Automated Test Results Processing

The Art of Automated Crash Dump & Log Analysis

Edward Guy Smith
Consulting Engineer

Mangosoft Incorporated
1500 West Park Drive, Suite 190
Westborough, MA 01591

edsmith@mangosoft.com

Abstract

QA automation often leverages success on the ability to perform continuous, non-stop testing. When one test completes, its results are collected and a new test is begun. The collected results may include test logs, product logs, Dr. Watson® dumps, user mode process dumps, and kernel mode memory dumps. QA engineers then process the results, distilling them down into problem reports. As a manual process, it is difficult to achieve any level of efficiency in processing results and provide any consistency in problem reporting.

Consistency in problem reporting is key to being able to distinguish new problems from ones that have already been reported, and to provide the details development engineering needs to isolate and identify a defect. Automating the results analysis process provides an efficient method for producing problem reports that uniquely identify product failures by standardizing the format and content of failure summaries and their supporting details.

This paper introduces techniques used to automate the results analysis process. It examines the analysis of crash dump files and log files to extract consistent failure summaries and details, showing how these are used in problem reporting. It then studies the practical application of *Automated Test Results Processing* at Mangosoft Incorporated and presents data showing the impact this has had in product testing.

Introduction

Automated testing enables QA teams to find more product defects in a shorter period of time than manual testing. It can however produce a mountain of results to analyze. How are these results processed? Today that is mostly the job of a QA engineer; reading through product logs, test logs, and examining crash dumps produced by the failing product. Processing these results manually is terribly inefficient and produces inconsistent results from engineer to engineer. The key role of a QA engineer in examining test failures is to determine if the failure represents a product defect and to identify if the failure represents a new defect or a duplicate of one already reported. Automated testing often exposes the same product defect multiple times, so it becomes critical that each failure is described uniquely and consistently. Poorly or inconsistently described problem reports prevent QA engineers from easily identifying these duplicates. Reporting the same defect multiple times wastes engineering resources re-examining the same problem.

Customer observable quality and robustness is directly affected by the number of defects left in the product that ships. Automated and manual testing are still the methods most relied upon for identifying software defects. An effective model of automated testing is *Continuous Testing [1]*. In this model, tests run 24 hours a day, 7 days a week, and the results of this testing are efficiently processed. A key element of the *Continuous Testing* methodology is the ability to automate the results analysis. This paper explores automating the results analysis process to efficiently produce consistent problem reports with well-described failure summaries and supporting details. The paper examines the practical application of *Automated Test Results Processing* at Mangosoft and reviews data collected from this effort. Using *Automated Test Results Processing*, Mangosoft increased their test execution capacity from 2500 tests per month to over 5000 and reduced the number of QA engineers required to process the test results from seven engineers down to three!

The Process

Successful automated testing produces scores of results to process. Manually working through these results to locate product failures and describe them is a cumbersome and error prone process. Automation can help reduce the effort, provide consistency in problem reporting, and allow many more tests to be executed with fewer QA engineers. Achieving *Automated Test Results Processing* requires three steps:

1. Apply *Consistent Methods* to reveal defects in the product, improving the effectiveness of automated testing and the automated results analysis.
2. Design *Concise Problem Reports* to allow QA to quickly identify duplicates problems and provide developers with the information they need to isolate defects.
3. *Automate the Analysis* of test results to efficiently collect the data required to construct consistent problem reports.

The following sections examine these steps and suggest a design and format to achieve *Automated Test Results Processing.*

Consistent Methods

To extract meaningful results in an automated fashion requires the use of consistent defect-revealing methods. Test and product log files must record significant events and clearly flag failure events. Within the product, effective use and placement of debug assertions *[2]* can be instrumental in revealing defects. An assertion is a Boolean expression evaluating some current program state. If the assertion evaluates to FALSE, the program can take evasive action, recording useful information and capturing the program state in a user mode or kernel mode crash dump file.

Carefully placed assertions and logging using consistent methods can greatly improve the effectiveness of automated testing and the ability to automate the results analysis process.

Assertion Methods

To make the most of assertions, consider using Windows® NT as the main test platform and collect crash dumps. A crash dump provides a snapshot representing the exact state of the product at the time of the assertion. Windows NT provides the ability to capture a complete kernel mode crash dump or a user mode process dump that can later be analyzed in a debugger to determine the exact cause of the failure. This is a huge advantage to automated testing, allowing the automation to collect all of the necessary information to diagnose a test failure, move the information off the test platform, and proceed to the next test.

To take advantage of crash dumps in kernel mode, configure Windows NT to create dumps. In the control panel *System* applet, enable the writing of debugging information to a memory dump file and the automatic reboot option so the system restarts automatically after a crash. When a kernel mode application triggers an assertion or causes a bugcheck, the contents of memory are written to the page file, the system is rebooted, and the dump is copied out of the page file and written to the Windows NT system directory as `MEMORY.DMP`. Ensure Windows NT has enough disk space to hold a file the size of memory and that the page file is approximately 1½ times the memory size. This is required so Windows NT has room for the dump file and enough paging space to reboot while copying the dump to disk. Check the page file size from the same control panel *System* applet.

