Memory Dump Analysis Anthology

Volume 8b

Dmitry Vostokov
Software Diagnostics Institute

OpenTask

Published by OpenTask, Republic of Ireland

A CIP catalog record for this book is available from the British Library.

ISBN-13: 978-1-908043-54-2 (Paperback)

First printing, 2015

Revision 1.06 (March 2017)

Table of Contents

Preface

This reference volume consists of revised, edited, cross-referenced and thematically organized articles from Software Diagnostics Institute (DumpAnalysis.org + TraceAnalysis.org) and Software Diagnostics Library (former Crash Dump Analysis blog, DumpAnalysis.org/blog). Most of the selected articles are about software diagnostics, debugging, crash dump analysis, software trace and log analysis, malware analysis, and memory forensics. They were written in December 2014 - July 2015. We hope this reference is useful for:

- Software engineers developing and maintaining products on Windows platforms;
- Technical support and escalation engineers dealing with complex software issues;
- Quality assurance engineers testing software on Windows platforms;
- Security researchers, reverse engineers, malware and memory forensics analysts;
- Trace and log analysis articles will be of interest to users of any platform.

If you encounter any error, please contact me using this form:

http://www.dumpanalysis.org/contact

or send me a personal message using this contact e-mail:

dmitry.vostokov@dumpanalysis.org

Alternatively, via Twitter @DumpAnalysis

Facebook page and group:

http://www.facebook.com/DumpAnalysis

http://www.facebook.com/TraceAnalysis

http://www.facebook.com/groups/dumpanalysis

[This page is intentionally left blank]

About the Author

Dmitry Vostokov is an internationally recognized expert, speaker, educator, scientist and author. He is the founder of pattern-oriented software diagnostics, forensics and prognostics discipline and Software Diagnostics Institute (DA+TA: DumpAnalysis.org + TraceAnalysis.org). Vostokov has also authored more than 30 books on software diagnostics, forensics and problem-solving, memory dump analysis, debugging, software trace and log analysis, reverse engineering, and malware analysis. He has more than 20 years of experience in software architecture, design, development and maintenance in a variety of industries including leadership, technical and people management roles. Dmitry also founded DiaThings, Logtellect, OpenTask Iterative and Incremental Publishing (OpenTask.com), Software Diagnostics Services (former Memory Dump Analysis Services) PatternDiagnostics.com and Software Prognostics. In his spare time, he presents various topics on Debugging.TV and explores Software Narratology, an applied science of software stories that he pioneered, and its further development as Narratology of Things and Diagnostics of Things (DoT). His current area of interest is theoretical software diagnostics.

[This page is intentionally left blank]

Win32 Start Address Fallacy

One of the common mistakes is not double-checking symbolic output (Volume 5, page 21). Another example here is related to *Win32 Start Address*. In the output of **!thread** WinDbg command (or **!process** and **!sprocess Stack Trace Collection** commands, Volume 1, page 409) we can see *Win32 Start Address* and, in cases of **Truncated Stack Traces** (Volume 6, page 86) or **No Component Symbols** (Volume 1, page 298), we may use this information to guess the purpose of the thread. Unfortunately, it is shown without function offsets and may give a false sense of the thread purpose.

For example, this *Win32 Start Address ModuleA!DoSomething* may suggest that the purpose of the thread was to *DoSomething*:

```
THREAD fffffa803431cb50 Cid 03e8.2718 Teb: 000007fffff80000
Win32Thread: 0000000000000000 WAIT: (UserRequest) UserMode Non-
Alertable
fffffa80330e0500 SynchronizationEvent
Impersonation token: fffff8a00b807060 (Level Impersonation)
Owning Process fffffa8032354c40 Image: ServiceA.exe
Attached Process N/A            Image: N/A
Wait Start TickCount 107175     Ticks: 19677 (0:00:05:06.963)
Context Switch Count 2303       IdealProcessor: 1
UserTime        00:00:00.218
KernelTime      00:00:00.109
Win32 Start Address ModuleA!DoSomething (0x000007fef46b4cde)
Stack Init fffff88008e5fdb0 Current fffff88008e5f900
Base fffff88008e60000 Limit fffff88008e5a000 Call 0
Priority 10 BasePriority 10 UnusualBoost 0 ForegroundBoost 0
IoPriority 2 PagePriority 5
Kernel stack not resident.
Child-SP RetAddr Call Site
fffff880`08e5f940 fffff800`01c7cf72 nt!KiSwapContext+0x7a
fffff880`08e5fa80 fffff800`01c8e39f nt!KiCommitThreadWait+0x1d2
fffff880`08e5fb10 fffff800`01f7fe3e nt!KeWaitForSingleObject+0x19f
fffff880`08e5fbb0 fffff800`01c867d3 nt!NtWaitForSingleObject+0xde
fffff880`08e5fc20 00000000`76e5067a nt!KiSystemServiceCopyEnd+0x13
(TrapFrame @ fffff880`08e5fc20)
00000000`0427cca8 000007fe`f46a4afe ntdll!NtWaitForSingleObject+0xa
00000000`0427ccb0 000007fe`f46c68d4 ModuleA!DoSomething+0xc68d4
00000000`0427cd60 000007fe`f46c6ade ModuleA!DoSomething+0xc5ee8
```

But if we look at fragments of the stack trace we see function huge offsets and this means that this function was just some function from *ModuleA* export table. It was chosen because return addresses fall into an address range between exported functions. Because *Win32 Start Address* also falls into such an address range it is listed as *ModuleA!DoSomething* but without an offset. In our case, an engineer made the wrong assumption about the possible root cause and provided unnecessary troubleshooting instructions.

Multidimensionality of Exceptions

Multiple Exceptions pattern (Volume 1, page 255) can happen horizontally (different threads) and vertically (**Nested Exceptions** pattern, Volume 2, page 305) in one thread. The 3rd dimension is across processes (**Error Reporting Fault** pattern, Volume 7, page 152).

[This page is intentionally left blank]

PART 2: Crash Dump Analysis Patterns

Reference Leak

Objects such as processes may be referenced internally in addition to using handles. If their reference counts are unbalanced, we may have this pattern. For example, we have an instance of thousands of **Zombie Processes** (Volume 2, page 196) but we don't see **Handle Leaks** (Volume 7, page 164) from their parent processes if we analyze *ParentCid*s:

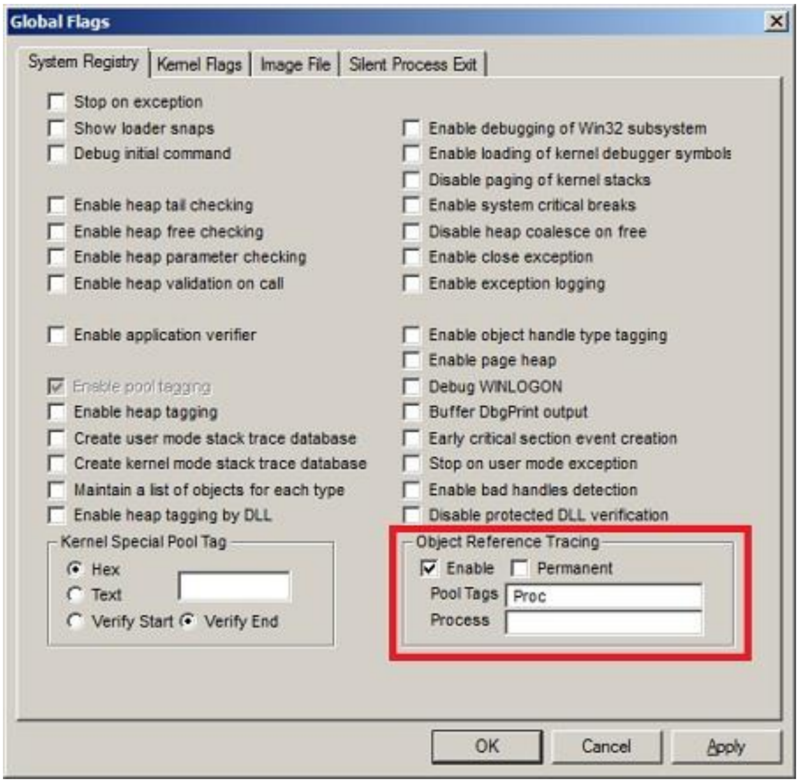
```
0: kd> !process 0 0
[...]
PROCESS fffffa801009a060
SessionId: 0 Cid: 2e270 Peb: 7fffffdb000 ParentCid: 032c
DirBase: 12ba37000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe

PROCESS fffffa8009b7e8e0
SessionId: 1 Cid: 2e0c8 Peb: 7fffffd9000 ParentCid: 10a0
DirBase: 21653e000 ObjectTable: 00000000 HandleCount: 0.
Image: taskmgr.exe

PROCESS fffffa8009e7a450
SessionId: 0 Cid: 2e088 Peb: 7efdf000 ParentCid: 0478
DirBase: 107f02000 ObjectTable: 00000000 HandleCount: 0.
Image: AppA.exe

PROCESS fffffa8009e794b0
SessionId: 0 Cid: 2e394 Peb: 7fffffd3000 ParentCid: 032c
DirBase: 210ffc000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe

PROCESS fffffa8009ed4060
SessionId: 0 Cid: 2dee4 Peb: 7efdf000 ParentCid: 0478
DirBase: 11b7c7000 ObjectTable: 00000000 HandleCount: 0.
Image: AppB.exe

PROCESS fffffa800a13bb30
SessionId: 0 Cid: 2e068 Peb: 7fffffd5000 ParentCid: 032c
DirBase: 1bb8c1000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe
```

```
PROCESS fffffa80096f26b0
SessionId: 0 Cid: 2e320 Peb: 7efdf000 ParentCid: 0478
DirBase: 6ad4c000 ObjectTable: 00000000 HandleCount: 0.
Image: AppC.exe

PROCESS fffffa8009c44060
SessionId: 0 Cid: 2e300 Peb: 7fffffdd000 ParentCid: 032c
DirBase: 10df06000 ObjectTable: 00000000 HandleCount: 0.
Image: conhost.exe
[...]

0: kd> !object fffffa800a13bb30
Object: fffffa800a13bb30 Type: (fffffa8006cecf30) Process
ObjectHeader: fffffa800a13bb00 (new version)
HandleCount: 0 PointerCount: 1

0: kd> !object fffffa8009b7e8e0
Object: fffffa8009b7e8e0 Type: (fffffa8006cecf30) Process
ObjectHeader: fffffa8009b7e8b0 (new version)
HandleCount: 0 PointerCount: 1
```

Such number of processes correlates with non-paged pool usage for process structures:

```
0: kd> !poolused 3
....
Sorting by NonPaged Pool Consumed

NonPaged Paged
Tag  Allocs    Frees     Diff  Used      Allocs Frees Diff Used

Proc 55488     60        55428 80328320  0      0     0    0    Process objects , Binary: nt!ps
File 51733526 51708737 24789 7150416   0      0     0    0    File objects
[...]
```

Here we recommend enabling object reference tracing either using *gflags.exe* or directly modifying registry:


Global Flags

System Registry | Kernel Flags | Image File | Silent Process Exit

- [ ] Stop on exception
- [ ] Show loader snaps
- [ ] Debug initial command
- [ ] Enable heap tail checking
- [ ] Enable heap free checking
- [ ] Enable heap parameter checking
- [ ] Enable heap validation on call
- [ ] Enable application verifier
- [✓] Enable pool tagging
- [ ] Enable heap tagging
- [ ] Create user mode stack trace database
- [ ] Create kernel mode stack trace database
- [ ] Maintain a list of objects for each type
- [ ] Enable heap tagging by DLL
- [ ] Enable debugging of Win32 subsystem
- [ ] Enable loading of kernel debugger symbols
- [ ] Disable paging of kernel stacks
- [ ] Enable system critical breaks
- [ ] Disable heap coalesce on free
- [ ] Enable close exception
- [ ] Enable exception logging
- [ ] Enable object handle type tagging
- [ ] Enable page heap
- [ ] Debug WINLOGON
- [ ] Buffer DbgPrint output
- [ ] Early critical section event creation
- [ ] Stop on user mode exception
- [ ] Enable bad handles detection
- [ ] Disable protected DLL verification

Kernel Special Pool Tag: [✓ Hex] Text; Verify Start [✓ Verify End]

Object Reference Tracing: [✓] Enable [ ] Permanent
Pool Tags: Proc
Process:

```
Key: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session
Manager\Kernel
Value: ObTracePoolTags
Type: REG_SZ
Data: Proc
```

Note: after troubleshooting or debugging please disable tracing because it consumes pool (another variant of **Instrumentation Side Effect** pattern, Volume 6, page 77, and may lead to similar **Insufficient Memory** pattern for stack trace database, Volume 8a, page 57):

```
0: kd> !poolused 3
....
Sorting by NonPaged Pool Consumed


NonPaged Paged
Tag  Allocs   Frees    Diff  Used        Allocs Frees Diff Used


ObRt 5688634 5676109 12525 4817288240 0 0 0 0 object reference stack tracing , Binary: nt!ob
Proc 22120 101 22019 25961168 0 0 0 0 Process objects , Binary: nt!ps
[...]
```

After enabling tracing, we collect a complete memory dump (in case of postmortem debugging) to analyze another variant of **Stack Trace** pattern using **!obtrace** WinDbg command (Volume 8a, page 51):

```
0: kd> !obtrace fffffa800af9e220
Object: fffffa800af9e220
Image: AppD.exe
Sequence (+/-) Tag Stack
-------- ----- ---- ---------------------------------------------
ad377858 +1 Dflt nt! ?? ::NNGAKEGL::`string'+21577
nt!PspAllocateProcess+185
nt!NtCreateUserProcess+4a3
nt!KiSystemServiceCopyEnd+13

ad37787d +1 Dflt nt! ?? ::FNODOBFM::`string'+18f1d
nt!NtCreateUserProcess+569
nt!KiSystemServiceCopyEnd+13

ad377882 +1 Dflt nt! ?? ::NNGAKEGL::`string'+1f9d8
nt!NtProtectVirtualMemory+119
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
nt!RtlCreateUserStack+1e4
nt!PspAllocateThread+299
nt!NtCreateUserProcess+65d
nt!KiSystemServiceCopyEnd+13

ad377884 -1 Dflt nt! ?? ::FNODOBFM::`string'+4886e
nt!NtProtectVirtualMemory+161
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
nt!RtlCreateUserStack+1e4
[...]
```

Analysis of such traces may be complicated due to **Truncated Stack Traces** (Volume 6, page 86). We plan to show one counting trick in the next pattern.

Origin Module

To complement Module patterns sub-catalogue (Volume 7, page 510) we introduce **Origin Module** pattern. This is a module that may have originated the problem behavior. For example, when we look at a stack trace we may skip **Top Modules** (Volume 6, page 62) due to our knowledge of the product, for example, if they are not known as **Problem Modules** (Volume 7, page 85) or known as **Well-Tested Modules** (Volume 6, page 48). In case of **Truncated Stack Traces** (Volume 7, page 86) we may designate bottom modules as possible problem origins. For example, for **Reference Leak** (page 15) pattern example we may consider checking reference counting for selected modules such as *ModuleA* and *ModuleB*:

```
ad377ae8 +1 Dflt nt! ?? ::FNODOBFM::`string'+18f1d
nt!ObpCallPreOperationCallbacks+4e
nt!ObpPreInterceptHandleCreate+af
nt! ?? ::NNGAKEGL::`string'+2c31f
nt!ObOpenObjectByPointerWithTag+109
nt!PsOpenProcess+1a2
nt!NtOpenProcess+23
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
ModuleA+dca63
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7

ad377aeb -1 Dflt nt! ?? ::FNODOBFM::`string'+4886e
nt!ObpCallPreOperationCallbacks+277
nt!ObpPreInterceptHandleCreate+af
nt! ?? ::NNGAKEGL::`string'+2c31f
nt!ObOpenObjectByPointerWithTag+109
nt!PsOpenProcess+1a2
nt!NtOpenProcess+23
nt!KiSystemServiceCopyEnd+13
nt!KiServiceLinkage+0
ModuleA+dca63
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7
```

```
ad377af7 +1 Dflt nt! ?? ::NNGAKEGL::`string'+1fb41
nt!ObReferenceObjectByHandle+25
ModuleA+dcade
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7
ModuleA+87ca
ModuleA+834a
ModuleA+a522c
ModuleA+a51b6
ModuleA+a4787
ModuleB+19c0c
ModuleB+19b28

ad377afa -1 Dflt nt! ?? ::FNODOBFM::`string'+4886e
ModuleA+dcbbe
ModuleA+b5bc
ModuleA+c9c2e
ModuleA+bae56
ModuleA+b938d
ModuleA+c0ec6
ModuleA+afce7
ModuleA+87ca
ModuleA+834a
ModuleA+a522c
ModuleA+a51b6
ModuleA+a4787
ModuleB+19c0c
ModuleB+19b28
ModuleB+b652
```

Hidden Call

Sometimes, due to optimization or indeterminate stack trace reconstruction, we may not see all stack trace frames. In some cases it is possible to reconstruct such **Hidden Calls**. For example, we have the following unmanaged **Stack Trace** (Volume 1, page 395) of **CLR Thread** (Volume 4, page 163):

```
0:000> k
ChildEBP RetAddr
0011d6b8 66fdee7c mscorwks!JIT_IsInstanceOfClass+0xd
0011d6cc 67578500 PresentationCore_ni!`string'+0x4a2bc
0011d6e0 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778500)
0011d6f4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011d708 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d71c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d730 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d744 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d758 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d76c 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d780 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011d794 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7a8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7bc 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7d0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7e4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d7f8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d80c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d820 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d834 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d848 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d85c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d870 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d884 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d898 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8ac 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8c0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8d4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8e8 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d8fc 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011d910 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d924 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d938 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d94c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d960 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d974 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d988 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d99c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9b0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9c4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9d8 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011d9ec 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011da00 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da14 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da28 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da3c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da50 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da64 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da78 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011da8c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011daa0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dab4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dac8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dadc 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
```

```
0011daf0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db04 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db18 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db2c 67578527 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db40 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x778527)
0011db54 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db68 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db7c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011db90 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dba4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbb8 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbcc 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbe0 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dbf4 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc08 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc1c 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc30 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc44 6757850d PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
0011dc58 66fc3282 PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x77850d)
*** WARNING: Unable to verify checksum for PresentationFramework.ni.dll
0011dd28 662a75e6 PresentationCore_ni!`string'+0x2e6c2
0011de08 662190a0 PresentationFramework_ni+0x2675e6
0011dffc 66fc35e2 PresentationFramework_ni+0x1d90a0
0011e0ec 66fd9dad PresentationCore_ni!`string'+0x2ea22
0011e214 66fe0459 PresentationCore_ni!`string'+0x451ed
0011e238 66fdfd40 PresentationCore_ni!`string'+0x4b899
0011e284 66fdfc9b PresentationCore_ni!`string'+0x4b180
*** WARNING: Unable to verify checksum for WindowsBase.ni.dll
0011e2b0 723ca31a PresentationCore_ni!`string'+0x4b0db
0011e2cc 723ca20a WindowsBase_ni+0x9a31a
0011e30c 723c8384 WindowsBase_ni+0x9a20a
0011e330 723cd26d WindowsBase_ni+0x98384
0011e368 723cd1f8 WindowsBase_ni+0x9d26d
0011e380 72841b4c WindowsBase_ni+0x9d1f8
0011e390 728589ec mscorwks!CallDescrWorker+0x33
0011e410 72865acc mscorwks!CallDescrWorkerWithHandler+0xa3
0011e54c 72865aff mscorwks!MethodDesc::CallDescr+0x19c
0011e568 72865b1d mscorwks!MethodDesc::CallTargetWorker+0x1f
0011e580 728bd9c8 mscorwks!MethodDescCallSite::CallWithValueTypes+0x1a
0011e74c 728bdb1e mscorwks!ExecuteCodeWithGuaranteedCleanupHelper+0x9f
*** WARNING: Unable to verify checksum for mscorlib.ni.dll
0011e7fc 68395887 mscorwks!ReflectionInvocation::ExecuteCodeWithGuaranteedCleanup+0x10f
0011e818 683804b5 mscorlib_ni+0x235887
0011e830 723cd133 mscorlib_ni+0x2204b5
0011e86c 723c7a27 WindowsBase_ni+0x9d133
0011e948 723c7d13 WindowsBase_ni+0x97a27
0011e984 723ca4fe WindowsBase_ni+0x97d13
0011e9d0 723ca42a WindowsBase_ni+0x9a4fe
0011e9f0 723ca31a WindowsBase_ni+0x9a42a
0011ea0c 723ca20a WindowsBase_ni+0x9a31a
0011ea4c 723c8384 WindowsBase_ni+0x9a20a
0011ea70 723c74e1 WindowsBase_ni+0x98384
0011eaac 723c7430 WindowsBase_ni+0x974e1
0011eadc 723c9b6c WindowsBase_ni+0x97430
0011eb2c 757462fa WindowsBase_ni+0x99b6c
0011eb58 75746d3a user32!InternalCallWinProc+0x23
0011ebd0 757477c4 user32!UserCallWinProcCheckWow+0x109
0011ec30 7574788a user32!DispatchMessageWorker+0x3bc
0011ec40 0577304e user32!DispatchMessageW+0xf
WARNING: Frame IP not in any known module. Following frames may be wrong.
0011ec5c 723c7b24 0x577304e
0011eccc 723c71f9 WindowsBase_ni+0x97b24
0011ecd8 723c719c WindowsBase_ni+0x971f9
0011ece4 6620f07e WindowsBase_ni+0x9719c
0011ecf0 6620e37f PresentationFramework_ni+0x1cf07e
0011ed14 661f56d6 PresentationFramework_ni+0x1ce37f
0011ed24 661f5699 PresentationFramework_ni+0x1b56d6
0011ed80 72841b4c PresentationFramework_ni+0x1b5699
0011eda0 72841b4c mscorwks!CallDescrWorker+0x33
0011edb0 728589ec mscorwks!CallDescrWorker+0x33
```

```
0011ee30 72865acc mscorwks!CallDescrWorkerWithHandler+0xa3
0011ef6c 72865aff mscorwks!MethodDesc::CallDescr+0x19c
0011ef88 72865b1d mscorwks!MethodDesc::CallTargetWorker+0x1f
0011efa0 728fef01 mscorwks!MethodDescCallSite::CallWithValueTypes+0x1a
0011f104 728fee21 mscorwks!ClassLoader::RunMain+0x223
0011f36c 728ff33e mscorwks!Assembly::ExecuteMainMethod+0xa6
0011f83c 728ff528 mscorwks!SystemDomain::ExecuteMainMethod+0x45e
0011f88c 728ff458 mscorwks!ExecuteEXE+0x59
0011f8d4 70aef4f3 mscorwks!_CorExeMain+0x15c
0011f90c 70b77efd mscoreei!_CorExeMain+0x10a
0011f924 70b74de3 mscoree!ShellShim__CorExeMain+0x7d
0011f92c 754c338a mscoree!_CorExeMain_Exported+0x8
0011f938 77659f72 kernel32!BaseThreadInitThunk+0xe
0011f978 77659f45 ntdll!__RtlUserThreadStart+0x70
0011f990 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Its **Managed Stack Trace** (Volume 6, page 115) is the following:

```
0:000> !CLRStack
OS Thread Id: 0x1520 (0)
ESP       EIP
0011e7a0 728493a4 [HelperMethodFrame_PROTECTOBJ: 0011e7a0]
System.Runtime.CompilerServices.RuntimeHelpers.ExecuteCodeWithGuaranteedCleanup(TryCode,
CleanupCode, System.Object)
0011e808 68395887
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0011e824 683804b5 System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0011e83c 723cd133 System.Windows.Threading.DispatcherOperation.Invoke()
0011e874 723c7a27 System.Windows.Threading.Dispatcher.ProcessQueue()
0011e950 723c7d13 System.Windows.Threading.Dispatcher.WndProcHook(IntPtr, Int32, IntPtr,
IntPtr, Boolean ByRef)
0011e99c 723ca4fe MS.Win32.HwndWrapper.WndProc(IntPtr, Int32, IntPtr, IntPtr, Boolean ByRef)
0011e9e8 723ca42a MS.Win32.HwndSubclass.DispatcherCallbackOperation(System.Object)
0011e9f8 723ca31a
System.Windows.Threading.ExceptionWrapper.InternalRealCall(System.Delegate, System.Object,
Boolean)
0011ea1c 723ca20a System.Windows.Threading.ExceptionWrapper.TryCatchWhen(System.Object,
System.Delegate, System.Object, Boolean, System.Delegate)
0011ea64 723c8384 System.Windows.Threading.Dispatcher.WrappedInvoke(System.Delegate,
System.Object, Boolean, System.Delegate)
0011ea84 723c74e1
System.Windows.Threading.Dispatcher.InvokeImpl(System.Windows.Threading.DispatcherPriority,
System.TimeSpan, System.Delegate, System.Object, Boolean)
0011eac8 723c7430
System.Windows.Threading.Dispatcher.Invoke(System.Windows.Threading.DispatcherPriority,
System.Delegate, System.Object)
0011eaec 723c9b6c MS.Win32.HwndSubclass.SubclassWndProc(IntPtr, Int32, IntPtr, IntPtr)
0011ec74 00270b04 [NDirectMethodFrameStandalone: 0011ec74]
MS.Win32.UnsafeNativeMethods.DispatchMessage(System.Windows.Interop.MSG ByRef)
0011ec84 723c7b24
System.Windows.Threading.Dispatcher.PushFrameImpl(System.Windows.Threading.DispatcherFrame)
0011ecd4 723c71f9
System.Windows.Threading.Dispatcher.PushFrame(System.Windows.Threading.DispatcherFrame)
0011ece0 723c719c System.Windows.Threading.Dispatcher.Run()
0011ecec 6620f07e System.Windows.Application.RunDispatcher(System.Object)
0011ecf8 6620e37f System.Windows.Application.RunInternal(System.Windows.Window)
0011ed1c 661f56d6 System.Windows.Application.Run(System.Windows.Window)
0011ed2c 661f5699 System.Windows.Application.Run()
[...]
```

Caller-n-Callee (Volume 6, page 138) traces also don’t reveal anything more:

```
Thread   0
Current frame: mscorwks!JIT_IsInstanceOfClass+0xd
ChildEBP RetAddr  Caller,Callee
0011d6b8 66fdee7c (MethodDesc 0x66ee2954 +0x3c
MS.Internal.DeferredElementTreeState.GetLogicalParent(System.Windows.DependencyObject,
MS.Internal.DeferredElementTreeState)), calling mscorwks!JIT_IsInstanceOfClass
0011d6cc 67578500 (MethodDesc 0x66ee1270 +0x110
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)),
calling (MethodDesc 0x66ee2954 +0
MS.Internal.DeferredElementTreeState.GetLogicalParent(System.Windows.DependencyObject,
MS.Internal.DeferredElementTreeState))
0011d6e0 67578527 (MethodDesc 0x66ee1270 +0x137
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)),
calling (MethodDesc 0x66ee1270 +0
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
0011d6f4 6757850d (MethodDesc 0x66ee1270 +0x11d
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)),
calling (MethodDesc 0x66ee1270 +0
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
0011d708 6757850d (MethodDesc 0x66ee1270 +0x11d
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)),
calling (MethodDesc 0x66ee1270 +0
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
0011d71c 6757850d (MethodDesc 0x66ee1270 +0x11d
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject)),
calling (MethodDesc 0x66ee1270 +0
MS.Internal.UIElementHelper.InvalidateAutomationAncestors(System.Windows.DependencyObject))
[...]
```

However, if we check the return address for **Top Module** (Volume 6, page 62) *mscorwks* (66fdee7c) we will see a call possibly related to 3D processing:

```
0:000> k
ChildEBP RetAddr
0011d6b8 66fdee7c mscorwks!JIT_IsInstanceOfClass+0xd
0011d6cc 67578500 PresentationCore_ni!`string’+0×4a2bc
0011d6e0 67578527 PresentationCore_ni!`string’ <PERF>
(PresentationCore_ni+0×778500)
0011d6f4 6757850d PresentationCore_ni!`string’ <PERF>
(PresentationCore_ni+0×778527)
[…]


0:000> ub 66fdee7c
PresentationCore_ni!`string'+0x4a2a2:
66fdee62 740c            je      PresentationCore_ni!`string'+0x4a2b0
(66fdee70)
66fdee64 8bc8            mov     ecx,eax
66fdee66 8b01            mov     eax,dword ptr [ecx]
66fdee68 ff90d8030000    call    dword ptr [eax+3D8h]
66fdee6e 8bf0            mov     esi,eax
66fdee70 8bd7            mov     edx,edi
66fdee72 b998670467      mov     ecx,offset
PresentationCore_ni!`string'+0xb1bd8 (67046798)
66fdee77
e82c7afaff       call    PresentationCore_ni!?System.Windows.Media.Media3D.Viewpo
rt3DVisual.PrecomputeContent@@200001+0×3c (66f868a8)
```

The call structure seems to be valid when we check the next return address from the stack trace (67578500):

```
0:000> ub 67578500
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784e7):
675784e7
e8f4a2a0ff      call    PresentationCore_ni!?System.Windows.Media.Media3D.Scale
Transform3D.UpdateResource@@2002011280M802+0x108 (66f827e0)
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784ec):
675784ec eb05            jmp     PresentationCore_ni!`string' <PERF>
(PresentationCore_ni+0x7784f3) (675784f3)
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784ee):
675784ee b801000000      mov     eax,1
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f3):
675784f3 85c0            test    eax,eax
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f5):
675784f5 74b1            je      PresentationCore_ni!`string' <PERF>
(PresentationCore_ni+0x7784a8) (675784a8)
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f7):
675784f7 8bcb            mov     ecx,ebx
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784f9):
675784f9 33d2            xor     edx,edx
PresentationCore_ni!`string' <PERF> (PresentationCore_ni+0x7784fb):
675784fb e84069a6ff      call    PresentationCore_ni!`string’+0×4a280
(66fdee40)
```

Corrupt Structure

Corrupt Structure pattern is added for completeness of pattern discourse. We mentioned it a few times, for example, in **Self-Diagnosis** (kernel mode, Volume 6, 89), and **Critical Section Corruption** (Volume 2, page 324). Typical signals of the corrupt structure include:

- **Regular Data** (Volume 7, page 106) such as ASCII and UNICODE fragments over substructures and pointer areas
- Large values where you expect small and vice versa
- User space address values where we expect kernel space and vice versa
- Malformed and partially zeroed _LIST_ENTRY data (see exercise C3[1] for linked list navigation)
- Memory read errors for pointer dereferences or inaccessible memory indicators (??)
- Memory read error at the end of the linked list while traversing structures

```
0: kd> dt _ERESOURCE ffffd0002299f830
ntdll!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY [ 0xffffc000`07b64800 -
0xffffe000`02a79970 ]
+0x010 OwnerTable         : 0xffffe000`02a79940 _OWNER_ENTRY
+0x018 ActiveCount        : 0n0
+0x01a Flag               : 0
+0x01a ReservedLowFlags : 0 ''
+0x01b WaiterPriority     : 0 ''
+0x020 SharedWaiters      : 0x00000000`00000001 _KSEMAPHORE
+0x028 ExclusiveWaiters : 0xffffe000`02a79a58 _KEVENT
+0x030 OwnerEntry         : _OWNER_ENTRY
+0x040 ActiveEntries      : 0
+0x044 ContentionCount    : 0
+0x048 NumberOfSharedWaiters : 0x7b64800
+0x04c NumberOfExclusiveWaiters : 0xffffc000
+0x050 Reserved2          : (null)
+0x058 Address            : 0xffffd000`2299f870 Void
+0x058 CreatorBackTraceIndex : 0xffffd000`2299f870
+0x060 SpinLock           : 1
```

[1] http://www.patterndiagnostics.com/advanced-windows-memory-dump-analysis-book

```
0: kd> dt _ERESOURCE ffffd0002299d830
ntdll!_ERESOURCE
+0×000 SystemResourcesList : _LIST_ENTRY [ 0×000001e0`00000280 -
0×00000000`00000004 ]
+0×010 OwnerTable          : 0×00000000`0000003c _OWNER_ENTRY
+0×018 ActiveCount         : 0n0
+0×01a Flag                : 0
+0×01a ReservedLowFlags : 0 ”
+0×01b WaiterPriority      : 0 ”
+0×020 SharedWaiters       : 0×0000003c`000001e0 _KSEMAPHORE
+0×028 ExclusiveWaiters : (null)
+0×030 OwnerEntry          : _OWNER_ENTRY
+0×040 ActiveEntries       : 0
+0×044 ContentionCount     : 0×7f
+0×048 NumberOfSharedWaiters : 0×7f
+0×04c NumberOfExclusiveWaiters : 0×7f
+0×050 Reserved2           : 0×00000001`00000001 Void
+0×058 Address             : 0×00000000`00000005 Void
+0×058 CreatorBackTraceIndex : 5
+0×060 SpinLock            : 0
```

However, we need to be sure that we supplied the correct pointer to **dt** WinDbg command. One of the signs that the pointer was incorrect are memory read errors or all zeroes:

```
0: kd> dt _ERESOURCE ffffd000229af830
ntdll!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY [ 0x00000000`00000000 -
0x00000000`00000000 ]
+0x010 OwnerTable : (null)
+0x018 ActiveCount : 0n0
+0x01a Flag : 0
+0x01a ReservedLowFlags : 0 ''
+0x01b WaiterPriority : 0 ''
+0x020 SharedWaiters : (null)
+0x028 ExclusiveWaiters : (null)
+0x030 OwnerEntry : _OWNER_ENTRY
+0x040 ActiveEntries : 0
+0x044 ContentionCount : 0
+0x048 NumberOfSharedWaiters : 0
+0x04c NumberOfExclusiveWaiters : 0
+0x050 Reserved2 : (null)
+0x058 Address : (null)
+0x058 CreatorBackTraceIndex : 0
+0x060 SpinLock : 0
```

```
0: kd> dt _ERESOURCE ffffd00022faf830
ntdll!_ERESOURCE
+0x000 SystemResourcesList : _LIST_ENTRY
+0x010 OwnerTable        : ????
+0x018 ActiveCount       : ??
+0x01a Flag              : ??
+0x01a ReservedLowFlags : ??
+0x01b WaiterPriority    : ??
+0x020 SharedWaiters     : ????
+0x028 ExclusiveWaiters : ????
+0x030 OwnerEntry        : _OWNER_ENTRY
+0x040 ActiveEntries     : ??
+0x044 ContentionCount   : ??
+0x048 NumberOfSharedWaiters : ??
+0x04c NumberOfExclusiveWaiters : ??
+0x050 Reserved2         : ????
+0x058 Address           : ????
+0x058 CreatorBackTraceIndex : ??
+0x060 SpinLock          : ??
Memory read error ffffd00022faf890
```

Software Exception

Software Exception is added for completeness of pattern discourse. We mentioned it a few times before, for example, in **Activation Context** (Volume 6, page 117), **Exception Module** (Volume 8a, page 80), **Missing Component** (static linkage, Volume 2, page 283), **Self-Dump** (Volume 2, page 181), **Stack Overflow** (software implementation, Volume 6, page 82), and **Translated Exception** (Volume 7, page 107) patterns. A typical example of software exceptions is **C++ Exception** (Volume 3, page 84) pattern.

Software exceptions, such as *not enough memory*, are different from the so-called *hardware exceptions* by being predictable, synchronous, and detected by software code itself. Hardware exceptions such as *divide by zero*, *access violation*, and *memory protection*, on the contrary, are unpredictable and detected by hardware. Of course, it is possible to do some checks before code execution, and then throw a software exception or some diagnostic message for a would be hardware exception. See, for example, **Self-Diagnosis** pattern for user mode (Volume 2, page 318) and its corresponding equivalent for kernel mode (Volume 6, page 89).

In Windows memory dumps we may see *RaiseException* call in user space stack trace, such as from **Data Correlation** (Volume 6, page 84) pattern example:

```
0:000> kL
ChildEBP RetAddr
0012e950 78158e89 kernel32!RaiseException+0×53
0012e988 7830770c msvcr80!_CxxThrowException+0×46
0012e99c 783095bc mfc80u!AfxThrowMemoryException+0×19
0012e9b4 02afa8ca mfc80u!operator new+0×27
0012e9c8 02b0992f ModuleA!std::_Allocate<…>+0×1a
0012e9e0 02b09e7c ModuleA!std::vector<double,std::allocator
>::vector<double,std::allocator >+0×3f
[…]</double,std::allocator</double,std::allocator
```

When looking for **Multiple Exceptions** (Volume 1, page 255) or **Hidden Exceptions** (Volume 1, page 271) we may also want to check for such calls.

Crashed Process

Sometimes we can see signs of **Crashed Processes** in the kernel and complete memory dumps. By crashes (Volume 1, page 36) we mean the sudden disappearance of processes from Task Manager, for example. In memory dumps, we can still see such processes as **Zombie Processes** (Volume 2, page 196). **Special Processes** (Volume 2, page 164) found in the process list may help to select the possible candidate among many **Zombie Processes**. If a process is supposed to be launched only once (as a service) but found several times as **Zombie Process** and also as a normal process later in the process list (for example, as **Last Object**, Volume 8a, page 37), then this may point to possible past crashes (or silent terminations). We also have a similar trace analysis pattern: **Singleton Event** (Volume 8a, page 108). The following example illustrates both signs:

```
0: kd> !process 0 0

[...]

