issues can be fixed in a similar way. In case the application hangs in some scenario, the core can be generated forcibly. Then the core dump can be analyzed to see what the call stack etc. is and accordingly see which threads are in hang state and what they were doing – same as per the above core dump analysis.

Following dbx commands can be used for thread related information:

threads - to view all threads and see which crashed or threw sigsegv signal

thread t@<thead #> - will set it as current thread

Also, the information of the thread causing the segmentation fault is available in pflags.

General guidelines

1. Symbols should be loaded correctly before analyzing the core dump so that correct call stack etc. is displayed.
2. Call stack, disassembly, registers contents, memory contents can be helpful in debugging the crash issue and fixing it.
3. Core can be debugged on the different machine and need not to be the same machine where core was created.

About the Author

Ramanpreet Singh Lamba

Raman is working as Technology Architect with Infosys. He has more than 8.5 years of experience in the IT industry. His areas of specialization include Access Management and Web security. He has worked on various critical issues in well-known Access Management product that includes crash dump issues.

Raman has wide experience in working on multiple complex projects for Software Development and Maintenance. He took his degree in Computer Science Engineering in 2004. He can be reached at Ramanpreet_Lamba@infosys.com