For collecting process dumps from user mode applications, use the *Userdump* utility included in the Microsoft OEM Support Tools *[3]*. Dr. Watson, which is included with Windows NT, also has the ability to write process dumps and works well on Windows 2000 but can have symbol problems on Windows NT 4.0 (Microsoft Systems Journal, January 2000 Issue *[4]*). Userdump is used to take dumps of running or hung processes without terminating them, and to monitor processes for exceptions, generating a dump file when an exception occurs. This exception monitoring can be used to collect a process dump when an assertion triggers. Install Userdump and configure it to monitor all exceptions for the application you are testing. Uncheck the *Ignore exceptions that occur inside Kernel32.dll* check box so Userdump catches the RaiseException() calls, which raise an exception in the kernel32.dll module.

When a user mode application triggers an assertion or causes an exception, Windows displays an exception dialog to inform the user. This is helpful in manual testing but inconvenient for automated testing. Although Userdump catches exceptions and writes a process dump, it does not prevent the exception dialogs. To avoid these, configure Dr. Watson as default debugger by running `drwtsn32.exe -i` so when an exception occurs, Dr. Watson is called to handle it. Now run `drwtsn32.exe` and uncheck all the option check boxes except *Append To Existing Log File* to configure Dr. Watson to create logs, but no dumps or notifications, since Userdump handles this.

Microsoft's C++ ASSERT macros also display a dialog when an assertion triggers in a windows application. These dialogs can be avoided by creating your own assertion macro. This provides greater flexibility in what actions to take when an assertion is triggered, including saving the assertion expression so it is easier to locate in the dump file. Here is an example showing how to create an assertion macro which, when triggered, saves the assertion expression in a global symbol and then raises an exception. Userdump will catch the exception, write the contents of the process into a user mode dump file, and then pass the exception to Dr. Watson, which adds it to its log; all without a single dialog box for the automation to deal with.

```
// AssertMacro.h //

#ifndef ASSERTMACRO_H
#define ASSERTMACRO_H

#ifdef __cplusplus
extern "C" {
#endif

const char* g_assert = 0;
const unsigned KMODE_ASSERTION = 0x0000DEAD;
int LastRites(const char* expr, const char* file, unsigned line);

#ifdef __cplusplus
};
#endif
```

```
#ifdef _DEBUG
#define ASSERTDUMP(expr)  (int)((expr) || LastRites(#expr, __FILE__, __LINE__))
#else
#define ASSERTDUMP(expr)  (int)((expr) || 0)
#endif

int LastRites(const char* expr, const char* file, unsigned line)
{
    char assertLine[20];
    itoa(line, assertLine, 10);     // Line # in "file" where ASSERTDUMP occurred
    g_assert = expr;                // Assert expression that evaluated to FALSE

    // Output the g_assert string, file and assertLine here if you like
    // Uncomment appropriate line, User mode - RaiseException(), Kernel mode - KeBugCheckEx()
    // RaiseException(EXCEPTION_BREAKPOINT, EXCEPTION_NONCONTINUABLE, 0, 0);
    // KeBugCheckEx(KMODE_ASSERTION, 0, 0, 0, 0);

    g_assert = 0;
    return 0;
}

#endif //ASSERTMACRO_H
```

The code starts off defining an ASSERTDUMP macro so that, in the debug build, if the expression passed to it evaluates FALSE, LastRites() is called passing it the string value of the assertion expression, the file name, and the line number where the ASSERTDUMP occurred. In the release build, it simply evaluates the expression and returns. The LastRites() routine saves the assert expression in the global symbol g_assert. This makes it incredibly simple to find the assertion expression in automated dump analysis. Lastly, in a user mode application, call RaiseException() passing it an EXCEPTION_BREAKPOINT. This causes an exception to occur in kernel32.dll that Userdump will catch and write a user mode process dump. Userdump then passes the exception to Dr. Watson, which exits the application and prevents the exception dialog from appearing.

To use this macro in kernel mode applications, call KeBugCheckEx() instead of RaiseException(). Use the custom bugcheck type KMODE_ASSERTION to identify the crash was caused by a product assertion. The KeBugCheckEx() call causes Windows NT to crash and write a kernel mode dump file. There are no dialogs to deal with in kernel mode.

Logging Methods

Logging is generally used to report interesting informational events as well as failure events. In order for log files to be useful in identifying product defects, there must be a consistent method of reporting events. The easiest way to accomplish this is to use common code to provide support for logging. With everyone using the same logging methods, consistency in the logging can be achieved. Here are some suggested methods:

- Provide a LogInfo() call to record informational events.
- Provide a LogError() call to record recoverable failure events with the ERROR: keyword prefix.
- Provide a LogFatal() call to record unrecoverable failure events with the FATAL: keyword prefix.
- Have the logging code prefix a timestamp to record when the event occurred.