PROCESS fffffa80088a5640
SessionId: 0 Cid: 2184 Peb: 7fffffd7000 ParentCid: 0888
DirBase: 381b8000 ObjectTable: 00000000 HandleCount: 0.
Image: WerFault.exe

PROCESS fffffa8007254b30
SessionId: 0 Cid: 20ac Peb: 7fffffdf000 ParentCid: 02cc
DirBase: b3306000 ObjectTable: 00000000 HandleCount: 0.
Image: ServiceA.exe

[...]

PROCESS fffffa8007fe2b30
SessionId: 0 Cid: 2a1c Peb: 7fffffdf000 ParentCid: 02cc
DirBase: 11b649000 ObjectTable: fffff8a014939530 HandleCount: 112.
Image: ServiceA.exe
```

Variable Subtrace

When analyzing **Spiking Threads** (Volume 1, page 305) across **Snapshot Collection** (Volume 5, page 346) we are interested in finding a module (or a function) that was most likely responsible (for example, “looping” inside). Here we can compare the same thread stack trace from different memory dumps and find their **Variable Subtrace**. For such subtraces, we have changes in the **kv**-style output: in return addresses, stack frame values, and possible arguments. The call site that starts the variable subtrace is the most likely candidate (subject to the number of snapshots). For example, consider the following pseudo code:

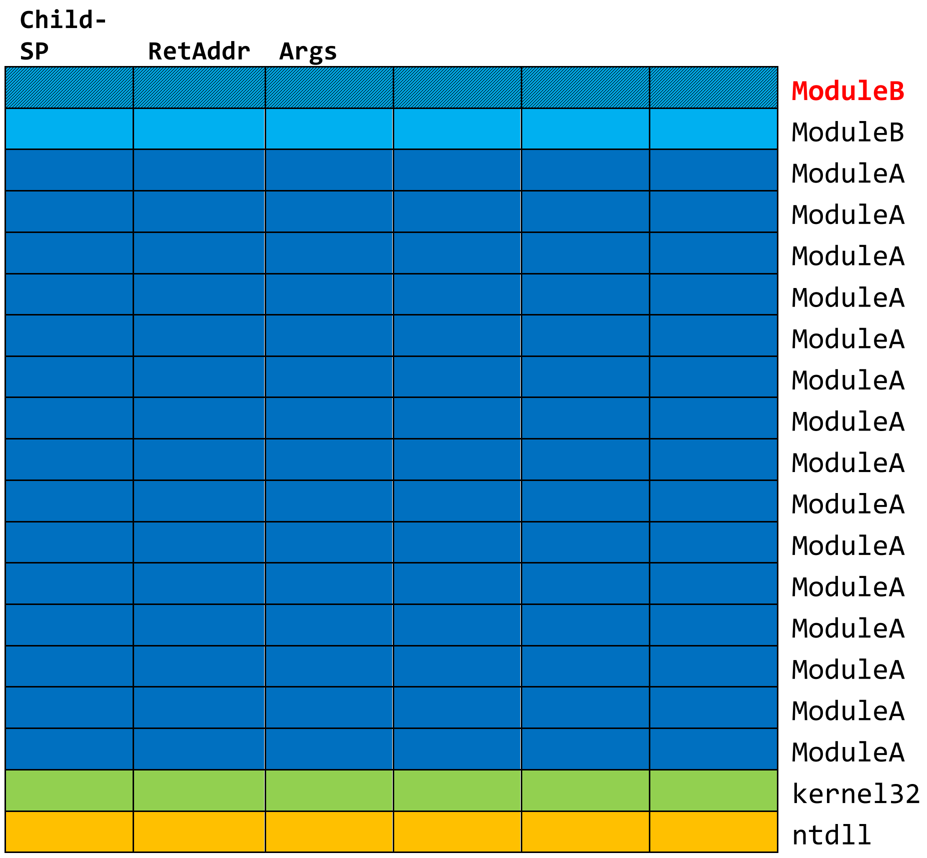
```
ModuleA!start()
{
    ModuleA!func1();
}
ModuleA!func1()
{
    ModuleB!func2();
}
ModuleB!func2()
{
    while (…)
    {
        ModuleB!func3();
    }
}
ModuleB!func3()
{
    ModuleB!func4();
}
ModuleB!func4()
{
    ModuleB!func5();
}
ModuleB!func5()
{
    // ...
}
```

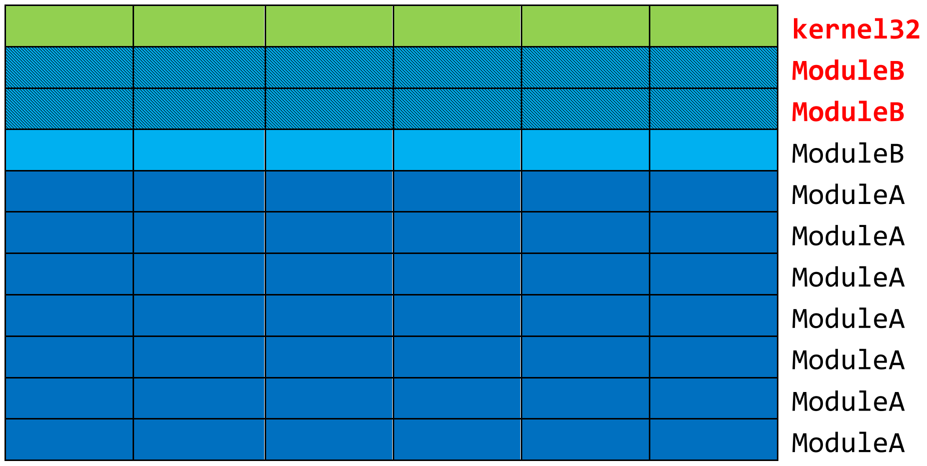
Here, the variable stack trace part will correspond to *ModuleB* frames. The memory dump can be saved anywhere inside the “while” loop and down the calls, and the last variable return address down the stack trace will belong to *ModuleB!func2* address range. The non-variable part will start with *ModuleA!func1* address range:

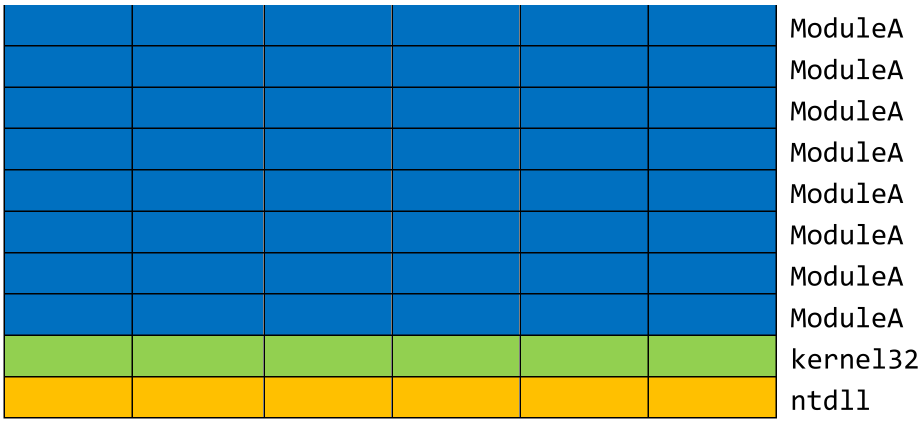
```
// snapshot 1

RetAddr
ModuleB!func4+0×20
ModuleB!func3+0×10
ModuleB!func2+0×40
ModuleA!func1+0×10
ModuleA!start+0×300

// snapshot 2

RetAddr
ModuleB!func2+0×20
ModuleA!func1+0×10
ModuleA!start+0×300

// snapshot 3

RetAddr
ModuleB!func3+0×20
ModuleB!func2+0×40
ModuleA!func1+0×10
ModuleA!start+0×300
```

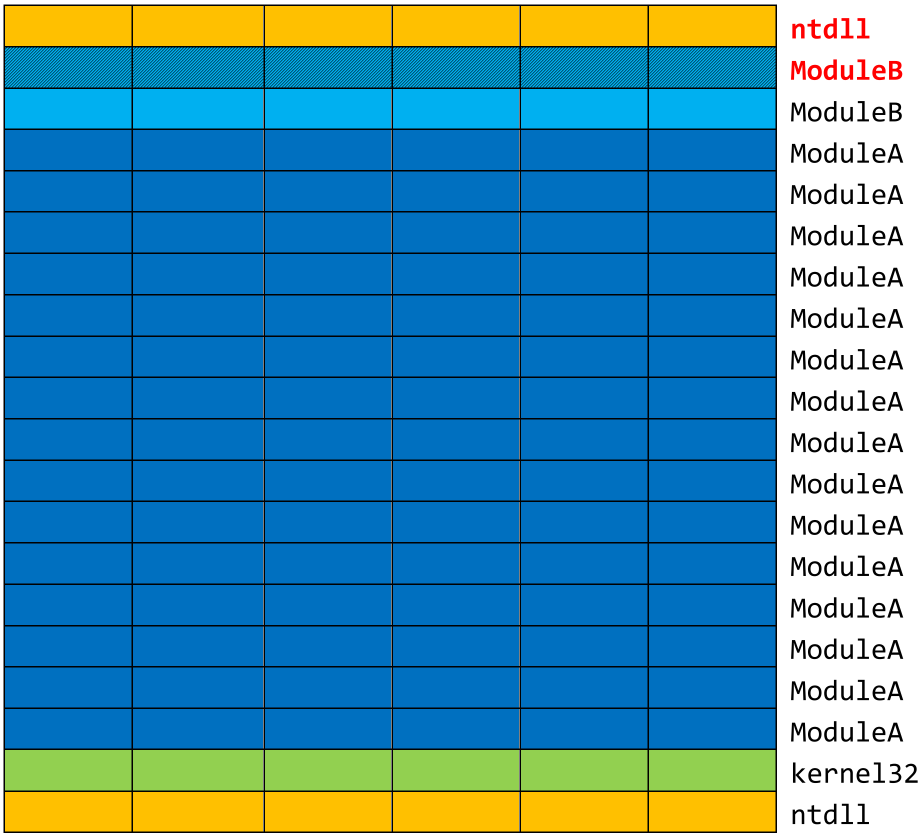
To illustrate this analysis pattern we adopted Memory Cell Diagram (MCD) approach from Accelerated Disassembly, Reconstruction and Reversing[2] training and introduce here Abstract Stack Trace Notation (ASTN) diagrams where different colors are used for different modules and changes are highlighted with different fill patterns. The following three ASTN diagrams from subsequently saved process memory dumps illustrate real stack traces we analyzed some time ago. We see that the variable subtrace contains only the 3rd-party *ModuleB* calls. Moreover, the loop is possibly contained inside *ModuleB* because all *ModuleA* frames are non-variable including *Child-SP* and *Args* column values.

[2] http://www.dumpanalysis.org/accelerated-disassembly-reconstruction-reversing-book

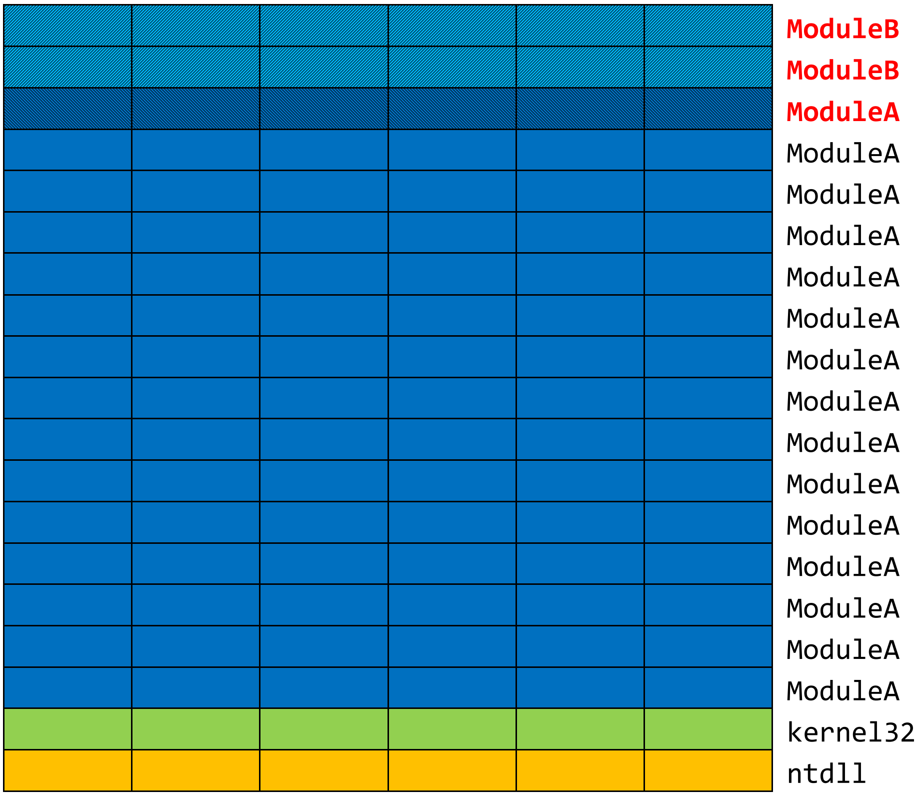

Child-SP
RetAddr
Args
ModuleB
ModuleB
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
kernel32
ntdll

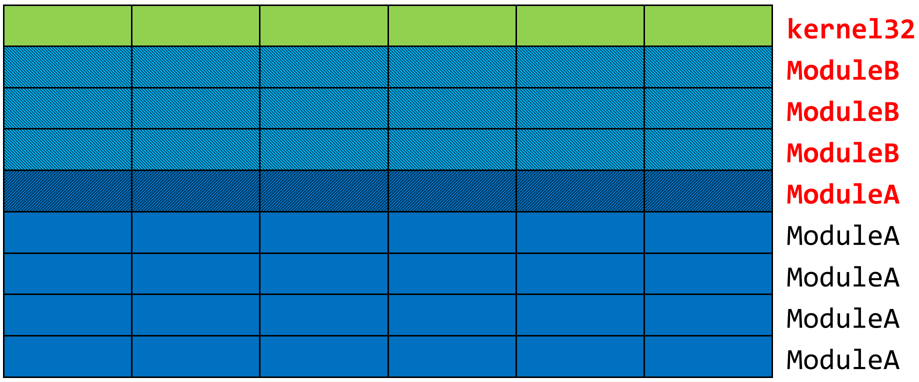

kernel32
ModuleB
ModuleB
ModuleB
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA

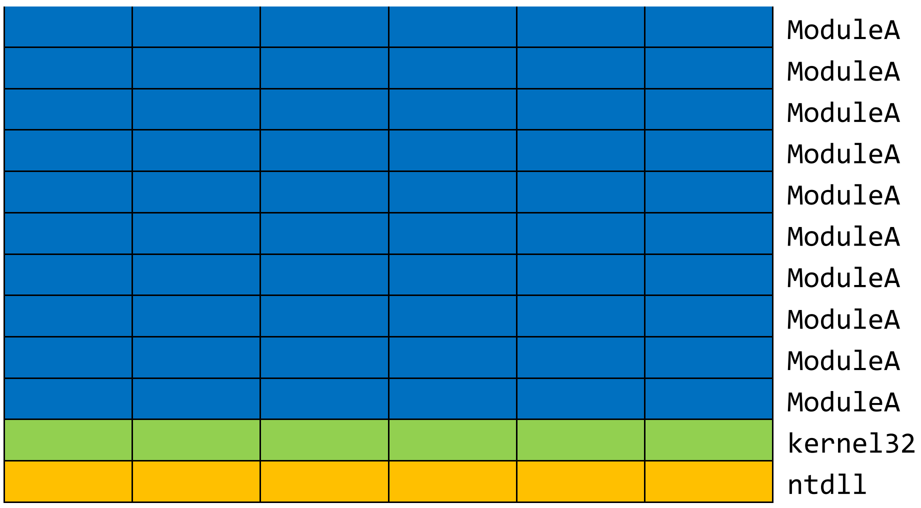

ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
kernel32
ntdll

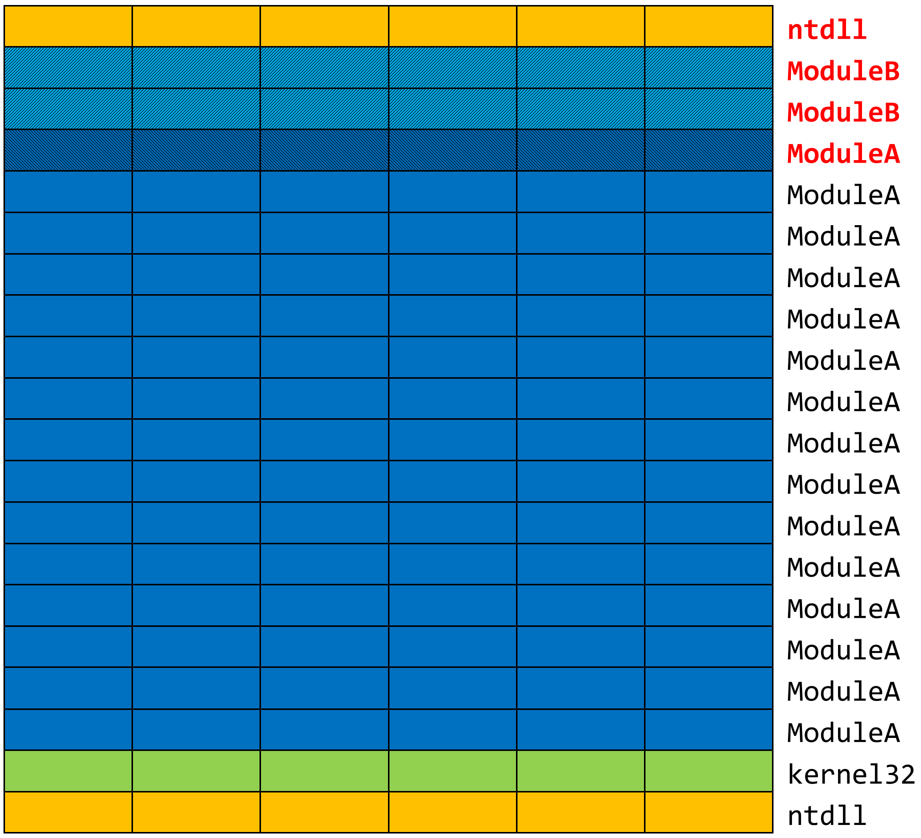

ntdll
ModuleB
ModuleB
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
kernel32
ntdll

If we had ASTN diagrams below instead we would have concluded that the loop was in *ModuleA* with changes in *ModuleB* columns as an execution side effect:






ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
kernel32
ntdll


ntdll
ModuleB
ModuleB
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
ModuleA
kernel32
ntdll

User Space Evidence

One of the questions asked was what can we do if we got a kernel memory dump instead of the requested complete memory dump? Can it be useful? Of course, if we requested a complete memory dump after analyzing a kernel memory dump then the second kernel dump may be useful for double checking. Therefore, we assume that we just got a kernel memory dump for the first time, and the issue is some performance issue or system freeze and not a bugcheck. If we have a bugcheck, then kernel memory dumps are sufficient most of the time, and we do not consider them for this pattern.

Such a kernel memory dump is still useful because of user space diagnostic indicators pointing to possible patterns in user space or “interspace”. We call this pattern **User Space Evidence**. It is a collective super-pattern like **Historical Information** (Volume 1, page 458).

We can see patterns in kernel memory dumps such as **Wait Chains** (for example, **ALPC**, Volume 3, page 97, or **Process Objects**, Volume 5, page 49), **Deadlocks** (for example **ALPC**, Volume 1, page 474), kernel stack traces corresponding to specific **Dual Stack Traces** (Volume 6, page 52, for example, exception processing), **Handle Leaks** (Volume 1, page 327), **Missing Threads** (Volume 1, page 362), **Module Product Process** (Volume 7, page 189), **One-Thread Processes** (Volume 7, page 187), **Spiking Thread** (Volume 1, page 305), **Process Factory** (Volume 3, page 112, for example, PPID for **Zombie Processes**, Volume 2, page 196), and others.

Found evidence may point to specific processes and process groups (**Couples Processes**, Volume 1, page 419, and session processes) and suggest process memory dump collection (especially forcing further complete memory dumps is problematic) or troubleshooting steps for diagnosed processes.

Technology-Specific Subtrace (COM Client Call)

Here we add yet another **Technology-Specific Subtrace** pattern for **COM client calls** (as compared to COM interface invocation for servers, Volume 6, page 67). We recently got a complete memory dump where we had to find the destination server process, and we used the old technique described in the article **In Search of Lost CID** (Volume 2, page 136). We reprint the 32-bit stack subtrace trace here:

```
[...]
00faf828 7778c38b
ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0x112
00faf908 776c0565 ole32!CRpcChannelBuffer::SendReceive2+0xd3
00faf974 776c04fa ole32!CAptRpcChnl::SendReceive+0xab
00faf9c8 77ce247f ole32!CCtxComChnl::SendReceive+0×1a9
00faf9e4 77ce252f RPCRT4!NdrProxySendReceive+0×43
00fafdcc 77ce25a6 RPCRT4!NdrClientCall2+0×206
[...]
```

Here's also an x64 fragment from **Semantic Structures** (PID.TID) pattern (Volume 6, page 73):

```
[...]
00000000`018ce450 000007fe`ffee041b
ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0xa3
00000000`018ce4f0 000007fe`ffd819c6 ole32!CRpcChannelBuffer::SendReceive2+0×11b
00000000`018ce6b0 000007fe`ffd81928 ole32!CAptRpcChnl::SendReceive+0×52
00000000`018ce780 000007fe`ffedfcf5 ole32!CCtxComChnl::SendReceive+0×68
00000000`018ce830 000007fe`ff56ba3b ole32!NdrExtpProxySendReceive+0×45
00000000`018ce860 000007fe`ffee02d0 RPCRT4!NdrpClientCall3+0×2e2
[...]
```

If we have the call over ALPC it is easy to find the server process and thread (**Wait Chain**, Volume 3, page 97). In case of a modal loop, we can use the raw stack analysis technique mentioned above (see also the case study from Volume 3, page 205).

Other subtrace examples can be found in pattern examples for **High Contention** (.NET CLR monitors, Volume 7, page 142), **Wait Chain** (RTL_RESOURCE, Volume 8a, page 29), and in the case study from Volume 4, page 182.

Internal Stack Trace

Occasionally, we look at **Stack Trace Collection** (Volume 1, page 409) and notice **Internal Stack Trace**. This is a stack trace that is shouldn't be seen in a normal crash dump because statistically it is rare (we planned to name this pattern **Rare Stack Trace** initially). This stack trace is also not **Special Stack Trace** (Volume 1, page 479) because it is not associated with the special system events or problems. It is also not a stack trace that belongs to various **Wait Chains** (Volume 1, page 482) or **Spiking Threads** (Volume 1, page 305). This is also a real stack trace and not a reconstructed or hypothetical stack trace such as **Rough Stack Trace** (Volume 8a, page 39) or **Past Stack Trace** (Volume 8a, page 43). This is simply a thread stack trace that shows some internal operation, for example, where it suggests that message hooking was involved:

```
THREAD fffffa8123702b00 Cid 11cc.0448 Teb: 000007fffffda000 Win32Thread:
fffff900c1e6ec20 WAIT: (WrUserRequest) UserMode Non-Alertable
fffffa81230cf4e0 SynchronizationEvent
Not impersonating
DeviceMap fffff8a0058745e0
Owning Process fffffa81237a8b30 Image: ProcessA.exe
Attached Process N/A Image: N/A
Wait Start TickCount 1258266 Ticks: 18 (0:00:00:00.280)
Context Switch Count 13752 IdealProcessor: 1 NoStackSwap LargeStack
UserTime 00:00:00.468
KernelTime 00:00:00.187
Win32 Start Address ProcessA!ThreadProc (0×000007feff17c608)
Stack Init fffff8800878c700 Current fffff8800878ba10
Base fffff8800878d000 Limit fffff88008781000 Call fffff8800878c750
Priority 12 BasePriority 8 UnusualBoost 0 ForegroundBoost 2 IoPriority 2
PagePriority 5
Child-SP RetAddr Call Site
fffff880`0878ba50 fffff800`01a6c8f2 nt!KiSwapContext+0×7a
fffff880`0878bb90 fffff800`01a7dc9f nt!KiCommitThreadWait+0×1d2
fffff880`0878bc20 fffff960`0010dbd7 nt!KeWaitForSingleObject+0×19f
fffff880`0878bcc0 fffff960`0010dc71 win32k!xxxRealSleepThread+0×257
fffff880`0878bd60 fffff960`000c4bf7 win32k!xxxSleepThread+0×59
fffff880`0878bd90 fffff960`000d07a5 win32k!xxxInterSendMsgEx+0×112a
fffff880`0878bea0 fffff960`00151bf8 win32k!xxxCallHook2+0×62d
fffff880`0878c010 fffff960`000d2454 win32k!xxxCallMouseHook+0×40
fffff880`0878c050 fffff960`0010bf23 win32k!xxxScanSysQueue+0×1828
fffff880`0878c390 fffff960`00118fae win32k!xxxRealInternalGetMessage+0×453
fffff880`0878c470 fffff800`01a76113 win32k!NtUserRealInternalGetMessage+0×7e
fffff880`0878c500 00000000`771b913a nt!KiSystemServiceCopyEnd+0×13 (TrapFrame @
fffff880`0878c570)
00000000`053ff258 000007fe`fac910f4 USER32!NtUserRealInternalGetMessage+0xa
00000000`053ff260 000007fe`fac911fa DUser!CoreSC::xwProcessNL+0×173
00000000`053ff2d0 00000000`771b9181 DUser!MphProcessMessage+0xbd
00000000`053ff330 00000000`774111f5 USER32!_ClientGetMessageMPH+0×3d
00000000`053ff3c0 00000000`771b908a ntdll!KiUserCallbackDispatcherContinue
(TrapFrame @ 00000000`053ff288)
00000000`053ff438 00000000`771b9055 USER32!NtUserPeekMessage+0xa
```

```
00000000`053ff440 000007fe`ebae03fa USER32!PeekMessageW+0×105
00000000`053ff490 000007fe`ebae4925 ProcessA+0×5a
[…]
00000000`053ff820 00000000`773ec541 kernel32!BaseThreadInitThunk+0xd
00000000`053ff850 00000000`00000000 ntdll!RtlUserThreadStart+0×1d
```

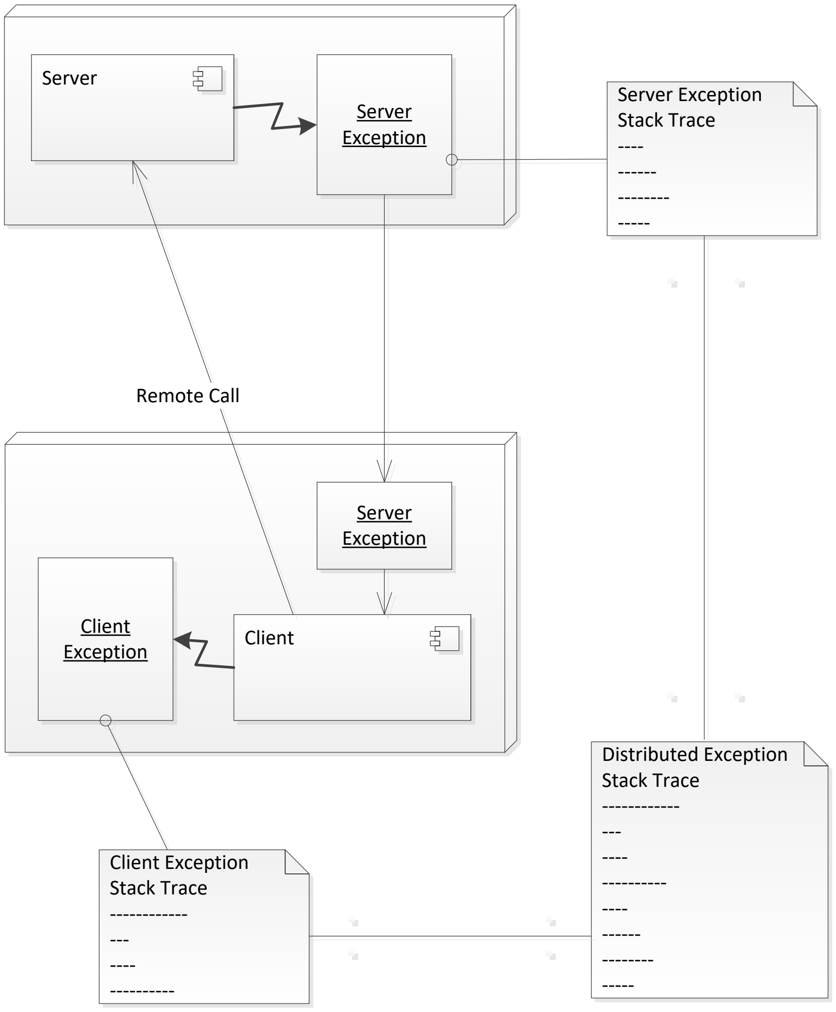
We see that this thread was neither waiting for significant time nor consuming CPU. It was reported that *ProcessA.exe* was very slow responding. So perhaps this was slowly punctuated thread execution with periodic small waits. In fact, **Execution Residue** (Volume 2, page 239) analysis revealed Non-**Coincidental Symbolic Information** (Volume 1, page 390) of the 3rd-party **Message Hook** (Volume 5, page 76) and its **Module Product Process** (Volume 7, page 189) was identified. Its removal resolved the problem.

Distributed Exception (Managed Code)

Managed code **Nested Exceptions** (Volume 2, page 310) give us process virtual space bound stack traces. However, exception objects may be marshaled across processes and even computers. The remote stack trace return addresses don't have the same validity in different process contexts. Fortunately, there is a *_remoteStackTraceString* field in exception objects, and it contains the original stack trace. Default analysis command sometimes uses it:

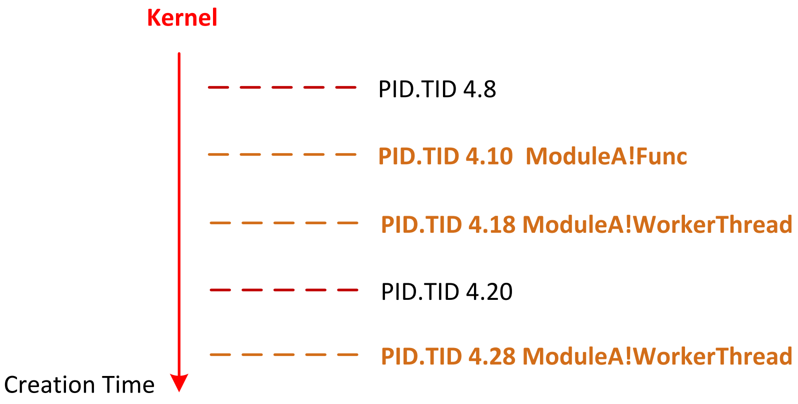
```
0:013> !analyze -v

[...]

