OSR
open
systems
resources
inc.

[CLIENT]

Crash Dump Analysis Report

The following report offers insight and root-cause analysis for a specific, client-provided crash scenario.

OSR Open Systems Resources, Inc.
105 Route 101A Suite 19
Amherst, New Hampshire 03031
+1 (603) 595-6500

Printed in the United States of America

Document Identifier: UL219

LIMITED WARRANTY

OSR Open Systems Resources, Inc. (OSR) expressly disclaims any warranty for the information presented herein. This material is presented “as is” without warranty of any kind, either express or implied, including, without limitation, the implied warranties of merchantability or fitness for a particular purpose. The entire risk arising from the use of this material remains with you. OSR’s entire liability and your exclusive remedy shall not exceed the price paid for this material. In no event shall OSR or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profit, business interruption, loss of business information, or any other pecuniary loss) arising out of the use or inability to use this information, even if OSR has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

U.S. GOVERNMENT RESTRICTED RIGHTS

This material is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Right in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software--Restricted Rights 48 CFR 52.227-19, as applicable. Manufacturer is OSR Open Systems Resources, Inc. Amherst, New Hampshire 03031.

Table of Contents

PROBLEM REPORT

<Client> has indicated to OSR that they are experiencing sporadic crashes when running their software on a particular Windows 2000 installation. The crash so far has been 100% reproducible while performing stress testing, though the crash location is not always the same.

Initial triage by *<Client>'s* engineers has been performed and they believe the issue to be a possible memory corruption.

ANALYSIS RESULTS

After our analysis, we do not believe this to be a memory corruption issue. Instead, we believe that the *<CLIENT DRIVER>* driver is allocating dispatcher objects out of paged pool. Dispatcher Objects are wait locks (such as events, semaphores, and mutexes) that are used for synchronization. Because dispatcher objects are manipulated by the dispatcher at interrupt request levels (IRQLs) greater than or equal to DISPATCH_LEVEL, all dispatcher objects must be non-pageable.

This also explains why the problem was not reproducible across all machines tested. This type of crash will only appear if the pageable memory is actually paged out at the time of access. In the interest of performance, the Memory Manager will only page out pageable memory when memory pressure warrants. It could be that the stress testing being performed on this platform (which had limited memory) resulted in greater than usual memory pressure, flushing out the bug.

ANALYSIS DETAILS

In order to understand the environment, we have identified the crash to be from a Windows 2000 Service Pack 4 system. The system is running with a single processor and 256MB of RAM. A scan of the loaded module list indicates that there are several third party drivers present, including drivers other than those supplied by *<CLIENT>*.

Analysis always begins with the *!analyze -v* instruction, which performs an automated analysis of the crash:

```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

IRQL_NOT_LESS_OR_EQUAL (a)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If a kernel debugger is available get the stack backtrace.
Arguments:
Arg1: e1d1d800, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, bitfield :
       bit 0 : value 0 = read operation, 1 = write operation
```



```
      bit 3 : value 0 = not an execute operation, 1 = execute operation (only on
chips which support this level of status)
Arg4: 80431079, address which referenced memory

Debugging Details:
------------------

WRITE_ADDRESS:  e1d1d800 Paged pool

CURRENT_IRQL:  2

FAULTING_IP:
nt!KiUnwaitThread+d
80431079 8916            mov     dword ptr [esi],edx

DEFAULT_BUCKET_ID:  DRIVER_FAULT

BUGCHECK_STR:  0xA

PROCESS_NAME:  Idle

TRAP_FRAME:  804704dc -- (.trap 0xffffffff804704dc)
ErrCode = 00000002
eax=ff983e0c ebx=ff983da0 ecx=ff983da0 edx=e1d1d800 esi=e1d1d800 edi=ff983e90
eip=80431079 esp=80470550 ebp=80470574 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000302
nt!KiUnwaitThread+0xd:
80431079 8916            mov     dword ptr [esi],edx  ds:0023:e1d1d800=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 80431079 to 80466b77

STACK_TEXT:
804704dc 80431079 00000001 bfc750f9 816d91e4 nt!KiTrap0E+0x20b
80470554 804312a5 00000000 804705a4 ff983e88 nt!KiUnwaitThread+0xd
80470574 804306cd 00000051 818a2220 80470680 nt!KiWaitTest+0xdf
80470664 80430658 8046c470 8046c700 ffdff000 nt!KiTimerListExpire+0x6d
80470690 80463247 8047fd60 00000000 00033c50 nt!KiTimerExpiration+0xb4
804706a4 804631e2 0000000e 00000000 00000000 nt!KiRetireDpcList+0x30
804706a8 00000000 00000000 00000000 00000000 nt!KiIdleLoop+0x26


STACK_COMMAND:  kb

FOLLOWUP_IP:
nt!KiUnwaitThread+d
80431079 8916            mov     dword ptr [esi],edx

SYMBOL_STACK_INDEX:  1

SYMBOL_NAME:  nt!KiUnwaitThread+d

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: nt

IMAGE_NAME:  ntoskrnl.exe

DEBUG_FLR_IMAGE_TIMESTAMP:  40d1d183

FAILURE_BUCKET_ID:  0xA_nt!KiUnwaitThread+d

BUCKET_ID:  0xA_nt!KiUnwaitThread+d
```

```
Followup: MachineOwner
---------
```