Having identifiable keywords in the log makes the job of a parser easier and more accurate. It is easy to see how some simple parsing code could locate the failure events in the following example:

```
11/7/2000 11:48:30 AM   Starting script SparseRead on machine DAX
11/7/2000 11:48:30 AM
11/7/2000 11:48:37 AM   SparseRead Test started
11/7/2000 11:48:56 AM   Starting Recursive Copy
11/7/2000 11:49:17 AM   Awaiting Reconciliation
11/7/2000 11:49:18 AM   FATAL: AwaitReconciliation() failed, Status = 0xE6600056
11/7/2000 11:49:18 AM
11/7/2000 11:49:18 AM   Finished script SparseRead on machine DAX
```

Using these logging and assertion methods to reveal defects provides the data required to construct concise problem reports. Defining a consistent format and content of problem reports will improve the effectiveness of automated results analysis.

Concise Problem Reports

Problem reports consist of *identity* data that describe the failure and supporting *analysis* data used to isolate the defect. An effective QA team uses the identity data to filter out duplicate occurrences of failures and pass only unique failures to the development engineering team for analysis. Developers use the analysis data to diagnose a failure and isolate the defect. Define the content of the analysis data with the development engineering team to ensure they have what they need to perform a diagnosis. The analysis data usually consists of: configuration information, test logs, product logs, and product crash dump files.

The format of the identity data defines what the automated results analysis will need to collect. The identity data consists of two items, the *failure summary*, and the *failure details*. The failure summary provides a one-line summary of the problem that precisely describes the failure. This is the initial item used in matching problem reports so it must be precise enough to distinguish a duplicate problem from a new one. It should not include specific data that may change from test run to test run such as file names, file sizes, line numbers, module offsets, etc. The failure details provide specific data about where the software failed and the events leading up to the failure. This is used in second level matching to compare more details about a specific failure with other reported instances that have the same failure summary.

Here is a suggested format for the summary and details when reporting failures diagnosed from crash dumps:

```
Summary: {assertion|bugcheck|exception} {in|from} calling routine
Details: source module @ line number
         stack trace
```

Here is a suggested format for the summary and details when reporting failures events diagnosed from product or test logs:

```
Summary: error event
Details: log file name @ line number
         log trace
```

As an example, consider the following code segment:

```
HRESULT DUploadState::FinishSegment(DOp* pOp) {
    .
    .
    if (!AtEndOfFile()) {
        ASSERTDUMP(m_filelength > m_readpos);
        m_total_in += m_compressor->total_in;
        m_total_out += m_compressor->total_out;
    }
    .
    .
}
```

If the ASSERTDUMP macro triggers in this code, a failure summary can be assembled from the assertion string and calling routine name. To construct the failure details, identify where the assertion occurred and the events that led up to the failure. The source module and line number state exactly where in the source code the assertion triggered, and a stack trace shows the events leading up to the failure. The completed failure summary and details would appear as follows:

```
Summary: m_filelength > m_readpos in webman!DuploadState::FinishSegment
Details: Assertion in uploadstate.cpp @ 492

         Stack Trace
         ---------------------------------------
         webman!LastRites+0x1de
         webman!DUploadState__FinishSegment+0x152
         webman!DUploadState__PrepareAPacket+0xdd
         webman!DTcb__UploadDataBlock+0xe2
         webman!DTcb__Transfer+0xa3
         webman!DTcbDaemon__Main+0x36a
         webman!DTcbDaemon__InitDaemon+0x12
         webman!palThreadWrapper+0x5a
         ntoskrnl!PspSystemThreadStartup+0x54
        ntoskrnl!KiThreadStartup+0x16
```

The same technique can be applied to processing log files. In the following example, parsing the WebAgent.log file revealed a failure event at line 201. Use the error string from the log to specify a failure summary. To construct the failure details, identify where the failure event occurred and the events that led up to the failure. The log file name and line number state exactly where in the log the event occurred, and a log trace shows the events leading up to the failure. The completed failure summary and details would appear as follows:

```
Summary: WaitingForCount was non-zero(=1) in CHttpPipe::InstanceQueueRemove
Details: Error in WebAgent.log @ 201

         Log Trace
         ----------------------------------------
         21:37:37 Heap Pool Internal Statistics:
         21:37:37  CNetworkBufPool
         21:37:37   Extent Size = 1000000
         21:37:37   Maximum Allocation Reached = 15 percent
         21:37:40 ERROR: WaitingForCount was non-zero(=1) in CHttpPipe::InstanceQueueRemove
```

To maintain consistency in problem reporting, it is vital to have a standard format for reporting failure summaries and details. Automating the analysis process to gather this data ensures a consistency that is difficult to achieve manually.

Automate the Analysis

The next few sections will explore some techniques of automating the processing of crash dump files and log files to extract the data required to construct failure summaries and details.