EXCEPTION_OBJECT: !pe 25203b0
Exception object: 00000000025203b0
Exception type: System.Reflection.TargetInvocationException
Message: Exception has been thrown by the target of an invocation.
InnerException: System.Management.Instrumentation.WmiProviderInstallationException, Use
!PrintException 0000000002522cf0 to see more.
StackTrace (generated):
SP IP Function
000000001D39E720 0000000000000001 Component!Proxy.Start()+0x20
000000001D39E720 000007FEF503D0B6
mscorlib_ni!System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)+0x286
000000001D39E880 000007FEF503CE1A
mscorlib_ni!System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object, Boolean)+0xa
000000001D39E8B0 000007FEF503CDD8
mscorlib_ni!System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)+0x58
000000001D39E900 000007FEF4FB0302
mscorlib_ni!System.Threading.ThreadHelper.ThreadStart()+0x52

[...]

MANAGED_STACK_COMMAND: ** Check field _remoteStackTraceString **;!do 2522cf0;!do 2521900

[...]

0:013> !DumpObj 2522cf0
[...]
000007fef51b77f0 4000054 2c System.String 0 instance 2521900 _remoteStackTraceString
[…]

0:013> !DumpObj 2521900
Name: System.String
[…]
String: at System.Management.Instrumentation.InstrumentationManager.RegisterType(Type
managementType)
at Component.Provider..ctor()
at Component.Start()
```

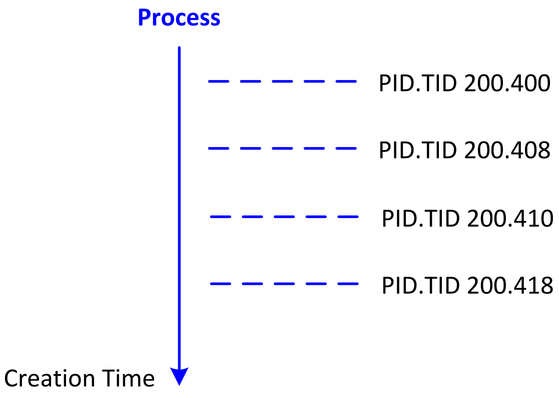
Checking this field may also be necessary for exceptions of interest from managed space **Execution Residue** (Volume 6, page 149). We call this pattern **Distributed Exception**. The basic idea is illustrated in the following diagram using the borrowed UML notation (not limited to just two computers):



Thread Poset

Predicate Stack Trace Collections (Volume 7, page 100) allow us to get a subset of stack traces, for example, by showing only stack traces where a specific module is used (for example, **!stacks 2** *module* WinDbg command). From diagnostic analysis perspective, the order in which threads from the subset appear is also important, especially when the output is sorted by thread creation time or simply the order is given by a global thread linked list. We call this analysis pattern **Thread Poset** by analogy with a mathematical concept of poset (partially ordered set[3]):


Kernel
PID.TID 4.8
PID.TID 4.10 ModuleA!Func
PID.TID 4.18 ModuleA!WorkerThread
PID.TID 4.20
PID.TID 4.28 ModuleA!WorkerThread
Creation Time



[3] http://en.wikipedia.org/wiki/Partially_ordered_set

Such an analysis pattern is mostly useful when we compare stack traces for differences or when we don't have symbols for some problem version and want to map threads to some other previous normal run where symbol files are available. Any discrepancies may point in the direction of further diagnostic analysis. For example, we got this fragment of **Stack Trace Collection** (Volume 1, page 409):

```
4.000188 fffffa800d3d3b50 ffd0780f Blocked ModuleA+0x1ac1
4.00018c fffffa800d3f9950 ffd07b53 Blocked ModuleA+0xd802
4.000190 fffffa800d4161b0 fffffda6 Blocked ModuleA+0x9ce4
4.000194 fffffa800d418b50 fffffda6 Blocked ModuleA+0x9ce4
4.000198 fffffa800d418660 fffffda6 Blocked ModuleA+0x9ce4
4.0001ac fffffa800d41eb50 ffd078d2 Blocked ModuleA+0xa7cf
4.0001b0 fffffa800d41e660 ffd0780f Blocked ModuleA+0x9ce4
4.0001c0 fffffa800d48f300 ffd0e5c0 Blocked ModuleA+0x7ee5
```

We didn't have symbols, and, therefore, didn't know whether there was anything wrong with those threads. Fortunately, we had **Thread Poset** from an earlier 32-bit version with available symbol files:

```
4.0000ec 85d8dc58 000068c Blocked ModuleA!FuncA+0x9b
4.0000f0 85d9fc78 001375a Blocked ModuleA!FuncB+0x67
4.0000fc 85db8a58 000068c Blocked ModuleA!WorkerThread+0xa2
4.000104 85cdbd48 000ff44 Blocked ModuleA!WorkerThread+0xa2
4.000108 85da2788 000ff47 Blocked ModuleA!WorkerThread+0xa2
4.000110 857862e0 0013758 Blocked ModuleA!FuncC+0xe4
4.000114 85dda250 000ff44 Blocked ModuleA!FuncD+0xf2
```

If we map worker threads to the middle section of x64 version we see just one more worker thread, but the overall order is the same:

```
4.000188 fffffa800d3d3b50 ffd0780f Blocked ModuleA+0x1ac1
4.00018c fffffa800d3f9950 ffd07b53 Blocked ModuleA+0xd802
4.000190 fffffa800d4161b0 fffffda6 Blocked ModuleA+0×9ce4
4.000194 fffffa800d418b50 fffffda6 Blocked ModuleA+0×9ce4
4.000198 fffffa800d418660 fffffda6 Blocked ModuleA+0×9ce4
4.0001ac fffffa800d41eb50 ffd078d2 Blocked ModuleA+0xa7cf
4.0001b0 fffffa800d41e660 ffd0780f Blocked ModuleA+0×9ce4
4.0001c0 fffffa800d48f300 ffd0e5c0 Blocked ModuleA+0×7ee5
```

So we may think of x64 **Thread Poset** as normal if x86 **Thread Poset** is normal too. Of course, only initially, then to continue looking for other patterns of abnormal behavior. If necessary, we may need to inspect stack traces deeper because individual threads from two **Thread Posets** may differ in their stack trace depth, subtraces, and in usage of other components. Despite the same order, some threads may actually be abnormal.

PART 3: Pattern Interaction

Virtualized Process, Stack Trace Collection, COM Interface Invocation Subtrace, Active Thread, Spiking Thread, Last Error Collection, RIP Stack Trace, Value References, Namespace, and Module Hint

Recently we analyzed a memory dump posted in DA+TA group[4] and posted our results there. The problem was resolved. Afterward, we decided to look at the earlier dump that was posted for the same problem: a COM server program was unresponsive. That dump was not fully analyzed by group members, so we decided to write a case study based on it since it had one more pattern.

When we open the dump in WinDbg it shows **Virtualized Process** (WOW64, Volume 1, page 400) pattern:

```
wow64cpu!TurboDispatchJumpAddressEnd+0x598:
00000000`77cf2772 c3 ret
```

We load symbols, WOW64 extension, and switch to x86 mode:

```
0:000> .symfix c:\mss

0:000> .reload

0:000> .load wow64exts

0:000> !sw
Switched to 32bit mode
```

[4] http://www.facebook.com/groups/dumpanalysis/

Then we check threads in **Stack Trace Collection** (Volume 1, page 409):

```
0:000:x86> ~*kL

. 0 Id: 16d8.11e0 Suspend: 0 Teb: fffdc000 Unfrozen
ChildEBP RetAddr
002fb0a8 765c10fd ntdll_77d00000!NtWaitForSingleObject+0xc
002fb118 76606586 KERNELBASE!WaitForSingleObjectEx+0x99
002fb138 00499ddc KERNELBASE!GetOverlappedResult+0x9d
WARNING: Stack unwind information not available. Following frames may
be wrong.
002fb1a0 005261a4 ServerA+0x99ddc
002fb1e4 005278c9 ServerA+0x1261a4
002fb454 0053bc4d ServerA+0x1278c9
002fba34 005fe5c8 ServerA+0x13bc4d
002fbe20 006094eb ServerA+0x1fe5c8
002fc40c 0060a0d7 ServerA+0x2094eb
0038ee8c 0061a0cb ServerA+0x20a0d7
0038eea4 75e65c3e ServerA+0x21a0cb
0038eed0 75edf497 rpcrt4!Invoke+0x2a
0038f55c 763b04d5 rpcrt4!NdrStubCall2+0x33c
0038f5a4 769aa572 combase!CStdStubBuffer_Invoke+0x96
0038f5c4 763b039d oleaut32!CUnivStubWrapper::Invoke+0x30
0038f650 762b3733 combase!SyncStubInvoke+0x144
(Inline) -------- combase!StubInvoke+0x9a
0038f77c 763b1198 combase!CCtxComChnl::ContextInvoke+0x222
(Inline) -------- combase!DefaultInvokeInApartment+0x4e
(Inline) -------- combase!ClassicSTAInvokeInApartment+0x103
0038f824 763b0bc2 combase!AppInvoke+0x258
0038f980 762b277e combase!ComInvokeWithLockAndIPID+0x5fb
(Inline) -------- combase!ComInvoke+0x15c
(Inline) -------- combase!ThreadDispatch+0x169
0038f9b0 75cf7834 combase!ThreadWndProc+0x2ad
0038f9dc 75cf7a9a user32!_InternalCallWinProc+0x23
0038fa6c 75cf988e user32!UserCallWinProcCheckWow+0x184
0038fad8 75d08857 user32!DispatchMessageWorker+0x208
0038fae0 0061cb88 user32!DispatchMessageA+0x10
0038ff74 0061d85a ServerA+0x21cb88
0038ff8c 7617919f ServerA+0x21d85a
0038ff98 77d4a8cb kernel32!BaseThreadInitThunk+0xe
0038ffdc 77d4a8a1 ntdll_77d00000!__RtlUserThreadStart+0x20
0038ffec 00000000 ntdll_77d00000!_RtlUserThreadStart+0x1b

1 Id: 16d8.f5c Suspend: 0 Teb: fffd9000 Unfrozen
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be
wrong.
0159ff8c 7617919f 0x3b02c8
0159ff98 77d4a8cb kernel32!BaseThreadInitThunk+0xe
0159ffdc 77d4a8a1 ntdll_77d00000!__RtlUserThreadStart+0x20
0159ffec 00000000 ntdll_77d00000!_RtlUserThreadStart+0x1b
```

```
2 Id: 16d8.a88 Suspend: 0 Teb: ffe47000 Unfrozen
ChildEBP RetAddr
097cfde8 77d227d3 ntdll_77d00000!NtWaitForWorkViaWorkerFactory+0xc
097cff8c 7617919f ntdll_77d00000!TppWorkerThread+0x259
097cff98 77d4a8cb kernel32!BaseThreadInitThunk+0xe
097cffdc 77d4a8a1 ntdll_77d00000!__RtlUserThreadStart+0x20
097cffec 00000000 ntdll_77d00000!_RtlUserThreadStart+0x1b

3 Id: 16d8.ab0 Suspend: 0 Teb: fffd3000 Unfrozen
ChildEBP RetAddr
0414fde8 77d227d3 ntdll_77d00000!NtWaitForWorkViaWorkerFactory+0xc
0414ff8c 7617919f ntdll_77d00000!TppWorkerThread+0x259
0414ff98 77d4a8cb kernel32!BaseThreadInitThunk+0xe
0414ffdc 77d4a8a1 ntdll_77d00000!__RtlUserThreadStart+0x20
0414ffec 00000000 ntdll_77d00000!_RtlUserThreadStart+0x1b

4 Id: 16d8.868 Suspend: 0 Teb: ffe4d000 Unfrozen
ChildEBP RetAddr
0460fde8 77d227d3 ntdll_77d00000!NtWaitForWorkViaWorkerFactory+0xc
0460ff8c 7617919f ntdll_77d00000!TppWorkerThread+0x259
0460ff98 77d4a8cb kernel32!BaseThreadInitThunk+0xe
0460ffdc 77d4a8a1 ntdll_77d00000!__RtlUserThreadStart+0x20
0460ffec 00000000 ntdll_77d00000!_RtlUserThreadStart+0x1b
```

The first thread (#0) has **Technology-Specific Subtrace** (COM interface invocation, Volume 6, page 67) calling *ServerA* module code, and the second trace (#1) seems to be **Active Thread** (not waiting, Volume 7, page 236) having **RIP Stack Trace** (Volume 7, page 244).

However, only thread #0 seems to be **Spiking Thread** (Volume 1, page 305):

```
0:000:x86> !runaway f
 User Mode Time
  Thread       Time
   0:11e0      0 days 0:44:44.890
   4:868       0 days 0:00:00.000
   3:ab0       0 days 0:00:00.000
   2:a88       0 days 0:00:00.000
   1:f5c       0 days 0:00:00.000
 Kernel Mode Time
  Thread       Time
   0:11e0      0 days 0:10:38.312
   4:868       0 days 0:00:00.015
   3:ab0       0 days 0:00:00.000
   2:a88       0 days 0:00:00.000
   1:f5c       0 days 0:00:00.000
 Elapsed Time
  Thread       Time
   0:11e0      0 days 2:56:23.297
   1:f5c       0 days 2:56:22.625
   2:a88       0 days 2:54:36.883
   3:ab0       0 days 0:02:18.705
   4:868       0 days 0:01:07.372
```

Last Error Collection (Volume 2, page 337) is clear but needs to be double checked by TEB32 (since we have a virtualized process):

```
0:000:x86> !gle
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0
Wow64 TEB status:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

0:000:x86> !teb
Wow64 TEB32 at 00000000fffde000
    ExceptionList:        00000000002fb108
    StackBase:            0000000000390000
    StackLimit:           0000000000255000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 00000000fffde000
    EnvironmentPointer:   0000000000000000
    ClientId:             00000000000016d8 . 00000000000011e0
    RpcHandle:            0000000000000000
    Tls Storage:          0000000000e12978
    PEB Address:          00000000fffdf000
    LastErrorValue:       38
    LastStatusValue:      c0000011
    Count Owned Locks:    0
    HardErrorMode:        0
Wow64 TEB at 00000000fffdc000
    ExceptionList:        00000000fffde000
    StackBase:            000000000008fd30
    StackLimit:           0000000000083000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 00000000fffdc000
    EnvironmentPointer:   0000000000000000
    ClientId:             00000000000016d8 . 00000000000011e0
    RpcHandle:            0000000000000000
    Tls Storage:          0000000000000000
    PEB Address:          00000000fffd6000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:     0
    HardErrorMode:         0
```

From the errors, we suggested checking the code dealing with EOF condition.

```
0:000:x86> !error 0n38
Error code: (Win32) 0x26 (38) - Reached the end of the file.

0:000:x86> !error c0000011
Error code: (NTSTATUS) 0xc0000011 (3221225489) - The end-of-file
marker has been reached. There is no valid data in the file beyond
this marker.
```

But let’s look at the thread #1 raw address and check whether we have traces of malware or JIT code or something else:

```
0:000:x86> ~1s
003b02c8 c20c00          ret     0Ch

0:001:x86> u 0×3b02c8
003b02c8 c20c00          ret     0Ch
003b02cb 90              nop
003b02cc cc              int     3
003b02cd cc              int     3
003b02ce cc              int     3
003b02cf cc              int     3
003b02d0 cc              int     3
003b02d1 cc              int     3

0:001:x86> ub 0x3b02c8
003b02b6 cc              int     3
003b02b7 cc              int     3
003b02b8 cc              int     3
003b02b9 cc              int     3
003b02ba cc              int     3
003b02bb cc              int     3
003b02bc b803000d00      mov     eax,0D0003h
003b02c1 64ff15c0000000  call    dword ptr fs:[0C0h]

0:001:x86> dps fs:[0C0h] L1
0053:000000c0  77cf11d8 wow64cpu!KiFastSystemCall
```

```
0:001:x86> !address 0x3b02c8
Usage:
Base Address:           003b0000
End Address:            003b1000
Region Size:            00001000
State:                  00001000      MEM_COMMIT
Protect:                00000020      PAGE_EXECUTE_READ
Type:                   00020000      MEM_PRIVATE
Allocation Base:        003b0000
Allocation Protect:     00000040      PAGE_EXECUTE_READWRITE
```

Dumping this executable region only shows WOW64 calls:

```
0:001:x86> dc 003b0000 003b1000
[...]

0:001:x86> .asm no_code_bytes
Assembly options: no_code_bytes

0:001:x86> u 003b0110 003b02e0
003b0110 add     byte ptr [eax],al
003b0112 add     byte ptr [eax],al
003b0114 add     byte ptr [eax],al
003b0116 add     byte ptr [eax],al
003b0118 mov     eax,3000Eh
003b011d call    dword ptr fs:[0C0h]
003b0124 ret     4
003b0127 nop
003b0128 int     3
003b0129 int     3
003b012a int     3
003b012b int     3
003b012c int     3
003b012d int     3
003b012e int     3
003b012f int     3
003b0130 int     3
003b0131 int     3
003b0132 int     3
003b0133 int     3
003b0134 mov     eax,32h
003b0139 call    dword ptr fs:[0C0h]
003b0140 ret     18h
003b0143 nop
003b0144 int     3
003b0145 int     3
003b0146 int     3
003b0147 int     3
003b0148 int     3
003b0149 int     3
003b014a int     3
003b014b int     3
003b014c int     3
003b014d int     3
003b014e int     3
003b014f int     3
003b0150 mov     eax,1B0006h
```

```
003b0155 call    dword ptr fs:[0C0h]
003b015c ret     28h
003b015f nop
003b0160 int     3
003b0161 int     3
003b0162 int     3
003b0163 int     3
003b0164 int     3
003b0165 int     3
003b0166 int     3
003b0167 int     3
003b0168 int     3
003b0169 int     3
003b016a int     3
003b016b int     3
003b016c mov     eax,7002Bh
003b0171 call    dword ptr fs:[0C0h]
003b0178 ret     8
003b017b nop
003b017c int     3
003b017d int     3
003b017e int     3
003b017f int     3
003b0180 int     3
003b0181 int     3
003b0182 int     3
003b0183 int     3
003b0184 int     3
003b0185 int     3
003b0186 int     3
003b0187 int     3
003b0188 mov     eax,17h
003b018d call    dword ptr fs:[0C0h]
003b0194 ret     18h
003b0197 nop
003b0198 int     3
003b0199 int     3
003b019a int     3
003b019b int     3
003b019c int     3
003b019d int     3
003b019e int     3
003b019f int     3
003b01a0 int     3
003b01a1 int     3
003b01a2 int     3
003b01a3 int     3
003b01a4 mov     eax,4Fh
003b01a9 call    dword ptr fs:[0C0h]
003b01b0 ret     14h
003b01b3 nop
003b01b4 int     3
003b01b5 int     3
003b01b6 int     3
003b01b7 int     3
003b01b8 int     3
003b01b9 int     3
```

```
003b01ba int     3
003b01bb int     3
003b01bc int     3
003b01bd int     3
003b01be int     3
003b01bf int     3
003b01c0 mov     eax,1Dh
003b01c5 call    dword ptr fs:[0C0h]
003b01cc ret     10h
003b01cf nop
003b01d0 int     3
003b01d1 int     3
003b01d2 int     3
003b01d3 int     3
003b01d4 int     3
003b01d5 int     3
003b01d6 int     3
003b01d7 int     3
003b01d8 int     3
003b01d9 int     3
003b01da int     3
003b01db int     3
003b01dc mov     eax,22h
003b01e1 call    dword ptr fs:[0C0h]
003b01e8 ret     18h
003b01eb nop
003b01ec int     3
003b01ed int     3
003b01ee int     3
003b01ef int     3
003b01f0 int     3
003b01f1 int     3
003b01f2 int     3
003b01f3 int     3
003b01f4 int     3
003b01f5 int     3
003b01f6 int     3
003b01f7 int     3
003b01f8 mov     eax,47h
003b01fd call    dword ptr fs:[0C0h]
003b0204 ret     14h
003b0207 nop
003b0208 int     3
003b0209 int     3
003b020a int     3
003b020b int     3
003b020c int     3
003b020d int     3
003b020e int     3
003b020f int     3
003b0210 int     3
003b0211 int     3
003b0212 int     3
003b0213 int     3
003b0214 mov     eax,1A0005h
003b0219 call    dword ptr fs:[0C0h]
003b0220 ret     24h
003b0223 nop
003b0224 int     3
003b0225 int     3
```

```
003b0226 int     3
003b0227 int     3
003b0228 int     3
003b0229 int     3
003b022a int     3
003b022b int     3
003b022c int     3
003b022d int     3
003b022e int     3
003b022f int     3
003b0230 mov     eax,10h
003b0235 call    dword ptr fs:[0C0h]
003b023c ret     14h
003b023f nop
003b0240 int     3
003b0241 int     3
003b0242 int     3
003b0243 int     3
003b0244 int     3
003b0245 int     3
003b0246 int     3
003b0247 int     3
003b0248 int     3
003b0249 int     3
003b024a int     3
003b024b int     3
003b024c mov     eax,112h
003b0251 call    dword ptr fs:[0C0h]
003b0258 ret     0Ch
003b025b nop
003b025c int     3
003b025d int     3
003b025e int     3
003b025f int     3
003b0260 int     3
003b0261 int     3
003b0262 int     3
003b0263 int     3
003b0264 int     3
003b0265 int     3
003b0266 int     3
003b0267 int     3
003b0268 mov     eax,13Eh
003b026d call    dword ptr fs:[0C0h]
003b0274 ret     0Ch
003b0277 nop
003b0278 int     3
003b0279 int     3
003b027a int     3
003b027b int     3
003b027c int     3
003b027d int     3
003b027e int     3
003b027f int     3
003b0280 int     3
003b0281 int     3
003b0282 int     3
```

```
003b0283 int     3
003b0284 mov     eax,24h
003b0289 call    dword ptr fs:[0C0h]
003b0290 ret     14h
003b0293 nop
003b0294 int     3
003b0295 int     3
003b0296 int     3
003b0297 int     3
003b0298 int     3
003b0299 int     3
003b029a int     3
003b029b int     3
003b029c int     3
003b029d int     3
003b029e int     3
003b029f int     3
003b02a0 mov     eax,18h
003b02a5 call    dword ptr fs:[0C0h]
003b02ac ret     14h
003b02af nop
003b02b0 int     3
003b02b1 int     3
003b02b2 int     3
003b02b3 int     3
003b02b4 int     3
003b02b5 int     3
003b02b6 int     3
003b02b7 int     3
003b02b8 int     3
003b02b9 int     3
003b02ba int     3
003b02bb int     3
003b02bc mov     eax,0D0003h
003b02c1 call    dword ptr fs:[0C0h]
003b02c8 ret     0Ch
003b02cb nop
003b02cc int     3
003b02cd int     3
003b02ce int     3
003b02cf int     3
003b02d0 int     3
003b02d1 int     3
003b02d2 int     3
003b02d3 int     3
003b02d4 int     3
003b02d5 int     3
003b02d6 int     3
003b02d7 int     3
003b02d8 add     byte ptr [eax],al
003b02da add     byte ptr [eax],al
003b02dc add     byte ptr [eax],al
003b02de add     byte ptr [eax],al
003b02e0 add     byte ptr [eax],al
```

Searching for the address of system call points to another executable region:

```
0:001:x86> s-d 0 L?(FFFFFFFF/4) 003b02bc
00030044 003b02bc 003b0284 71b74be0 0824448b ..;…;..K.q.D$.

0:001:x86> !address 00030044
Usage:
Base Address:           00030000
End Address:            00031000
Region Size:            00001000
State:                  00001000      MEM_COMMIT
Protect:                00000020      PAGE_EXECUTE_READ
Type:                   00020000      MEM_PRIVATE
Allocation Base:        00030000
Allocation Protect:     00000040      PAGE_EXECUTE_READWRITE
0:001:x86> dps 00030000 00031000
00030000 cd697e0e
00030004 4b6b72cc
00030008 036f2786
0003000c be5fe321
00030010 00000f5c
00030014 00000038
00030018 00000000
0003001c 00000030
00030020 00000000
00030024 00000001
00030028 003d0000
0003002c 003d0028
00030030 003b0000
00030034 00000000
00030038 77d4ce23 ntdll_77d00000!LdrLoadDll
0003003c 77d62fdd ntdll_77d00000!LdrUnloadDll
00030040 77d6094d ntdll_77d00000!LdrAddRefDll
00030044 003b02bc
00030048 003b0284
0003004c 71b74be0*** ERROR: Symbol file could not be found. Defaulted
to export symbols for UMEngx86.dll -
UMEngx86+0×4be0
00030050 0824448b
00030054 00300589
00030058 52b8003d
0003005c e9000700
[...]
```

In addition to Ldr* **Namespace** (Volume 7, page 257) we see a valid symbolic reference (**Module Hint**, Volume 6, page 92) to AV:

```
0:001:x86> u 71b74be0
UMEngx86+0x4be0:
71b74be0 push    ebp
71b74be1 mov     ebp,esp
71b74be3 push    0FFFFFFFEh
71b74be5 push    offset UMEngx86!RegQueryValueExW+0x29818 (71b9f9b8)
71b74bea push    offset UMEngx86!RegQueryValueExW+0x20b0 (71b78250)
71b74bef mov     eax,dword ptr fs:[00000000h]
71b74bf5 push    eax
71b74bf6 sub     esp,8