Automated Crash Dump Analysis

Using Windows NT to test products provides the distinct advantage of being able to collect either a kernel mode crash dump or a user mode process dump, depending on whether the product is a kernel mode or user mode application. This allows the automated testing to collect any dumps and logs, move them off the test platform, and proceed with the next test.

Opening the dump in a debugger to locate the crash reason and collect a stack trace can be a time consuming mechanical process, perfect for automation! Microsoft recently restructured their debuggers, abstracting the debugging engine into a DLL and providing an SDK for developers to take advantage of its power in their Debugging Tools for Windows *[5]*. Today, their Windbg debugger is a graphical user interface sitting on top of the debugger engine (dbgeng.dll). Using the SDK, it is simple to construct an automated dump analysis routine to extract the crash reason and stack trace required to formulate the failure summary and details.

The SDK is poorly documented, but the included sample code provides the necessary clues to facilitate its use. Perhaps the most powerful function provided is the Execute() function, which accepts any of the documented debugger commands. Call the Execute() function passing it any debugger command and the resulting text is returned through an output callback. The sample code shows how to open the dump, load the symbols, collect a stack trace, and close the dump.

There are two things to determine from a dump, the failure summary that describes why the software crashed and failure details showing where it crashed, and the stack trace leading up to it. To collect the stack trace, either follow the example shown in the SDK sample code and specify exactly what to include in the trace output, or use the Execute() function to process one of the existing stack commands. For example, to collect a stack trace that looks like the output of the `"kb"` command in the Windbg debugger, add the following code to the SDK sample:

```
// Execute the kb command
if ((status = g_Control->Execute(DEBUG_OUTCTL_THIS_CLIENT,
                                 "kb",
                                 DEBUG_EXECUTE_ECHO)) != S_OK)
{
    Exit(1, "Execute kb command failed, 0x%X\n", status);
}
```

To obtain a failure summary requires a little more work. If the failing application is executing in kernel mode, the crash reason can be determined from the bugcheck code. To examine this code, use the same Execute() example shown above, except pass it the `".bugcheck"` command. For a description of the bugcheck codes, look at the *debugger.chm* help file included with the SDK. If the failing application is executing in user mode, the crash reason can be determined from the exception code using the `".lastevent"` command. For a complete list of exception codes, see the *ntstatus.h* file that comes in the Microsoft Windows Driver Development Kits *[6]*.

Ideally, the majority of crashes encountered are because of assertions coded into the product. Carefully selected and placed assertions *[2]* make all the difference. An effective practice here is to carefully examine every crash that was not caught by a product assertion, determine why, and then correct the code to properly detect this type of failure in the future. Using the ASSERTDUMP macro described in this paper, when an assertion is triggered, the LastRites() routine will appear on the stack. The caller to LastRites() on the stack contains the required *calling routine*, *source module*, and *line number* information. In a kernel mode dump, the crash will be from the KMODE_ASSERTION (0x0000DEAD) bugcheck. In a user mode dump, the crash will be from an exception code of EXCEPTION_BREAKPOINT (0x80000003). The *assertion* string can be located by calling Execute() with the `"da poi(g_assert)"` command to display the contents pointed to by the g_assert symbol.

Here is an example of a user mode program where an ASSERTDUMP macro triggered. The example shows the output of the dbgeng.dll Execute() commands:

```
kb
ChildEBP RetAddr  Args to Child
0012f6e0 004014f6 80000003 00000001 00000000 KERNEL32!RaiseException+0x55
0012f758 00401d44 005b442c 005b4434 000000bd Eman!LastRites+0x56 [d:\Eman\assertmacro.h @ 22]
0012f7c4 0047a5b4 0012fa60 00167060 006e5388 Eman!CEmanDlg__OpenProfile+0x44 [d:\Eman\emandlg.cpp @ 189]
0012f7fc 0047ace1 0012fe88 000003e8 00000000 Eman!_AfxDispatchCmdMsg+0xa2 [cmdtarg.cpp @ 88]
0012f854 0047ba11 000003e8 00000000 00000000 Eman!CCmdTarget__OnCmdMsg+0x272 [cmdtarg.cpp @ 302]
0012f884 004809cb 000003e8 00000000 00000000 Eman!CDialog__OnCmdMsg+0x24 [dlgcore.cpp @ 97]
0012f8e4 0047fbe1 000003e8 00030696 0012fa60 Eman!CWnd__OnCommand+0x138 [wincore.cpp @ 2088]
0012f9e0 0047fb64 00000111 000003e8 00030696 Eman!CWnd__OnWndMsg+0x53 [wincore.cpp @ 1597]
0012fa00 0047d709 00000111 000003e8 00030696 Eman!CWnd__WindowProc+0x2e [wincore.cpp @ 1585]
0012fa74 0047dba5 0012fe88 00040674 00000111 Eman!AfxCallWndProc+0x65 [wincore.cpp @ 215]
0012faa0 77e148dc 00040674 00000111 000003e8 Eman!AfxWndProc+0x81 [wincore.cpp @ 368]
0012fac0 77e157ef 0047db24 00040674 00000111 USER32!UserCallWinProc+0x18

.lastevent
Last event: Exception 80000003, second chance

da poi(g_assert)
005b442c  "i != 1"
```

Since the Exception code was an EXCEPTION_BREAKPOINT (0x80000003), examine the global symbol g_assert for an *assertion* string, collect the *calling routine*, *source module*, and *line number* using the caller to LastRites() on the stack. This user mode assertion would generate the following failure summary and details:

```
Summary: i != 1 in Eman!EmanDlg::OpenProfile
Details: Assertion in emandlg.cpp @ 189

         Stack Trace
         -----------------------------------
          KERNEL32!RaiseException+0x55
          Eman!LastRites+0x56
         Eman!EmanDlg__OpenProfile+0x44
         Eman!_AfxDispatchCmdMsg+0xa2
         Eman!CCmdTarget__OnCmdMsg+0x272
         Eman!CDialog__OnCmdMsg+0x24
         Eman!CWnd__OnCommand+0x138
         Eman!CWnd__OnWndMsg+0x53
         Eman!CWnd__WindowProc+0x2e
         Eman!AfxCallWndProc+0x65
         Eman!AfxWndProc+0x81
         USER32!UserCallWinProc+0x18
```

If the crash was caused by the product but not caught by a product assertion, the LastRites() routine will not appear on the stack. This means the product code caused an unexpected bugcheck in kernel mode or an unexpected exception in user mode. In either of these cases, examine the stack for routines containing the

product name and obtain the *calling routine*, *source module*, and *line number* information from last call out of the product. It is also possible to have a crash where the product doesn't show up on the stack. In this case, omit the *calling routine* information from the failure summary and just specify the bugcheck or exception encountered.

Below is a kernel mode dump where the product under test did not trigger an assertion but caused an IRQL_NOT_LESS_OR_EQUAL bugcheck, resulting in a KiTrap. This signals that the driver accessed paged memory at DISPATCH_LEVEL or above.

```
kb
ChildEBP RetAddr  Args to Child
fea62ac0 8012d2a9 00000000 fea62bcc fea62af4 ntoskrnl!KiTrap0E+0x27c
fea62b44 80128395 00000720 00000001 00000720 ntoskrnl!MiRemoveWsle+0x13
fea62bec 8012a841 c0385c80 00000001 00000080 ntoskrnl!MiDeleteSystemPagableVm+0xc1
fea62c20 8010a571 e1720000 e1721000 00000000 ntoskrnl!MiFreePoolPages+0x38f
fea62c4c 8010a469 e1720000 00000000 fe8939f5 ntoskrnl!ExFreePoolWithTag+0x85
fea62c58 fe8939f5 e1720000 00000001 fea62c84 ntoskrnl!ExFreePool+0xb
fea62c68 fe89b99c e1720000 81cf3d10 00000001 webman!palFreeBlock+0x41 [memory.cpp @ 90]
fea62c84 fe89a6d5 e1720000 00001000 81cf3ccc webman!DPoolSpaceBlock__FreeExtent+0x134 [poolspace.cpp @ 402]
fea62c98 fe89abb3 e1720000 00001000 00000000 webman!DPool__FreeExtent+0x20 [pool.cpp @ 128]
fea62cb4 fe88cae4 e1720000 81396b98 fea62cd0 webman!DPoolPage__PoolFree+0xd9 [poolpage.cpp @ 174]
fea62cc4 fe88d28f 81396b98 fea62cf8 fe88ca18 webman!DLdsPageCb__~DLdsPageCb+0x2e [ldspcb.cpp @ 75]

.bugcheck
Bugcheck code 0000000A
Arguments c0c413d0 00000001 00000000 8012d2a9
```

Since the webman product was found on the stack, this kernel mode dump would generate the following failure summary and details:

```
Summary: IRQL_NOT_LESS_OR_EQUAL from webman!palFreeBlock
Details: Bugcheck from memory.cpp @ 90

         Stack Trace
         -----------------------------------
         ntoskrnl!KiTrap0E+0x27c
         ntoskrnl!MiRemoveWsle+0x13
         ntoskrnl!MiDeleteSystemPagableVm+0xc1
         ntoskrnl!MiFreePoolPages+0x38f
         ntoskrnl!ExFreePoolWithTag+0x85
         ntoskrnl!ExFreePool+0xb
         webman!palFreeBlock+0x41
         webman!DPoolSpaceBlock__FreeExtent+0x134
         webman!DPool__FreeExtent+0x20
         webman!DPoolPage__PoolFree+0xd9
         webman!DLdsPageCb__~DLdsPageCb+0x2e
```