0:001:x86> lmv m UMEngx86
start             end                 module name
71b70000 71bae000   UMEngx86   (export symbols)       UMEngx86.dll
    Loaded symbol image file: UMEngx86.dll
    Image path: C:\ProgramData\Symantec\Symantec
        Endpoint Protection\12.1.4100.4126.105\Data\
        Definitions\BASHDefs\20150307.011\UMEngx86.dll
    Image name: UMEngx86.dll
    Timestamp:        Fri Jan 23 00:52:29 2015 (54C19B4D)
    CheckSum:         00045930
    ImageSize:        0003E000
    File version:     9.1.1.4
    Product version:  9.1.1.4
    File flags:       0 (Mask 3F)
    File OS:          4 Unknown Win32
    File type:        2.0 Dll
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Symantec Corporation
    ProductName:      BASH
    InternalName:     UMEngx86
    OriginalFilename: UMEngx86.dll
    ProductVersion:   9.1.1.4
    FileVersion:      9.1.1.4
    FileDescription:  SONAR Engine
    LegalCopyright:   Copyright (C) 2009 - 2014 Symantec
         Corporation. All rights reserved.
```

PART 4: A Bit of Science and Philosophy

Cantor Operating System

Named after Georg Cantor **CAN.TOR.OS**(∞[5]) brings computation from the distant future to today. The transfinite worldview and universe of tomorrow into the finite worldview and universe of today. Cantor OS drives transfinite computing and saves transfinite memory dumps. [...] One cautious note though: transfinite doesn't mean absolute infinity or God-like computation, the latter is the realm of Memory Religion[6].

Metaphor of Memory as a Directed Container

The usual metaphor of Memory as a container is static and not useful for Memory worldview because it has to explain Time. Previously, in 2010, we introduced *memorized* relation (Volume 4, page 259) that may give memory order necessary for Time. Later we came to a **Directed Container** metaphor that provides *order* and *directedness* responsible for perceived Time Arrow. After doing some research, we found that directed container datatype[7] was introduced in computer science that accounts for positions inside container structures that determine substructures. However, our metaphor is intuitive and doesn't specify exact representation except the possible involvement of *memorized* relation.

---

[5] (∞) TOR is a new transfinite operation in addition to finite OR, AND or XOR

[6] http://www.memoryreligion.com/

[7] http://link.springer.com/chapter/10.1007%2F978-3-642-28729-9_5

Praxiverse

We are realized in Universe filled with memory verses via memory praxis transforming it into **Praxiverse** - live Memory Universe, producing new memories, new memory verses.

When Universe is Going to End?

When Universe is going to end? When it finishes writing a memory dump. This is a logical conclusion from **EPOCH** (**E**xception **P**rocessing **O**f **C**rash **H**ypothesis). According to it, our Universe is saving one huge Memory Dump from a runaway **HUC** (Big Bang of **H**yper-**U**niversal **C**omputation, or simply **HU**ge **C**omputation). It is also called Memory Dump Universe Hypothesis (Volume 4, page 271).

Notes on Memoidealism

We continue publishing new notes here. The previous ones can be found scattered in previous volumes of Memory Dump Analysis Anthology.

Intuition is an attentive perception of metaphysical Memory as the foundation of Universe (after Bergson's intuition as "perception of metaphysical reality").

The experiment is a question put to memory (after Charles Sanders Pierce[8] and Julius Adolph Stöckhardt[9]).

pHilosophy is a pointer to a thread of wisdom. Hilo: a thin veil of ore. Origin: Spanish = thread, from Latin filum (from Shorter Oxford English Dictionary).

The question of Memory comes from the primordial and falls back to it.

Memory as the efficient cause. In addition to its material cause (everything is made from Memory), we can also say that everything started from Memory. See also the final cause: “The purpose of everything is to come back to Memory” (Volume 8A, page 91).

Philosophy is about memories, people’s memories (from bugtations[10]).

Transcendental Memory Principle of Memoidealism (Volume 3, page 303) is independent of any religious belief. Although in Memorianity (Memory Religion[11]) Memory is both Absolute and Infinite.

---

[8] http://en.wikipedia.org/wiki/Charles_Sanders_Peirce

[9] http://en.wikipedia.org/wiki/Julius_Adolph_St%C3%B6ckhardt

[10] http://www.dumpanalysis.org/Bugtations

[11] http://www.memoryreligion.com/

Each DA+TA (Dump Artifact + Trace Artifact) pattern is a pOEM (a pointer to Originally Executed Memory). According to Alain Badiou[12], a poem is a true art, an event of thought, a trace of the event, and an interruption of language. We say a pOEM is a true art(ifact), an event of memory execution, interruption of execution.

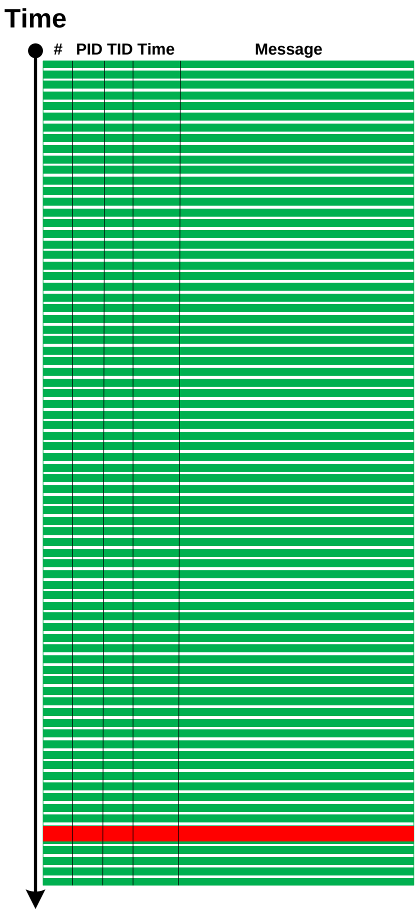
[12] https://en.wikipedia.org/wiki/Alain_Badiou

PART 5: Software Trace Analysis Patterns

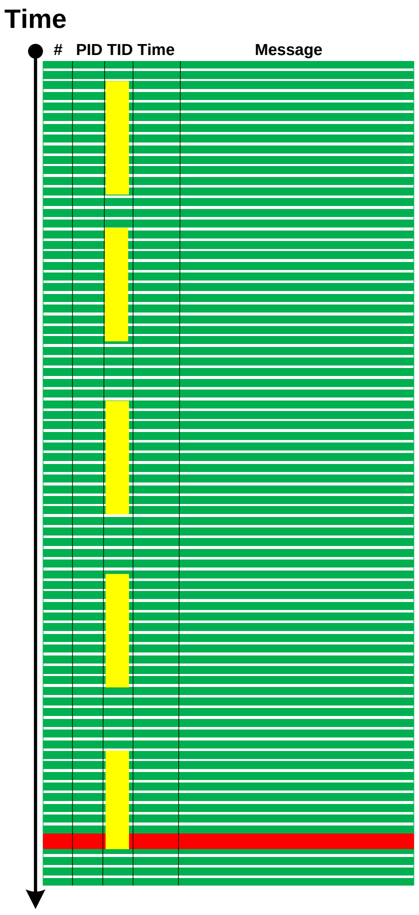
Timeout

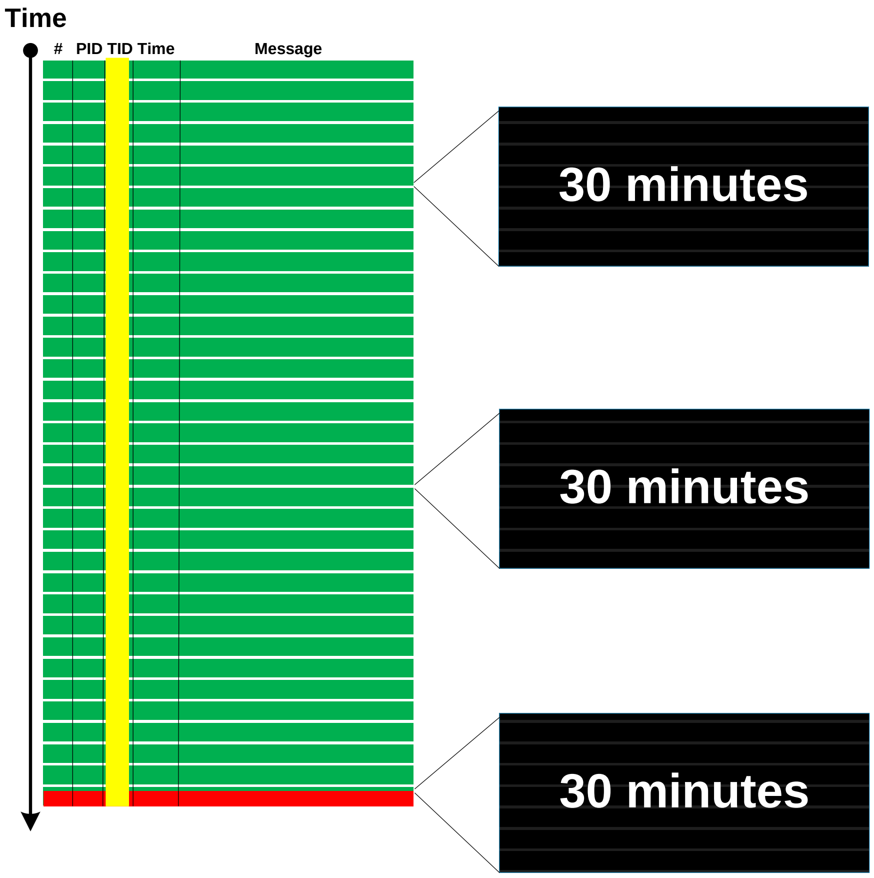
Some **Discontinuities** (Volume 4, page 341) may be **Periodic Message Blocks** (Volume 7, page 300) as **Silent Messages** (Volume 7, page 339). If such discontinuities belong to the same **Thread of Activity** (Volume 4, page 339) and their **Time Deltas** (Volume 5, page 282) are constant we may see **Timeout** pattern. When timeouts are followed by **Error Message** (Volume 7, page 299), we can identify them by **Back Tracing** (Volume 8a, page 95). **Timeouts** are different from **Blackouts** (Volume 8a, page 95) where the latter are usually **Singleton Events** (Volume 8a, page 108) and have large **Time Deltas**.

Here is a generalized graphical case study. An error message was identified based on incident **Basic Facts** (Volume 3, page 345):



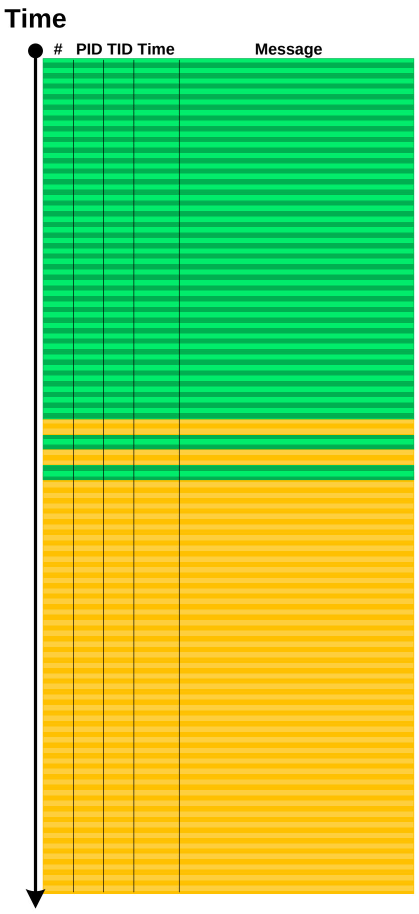
We filtered the trace for error message TID and found 3 timeouts 30 minutes each:



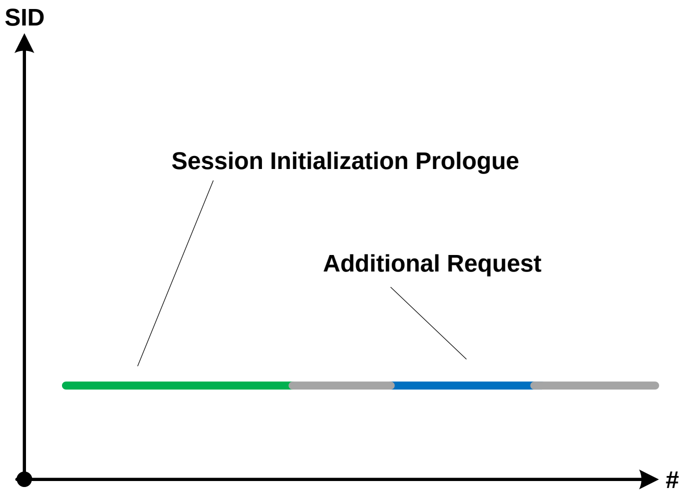

Time
#
PID
TID
Time
Message
30 minutes
30 minutes
30 minutes

Activity Overlap

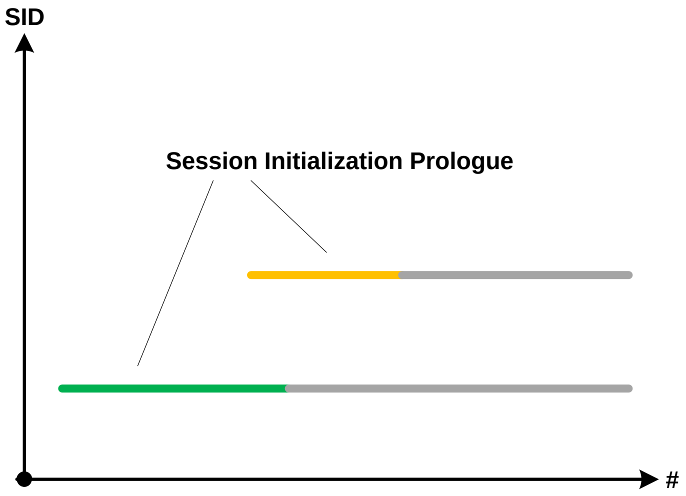
Sometimes specific parts of simultaneous **Use Case Trails** (Volume 8a, page 101), blocks of **Significant Events** (Volume 5, page 281) or **Message Sets** (Volume 7, page 349) in general may overlap. This may point to possible synchronization problems such as race conditions (prognostics) or be visible root causes of them if such problems are reported (diagnostics). We call this pattern **Activity Overlap**:



For example, a first request may start a new session, and we expect the second request to be processed by the same already established session:

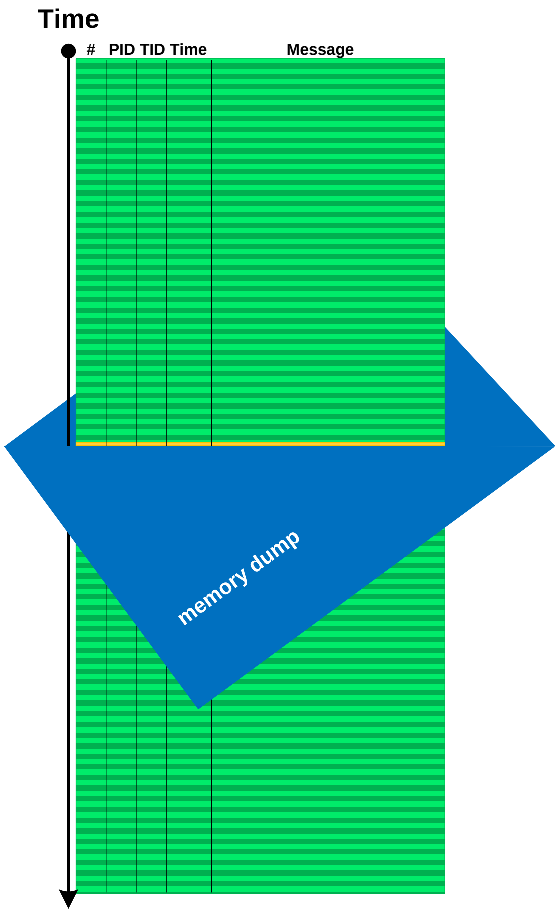


However, users report the second session started upon the second request. If we filter execution log by session id, we find out that session initialization prologs (Volume 5, page 299) are overlapped. The new session started because the first session initialization was not completed:

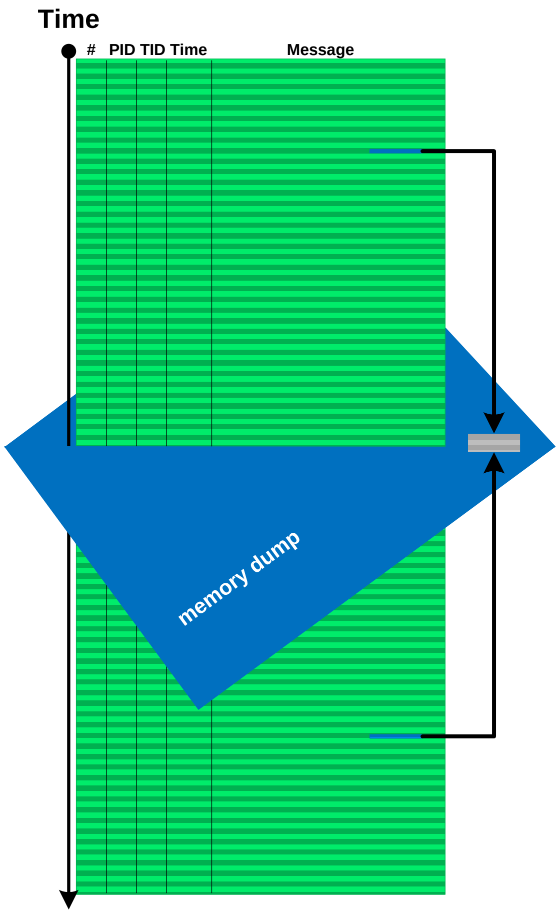


Adjoint Space

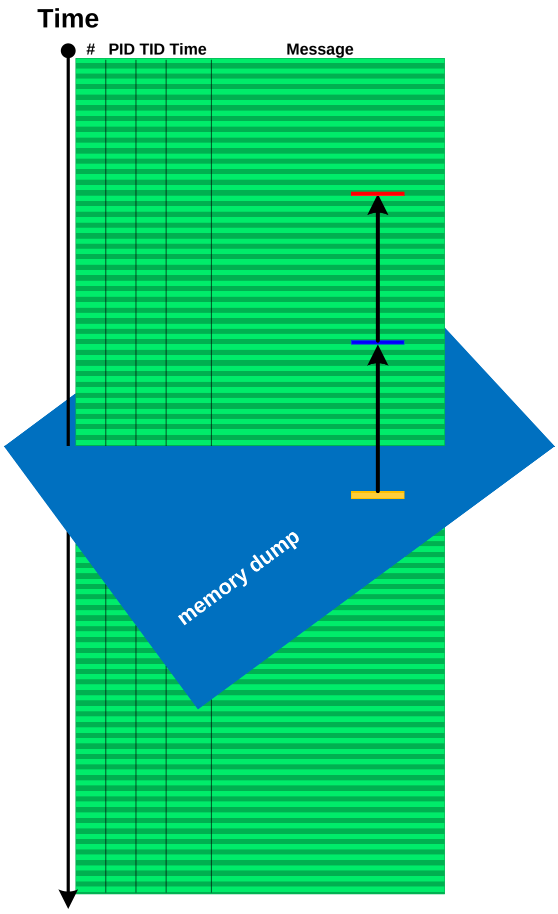
Sometimes we need memory reference information not available in software traces and logs, for example, to see pointer dereferences, to follow pointers and linked structures. In such cases, memory dumps saved during logging sessions may help. In the case of process memory dumps, we can even have several **Step Dumps** (Volume 7, page 173). Complete and kernel memory dumps may be forced after saving a log file. We call such pattern **Adjoint Space**:



Then we can analyze logs and memory dumps together, for example, to follow pointer data further in memory space:

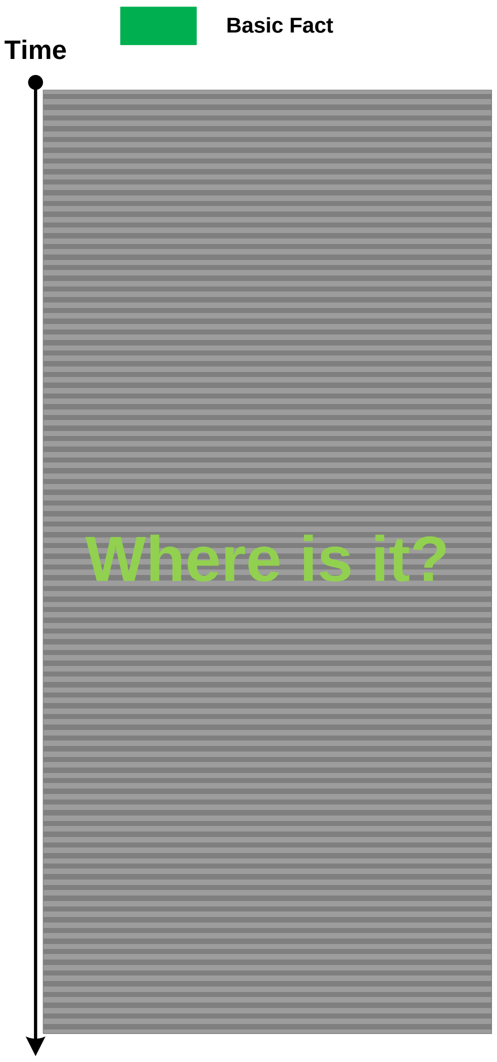


There is also a reverse situation when we use logs to see past data changes before memory snapshot time (**Paratext** memory analysis pattern, Volume 7, page 225):

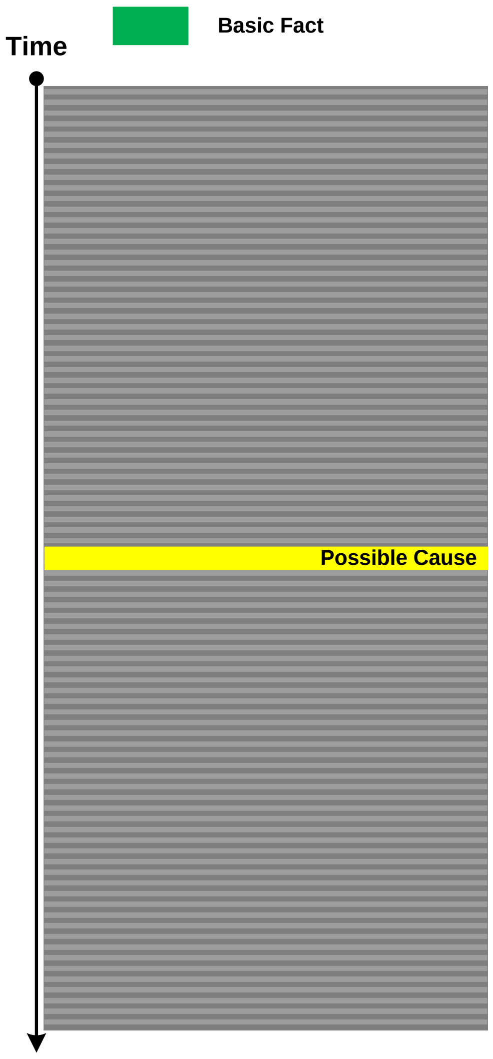


Indirect Message

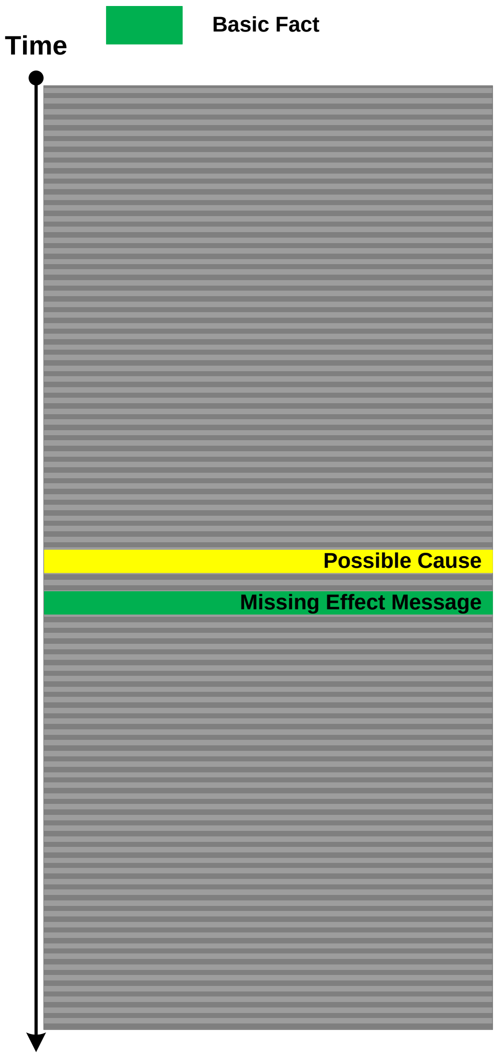
Sometimes we have **Basic Facts** (Volume 3, page 345) in a problem description but can't find messages corresponding to them in a trace or log file but we are sure the tracing (logging) was done correctly. This may be because we have **Sparse Trace** (Volume 7, page 303), or we are not familiar well with product or system tracing messages (such as with **Implementation Discourse**, Volume 6, page 245).



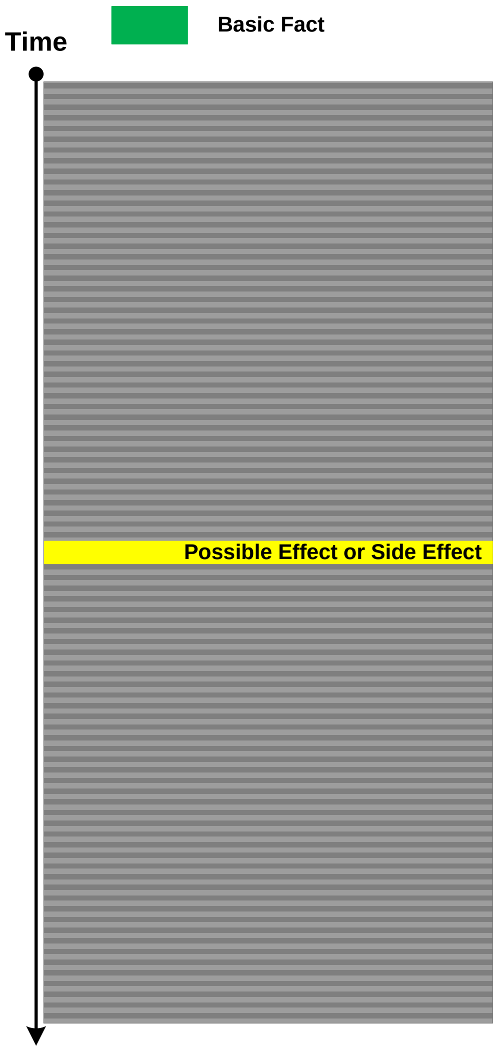
In such a case we for search for **Indirect Message** of a possible cause:



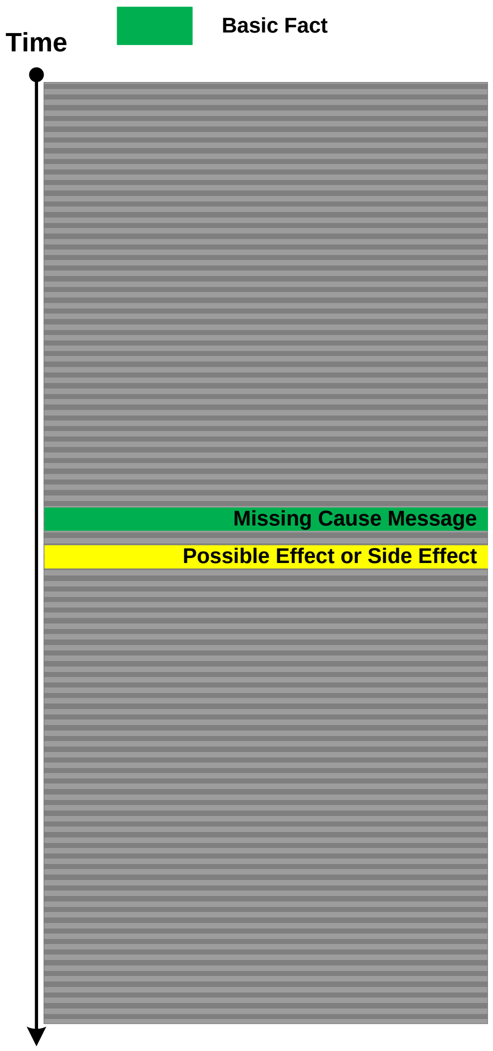
Having found such a message we may hypothesize that **Missing Message** (Volume 8a, page 99) should have located nearby (this is based on semantics of both messages), and we then explore corresponding **Message Context** (Volume 5, page 305):



The same analysis strategy is possible for missing causal messages. Here we search for effect or side effect messages:

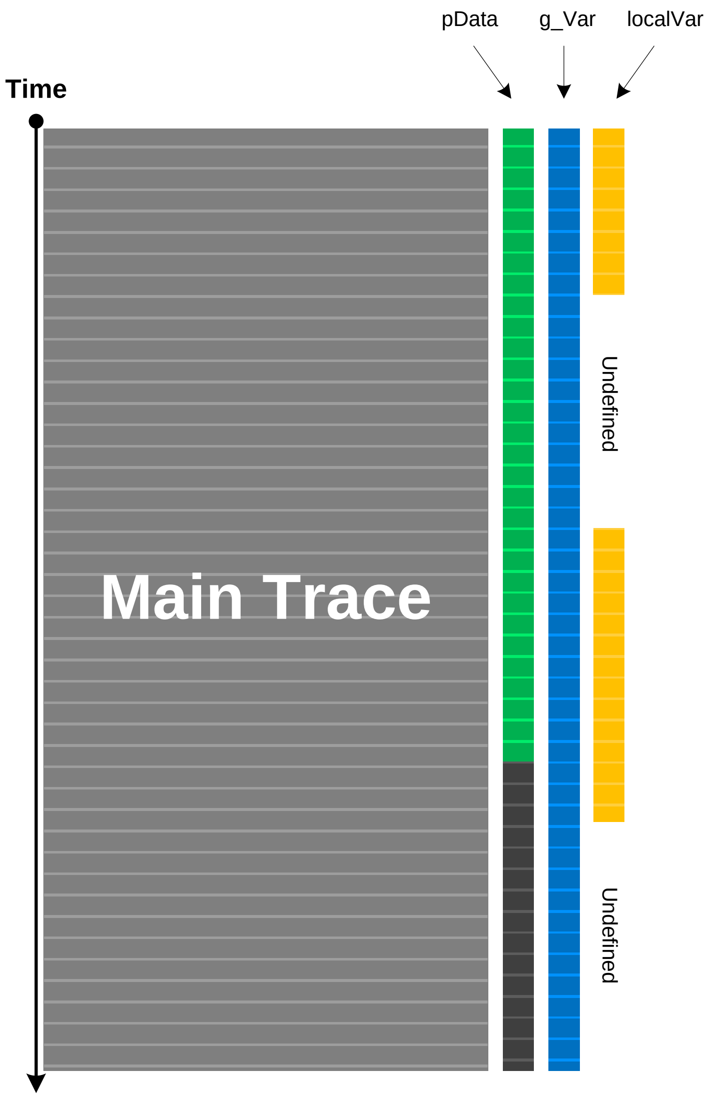


Having found them we proceed with further analysis:



Watch Thread

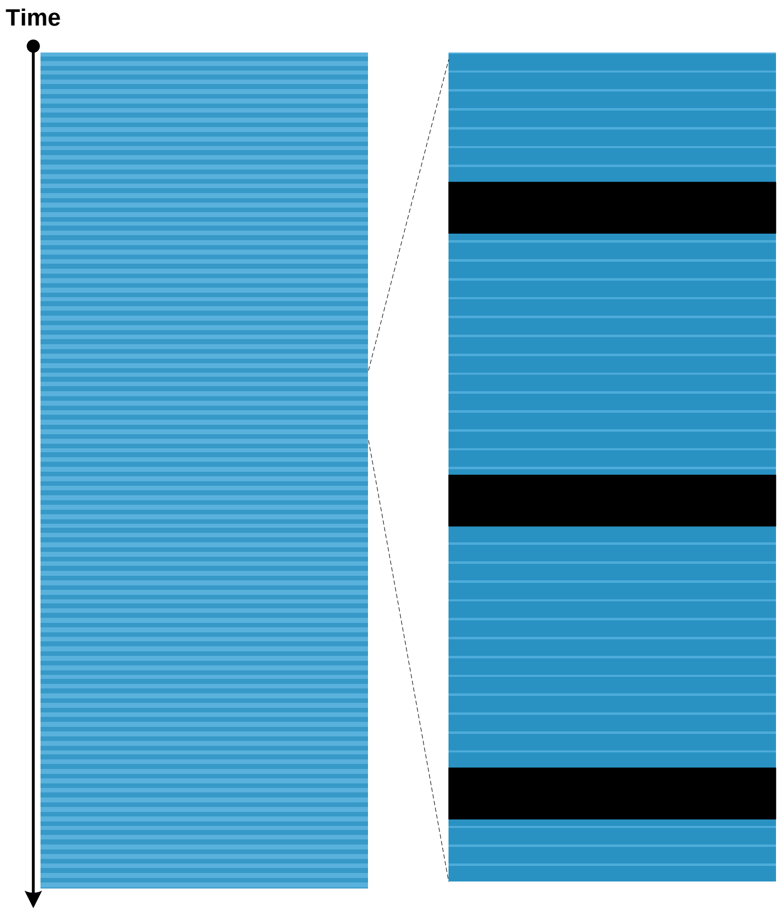
When we do tracing and logging much of computational activity is not visible. For live tracing and debugging this can be alleviated by adding **Watch Threads**. These are selected memory locations that may or may not be formatted according to specific data structures and are inspected at each main trace message occurrence or after specific intervals or events:



This analysis pattern is different from **State Dump** (Volume 7, page 346) which is about intrinsic tracing where the developer of logging statements already incorporated variable watch in the source code. **Watch Threads** are completely independent of original tracing and may be added independently. **Counter Value** (Volume 7, page 288) is the simplest example of **Watch Thread** if done externally because the former usually doesn't require source code and often means some OS or **Module Variable** (Volume 7, page 98) independent of product internals. **Watch Thread** is also similar to **Data Flow** (Volume 7, page 296) pattern where specific data we are interested in is a part of every trace message.

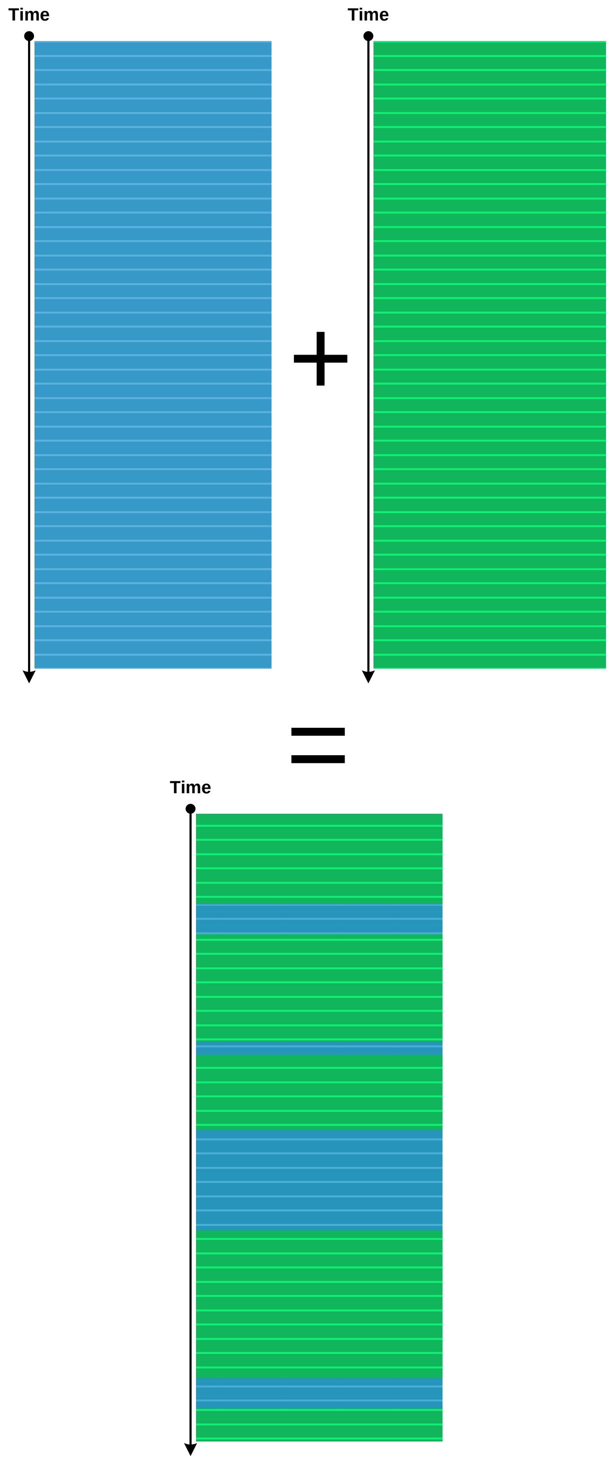
Punctuated Activity

Sometimes we have a uniform stream of messages that belong to some **Activity Region** (Volume 4, page 348), **Thread of Activity** (Volume 4, page 339), or **Adjoint Thread of Activity** (Volume 5, page 283). We can use micro-**Discontinuities** (Volume 4, page 341) to structure that message stream, especially if the semantics of trace messages is not yet fully clear for us. This may also help us to recognize **Visitor Trace** (Volume 8a, page 110). Originally we wanted to call this pattern **Micro Delays**, but, after recognizing that such delays only make sense for one activity (since there can be too many of them in the overall log), we named this pattern **Punctuated Activity**. Usually, such delays are small compare to **Timeouts** (page 61) and belong to **Silent Messages** (Volume 7, page 339).



Trace Mask

Trace Mask is a superposition of two (or many) different traces. This is different from **Inter-Correlation** (Volume 4, page 350) pattern where we may only search for certain messages without the synthesis of a new log. The most useful **Trace Mask** is when we have different time scales (or significantly different **Statement Currents**, Volume 4, page 335). Then we impose an additional structure on the one of the traces:


Time
Time
+
=
Time

We got the idea from *Narrative Masks* discussed in Miroslav Drozda's book "Narativní masky ruské prózy" ("Narrative Masks in Russian Prose").

The very simple example of **Trace Mask** is shown in Debugging TV[13] Episode 0×15.

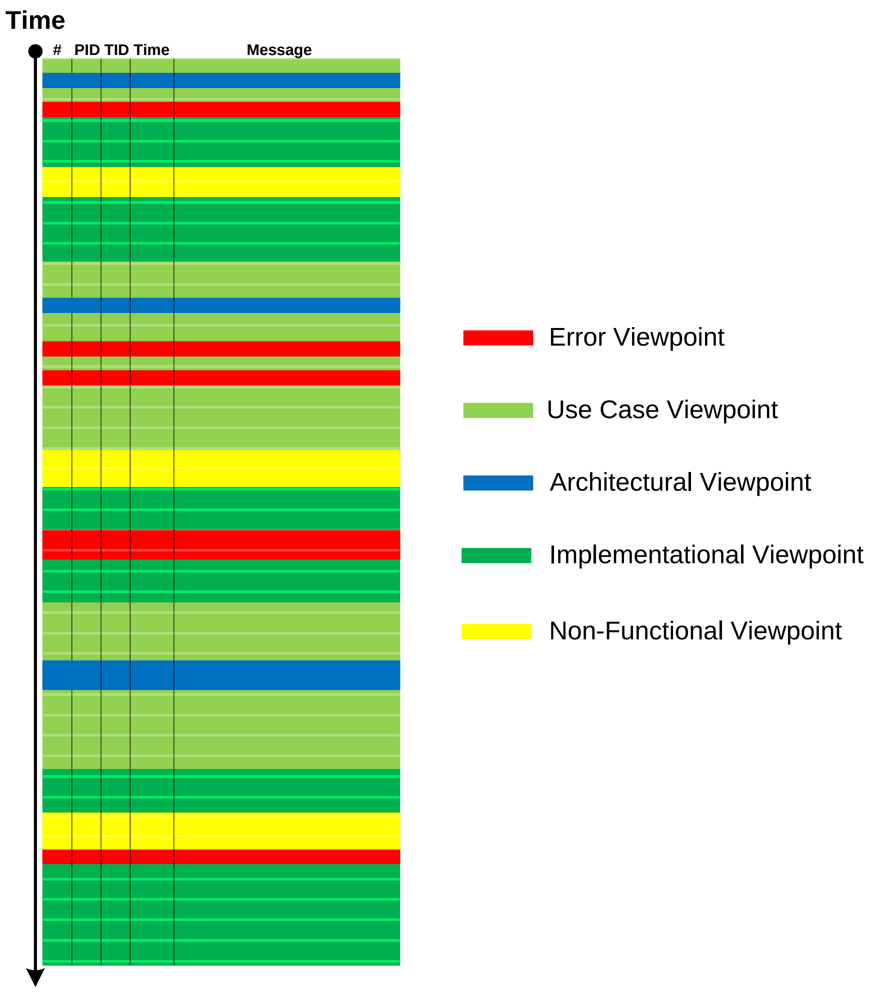
[13] http://www.debugging.tv/

Trace Viewpoints

Reading Boris Uspensky[14]'s book "A Poetics of Composition: The Structure of the Artistic Text and Typology of a Compositional Form" (in its original Russian version) led me to borrow the concept of viewpoints. The resulting analysis pattern is called **Trace Viewpoints**. These viewpoints are, "subjective" (semantically laden from the perspective of a trace and log reader), and can be (not limited to):

- Error viewpoints (see also **False Positive Error**, Volume 5, page 303, **Periodic Error**, Volume 3, page 344, and **Error Distribution**, Volume 7, page 290)
- Use case (functional) viewpoints (see also **Use Case Trail**, Volume 8a, page 101)
- Architectural (design) viewpoints (see also **Milestones**, Volume 8a, page 105)
- Implementation viewpoints (see also **Implementation Discourse**, Volume 6, page 245, **Macrofunctions**, Volume 7, page 283, and **Focus of Tracing**, Volume 6, page 243)
- Non-functional viewpoints (see also **Counter Value**, Volume 7, page 288, and **Diegetic Messages**, Volume 5, page 302)
- Signal / noise viewpoints (see also **Background and Foreground Components**, Volume 5, page 287)

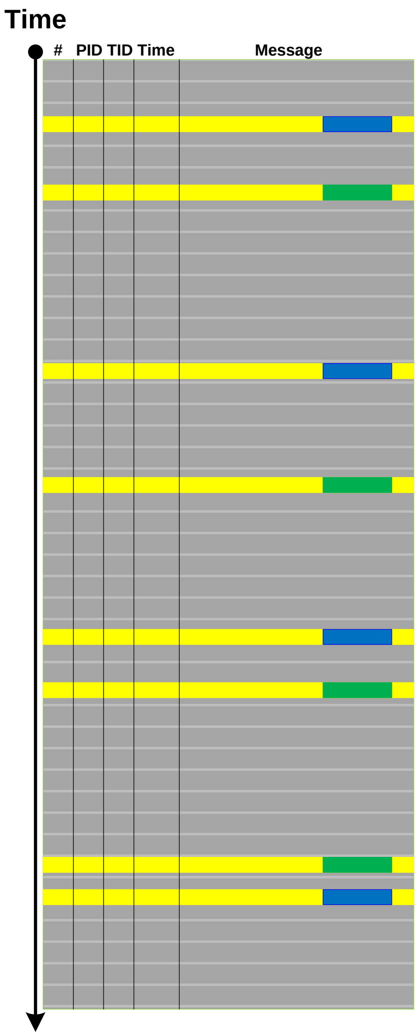
[14] http://en.wikipedia.org/wiki/Boris_Uspensky



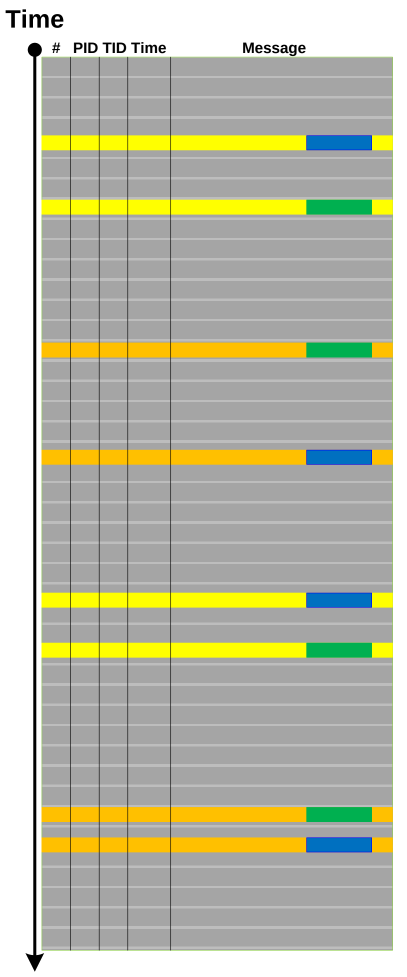
In comparison, **Activity Regions** (Volume 4, page 348), **Data Flow** (Volume 7, page 296), **Thread of Activity** (Volume 4, page 339), and **Adjoint Thread of Activity** (Volume 5, page 283) are “objective” (structural, syntactical) viewpoints.

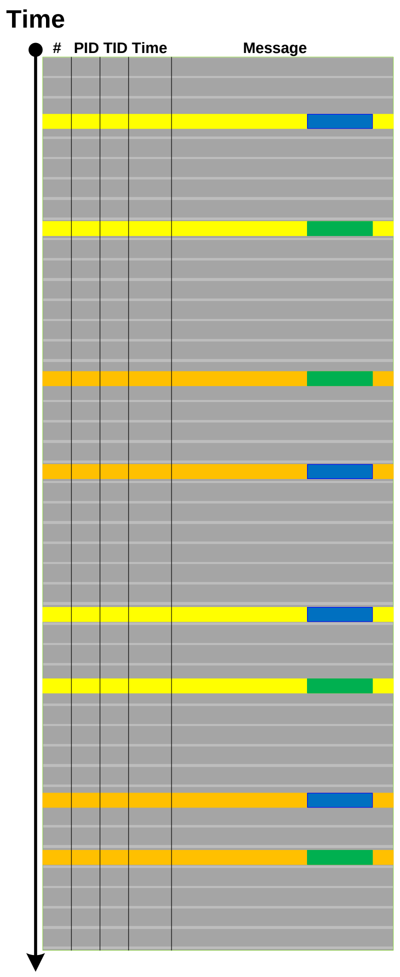
Data Reversal

Sometimes we notice that data values are in a different order than expected. We call this pattern **Data Reversal**. By data values, we mean some variable parts of a specific repeated message such the address of some structure or object. **Data Reversal** may happen for one message type:



But it can also happen for some message types and not for others. Typical example here are *Enter/Leave* trace messages for nested synchronization objects such as monitors and critical sections:



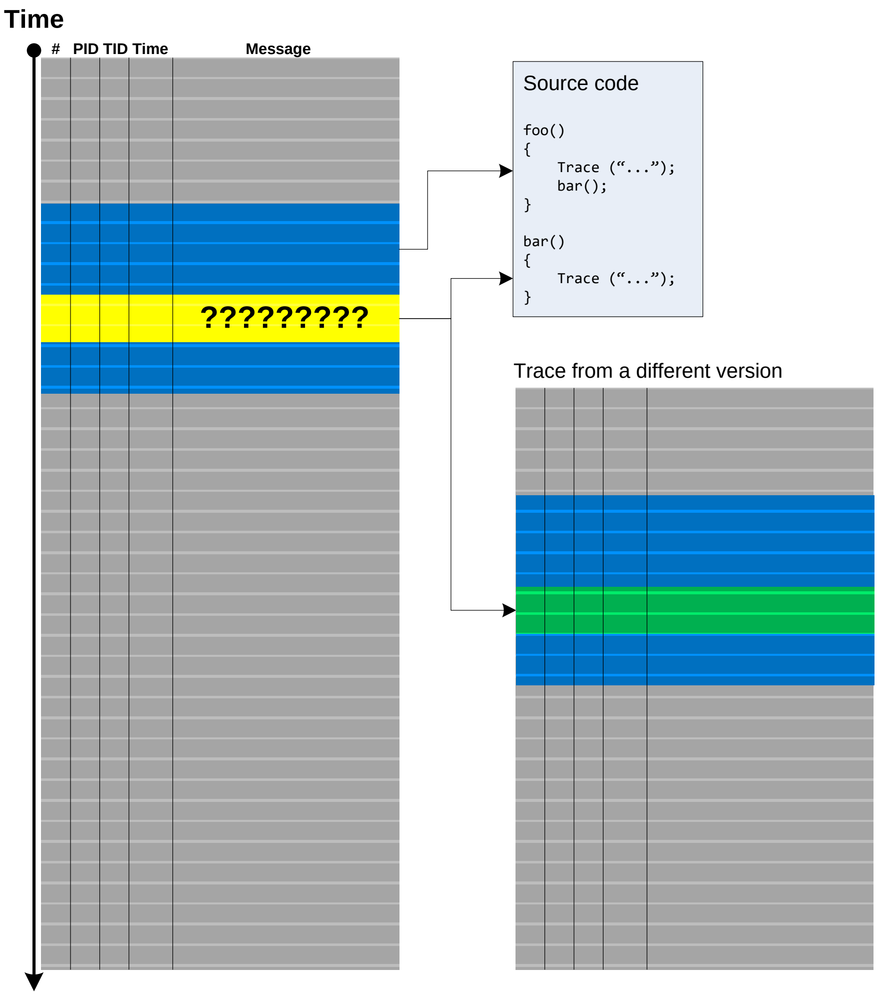


Since we talk about the same message type (the same **Message Invariant**, Volume 6, page 251), this pattern is different from **Event Sequence Order** (Volume 6, page 244) pattern.

In rare cases, we may observe Data Reversal inside one message with several variable parts but this may also be a case of **Data Association** (Volume 7, page 344).

Recovered Messages

If we analyze ETW-based traces such as CDF we may frequently encounter **No Trace Metafile** (Volume 5, page 296) pattern especially after product updates and fixes. This complicates pattern analysis because we may not be able to see **Significant Events** (Volume 5, page 281), **Anchor Messages** (Volume 5, page 293), and **Error Messages** (Volume 7, page 299). In some cases, we can recover messages by comparing **Message Context** (Volume 7, page 289) for unknown messages. If we have source code access, this may also help. Both approaches are illustrated in the following diagram:

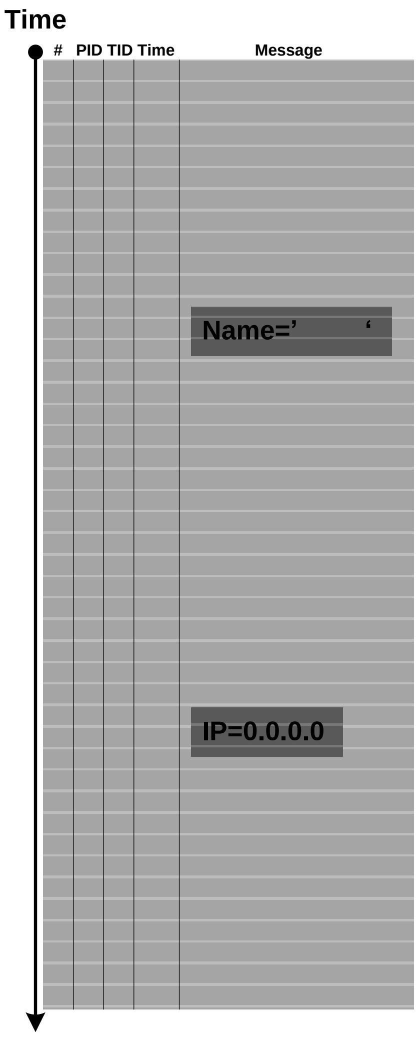


```
foo()
{
    Trace ("...");
    bar();
}

bar()
{
    Trace ("...");
}
```

The same approach may also be applied for a different kind of trace artifacts when some messages are corrupt. In such cases, it is possible to recover diagnostic evidence and, therefore, we call this pattern **Recovered Messages**.

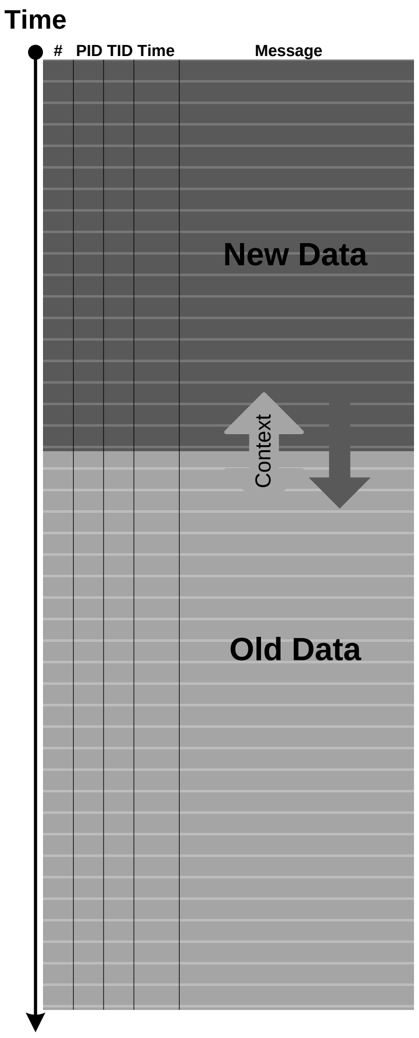
Palimpsest Messages

Palimpsest Messages are messages where some part or all of their content was erased or overwritten.



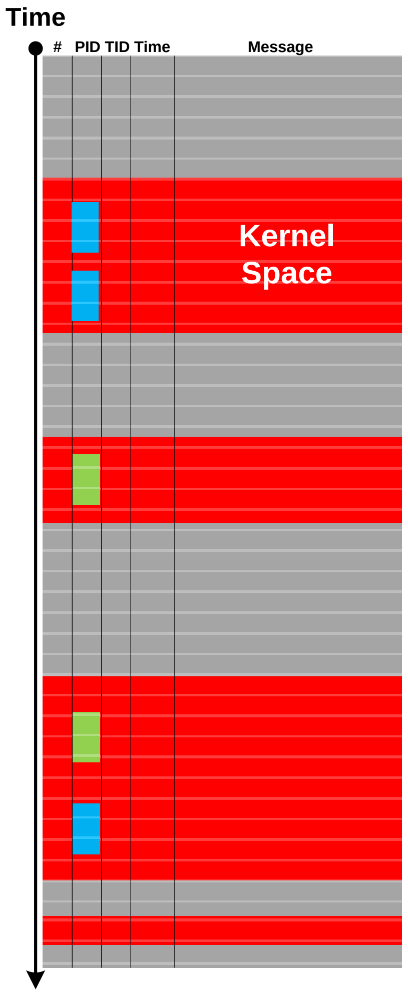
The name of this pattern comes from palimpsest[15] manuscript scrolls. Such messages may be a part of malnarratives (Volume 8a, page 121) or result from **Circular Tracing** (Volume 3, page 346) or trace buffer corruption. Sometimes, not all relevant data is erased and by using **Intra-** (Volume 3, page 347) and **Inter-Correlation** (Volume 4, page 350), and via the analysis of **Message Invariants** (Volume 6, page 251) it is possible to recover the original data. Also, as in **Recovered Messages** (page 86) pattern it may be possible to use **Message Context** (Volume 7, page 289) to infer some partial content.

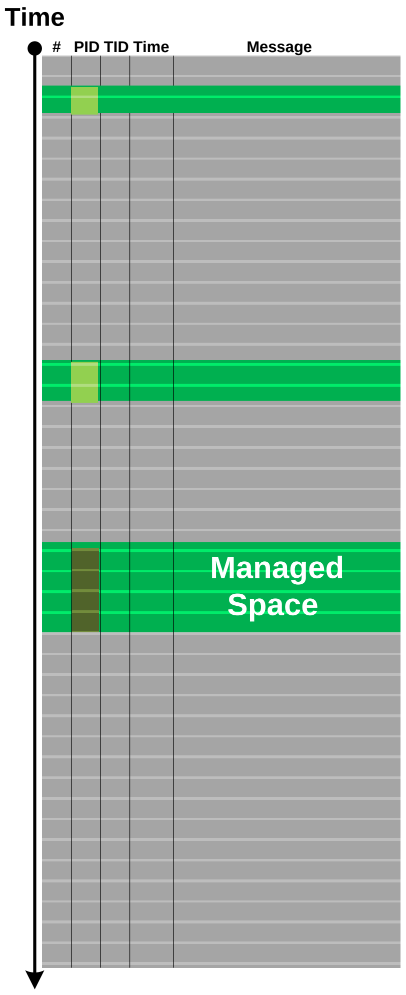
[15] http://en.wikipedia.org/wiki/Palimpsest


Time
#
PID
TID
Time
Message
New Data
Context
Old Data

Message Space

The message stream can be considered as a union of **Message Spaces**. A message space is an ordered set of messages preserving the structure of the overall trace. Such messages may be selected based on memory space they came from or can be selected by some other general attribute, or a combination of attributes and facts. The differences from **Message Set** (Volume 7, page 349) is that **Message Space** is usually much larger (with large scale structure) with various **Message Sets** extracted from it later for fine-grained analysis. This pattern also fits nicely with **Adjoint Spaces** (page 68). Here's an example of kernel and managed spaces in the same CDF / ETW trace from Windows platform where we see that kernel space messages came not only from System process but also from other process contexts:

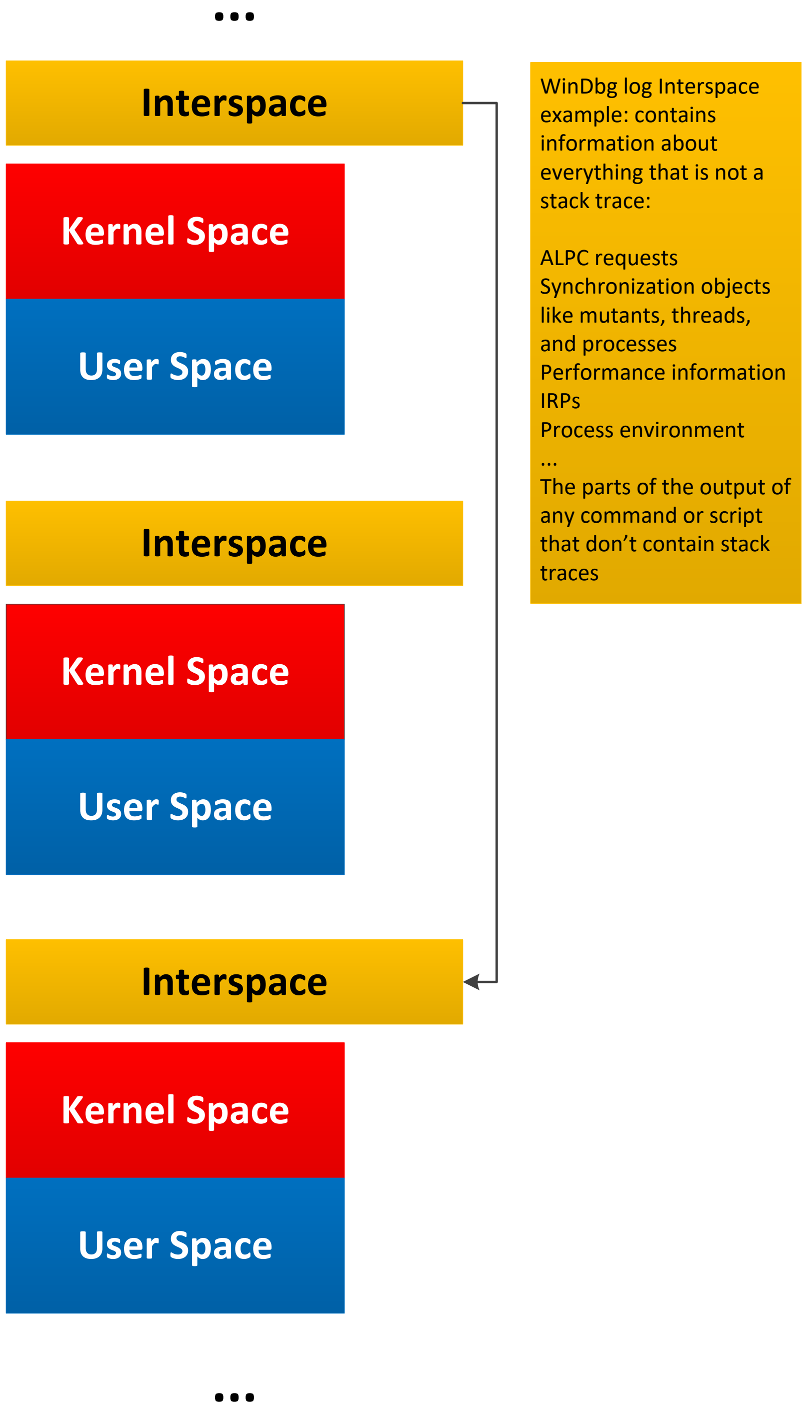




In the context of general traces and logs (page 121) such as debugger logs, separate **Message Space** regions may be linked (or “surrounded”) by **Interspace** (page 93).

Interspace

General traces and logs (page 121) may have **Message Space** (page 91) regions "surrounded" by the so-called **Interspace**. Such **Interspace** regions may link individual **Message Space** regions like in this diagram generalizing WinDbg **!process 0 3f** command output:

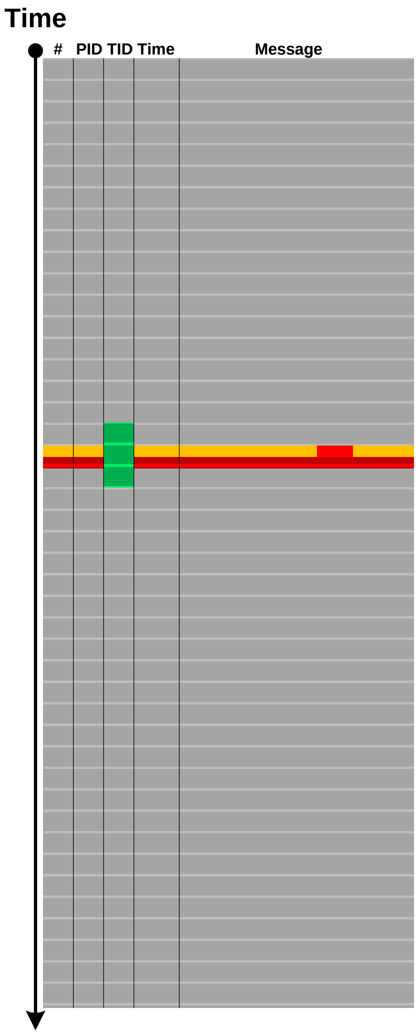


WinDbg log Interspace example: contains information about everything that is not a stack trace:

ALPC requests
Synchronization objects like mutants, threads, and processes
Performance information
IRPs
Process environment
...
The parts of the output of any command or script that don't contain stack traces

Translated Message

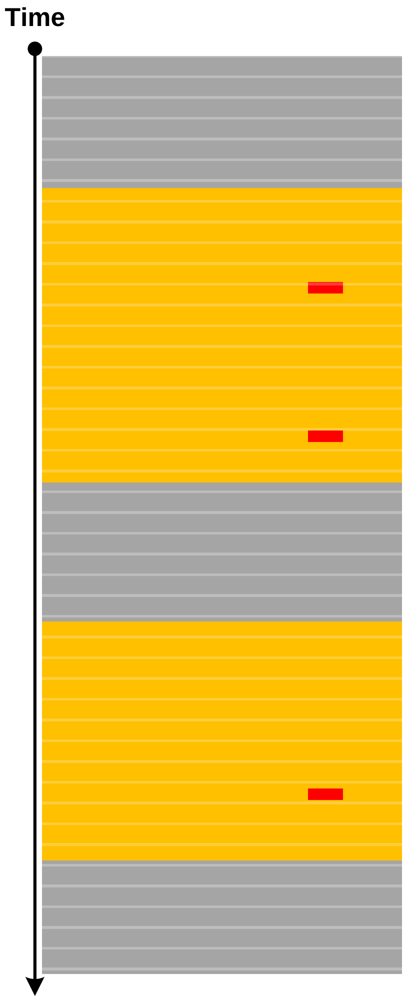
Sometimes we have messages that report about the error but do not give exact details. For example, “Communication error. The problem on the server side” or “Access denied error”. This may be the case of **Translated Messages**. Such messages are plain language descriptions or reinterpretations of flags, error and status codes contained in another log message. These descriptions may be coming from system API, for example, *FormatMessage* from Windows API, or may be from the custom formatting code. Since the code translating the message is in close proximity to the original message both messages usually follow each other with zero or very small **Time Delta** (Volume 5, page 282), come from the same component, file, function, and belong to the same **Thread of Activity** (Volume 4, page 339):



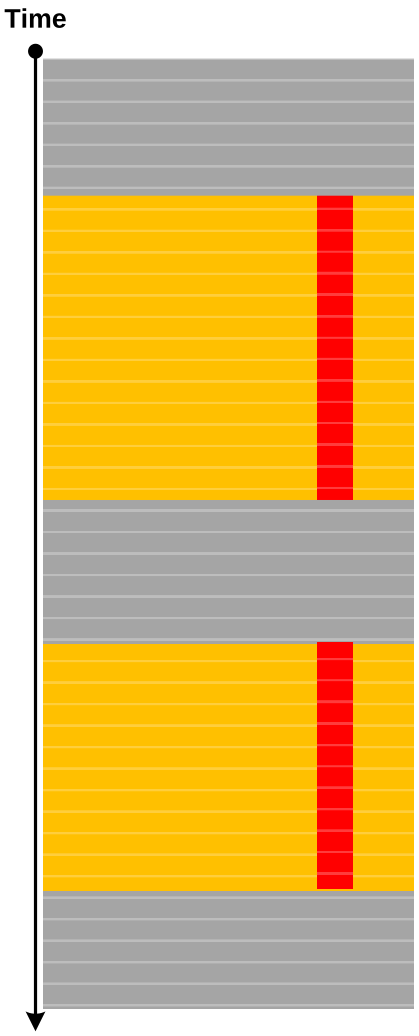
This pattern is different from **Gossip** (Volume 6, page 248) because the latter messages come from different modules, and, although they reflect some underlying event, they are independent of each.

Activity Disruption

Sometimes a few **Error Messages** (Volume 7, page 299) or **Periodic Errors** (Volume 3, page 344) with low **Statement Density** (Volume 4, page 335) for specific **Activity Regions** (Volume 4, page 348) or **Adjoint Threads of Activity** (for specific component, file or function, Volume 5, page 283) may constitute **Activity Disruption**. If the particular functionality was no longer available at the logging time then its unavailability may not be explained by such disruptions, and such messages may be considered **False Positive Errors** (Volume 5, page 303) in relation to the reported problem:



But, if we have **Periodic Message Blocks** (Volume 7, page 300) containing only **Periodic Errors** (Volume 3, page 344), **Activity Region** (Volume 4, page 348) or **Adjoint Thread** (Volume 5, page 283) **Discontinuity** (Volume 4, page 341), or simply **No Activity** (Volume 5, page 297), then we may have the complete cease of activity that may correlate with the unavailable functionality:


Time

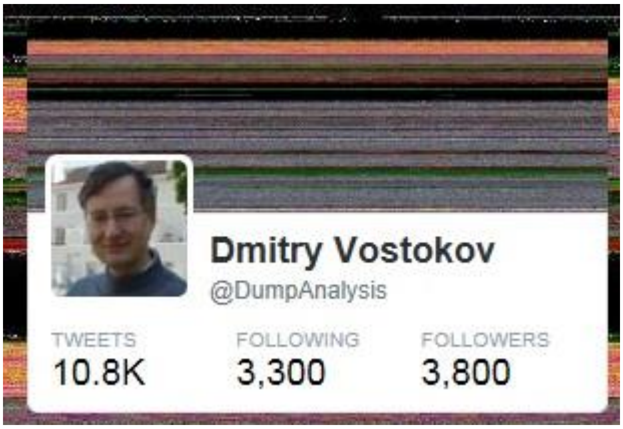
[This page is intentionally left blank]

PART 6: Fun with Debugging, Crash Dumps, and Traces

The Dump from the Future

```
deaddead: kd> !session
Sessions on machine: 3735936685
```

Exchange Rate on 16.12.14



Check the Plug

Debugging Slang

YAWE

YAWE - **Y**et **A**nother **W**indbg **E**xtension

Embedded Software Engineer

A software engineer embedded in a non-software team.

Compare with: Embedded Technical Support Engineer.

Minute-wise

When we have two logs from different time zones or generated using different time APIs, so we compare minutes.

Developer

One who uses bricks of books to build the walls of knowledge.

Multidigitalist

A person who simultaneously wears an Android watch, an Apple watch, and a Microsoft Band.

KgB

Hardcopy data measured in kilogram-bytes. Usage: We got plenty of KgB archives.

CIQ (Crash IQ)

Crash IQ (CIQ, pronounced "sick") is the measure of intellectuality of a crash dump and, reciprocally, the measure of system sickness. The less CIQ your crash dump has, the easier your analysis. For example, compare low CIQ NULL pointer issue with High CIQ system freeze.

Pat Ching

Pat Ching (patching) - from Pat (a light blow ... with ... instrument) and Ching (Book of Changes).

Explosive Mixture

PowerBuilder VM + Visual Basic VM

POEM

A pointer to OEM. Example: I found a few poems in this memory dump.

YearNormous Day

YNK day calculated via formula YYYY-MM-DD = N000 or YYYY-DD-MM = N000. Something eNormous may happen on that day.

eNormous

Normal in e-World (virtual) but may have a very big influence on Real World.

2015 - The Year of RAM

Because 2015 is the true year of RAM we greet you again in memory dump analysis style:

```
; random analysis of memory / reversing analysis of memory

0:001> g o a t 2015
Bp expression 'o ' could not be resolved, adding deferred bp
*** Bp expression 'o ' contains symbols not qualified with module
name.
Bp expression 't ' could not be resolved, adding deferred bp
*** Bp expression 't ' contains symbols not qualified with module
name.
Unable to insert breakpoint 10001 at 00000000`0000000a, Win32 error
0n299
    "Only part of a ReadProcessMemory or WriteProcessMemory request
was completed."
The breakpoint was set with BP.  If you want breakpoints
to track module load/unload state you must use BU.
go bp10001 at 00000000`0000000a failed
Unable to insert breakpoint 10003 at 00000000`00002015, Win32 error
0n299
    "Only part of a ReadProcessMemory or WriteProcessMemory request
was completed."
The breakpoint was set with BP.  If you want breakpoints
to track module load/unload state you must use BU.
go bp10003 at 00000000`00002015 failed
WaitForEvent failed
ntdll!DbgBreakPoint+0x1:
00000000`77280591 c3              ret
```

Diagnostics and Debugging in Science Fiction

Here’s an incomplete list of SF short stories, novellas, and novels I have read by the time of this writing with my summaries and thoughts.

"Guest of Honor" (by Robert Reed)

> The imperfect solution to the problem of immortality, stupid death accidents, and human curiosity. Humans designed for to bring memories back to their parents. The possibility of suicidal memory resonance. The terrible fear of being disassembled.

"The Man Who Walked Home" (by James Tiptree, Jr)

> A malfunction absorbed in the greater malfunction. No diagnostics of the former one is possible. What we see is "Dust from the future". Very sparse trace messages, one per year. Takes centuries to get enough for diagnostics. Diagnostics and Special Relativity. When Time becomes Space.

"Martian Blood" (by Allen M. Steele)

> About the consequences of diagnostics.

"The Clockwork Atom Bomb" (by Dominic Green)

> Set in post-WW3 African environment featuring black hole troubleshooting.

"Pathways" (by Nancy Kress)

> Prion debugging.

"The End of the World" (by Sushma Joshi)

> A Nepalese town population enjoys the last meal on hearing such prognostics. Then comes the end of the end.

"Fermi and Frost" (by Frederik Pohl)

Bifurcation points in the narrative of nuclear apocalypse.

"Dinner in Audoghast" (by Bruce Sterling)

Prognostics at its best.

"The Discovered Country" (by Ian MacLeod)

A too perfect program is most likely malware.

"When Sysadmins Ruled the Earth" (by Cory Doctorow)

Fighting entropy after a global disaster. But whodunit?

"The Waiting Stars" (by Aliette de Bodard)

One's freedom is another's slavery. Prenatally modified births of space ship control systems. Confucianism at cosmic scales. Deep memories that were not erased cause severe depression. Unconsciousness in the computer, consciousness in some human body. A body as a cage. The clash of civilizations and a personal relationship. Unfolding of being.

"Seveneves" (by Neal Stephenson)

"... I'm right in the middle of debugging this method..."

"Mortimer Gray's History of Death" (by Brian Stableford)

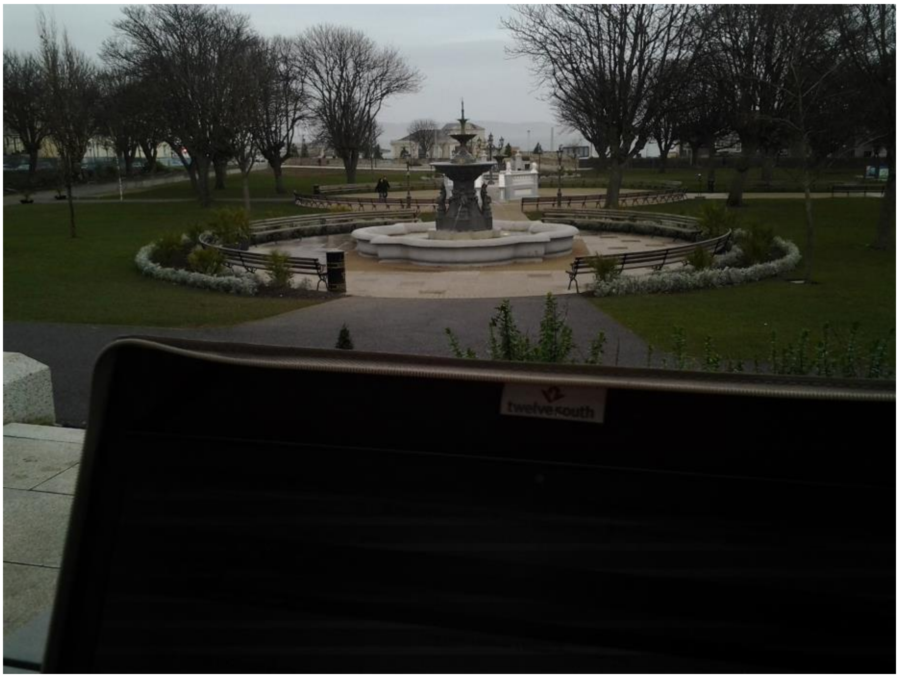
Interesting SF novella: it combines two narratives; one is a literary criticism of the multivolume history of death written millennia far into the future; the other narrative is some kind of a fragmented memoir from the author of that history book. The war on death is the main drive of humanity. When death problematic is solved for individuals (becomes "a social contract") and seems there's nothing to do (at least for the author) there comes a distant existential death threat to all humanity that needs to be fought off again (the author got the new meaning of life). What caught my attention is this workaround pattern for software behavior: "I handed over full responsibility for answering all my calls to a state-of-the-art Personal Simulation program, which grew so clever and so ambitious with practice that it began to give live interviews on broadcast television. Although it offered what was effectively no comment in a carefully elaborate fashion I eventually thought it best to introduce a block into its operating system – a block that ensured that my face dropped out of public sight for half a century."

"Guardians of the Phoenix" (by Eric Brown)

Horrors of cannibals encounter in the post-apocalyptic Earth (after China invaded India) reminded me "The 13th Warrior" movie. You still debug at the electronic high system level there (replace transistors when they blow).

Software and Hardware Exceptions

Gloomy outlook before I started my work on Pattern 0n222 (**Software Exception**)



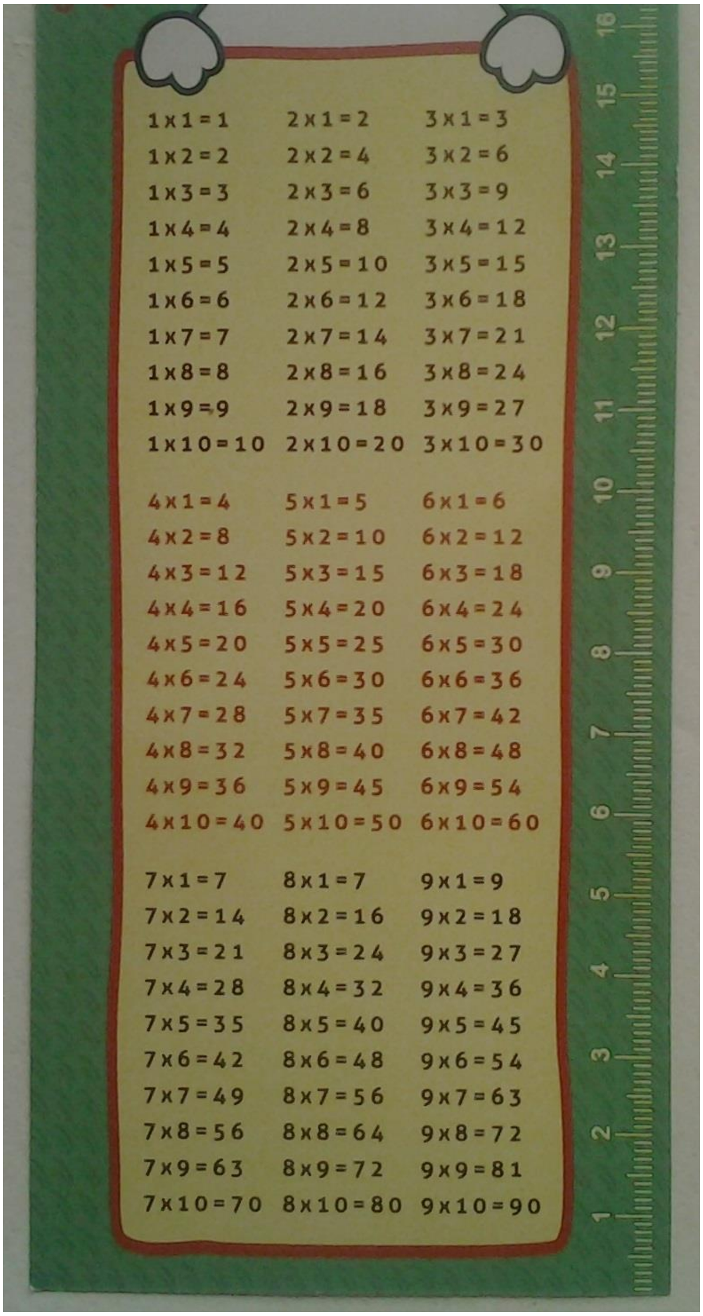
But after I finished my work on Pattern 0n222 (**Software Exception**) weather improved with **Hardware Exception**:

Logging for Kids

Trace in Space from the Log of Trace Malone Space Detective

- ISBN-13: 9780340626696
- Publisher: Hodder Headline plc
- Publication date: 2/29/2000
- Pages: 118
- Description: “Trace is the youngest-ever cadet in charge of her own spaceship. Unfortunately, her crew consists of a computer with an attitude, a droid with the personality chip of a surfer and an armadillo. Now the team has its first mission to find the missing computer chips from Planet Megalon.”

Find the Bug



1x1=1	2x1=2	3x1=3
1x2=2	2x2=4	3x2=6
1x3=3	2x3=6	3x3=9
1x4=4	2x4=8	3x4=12
1x5=5	2x5=10	3x5=15
1x6=6	2x6=12	3x6=18
1x7=7	2x7=14	3x7=21
1x8=8	2x8=16	3x8=24
1x9=9	2x9=18	3x9=27
1x10=10	2x10=20	3x10=30
4x1=4	5x1=5	6x1=6
4x2=8	5x2=10	6x2=12
4x3=12	5x3=15	6x3=18
4x4=16	5x4=20	6x4=24
4x5=20	5x5=25	6x5=30
4x6=24	5x6=30	6x6=36
4x7=28	5x7=35	6x7=42
4x8=32	5x8=40	6x8=48
4x9=36	5x9=45	6x9=54
4x10=40	5x10=50	6x10=60
7x1=7	8x1=7	9x1=9
7x2=14	8x2=16	9x2=18
7x3=21	8x3=24	9x3=27
7x4=28	8x4=32	9x4=36
7x5=35	8x5=40	9x5=45
7x6=42	8x6=48	9x6=54
7x7=49	8x7=56	9x7=63
7x8=56	8x8=64	9x8=72
7x9=63	8x9=72	9x9=81
7x10=70	8x10=80	9x10=90

Music for Debugging

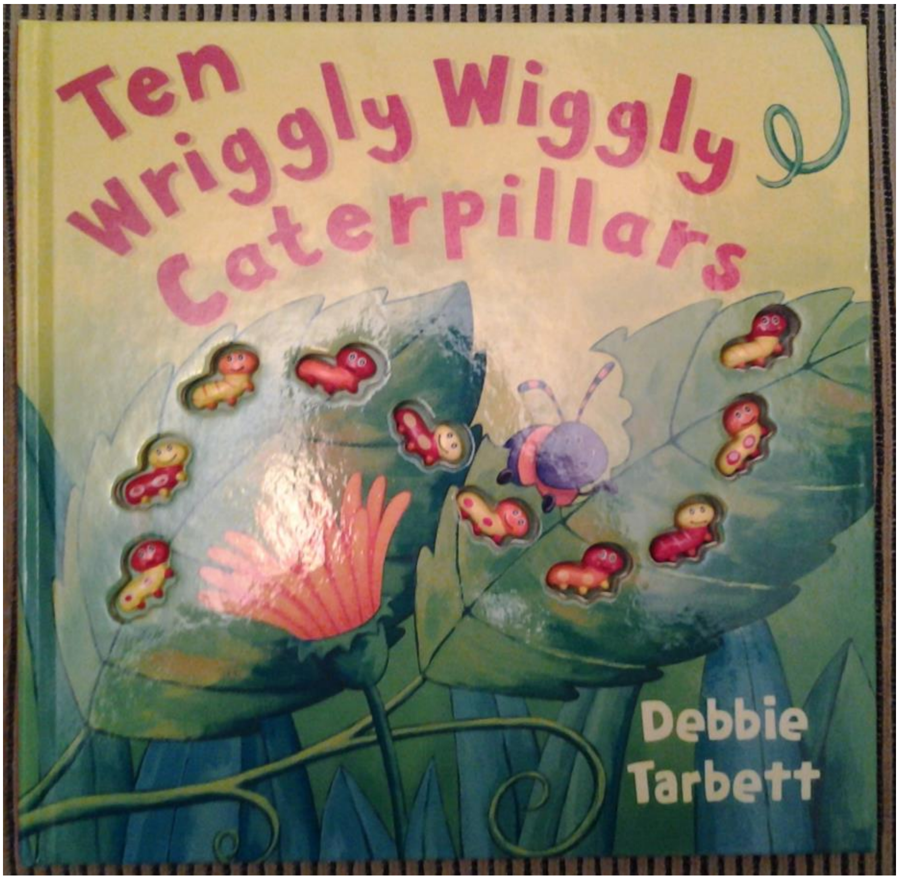
- “The 50 Darkest Pieces of Classical Music”. Great quality recordings on Fostex TH600 headphones and Marantz CD player.

- In addition to well-known composers, we also discovered Suite Gothique (Boëllmann) / Toccata for organ.

- Music for tracing. Gloria Coates String Quartet No. 5. The tracks:

  1. Through Time
  2. Through Space
  3. In the Fifth Dimension

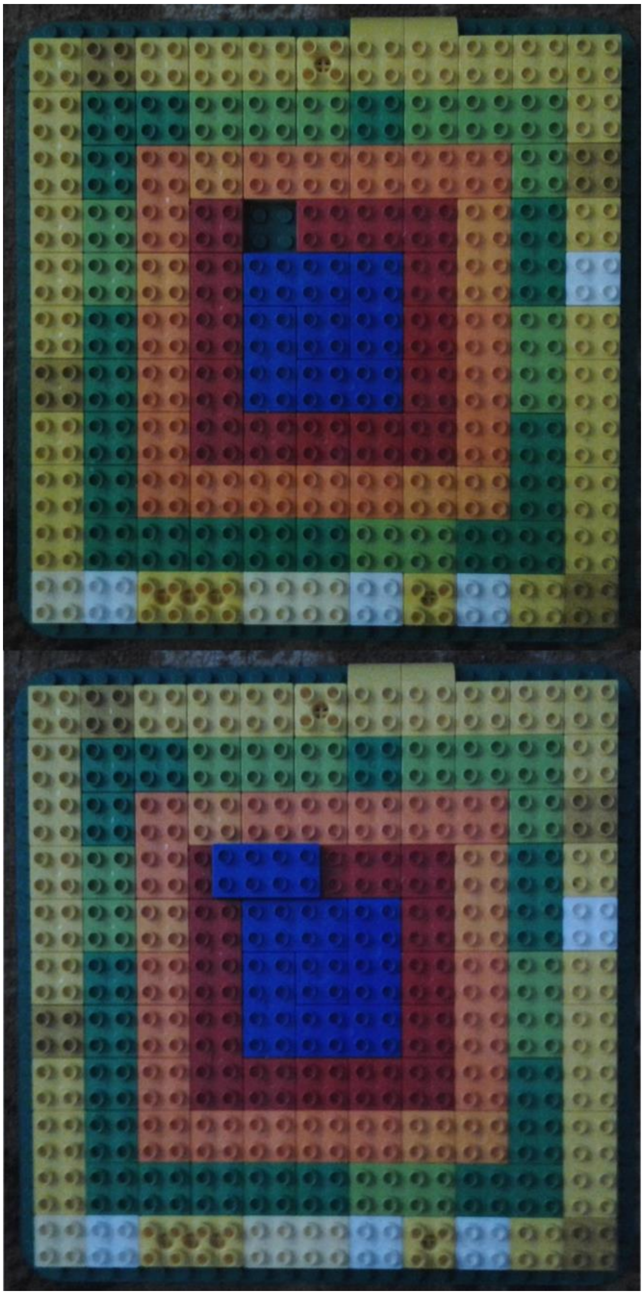
Tracing and Counting Book



The Last Error

Looks like I'm not the first "Vostokov" to talk about errors. Just found 1966 "L'ultimo errore" book (The Last Error) by Vladimir Vostokov and Oleg Smelov. Interesting that there is a crash dump analysis pattern called **Last Error Collection**. This seems to be a well-known Soviet espionage thriller and the author "Vostokov" is actually a pseudonym.

Patching the Hardware Defect



Pattern Match

PART 7: Software Narratology

Coding and Articoding

The analysis of software traces and logs is largely a qualitative activity. We look for specific problem domain patterns using general analysis patterns[16]. Some methodological aspects of this software defect research are similar to “qualitative research” method in social sciences[17]. The latter method uses the so-called “coding” techniques for data analysis[18]. Software traces and debugger logs from memory dumps are software execution artifacts we previously called DA+TA (Dump Artifact + Trace Artifact)[19]. We propose to use similar “coding” techniques to annotate them with diagnostic indicators, signal, and sign mnemonics, and patterns (such as software diagnosis codes[20]). We, therefore, call this software post-construction “coding” as **articoding** (**artecoding**), from **arti**fact (**arte**fact) + coding, to distinguish it from software construction coding. Such articoding forms a part of software post-construction problem solving[21]. Articodes form a second order software narrative[22] and can be articoded too.

Many software tools were developed for assisting qualitative research coding, and these can be reused for “coding” debugger logs, for example. In addition to those tools, general word and table processing programs can be used as well for some types of artifacts. Here we show MS Word for a WinDbg log example. The debugger log with stack traces from all processes and threads was

---

[16] Memory Dump Analysis Anthology, Volume 7, page 393

[17] http://en.wikipedia.org/wiki/Qualitative_research

[18] http://en.wikipedia.org/wiki/Qualitative_research#Coding

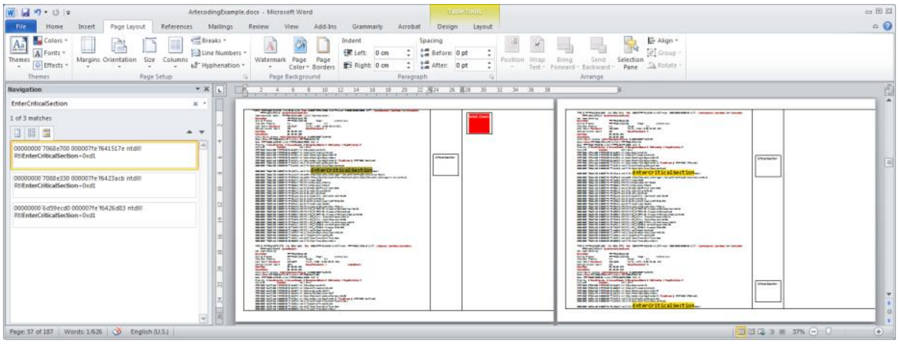
[19] Memory Dump Analysis Anthology, Volume 3, page 330

[20] Volume 7, page 446

[21] Introduction to Pattern-Driven Software Problem Solving, page 8

[22] Memory Dump Analysis Anthology, Volume 8a, page 123

loaded into MS Word template table with 3 columns. The first column is the log itself, the second column is for diagnostic indicators (such as critical section, CPU consumption, ALPC wait, etc.), and the third column is for pattern language articodes (here we use pattern names from Memory Analysis Pattern Catalogue[23], for traces we can use MS Excel and Trace and Log Analysis Pattern Catalogue[24]). Formatting and highlighting creativity here is unlimited. Irrelevant parts from the log can be deleted, and the final analysis log can have only relevant annotated tracing information.