Following is an example of a user mode dump where the product under test did not trigger an assertion but caused an EXCEPTION_ACCESS_VIOLATION:

```
kb
ChildEBP RetAddr  Args to Child
00bcfe18 00431511 001c55bc 00000001 00bcfe44 websrv!DOcb__IncrRefCount+0xc [ocb.cpp @ 1792]
00bcfe2c 0042595a 001c55bc 00bcfec0 0013a65c websrv!DOcbPtr__operator=+0x4a [ocb.cpp @ 3976]
00bcfe5c 0041cd99 00bcfe78 004bc400 004bc42c websrv!DVcb__OcbCleanup+0x12c [vcb.cpp @ 1938]
00bcff98 0041cc19 00bcffb8 004899d6 00133380 websrv!DVcbList__OcbCleanupDaemon+0x144 [vcblist.cpp @ 704]
00bcffa0 004899d6 00133380 00000000 00133380 websrv!DVcbList__OcbCleanupDaemonInit+0xd [vcblist.cpp @ 651]
00bcffb8 77f04ee8 00133380 00130680 77f8e4f1 websrv!_palThreadWrapper+0x46 [threads.c @ 28]
00bcffec 00000000 00489990 00133380 00000000 KERNEL32!lstrcmpiW+0xbe

.lastevent
Last event: Exception C0000005, second chance
```

In this user mode dump it would appear that the code in the IncrRefCount() routine is the culprit and this would generate the following failure summary and details:

```
Summary: EXCEPTION_ACCESS_VIOLATION from websrv!Docb::IncrRefCount
Details: Exception from ocb.cpp @ 1792

         Stack Trace
         -----------------------------------
          websrv!DOcb__IncrRefCount+0xc
         websrv!DOcbPtr__operator=+0x4a
         websrv!DVcb__OcbCleanup+0x12c]
         websrv!DVcbList__OcbCleanupDaemon+0x144
         websrv!DVcbList__OcbCleanupDaemonInit+0xd
         websrv!_palThreadWrapper+0x46
         KERNEL32!lstrcmpiW+0xbe
```

Automated Log File Analysis

The best way to parse text log files for failure events is to build a generic rules-based parsing engine. To parse a text log, the engine loads a configuration file for the log type that describes the rules to use when parsing the log. Each unique log type is described in a configuration file, defining to the parser how to identify key items. The generic parser code implements these rules and applies them as it reads the log file. The following section examines such a design.

The % character is used to delimit a special character sequence the parser needs to interpret. The parser code understands the following special character sequences that can be substituted in any of the string parsing rules:

- %d any numeric string
- %a any alpha character string
- %s any string value
- %S any white space
- %T timestamp
- %N logical NOT operator
- %A logical AND operator
- %O logical OR operator
- %% the % character

The configuration file contains three sections: *Events*, *Parsing*, and *Timestamp*. The log file is parsed a line at a time, keeping track of the line count to report the location of a failure event, and checking for matches using the Events section's *Contains* key. Once a failure event is found, the rules in the Parsing section are applied to refine the failure event. The Timestamp section provides a macro definition for how the timestamps are formatted in the logs to make them easier to specify in the Parsing and Events sections. Below are the rule specifications:

[Events]

This section identifies interesting events using the Contains key.

Contains=

This string key describes character sequences that must appear in a log line for it to be considered an interesting event. For example, to identify interesting events as lines that contain the string "ERROR: ", set Contains=ERROR:%S.

[Parsing]

Once an interesting event has been identified, this section uses keys defining character sequences to refine where the interesting information begins and where it ends within a line.

BeginAfter=

This key can be used to indicate where the interesting text within the event line begins. For example, if the event lines always begin with a timestamp, use this key to include only the text following the timestamp by setting BeginAfter=%T. If this key is left blank, the interesting text begins with the beginning of the line.

EndBefore=

This key can be used to indicate where the interesting text within the event line ends. For example, if the interesting events always end with the text " from address n", use this key to stop the text at the character before this string by setting EndBefore=%Sfrom%Saddress%S%d. If this key is left blank, the interesting text ends with the end of the line.

BeforeLines=

This key defines the number of lines to include before the interesting event in constructing the details. The default value is 5 lines.

AfterLines=

This key defines the number of lines to include after the interesting event in constructing the details. The default value is 0 lines.

[TimeStamp]
This section defines the macro %T, to describe the timestamp format to the parser.
Format=
This key defines the format of any timestamp entries in the log. Once the format is defined, substitute %T in any of the parsing rules to indicate a timestamp entry. The formatting for this key is a string using any of the % special characters. For example, set Format=%d/%d/%d%S%d:%d:%d%S%a%S if the log's timestamp appears as follows:

```
11/7/2000 11:48:56 AM   Starting Recursive Copy
```

The example below includes the log from a SparseRead test, the configuration that describes the parsing rules, and the resulting failure summary and details. The parsing rules capture any events containing the keywords ERROR: or FATAL:, remove the timestamp from the summary, and report three lines preceding the failure in the details.

```
11/7/2000 11:48:30 AM   Starting script SparseRead on machine DAX
11/7/2000 11:48:30 AM
11/7/2000 11:48:37 AM   SparseRead Test started
11/7/2000 11:48:56 AM   Starting Recursive Copy
11/7/2000 11:49:17 AM   Awaiting Reconciliation
11/7/2000 11:49:18 AM   FATAL: AwaitReconciliation() failed, Status = 0xE6600056
11/7/2000 11:49:18 AM
11/7/2000 11:49:18 AM   Finished script SparseRead on machine DAX
```