```
00000000`7068e700 000007fe`f641517e ntdll!RtlEnterCriticalSection+0xd1
00000000`7088e330 000007fe`f6423acb ntdll!RtlEnterCriticalSection+0xd1
00000000`6d59ecd0 000007fe`f6426d83 ntdll!RtlEnterCriticalSection+0xd1
```

The full-size picture can be found here:
http://www.dumpanalysis.org/blog/files/ArticodingExample.png

[23] Encyclopedia of Crash Dump Analysis Patterns

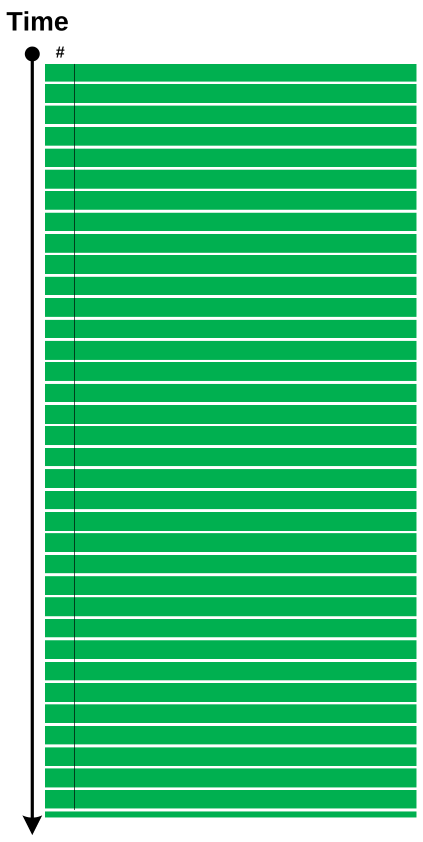
[24] Software Trace and Log Analysis: A Pattern Reference

PART 8: Software Diagnostics, Troubleshooting, and Debugging

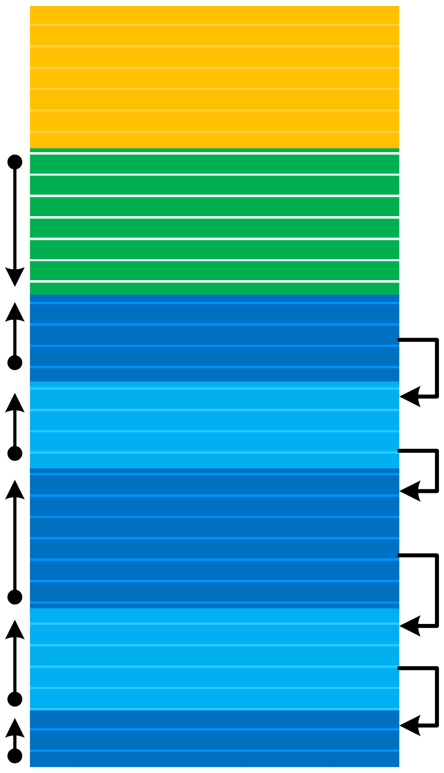
Special and General Trace and Log Analysis

Most software traces include message timestamps or have an implicit time arrow via sequential ordering. We call such traces **Special**. The analysis is special too because causality is easily seen. Typical examples of analysis patterns here are **Discontinuity** (Volume 4, page 341), **Time Delta** (Volume 5, page 282), **Event Sequence Order** (Volume 6, page 244), **Data Flow** (Volume 7, page 296, see also time dependency markers in the training course reference[25]), and more recently added patterns such as **Back Trace** (Volume 8a, page 95), **Timeout** (page 61), **Milestones** (Volume 8a, page 105), and **Event Sequence Phase** (Volume 8a, page 103). **Inter-** (Volume 4, page 350) and **Intra-Correlation** (Volume 3, page 347) analysis is also easy.

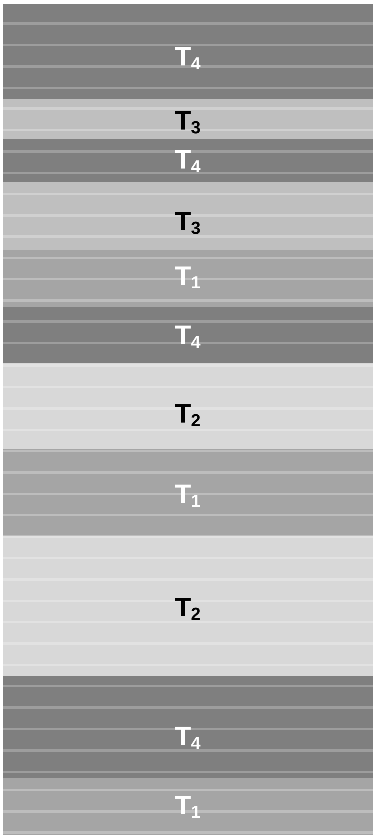
25 http://www.patterndiagnostics.com/Training/Accelerated-Windows-Software-Trace-Analysis-Public.pdf



On the other side, there are plenty of software logs or digital media artifacts with “chaotic” records where time arrow is missing or only partial. Typical examples are debugger logs from WinDbg debugger from Microsoft Debugging Tools for Windows or logs from debugging sessions on other platforms. Such logs may contain global ordering such as the list of processes and threads (**Last Object** memory analysis pattern, Volume 8a, page 37) interspaced with local pockets of stack traces that have reversed ordering. Some logging output may not have any ordering or timing information whatever.



In a more general case, logging may be completely arbitrarily. A typical example is raw stack analysis and its **Rough Stack Trace** (Volume 8a, page 39) and **Past Stack Trace** (Volume 8a, page 43) patterns.

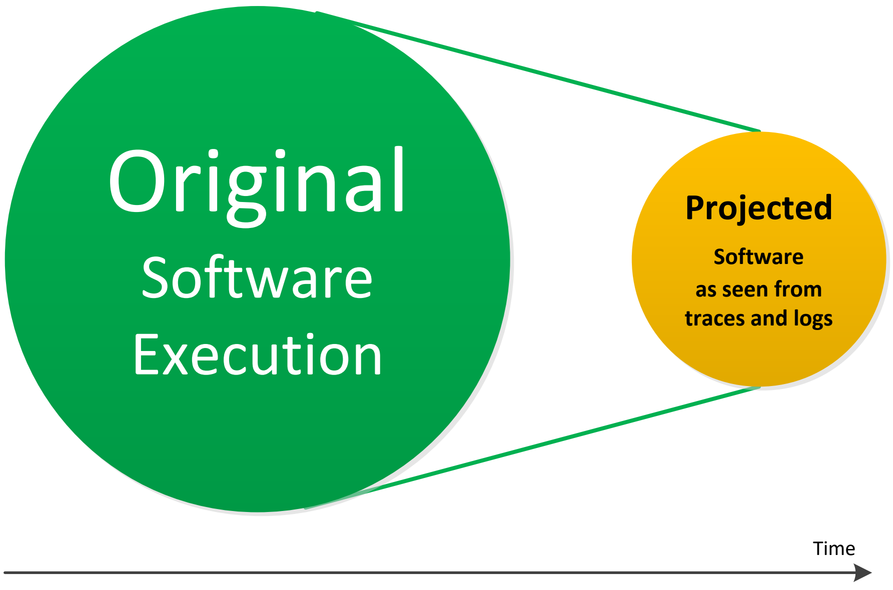


We call such traces **General**. The main task of general trace analysis is to recover causality. It may be possible if another analysis pattern is introduced called **Causality Markers**. The prototypes of such a pattern are various **Wait Chains**[26], **Waiting Thread Time** (Volume 1, page 343) memory analysis pattern and its process memory dump equivalent (Volume 2, page 319).

[26] Encyclopedia of Crash Dump Analysis Patterns, page 952

Projective Debugging

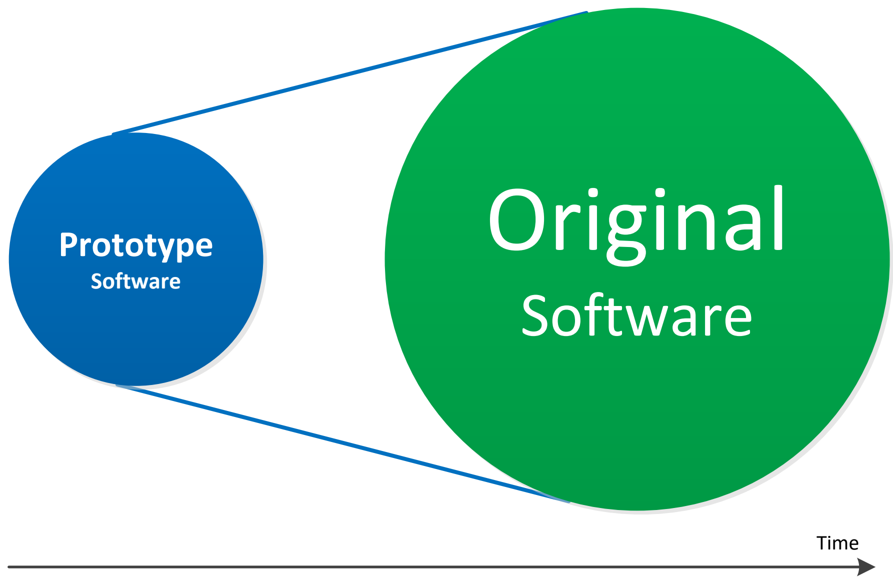
Modern software systems and products are hard to debug despite their elaborate tracing and logging facilities. Typical logs may include millions of trace messages from hundreds and thousands of components, processes, and threads. The postmortem diagnostic analysis became more structural after the introduction of **Trace and Log Analysis Patterns**[27] but live debugging requires a lot more efforts. Here we introduce **Projective Debugging** as a tool for trace-level debugging. Its main idea is to analyze, diagnose and debug the so-called "projected" execution of software as seen from the original software execution traces and logs:



Picture 1. Original software execution is mapped into projected software as seen from traces and logs.

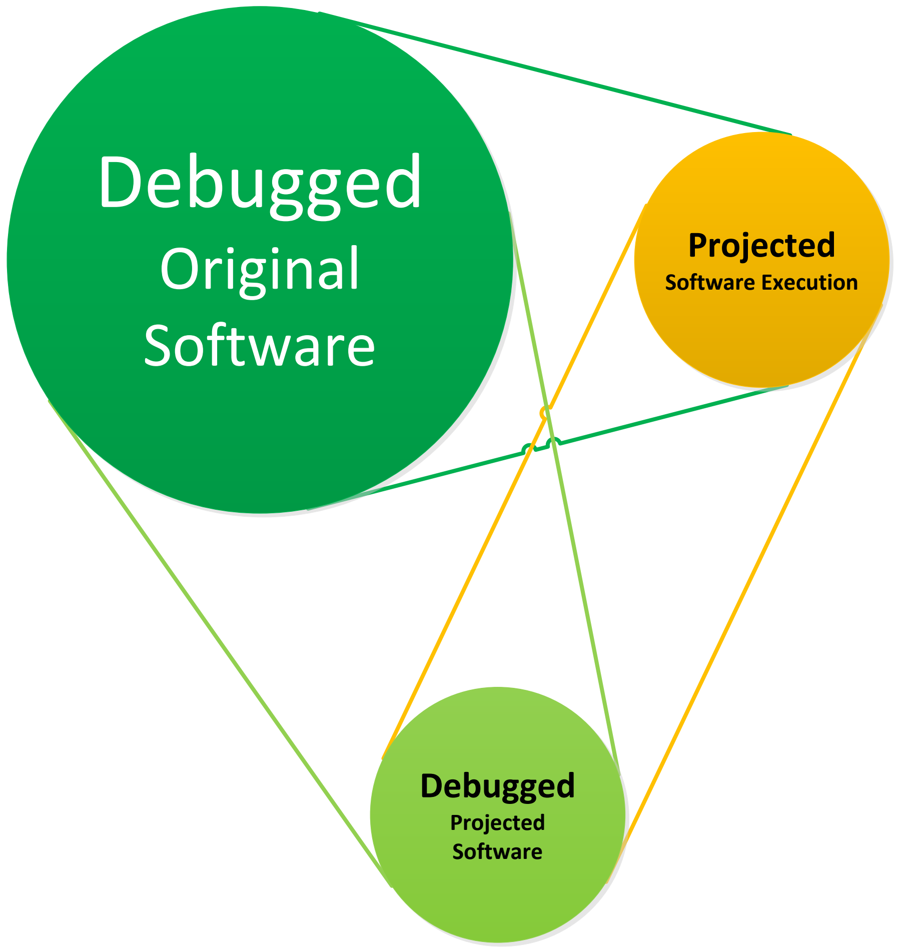
[27] Software Trace and Log Analysis: A Pattern Reference (ISBN: 978-1-908043801)

Please notice, that **Projective Debugging** is different from the so-called *Prototype Debugging* by creating models after the software product is built (some engineering methodologies prescribe that prototypes should be discarded before building the product):



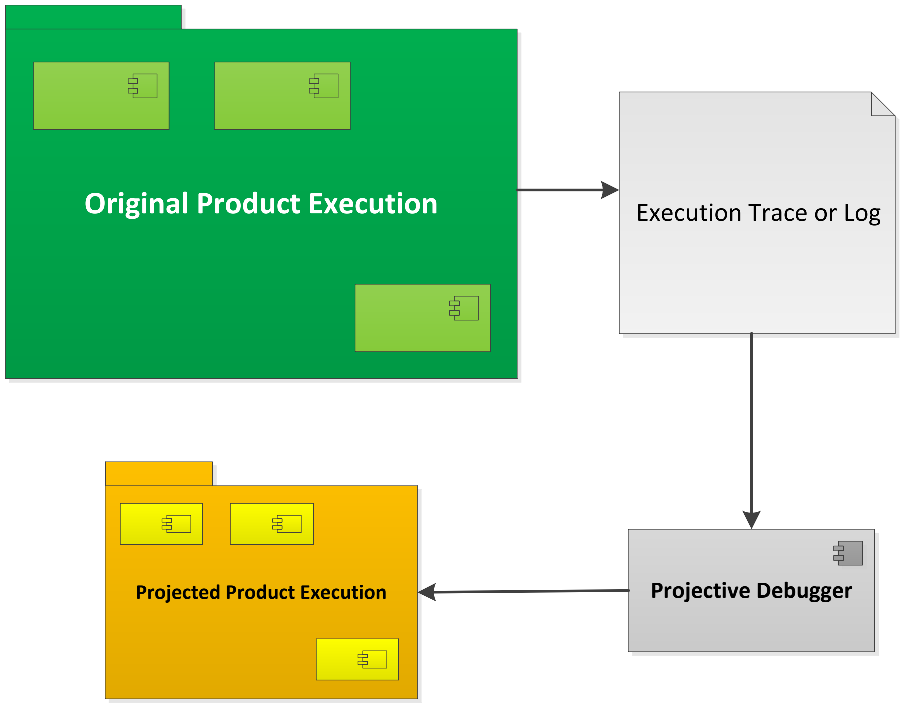
Picture 2. The prototype software is mapped into the product.

The problems diagnosed and solved in the projected system are fed back into the original system:



Picture 3. Debugged projected software is mapped into the original software.

The implementation of the main idea of **Projective Debugging** is that: we can take a trace or log, interpret every trace message according to some rules, and translate it into executable code mirroring components and execution entities such as user sessions, processes, and threads. This is a task for the **Projective Debugger**, and it is illustrated in the following diagram where we borrowed notation from UML:



Picture 4. The logs and traces from the original product execution are translated by Projective Debugger.

For example, the **Projective Debugger** (ProjectiveDebugger.exe) interprets these very simple messages below, and creates a process PID220.exe (we have only one thread), and then opens a file "data.txt". After 10 seconds, it closes the file.

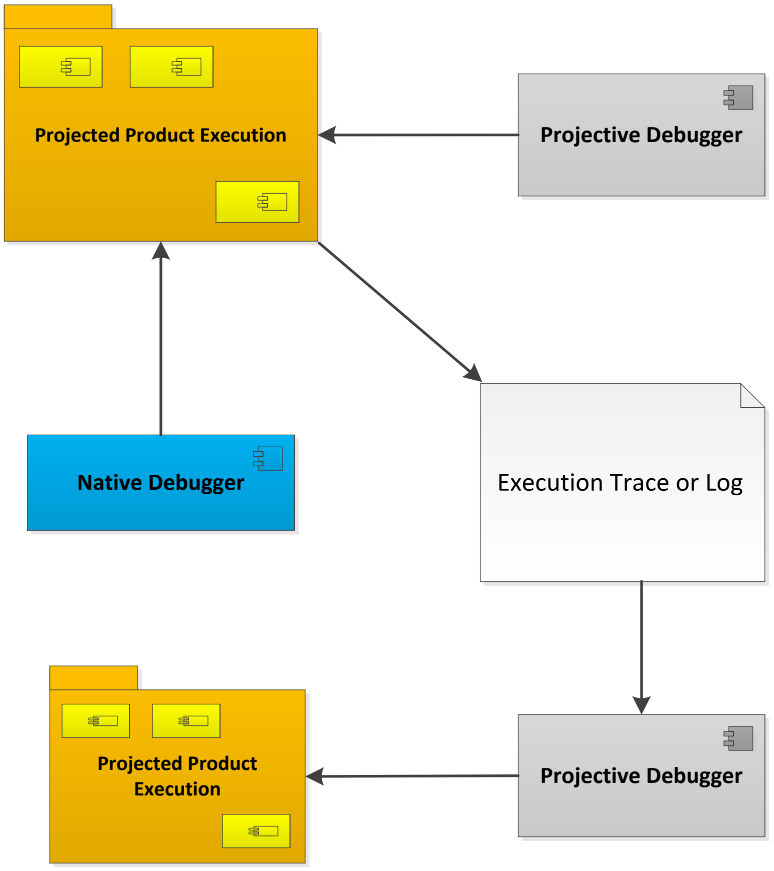
```
Time     PID  TID  Message
------------------------------------
11:00:00 220  330  Open "data.txt"
11:00:10 220  330  Close "data.txt"
```

In addition to executing code corresponding to messages using the same time deltas as read from the trace, it may scale execution time proportionally, for example, executing 2-day log in a matter of minutes. Such scaling may also uncover additional synchronization bugs.

The trace may be pre-processed, and all necessary objects created before execution or it may be interpreted line by line. For complex traces, the projected source code may be generated for later compilation, linking, and execution. Once the projected code is executed, breakpoints may be set on existing traces, and other types of **Debugging Implementation Patterns**[28] may be applied. Moreover, we may re-execute the trace several times and even have some limited form of backward execution.

[28] Accelerated Windows Debugging[3] (ISBN: 978-1908043566)

The resulting code model can be inspected by native debuggers for the presence of **Memory Analysis Patterns**[29] and can even have its own logging instrumentation with traces and logs analyzed by another instance of the **Projective Debugger**:



Picture 5. Projected Product Execution is inspected by a native debugger and also generates its own set of traces and logs to be projected to another model by another instance of the Projected Debugger.

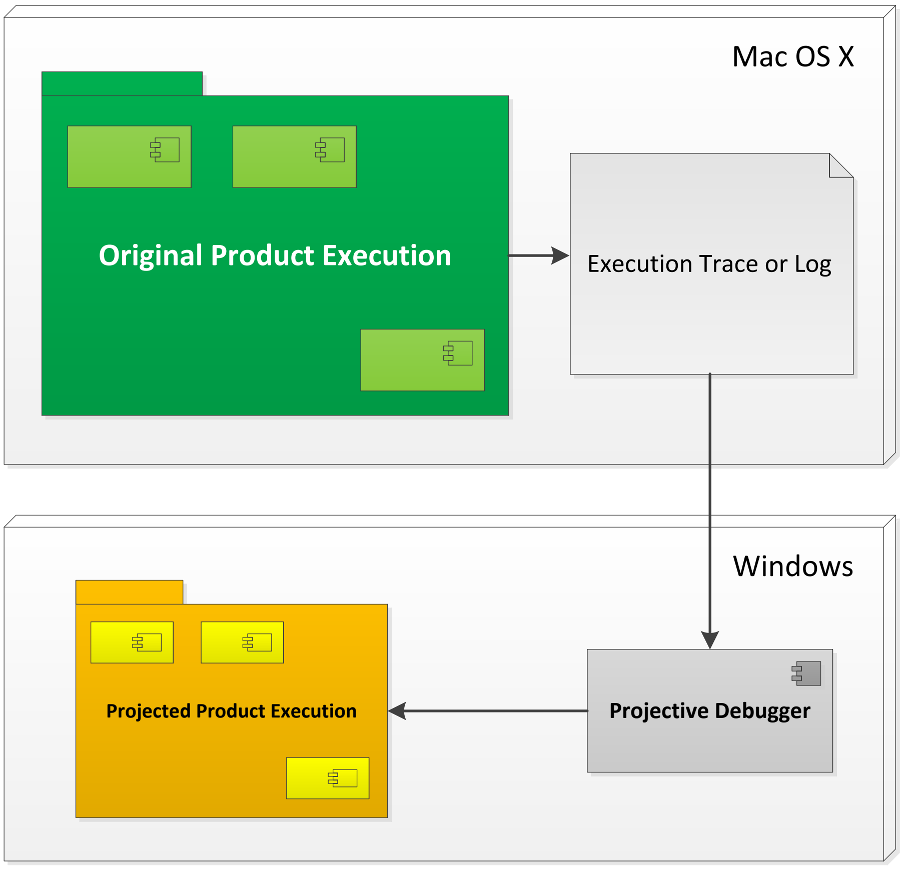
---

[29] Encyclopedia of Crash Dump Analysis Patterns: Detecting Abnormal Software Structure and Behavior in Computer Memory (ISBN: 978-1-906717216)

We created the first version of the **Projective Debugger** and successfully applied to a small trace involving synchronization primitives across several threads. The **Projective Debugger** was able to translate it into an executable model with the same number of threads and the same synchronization primitives and logic. The resulting process was hung as expected and we attached a native debugger (WinDbg for Windows) and found a deadlock.

Since traces are analyzed from platform-independent **Software Narratology**[30] perspective, it is possible to get a trace from one operating system, and then, after applying a set of rules, re-interpret it into an executable model in another operating system. We created the similar multithreading test program on Mac OS X that was hung and reinterpreted its trace into an executable model under Windows:

[30] Software Narratology: An Introduction to the Applied Science of Software Stories (ISBN: 978-1908043078)



Picture 6. The traces and logs from the original product execution on Mac OS X are projected by a Windows version of Projective Debugger into an executable model for a Windows platform.

Since resultant executable models can also have corresponding logging instrumentation mirroring original tracing, any problems found in executable models can be fixed iteratively and, once the problem disappears, the resulting fix or configuration may be incorporated into the original product or system.

If tracing involves kernel space and mode, a specialized projected executable can be created to model the operating system and driver level.

The more trace data you have, the more real your projected execution becomes. However, we want to have enough tracing statements but not to complicate the projected model. Then, ideally, we should trace only relevant behavior, for example, corresponding to use cases and architecture.

Projective Debugging may also improve your system or product maintainability by highlighting whether you need more tracing statements and whether you need more accurate and complete tracing statements.

Pattern! What Pattern?

There is confusion about patterns of diagnostics such as related to crash dump analysis and software trace and log analysis. We are often asked about pattern percentage detection rate or whether it is possible to automate pattern diagnostics. Before asking and answering such questions, it is important to understand what kinds of patterns are meant. Patterns of diagnostics can be subdivided into concrete and general problem patterns, and, also, into concrete and general analysis patterns.

Problem patterns are simply diagnostic patterns, and they can be defined as (fusing *Diagnostic Pattern*[31] and *Diagnostic Problem*[32] definitions):

A common recurrent identifiable set of indicators (signs) together with a set of recommendations and possible solutions to apply in a specific context.

Concrete Problem Patterns are particular sets of indicators, for example, an exception stack trace showing an invalid memory access in the particular function from the particular component/module code loaded and executed on Windows platform.

But such indicators can be generalized from different products and OS platforms giving rise to **General Problem Patterns** forming a pattern language. Our previous example can be generalized as **Exception Stack Trace** (Volume 4, page 337) showing **Invalid Pointer** (Volume 1, page 267) and **Exception Module** (Volume 8a, page 80). **Concrete Problem Patterns** are the implementation of the corresponding **General Problem Patterns**.

Now, it becomes clear why **Memory Analysis Pattern Catalog**[33] doesn't have any concrete BSOD bugcheck numbers. Most of such numbers are concrete implementations of **Self-Diagnosis** (Volume 6, page 89) pattern.

---

[31] Pattern-Oriented Software Forensics: A Foundation of Memory Forensics and Forensics of Things, page 13

[32] Ibid., page 14

Then we have **Concrete Analysis Patterns** as particular techniques to uncover **Concrete Problem Patterns.** For example, thread raw stack analysis for historical information to reconstruct a stack trace. Again, such techniques vary between OS platform and even between debuggers.

Generalizing again, we have **General Analysis Patterns**, for example, analyzing **Historical Information** (Volume 1, page 458) in **Execution Residue** (Volume 2, page 239) to construct **Glued Stack Trace** (Volume 7, page 178).

General Problem Pattern descriptions may already reference **General Analysis Patterns,** and in some cases, both may coincide. For example, **Hidden Exception** (Volume 1, page 271) pattern uses **Execution Residue** pattern as a technique to uncover such exceptions.

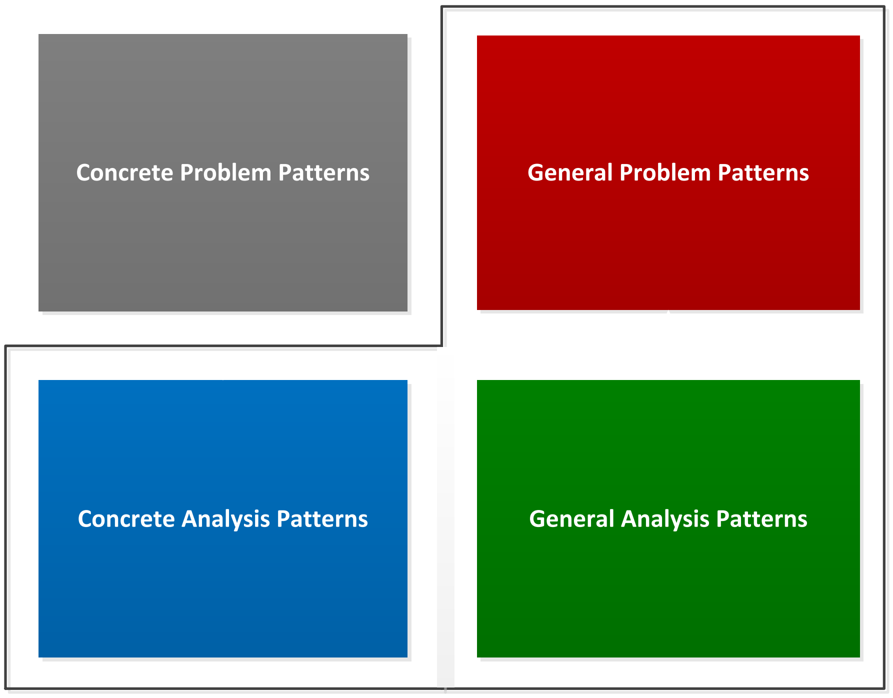
Most of **Software Trace and Log Analysis Patterns**[34] are **General Analysis Patterns** that were devised and cataloged to structure the analysis of the diverse logs from different products and OS platforms[35]. For example, a specific data value common to both working and problem logs that helps to find out the missing information from the problem description can be generalized to **Inter-Correlation** (Volume 4, page 350) analysis between the problem trace and **Master Trace** (Volume 6, page 247) using **Shared Point** (Volume 7, page 341).

[33] Encyclopedia of Crash Dump Analysis Patterns

[34] Software Trace and Log Analysis: A Pattern Reference

[35] Malware Narratives: An Introduction, page 14

This partitioning is depicted in the following diagram:


Concrete Problem Patterns
General Problem Patterns
Concrete Analysis Patterns
General Analysis Patterns

Software Diagnostics Institute[36] innovation is in devising and cataloging general problem and analysis patterns and providing some concrete analysis implementations on specific OS platforms such as Windows and Mac OS X.

---

[36] http://www.DumpAnalysis.org + http://www.TraceAnalysis.org

I Didn’t See Anything

How often do you hear that back from support departments when you submit your memory dumps and software logs? “Analysis inconclusive”, “logs or crash dumps are not good”, “I need more”, “liaise with another vendor”, and many others hide the same response behind the elaborate narrative façade. Based on the audit of memory dump analysis reports submitted to Software Diagnostics Services (former Memory Dump Analysis Services) by its customers over the course of the last few years I think such responses usually result from support engineers not utilizing the proper software diagnostics methodology, for example, using only what they remembered from their own past diagnostics, troubleshooting, and debugging practice. What is a solution to this problem?

Software Diagnostics Institute (DumpAnalysis.org + TraceAnalysis.org) has been collecting analysis patterns for the past 8 years in cooperation with Software Diagnostics Services providing research funds and software execution artifacts. Patterns are organized into pattern catalogs, and checklists are recommended. The approach is called Pattern-Oriented Diagnostics which has 3 parts: pattern-driven, systemic, and pattern-based. Pattern-driven means that you go through the list of patterns and report ones you found and not found. This may be done iteratively. Systemic part means you can apply the same general patterns across different software execution artifacts, products, and operating systems. Pattern-based means you iteratively extend and improve your pattern catalogs and use Pattern-View-Controller diagnostics architecture.

Please find the following presentations for each part:

- Pattern-Driven: http://www.patterndiagnostics.com/Introduction-Software-Diagnostics-materials
- Systemic: http://www.patterndiagnostics.com/systemic-diagnostics-materials
- Pattern-Based: http://www.patterndiagnostics.com/pattern-based-diagnostics-materials
- Pattern-View-Controller: http://www.dumpanalysis.org/patterns-view-controller

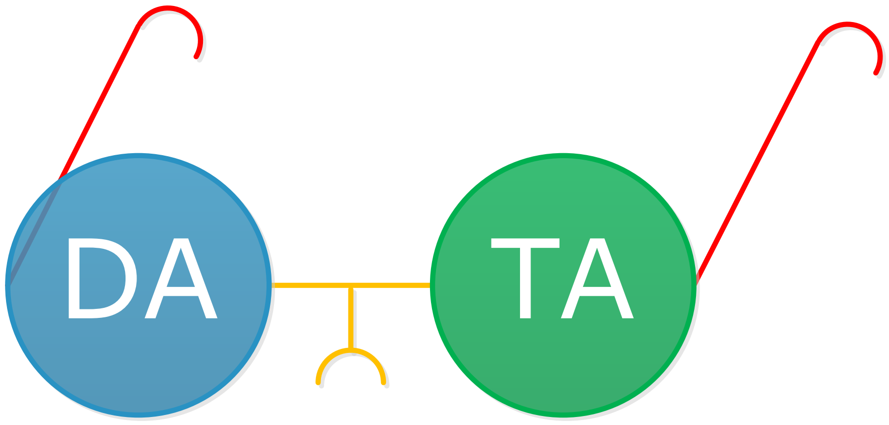
This is a very flexible approach already applied to malware detection and analysis, digital forensics, debugging, and network trace analysis. You can find more here:

http://www.patterndiagnostics.com/training-materials

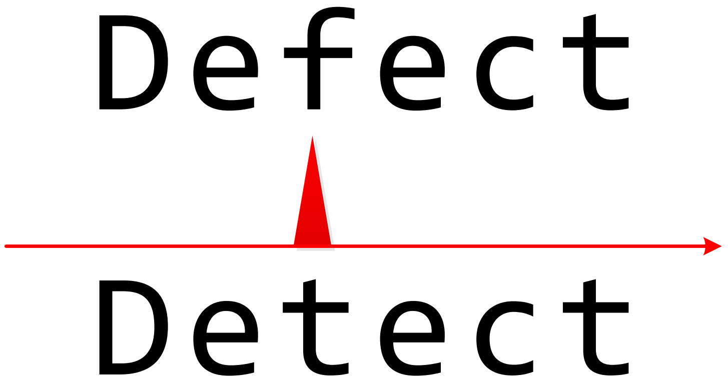
So the next time you hear a similar response from an engineer ask to provide the list of patterns not found in your memory dumps, traces, and logs.

PART 9: Art and Photography

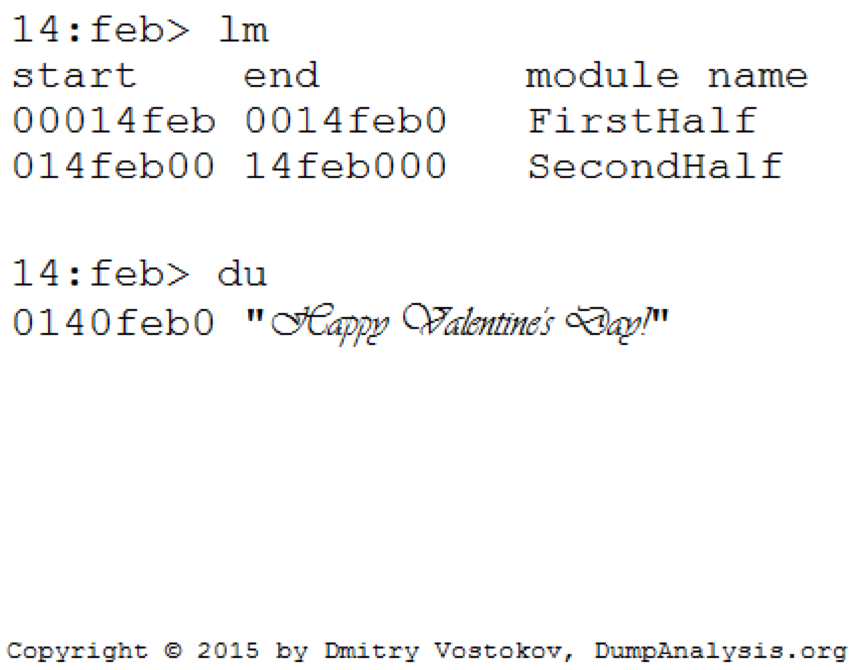
Diagnostics Designer Glasses



Pattern Diagnostics Logo

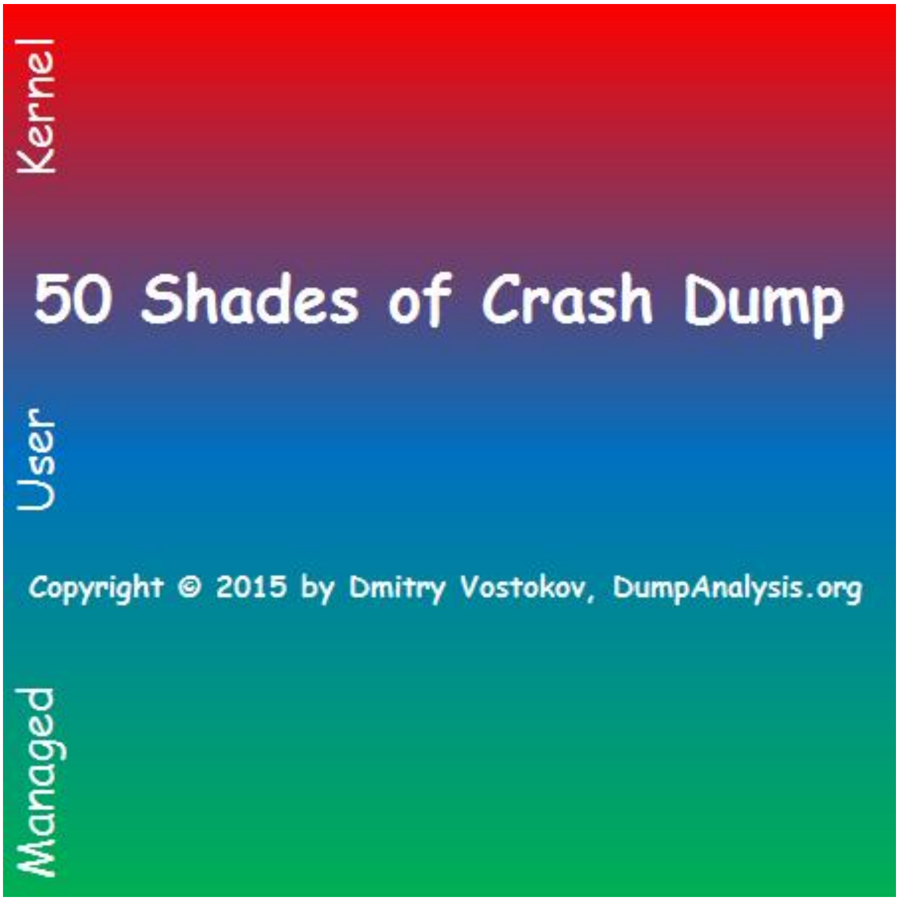


Happy Valentine's Day

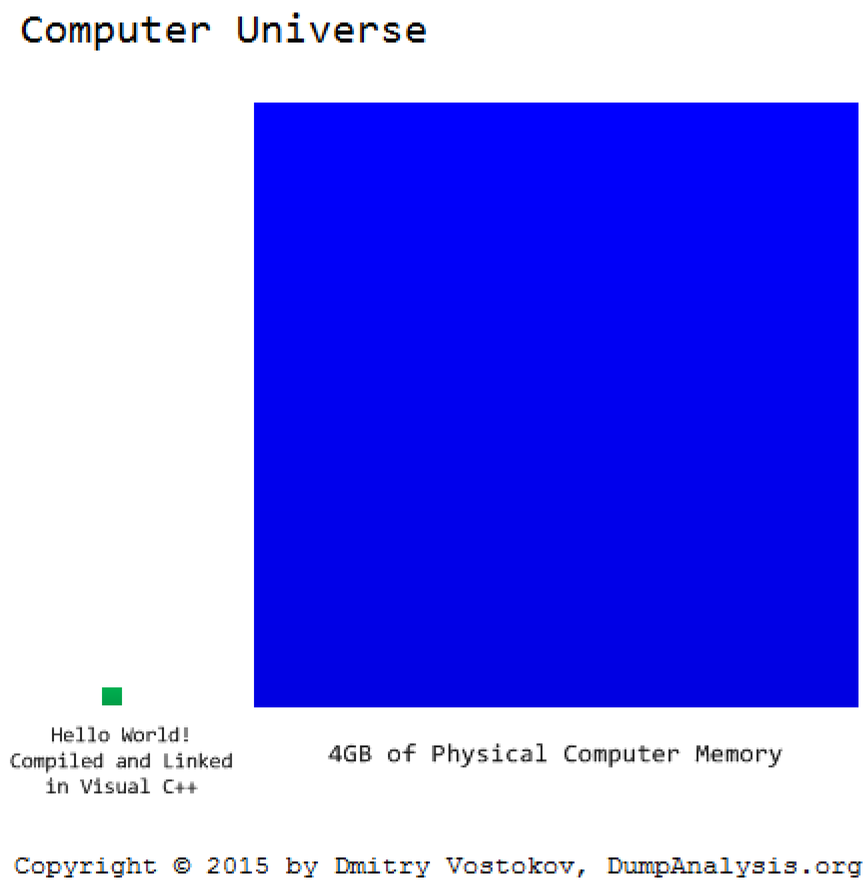


```
14:feb> lm
start    end         module name
00014feb 0014feb0    FirstHalf
014feb00 14feb000    SecondHalf

14:feb> du
0140feb0 "Happy Valentine's Day!"
```

50 Shades of Crash Dump

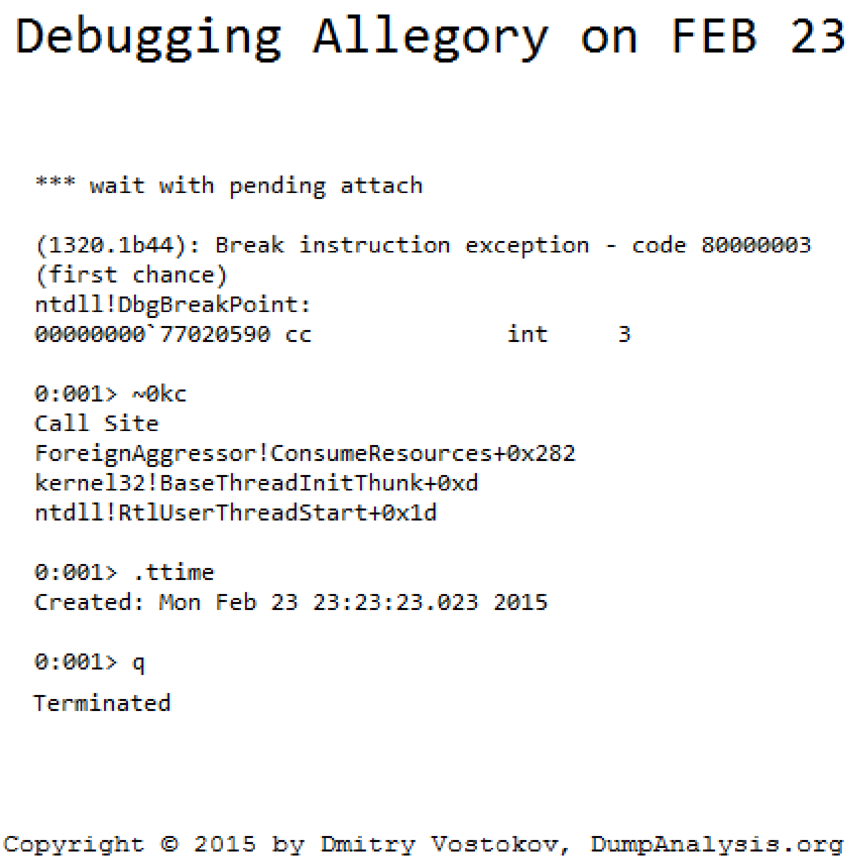


Computer Universe


Computer Universe
Hello World! Compiled and Linked in Visual C++
4GB of Physical Computer Memory
Copyright © 2015 by Dmitry Vostokov, DumpAnalysis.org

Failed Surveillance

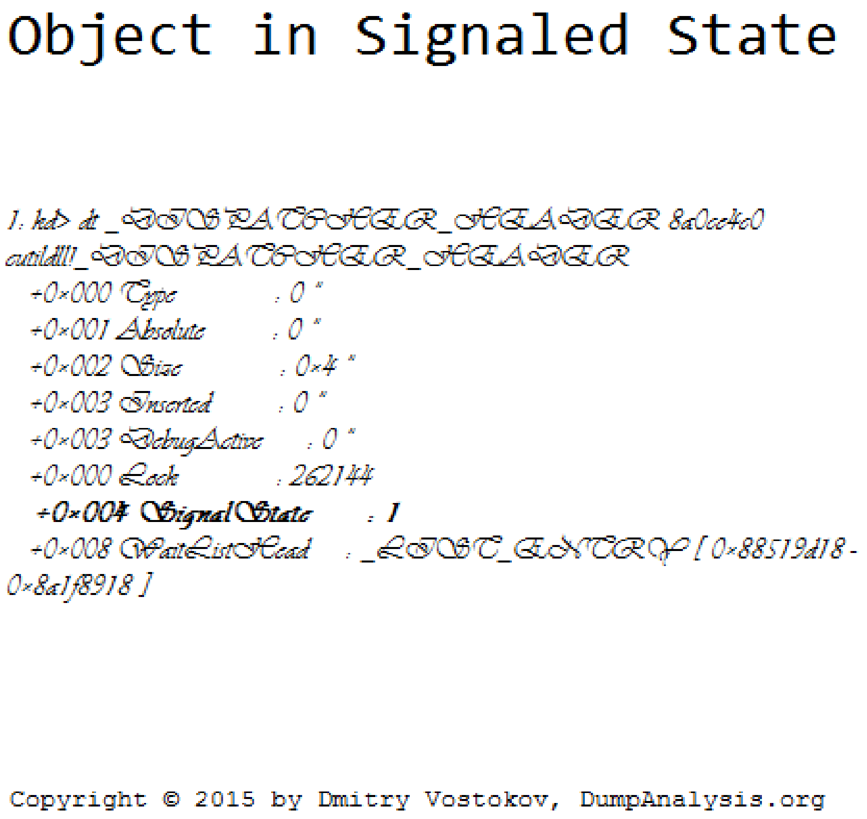
Debugging Allegory on FEB 23



Debugging Allegory on FEB 23

```
*** wait with pending attach

(1320.1b44): Break instruction exception - code 80000003
(first chance)
ntdll!DbgBreakPoint:
00000000`77020590 cc              int     3

0:001> ~0kc
Call Site
ForeignAggressor!ConsumeResources+0x282
kernel32!BaseThreadInitThunk+0xd
ntdll!RtlUserThreadStart+0x1d

0:001> .ttime
Created: Mon Feb 23 23:23:23.023 2015

0:001> q
Terminated
```

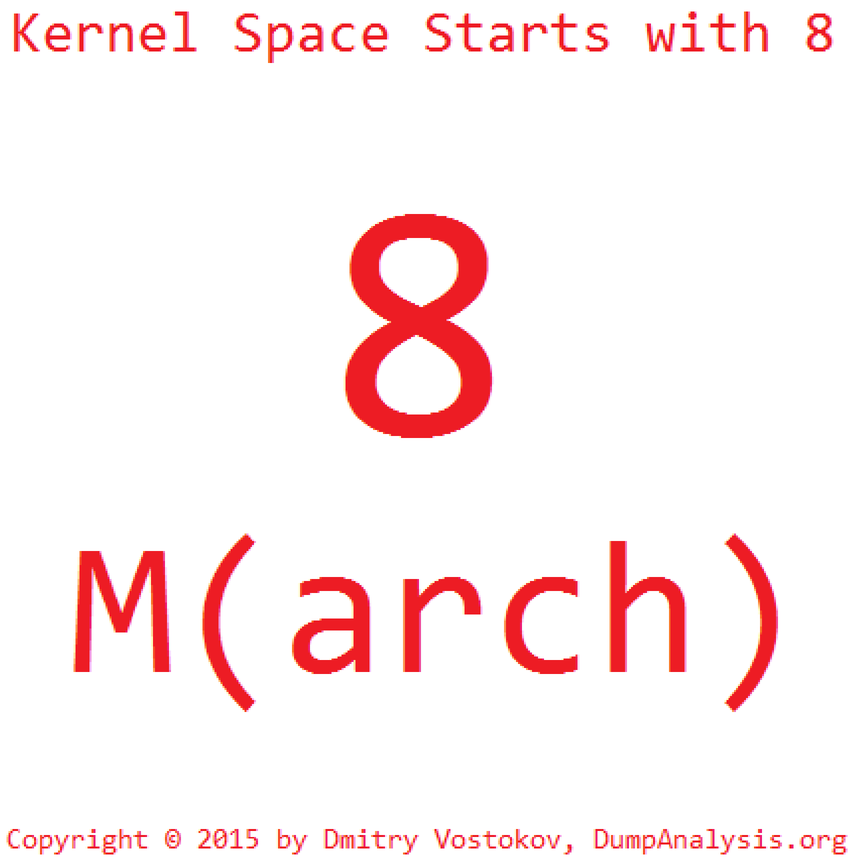
Object in Signaled State



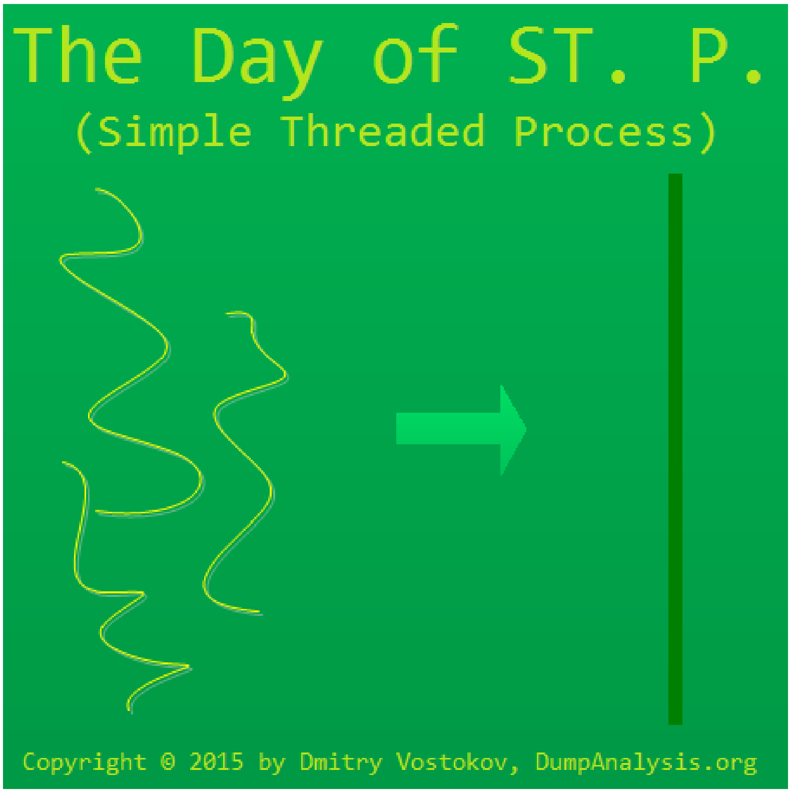
Object in Signaled State

```
1: kd> dt _DISPATCHER_HEADER 8a0ce4c0
ntdll!_DISPATCHER_HEADER
   +0x000 Type             : 0 ''
   +0x001 Absolute         : 0 ''
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0 ''
   +0x003 DebugActive      : 0 ''
   +0x000 Lock             : 262144
   +0x004 SignalState      : 1
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x88519d18 - 0x8a1f8918 ]
```

Kernel Space Starts with 8

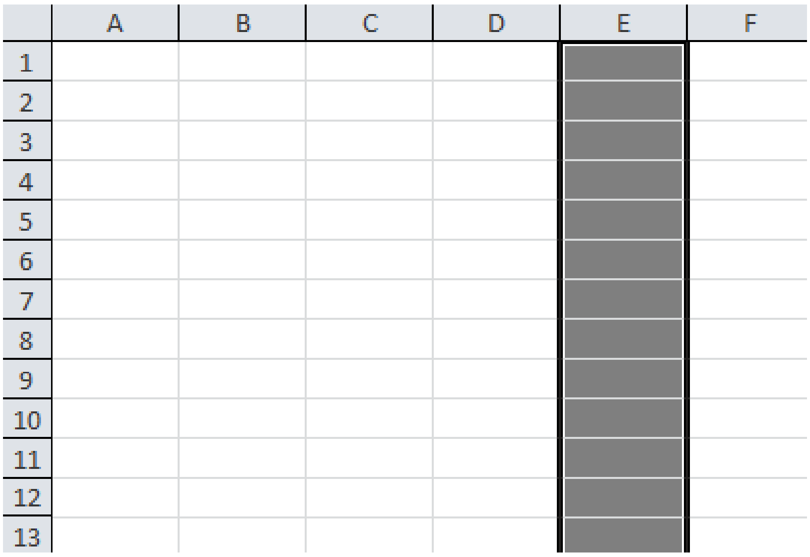

Kernel Space Starts with 8
8
M(arch)
Copyright © 2015 by Dmitry Vostokov, DumpAnalysis.org

The Day of ST. P. The Elimination of Snakes

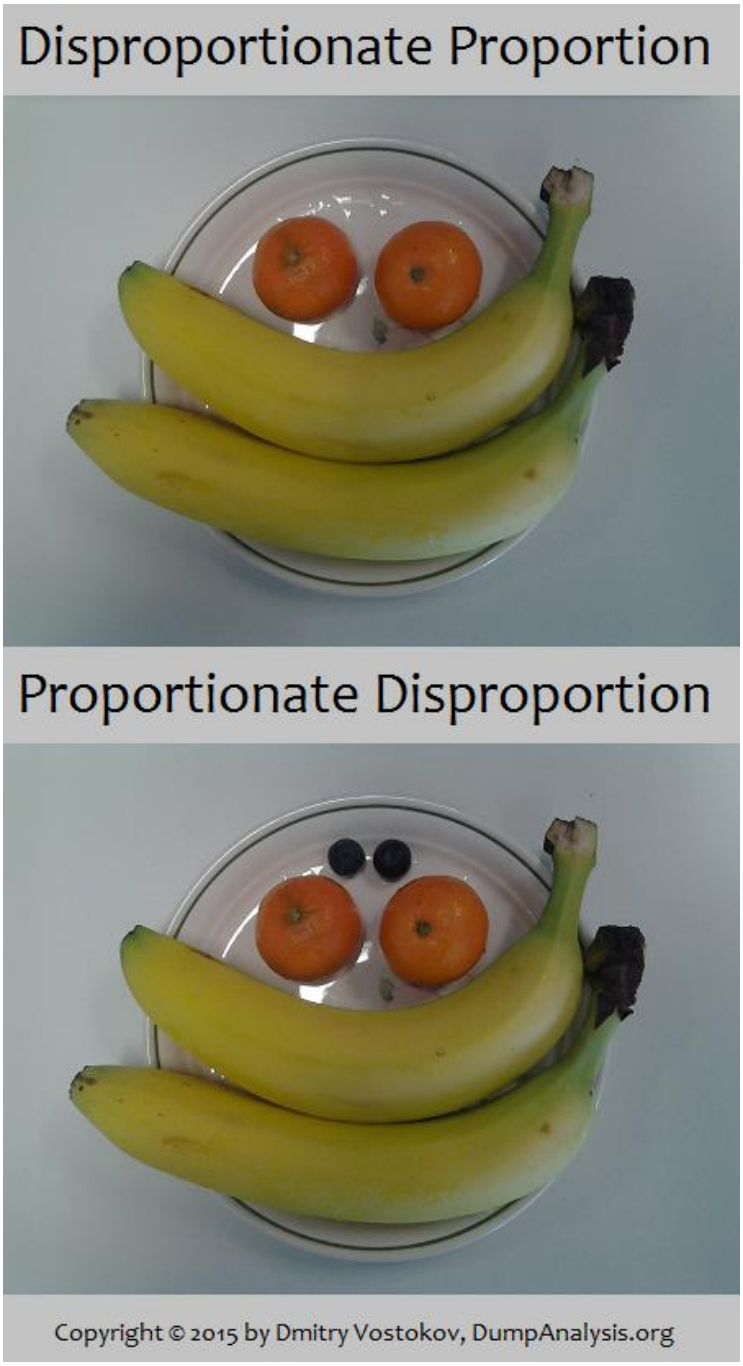

The Day of ST. P.
(Simple Threaded Process)
Copyright © 2015 by Dmitry Vostokov, DumpAnalysis.org

The Fifth Column

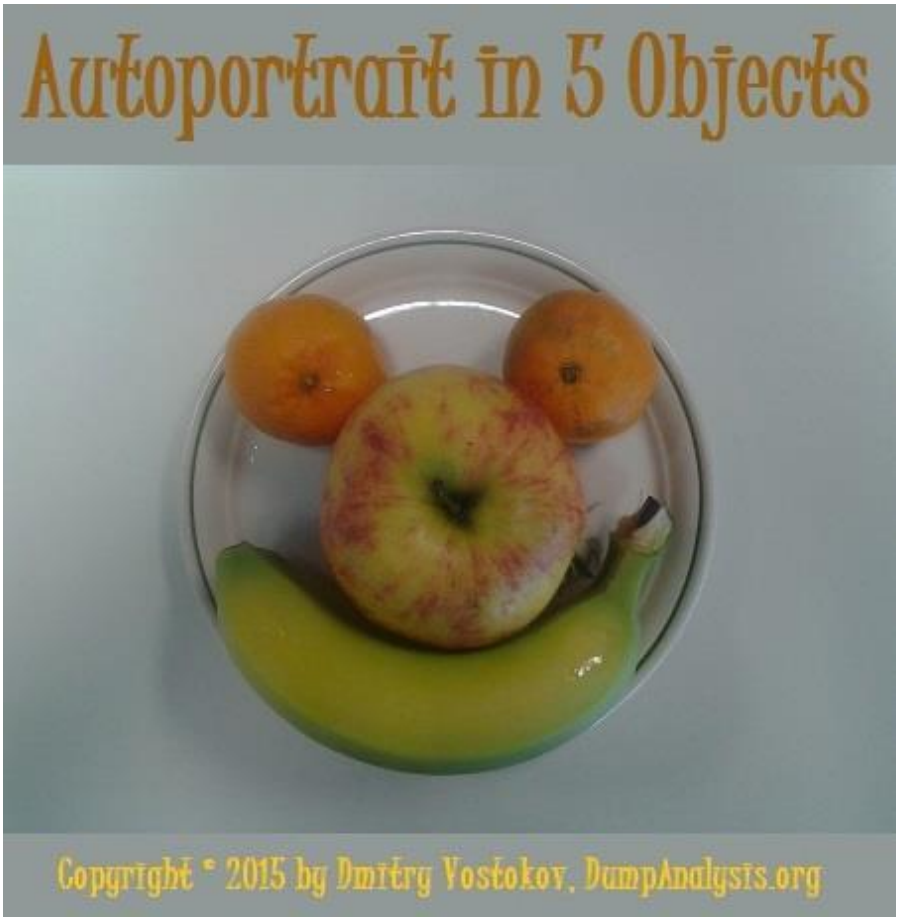
The Fifth Column



Proportionate Disproportionate Proportion

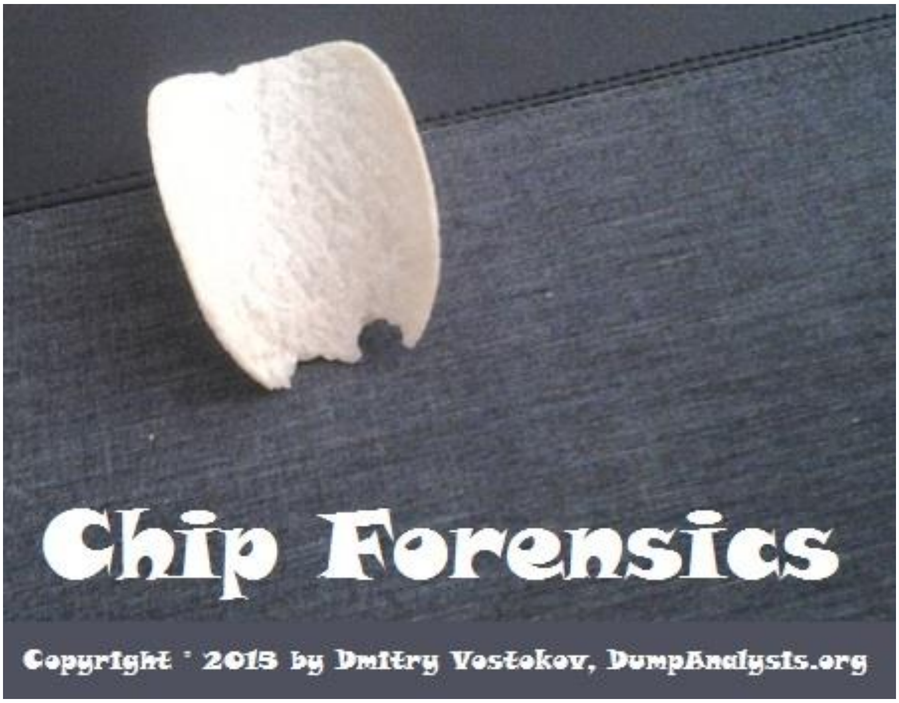

Disproportionate Proportion
Proportionate Disproportion
Copyright © 2015 by Dmitry Vostokov, DumpAnalysis.org

Autoportrait in 5 Objects


Autoportrait in 5 Objects
Copyright © 2015 by Dmitry Vostokov, DumpAnalysis.org

Kernel Works

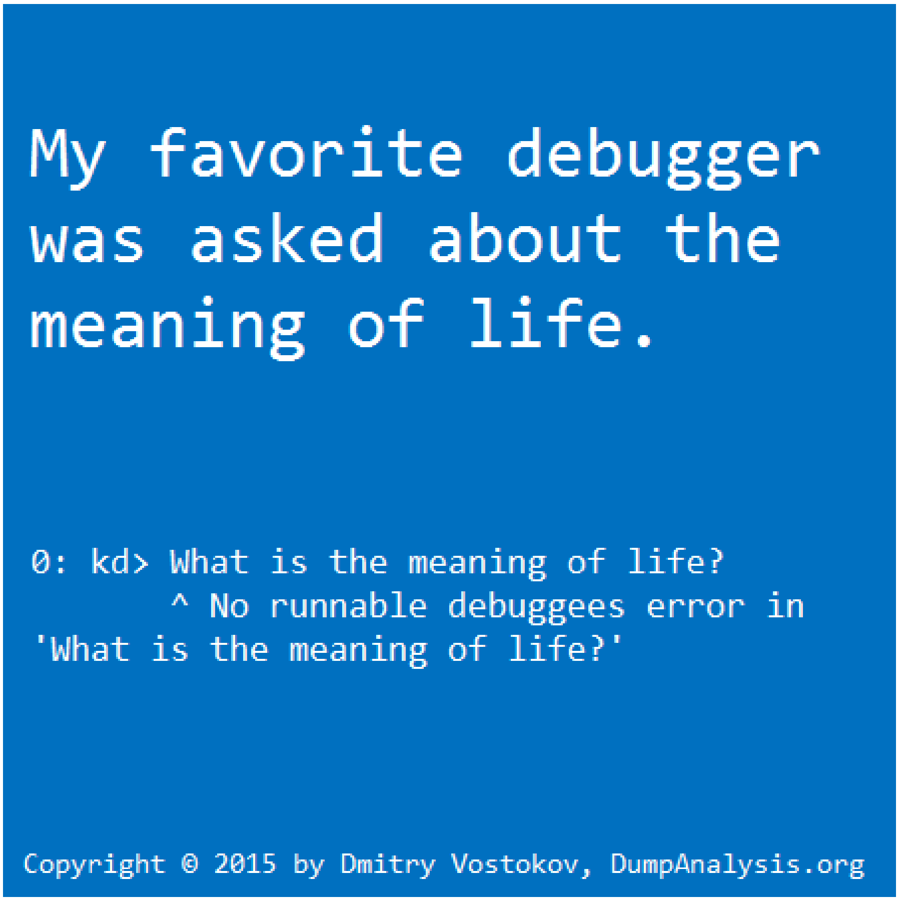
Chip Forensics



Industrial Windows

Industrial Windows
Copyright © 2015
Dmitry Vostokov
DumpAnalysis.org

The Meaning of Life



My favorite debugger was asked about the meaning of life.

```
0: kd> What is the meaning of life?
       ^ No runnable debuggees error in
'What is the meaning of life?'
```

Hidden Bug

Hidden Bug
Copyright © 2015 by Dmitry Vostokov, DumpAnalysis.org

PART 10: Memory Forensics

Artifact-Malware and its Primary and Secondary Effects

Once we saw an article in Facebook stream about trolling airline passengers. When they descend to an airport, they read a different city name written in large letters on the roof of some house.

An idea came to us to model this behavior for memory dump analysis: when we analyze crash dumps we usually rely on the output of some commands that redirect or reformat the contents of memory. For example, **lmv** WinDbg command shows module resource information such as its product name, copyright information, etc. What if that information were deliberately crafted to deceive and disturb software diagnostics and debugging process, and ultimately to explore possible vulnerabilities there? Popular debuggers have their own vulnerabilities[37] which may be used not only for anti-debugging purposes. When we say "deliberately crafted" we don't mean **Fake Module** (Volume 7, page 240) malware analysis pattern that is about a module that tries to present itself as another legitimate, well-known module. Also, we are not concerned with false positive decoy artifacts[38]. In our case **Artifact-Malware**, as we call it (or **Arti-Malware** for short, not to confuse with anti-malware), intentionally leaves malicious legitimate artifacts in software execution artifacts (such as memory dumps, traces, and logs) deliberately structured to alter execution of static analysis tools such as debuggers, disassemblers, reversing tools, etc. Such artifacts in artifacts may suggest exploring them further as possible culprits of abnormal software behavior thus triggering certain software and human vulnerabilities, and even social engineering attacks (when they suggest calling a phone number).

---

[37] M. Sikorski, A. Honig, Practical Malware Analysis, Debugger Vulnerabilities, page 363

[38] A. Walters, N. Petroni, Jr., Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process (http://www.blackhat.com/presentations/bh-dc-07/Walters/Paper/bh-dc-07-Walters-WP.pdf)

To model this, we quickly created a small Visual C++ project called TrollingApp and inserted version info resource. Normally WinDbg **lmv** command would show something like this:

```
0:000> lmv m TrollingModule
start             end                 module name
00000001`3ff50000 00000001`3ff58000   TrollingModule C (private pdb
symbols)  C:\Work\TrollingApp\x64\Release\TrollingModule.pdb
    Loaded symbol image file: TrollingModule.exe
    Image path: C:\Work\TrollingApp\x64\Release\TrollingModule.exe
    Image name: TrollingModule.exe
    Timestamp:        Sat Jun 27 10:28:47 2015 (558E6CCF)
    CheckSum:         00000000
    ImageSize:        00008000
    File version:     1.0.0.1
    Product version:  1.0.0.1
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        1.0 App
    File date:        00000000.00000000
    Translations:     1809.04b0
    CompanyName:      TODO: <Company name>
    ProductName:      TODO: <Product name>
    InternalName:     TrollingModule.exe
    OriginalFilename: TrollingModule.exe
    ProductVersion:   1.0.0.1
    FileVersion:      1.0.0.1
    FileDescription:  TODO: <File description>
    LegalCopyright:   Copyright © 2015 by Software Diagnostics
Institute
```

Since *LegalCopyright* is the last field shown in the formatted output, we changed it to contain the long string of "\r\n" characters intended to scroll away module information. The string was long as it was allowed by the resource compiler.

```
VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
 FILEFLAGS 0x0L
 FILEOS 0x40004L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "180904b0"
        BEGIN
            VALUE "CompanyName", "TODO: <Company name>"
            VALUE "FileDescription", "TODO: <File description>"
            VALUE "FileVersion", "1.0.0.1"
            VALUE "InternalName", "TrollingModule.exe"
            VALUE "LegalCopyright", "\r\n\r\n\r\n ... "
            VALUE "OriginalFilename", "TrollingModule.exe"
            VALUE "ProductName", "TODO: <Product name>"
            VALUE "ProductVersion", "1.0.0.1"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x1809, 1200
    END
END
```

The program itself is very simple triggering a NULL pointer exception to generate a crash dump (we configured *LocalDumps* registry key on Windows 7).

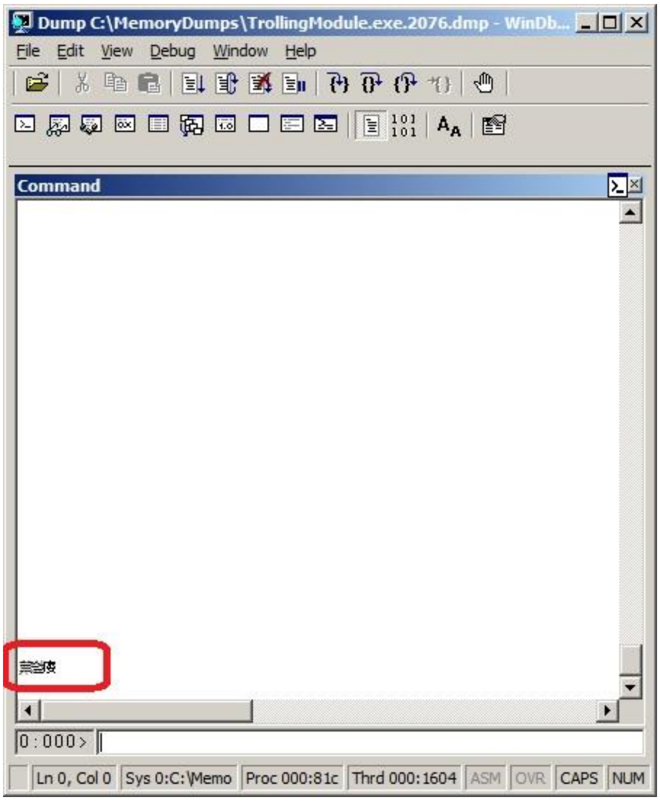
```
int _tmain(int argc, _TCHAR* argv[])
{
       int *p = 0;

       *p = 0;
       return 0;
}
```

So we opened a crash dump and checked the stack trace which suggested checking information about *TrollingModule* (as **Exception Module** memory analysis pattern, Volume 8a, page 80):

```
Loading Dump File [C:\MemoryDumps\TrollingModule.exe.2076.dmp]
User Mini Dump File with Full Memory: Only application data is available

Windows 7 Version 7601 (Service Pack 1) MP (4 procs) Free x64
Product: WinNt, suite: SingleUserTS Personal
Machine Name:
Debug session time: Sat Jun 27 10:28:58.000 2015 (UTC + 1:00)
System Uptime: 3 days 21:28:51.750
Process Uptime: 0 days 0:00:01.000
.....
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(81c.1604): Access violation - code c0000005 (first/second chance not
available)
ntdll!NtWaitForMultipleObjects+0xa:
00000000`7769186a c3              ret
0:000> .symfix c:\mss

0:000> .reload
.....
```

```
0:000> kL
Child-SP          RetAddr           Call Site
00000000`001fe6d8 000007fe`fd741430 ntdll!NtWaitForMultipleObjects+0xa
00000000`001fe6e0 00000000`77541723 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`001fe7e0 00000000`775bb5e5
kernel32!WaitForMultipleObjectsExImplementation+0xb3
00000000`001fe870 00000000`775bb767 kernel32!WerpReportFaultInternal+0x215
00000000`001fe910 00000000`775bb7bf kernel32!WerpReportFault+0x77
00000000`001fe940 00000000`775bb9dc kernel32!BasepReportFault+0x1f
00000000`001fe970 00000000`776d3398 kernel32!UnhandledExceptionFilter+0x1fc
00000000`001fea50 00000000`776585c8 ntdll! ?? ::FNODOBFM::`string'+0x2365
00000000`001fea80 00000000`77669d2d ntdll!_C_specific_handler+0x8c
00000000`001feaf0 00000000`776591cf ntdll!RtlpExecuteHandlerForException+0xd
00000000`001feb20 00000000`77691248 ntdll!RtlDispatchException+0x45a
00000000`001ff200 00000001`3ff51002 ntdll!KiUserExceptionDispatch+0x2e
00000000`001ff908 00000001`3ff51283 TrollingModule!wmain+0x2
00000000`001ff910 00000000`775359ed TrollingModule!__tmainCRTStartup+0x10f
00000000`001ff940 00000000`7766c541 kernel32!BaseThreadInitThunk+0xd
00000000`001ff970 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

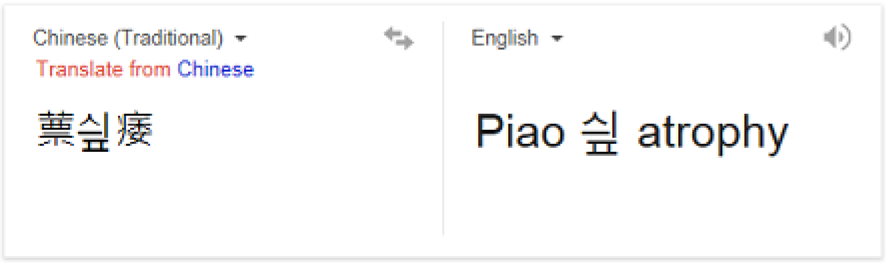
But when we executed **lmv** command we saw the blank screen with some UNICODE symbols at the end:


Dump C:\MemoryDumps\TrollingModule.exe.2076.dmp - WinDb...
File Edit View Debug Window Help
Command
0:000>
Ln 0, Col 0 | Sys 0:C:\Memo | Proc 000:81c | Thrd 000:1604 | ASM | OVR | CAPS | NUM

```
嵩娑
```

Not only we triggered the scroll but the artifact buffer somehow caused additional unintended consequences.

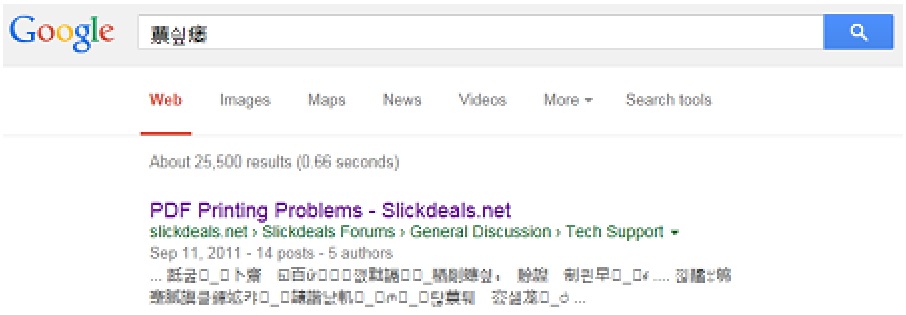
We were also surprised by the second order effects. We were curious about what that Unicode string was meant and copied it to Google translate page in IE. It was hanging afterward spiking CPU when we were switching to that tab. We tried to save a crash dump using Task Manager, but it failed with a message about an error in *ReadProcessMemory* API and, although, the crash dump was saved, it was corrupt. The tab was recovered, and we were not able to reproduce it again. Perhaps, the browser was already in an abnormal state because on the second attempt it behaved better:



蔈싋痿

Piao 싋 atrophy

Simple Google search shows that such output also appeared in different problems such as related to PDF printing:



蔈싋痿

About 25,500 results (0.66 seconds)

PDF Printing Problems - Slickdeals.net
slickdeals.net › Slickdeals Forums › General Discussion › Tech Support
Sep 11, 2011 - 14 posts - 5 authors
... [illegible] ...

In conclusion, we say that the primary effect of arti-malware is abnormal software behavior in static analysis tools. We have the secondary effect when information produced by a static analysis tool triggers abnormal software behavior in another analysis tool.

[This page is intentionally left blank]

PART 11: Miscellaneous

Quotes

These are my selected thoughts I posted previously on Facebook, Twitter, and where appropriate, on LinkedIn.

Strong AI (if it ever exists) does not pose a threat to humanity if designed with defects.

Pattern languages facilitate critical thinking and clear writing in software diagnostics and forensics.

If the software can eat your data, it can vomit it too.

The right question to ask when your program doesn't work is "Please give me a memory dump or trace/log or both". If you can't ask this question, then rethink your diagnostics strategy.

One letter makes a difference: cash dump vs. crash dump.

Windows kernel trivia: RIP results from IRP permutation.

There are numbers you see and instantly feel suspicion.

HIWORD is not 'Hi' word and LOWORD is not 'Lo' word (archaic greeting).

Most if not all of postmodern philosophy can be explained by trace and log analysis patterns.

Occupy Memory Space!

It is better if a process hangs earlier in the long journey. You have a chance to revise your strategy.

A network outage is an opportunity to read a book.

It is important to teach problem-solving skills. However, it is more important to teach the right solving of the correctly identified problems. It is only attainable through the broad diagnostics education.

A product intended to simplify user experience may actually complexify internal workings of the software environment.

A Hegelian approach to truth: being in opposition to the opposition.

A C++ object definition that is also grammatically correct: An obj;

Memory dump analysis is the job of the future.

One of the oldest TCP implementations was over pipes (in prisons).

Diagnostics: the science of focusing on problems, not solutions.

A diagnostics masterpiece is akin to the work of a sculptor. From the mass of possible patterns only a few relevant remain.

Computers value status more than humans do.

When you see a complete memory dump for the first time from the board of your debugger, you feel memory space sickness. But then you adapt.

In the future world without software crashes, hangs, spikes, leaks, and just bugs software engineers will indulge themselves with writing bad code.

Books and Beers have the same pattern: B[vvc]s.

You need a solid software architecture to support bug fixing.

Status Updates

Facebook or LinkedIn status updates are forms of software logs, and we can apply the whole pattern analysis apparatus to them:

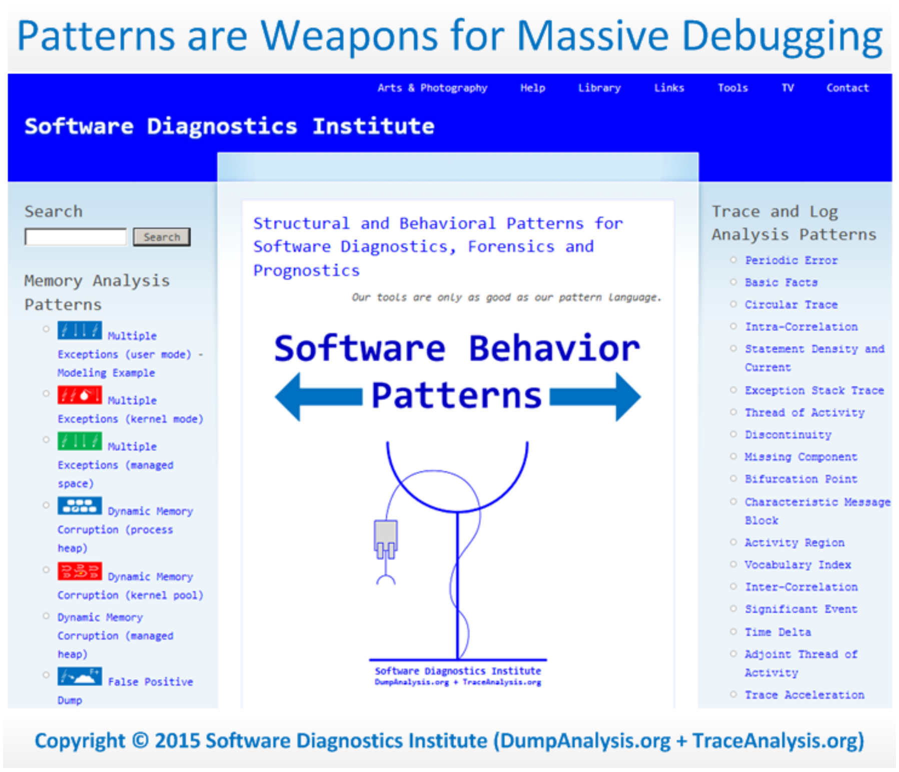
http://www.dumpanalysis.org/trace-log-analysis-pattern-reference

Execution Residue

Debugged! MZ/PE
March, 09
Memory Dump Analysis Anthology, Volume 1
Collector's Edition | Dmitry Vostokov | DumpAnalysis.org
Memory Dump Analysis Anthology Collector's Edition, Volume 1
Encyclopedia of Crash Dump Analysis Patterns
Dmitry Vostokov | Software Diagnostics Institute | DumpAnalysis.org
Windows Debugging Vostokov
x64 Windows Debugging Vostokov
Memory Dump Analysis Anthology, Volume 8a
Memory Dump Analysis Anthology, Volume 7
Memory Dump Analysis Anthology, Volume 6
Memory Dump Analysis Anthology, Volume 5
Memory Dump Analysis Anthology, Volume 4
Memory Dump Analysis Anthology, Volume 3
Memory Dump Analysis Anthology, Volume 2
Memory Dump Analysis Anthology, Volume 1
Pattern-Oriented Memory Forensics
Principles of Memory Dump Analysis
Software Diagnostics: The Collected Seminars
Accelerated .NET Memory Dump Analysis 2nd ed.
Advanced Windows Memory Dump Analysis 2nd ed.
Advanced Windows RT Memory Dump Analysis ARM ed.
Accelerated Windows Software Trace Analysis
Accelerated Disassembly, Reconstruction and Reversing
Accelerated Windows Debugging3
Accelerated Mac OS X Core Dump Analysis, 2nd ed.
Accelerated Windows Malware Analysis
Accelerated Windows Memory Dump Analysis 3rd ed.
Accelerated Mac OS X Core Dump Analysis
Accelerated Windows Memory Dump Analysis 2nd ed.

Appendix

Patterns are Weapons for Massive Debugging



Patterns are Weapons for Massive Debugging

Arts & Photography Help Library Links Tools TV Contact

Software Diagnostics Institute

Search

Search

Memory Analysis Patterns

- Multiple Exceptions (user mode) - Modeling Example
- Multiple Exceptions (kernel mode)
- Multiple Exceptions (managed space)
- Dynamic Memory Corruption (process heap)
- Dynamic Memory Corruption (kernel pool)
- Dynamic Memory Corruption (managed heap)
- False Positive Dump

Structural and Behavioral Patterns for Software Diagnostics, Forensics and Prognostics

Our tools are only as good as our pattern language.

Software Behavior
Patterns
Software Diagnostics Institute
DumpAnalysis.org + TraceAnalysis.org

Trace and Log Analysis Patterns

- Periodic Error
- Basic Facts
- Circular Trace
- Intra-Correlation
- Statement Density and Current
- Exception Stack Trace
- Thread of Activity
- Discontinuity
- Missing Component
- Bifurcation Point
- Characteristic Message Block
- Activity Region
- Vocabulary Index
- Inter-Correlation
- Significant Event
- Time Delta
- Adjoint Thread of Activity
- Trace Acceleration

Copyright © 2015 Software Diagnostics Institute (DumpAnalysis.org + TraceAnalysis.org)

Crash Dump Analysis Checklist

General:

- Symbol servers (*.symfix*)
- Internal database(s) search
- Google or Microsoft search for suspected components as this could be a known issue. Sometimes a simple search immediately points to the fix on a vendor's site
- The tool used to save a dump (to flag false positive, incomplete or inconsistent dumps)
- OS/SP version (*version*)
- Language
- Debug time
- System uptime
- Computer name (*dS srv!srvcomputername* or *!envvar COMPUTERNAME*)
- List of loaded and unloaded modules (*lmv* or *!dlls*)
- Hardware configuration (*!sysinfo*)
- *.kframes 1000*

Application or service:

- Default analysis (*!analyze -v* or *!analyze -v -hang* for hangs)
- Critical sections (*!cs -s -l -o*, *!locks*) for both crashes and hangs
- Component timestamps, duplication, and paths. DLL Hell? (*lmv* and *!dlls*)
- Do any newer components exist?
- Process threads (*~*kv* or *!uniqstack*) for multiple exceptions and blocking functions
- Process uptime
- Your components on the full raw stack of the problem thread
- Your components on the full raw stack of the main application thread
- Process size
- Number of threads
- Gflags value (*!gflag*)
- Time consumed by threads (*!runaway*)
- Environment (*!peb*)
- Import table (*!dh*)
- Hooked functions (*!chkimg*)
- Exception handlers (*!exchain*)

- Computer name (*!envvar COMPUTERNAME*)
- Process heap stats and validation (*!heap -s, !heap -s -v*)
- CLR threads? (*mscorwks* or *clr* modules on stack traces) Yes: use .NET checklist below
- Hidden (unhandled and handled) exceptions on thread raw stacks

System hang:

- Default analysis (*!analyze -v -hang*)
- ERESOURCE contention (*!locks*)
- Processes and virtual memory including session space (*!vm 4*)
- Important services are present and not hanging (for example, terminal or IMA services for Citrix environments)
- Pools (*!poolused*)
- Waiting threads (*!stacks*)
- Critical system queues (*!exqueue f*)
- I/O (*!irpfind*)
- The list of all thread stack traces (*!process 0 3f*)
- LPC/ALPC chain for suspected threads (*!lpc message* or *!alpc /m* after search for "*Waiting for reply to LPC*" or "*Waiting for reply to ALPC*" in *!process 0 3f* output)
- Mutants (search for "*Mutants - owning thread*" in *!process 0 3f* output)
- Critical sections for suspected processes (*!cs -l -o -s*)
- Sessions, session processes (*!session*, *!sprocess*)
- Processes (size, handle table size) (*!process 0 0*)
- Running threads (*!running*)
- Ready threads (*!ready*)
- DPC queues (*!dpcs*)
- The list of APCs (*!apc*)
- Internal queued spinlocks (*!qlocks*)
- Computer name (*dS srv!srvcomputername*)
- File cache, VACB (*!filecache*)
- File objects for blocked thread IRPs (*!irp* -> *!fileobj*)
- Network (*!ndiskd.miniports* and *!ndiskd.pktpools*)
- Disk (*!scsikd.classext* -> *!scsikd.classext class_device 2*)
- Modules rdbss, mrxdav, mup, mrxsmb in stack traces
- Functions Ntfs!Ntfs* and nt!Fs* in stack traces

BSOD:

- Default analysis (*!analyze -v*)
- Pool address (*!pool*)
- Component timestamps (*lmv*)
- Processes and virtual memory (*!vm 4*)
- Current threads on other processors
- Raw stack
- Bugcheck description (including ln exception address for corrupt or truncated dumps)
- Bugcheck callback data (*!bugdump* for systems prior to Windows XP SP1)
- Bugcheck secondary callback data (*.enumtag*)
- Computer name (*dS srv!srvcomputername*)
- Hardware configuration (*!sysinfo*)

.NET application or service:

- CLR module and SOS extension versions (*lmv* and *.chain*)
- Managed exceptions (*~*e !pe*)
- Nested managed exceptions (*!pe -nested*)
- Managed threads (*!Threads -special*)
- Managed stack traces (*~*e !CLRStack*)
- Managed execution residue (*~*e !DumpStackObjects* and *!DumpRuntimeTypes*)
- Managed heap (*!VerifyHeap*, *!DumpHeap -stat* and *!eeheap -gc*)
- GC handles (*!GCHandles*, *!GCHandleLeaks*)
- Finalizer queue (*!FinalizeQueue*)
- Sync blocks (*!syncblk*)

Index of WinDbg Commands