The configuration file would describe the parsing rules as follows:

```
[Events]
Contains=FATAL:%OERROR:

[Parsing]
BeginAfter=%T%SFATAL:%S%O%T%SERROR:%S
EndBefore=
BeforeLines=3
AfterLines=0

[TimeStamp]
Format=%d/%d/%d%S%d:%d:%d%S%a
```

After applying the parsing rules to the log, the following failure summary and details result:

```
Summary: AwaitReconciliation() failed, Status = 0xE6600056
Details: Error in SparseRead.log @ 6

         Log Trace
         ----------------------------------------
         11/7/2000 11:48:37 AM   SparseRead Test started
         11/7/2000 11:48:56 AM   Starting Recursive Copy
         11/7/2000 11:49:17 AM   Awaiting Reconciliation
         11/7/2000 11:49:18 AM   FATAL: AwaitReconciliation() failed, Status = 0xE6600056
```

Practical Application

Mangosoft Incorporated specializes in developing caching solutions. Our products include Cachelink®, a distributed web cache product, and Mangomind℠, a business Internet file service. Cachelink combines individual browser caches into a highly efficient web-cache server without the need for any additional hardware. Mangomind applies our caching technology to shared Internet storage in a client/server model. A Mangomind drive looks and operates just like a local drive and is completely integrated into Windows. Mangomind's Internet file-sharing service allows multiple users in any location to access and share files.

Automated Test Results Processing evolved during the development of our distributed caching technology. We design our products with generous use of debug assertions and a built-in fault infrastructure *[7]* for exercising the less frequently executed code paths and simulating common errors. QA designed distributed cache and file system tests that take advantage of this fault infrastructure and assertion usage. To capture a complete snapshot of a system failure, the test automation is designed to take crash dumps on all PCs participating in a test, and to collect these dumps and logs for later analysis. Collecting the results from one failure and proceeding with the next test allows us to test, 24 hours a day, 7 days a week.

Post processing these dump files by hand was an expensive, tedious, and error prone process for a team of seven QA engineers. We applied these *Automated Test Results Processing* techniques to analyze our test logs, product logs, and product dumps, extracting a failure summary and details. This immediately relieved the main pressure point in our process allowing us to double our testing volume and reduce the number of QA engineers processing test results by half. The quality of our problem reports improved, we could now provide consistent data to developers for defect diagnosis, and we were easily avoiding filing duplicate problem reports. This process was so effective when we were testing on Windows NT platforms that we searched for a solution to testing on Windows 9x platforms. We initially used Softice™ to stop in the debugger when we encountered a failure and have a developer diagnose the problem. This prevented any further testing on the platform and consumed the developer until the defect was isolated. We had become spoiled by the efficiency of testing on Windows NT platforms, with NT's ability to create dumps, allowing us to post process the failures. Since we also had to test our products on Windows 9x platforms, the development engineering team designed and implemented a Windows 9x memory dumper and debugger so we could collect crash dumps, and proceed with another test. This innovation made our products as easy to test and debug on Windows 9x platforms as they were on Windows NT platforms. We continue to use the custom memory dumper and debugger today with our Mangomind product, allowing us to maintain *Continuous Testing* on both platforms.

Successes

Mangosoft has achieved significant advances in automated software testing. Figure 1 shows the impact the *Automated Test Results Processing* had on the number of tests we were able to run. Before enabling this automation we were able to sustain a rate of 2500 tests per month with seven QA engineers. After the automation was in place we could now execute over 5000 tests per month with just three QA engineers. We eventually applied additional automation to assist in filing problem reports and now one QA engineer manages the scheduling and processing of over 5000 tests per month.

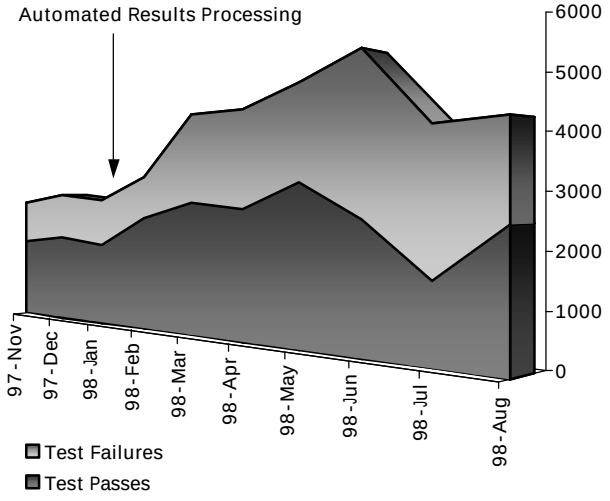

Figure 1 - Testing Volume

Figure 2 clearly shows the impact *Automated Test Results Processing* has had on our defect find rate. After implementing the automated analysis, our defect find rate climbed to over 700 problem reports in a single month, allowing us to quickly remove a large number of defects and greatly improve the quality of our product in a short period of time.

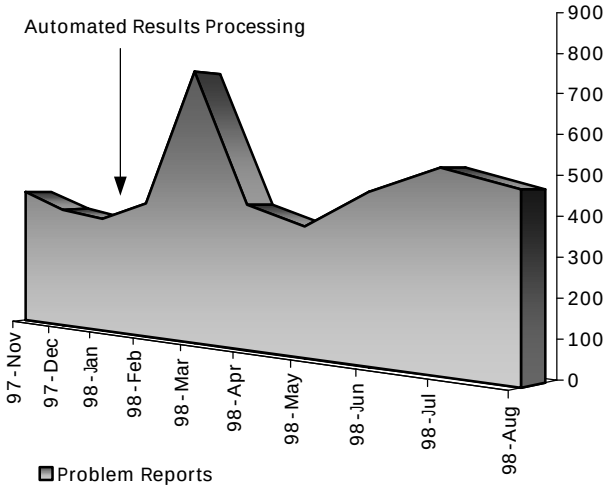

Figure 2 - Problem Report Volume

Conclusions and Future Work

This paper has described *Automated Test Results Processing*, a key automation innovation engineered at Mangosoft to achieve a *Continuous Testing* model. Our automated results processor provides a central analysis engine to handle common tasks, with plugin modules to handle product-specific analysis. This design allows us to quickly add new products and new result types for analysis.

At Mangosoft, we are continually refining our ideas and polishing our infrastructure. Having the ability to run and manage our automated testing with only a few QA engineers at the helm has dramatically changed the role of QA at Mangosoft. We are able to spend our time developing and exploring new areas of testing technologies such as:

- Targeted Software Fault Insertion *[7]* - We have developed a process for identifying, creating and inserting high-payback software faults. This has allowed us to explore many more code paths in our products in the search for defects.
- Generic Operation Sequencing - This technique breaks up product features into the modular operations required to test them. These operations can then be sequenced in various combinations to fully test a product feature. The beauty of this is the ability to reuse these modular operations across many product features to rapidly produce new test cases.
- Testing GUI without GUI - We are working with our development engineering team to implement this design methodology. The idea is to abstract the GUI presentation layer from the core code implementation through an API. This allows the product's GUI operations to be sequenced from test code as easily as they are by users. With this we can efficiently apply automated testing to more of the product without relying on cumbersome GUI-based testing tools, or requiring the time of hands-on testers.

Acknowledgment

I'd like to thank the incredible team of QA and Development engineers at Mangosoft for their efforts in evolving the *Automated Test Results Processing* into the success it is today, and providing the inspiration for this paper. Special thanks to Bill Goleman, Paul Houlihan, Jody Zolli,, Don Gaubatz, Mary Lennon, and Cara Torta for the valuable review comments during the development of this document.

References

1. Smith, E. 2000. *Continuous Testing*, Proceedings of the 17th International Conference on Testing Computer Software, also available at http://www.mangosoft.com/technology
2. Voas , J., and L. Kassab. 1999. *Using Assertions to Make Untestable Software More Testable*, Software Quality Professional, Volume 2 Issue 4.
3. Microsoft OEM Support Tools Phase 3 Service Release 2, http://support.microsoft.com/support/kb/articles/Q253/0/66.asp
4. Robbins, J. 2000. *Bugslayer*, Microsoft Systems Journal, January 2000 Issue. http://msdn.microsoft.com/library/periodic/period00/bugslayer0100.htm
5. Microsoft Debugging Tools for Windows, http://www.microsoft.com/ddk/debugging
6. Microsoft Windows Driver Development Kits, http://www.microsoft.com/ddk
7. Houlihan, P. 2001. *Targeted Software Fault Insertion*, Proceedings of the STAR*EAST* 2001 Conference, also available at http://www.mangosoft.com/technology

Edward Guy Smith

Ed is a Consulting Engineer at Mangosoft Incorporated specializing in automated test deployment and results processing for complex distributed systems. His design and development effort on automation infrastructure has allowed Mangosoft to achieve a *Continuous Testing* model where his *Automated Results Processing* plays a major role. Ed's paper on the *Continuous Testing* model was published in the Proceedings of the 17th International Conference on Testing Computer Software. He presented his paper at the conference in Washington, D.C. in June 2000.
Ed has designed and developed distributed cache and file system tests used to evaluate data coherency in the qualification of Mangosoft products. Ed is also interested in the optimization of the normal workflow between product development and product testing using automation where it makes sense.

Prior to Mangosoft, Ed worked at Digital Equipment Corporation for 19 years in a variety of Quality Assurance and Customer Support roles. He designed and developed tests for OpenVMS Clusters and their distributed I/O subsystems, worked as a Software Serviceability Engineer on OpenVMS, and as a Systems Support Engineer for Digital Equipment's General International Area.