The *!analyze* output indicates that this crash the result of an invalid memory reference. IRQL_NOT_LESS_OR_EQUAL is a common crash that is seen often in the field and can be a result of a violation of the IRQL rules enforced by the system.

If we view the interpreted arguments, we see that someone has attempted to write to address 0xe1d1d800 at IRQL DISPATCH_LEVEL:

```
Arg1: e1d1d800, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, bitfield :
       bit 0 : value 0 = read operation, 1 = write operation
```

We know that IRQL value 2 is DISPATCH_LEVEL from WDM.H:

```
#define DISPATCH_LEVEL 2            // Dispatcher level
```

From the *Debugging Details* output we also see that this address is a paged pool address, meaning that the data buffer being accessed was allocated using ExAllocatePoolWithTag specifying the paged pool value as the *PoolType*:

```
Debugging Details:
------------------

WRITE_ADDRESS:  e1d1d800 Paged pool
```

From those pieces of the *!analyze* information alone, we have the reason for the crash. An attempt was made to access pageable memory at DISPATCH_LEVEL, which will lead to a crash sooner or later as it is a violation of the IRQL rules. We continue the analysis however in the hopes of tracking this issue back to the driver responsible for the bug.

Unfortunately the call stack contains only the operating system and thus there is no obvious driver to blame. Thus, we must perform manual analysis from this stage on.

The first step in the manual analysis is to set the debugger context to that of the trap frame indicated in the *!analyze* output. This trap frame records the register state at the time of the invalid memory reference, allowing us to debug the crashing state:

```
kd> .trap 0xffffffff804704dc
ErrCode = 00000002
eax=ff983e0c ebx=ff983da0 ecx=ff983da0 edx=e1d1d800 esi=e1d1d800 edi=ff983e90
eip=80431079 esp=80470550 ebp=80470574 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000302
nt!KiUnwaitThread+0xd:
80431079 8916            mov     dword ptr [esi],edx  ds:0023:e1d1d800=????????
```

This indicates that the code is treating the value in ESI as a pointer and is trying to overwrite the pointer contents with the contents of the EDX register. Unfortunately the CPU is unable to translate the pointer value that is in ESI, leading to the crash.

The debugger indicates the inability to read the crashing address by displaying question marks. We can also examine the state of the pointer with the *!pte* command:

```
kd> !pte e1d1d800
E1D1D800  - PDE at C0300E1C        PTE at C0387474
          contains 0517E163      contains 06B2C080
        pfn 517e G-DA--KWV       not valid
                                 PageFile    0
                                 Offset 6b2c
                                 Protect:  4
```

Thus, the page is in fact paged out to a paging file and the system is guaranteed to crash on this memory access due to the current IRQL. While this validates our initial analysis, it still does not explain where the memory location came from, or what it was being used for. In order to determine that, we must examine the assembly instructions leading up to the crash to determine where the value in ESI came from.

```
kd> u nt!KiUnwaitThread nt!KiUnwaitThread+0xf
nt!KiUnwaitThread:
8043106c 095150          or      dword ptr [ecx+50h],edx
8043106f 8b4158          mov     eax,dword ptr [ecx+58h]
80431072 53              push    ebx
80431073 56              push    esi
80431074 8b7004          mov     esi,dword ptr [eax+4]
80431077 8b10            mov     edx,dword ptr [eax]
80431079 8916            mov     dword ptr [esi],edx
```

Working backwards from the faulting instruction, we see that ESI came from the contents of EAX plus 0x4:

```
80431074 8b7004          mov     esi,dword ptr [eax+4]
```

Moving back further, we see that EAX came from ECX plus 58 hex:

```
8043106f 8b4158          mov     eax,dword ptr [ecx+58h]
```

Because this function utilizes the ECX register without previously having loaded it with a value, this function can be identified as a *fastcall* function. From this, we can determine that ECX contains the first parameter to the function. It seems reasonable to believe that the first parameter to the routine *KiUnwaitThread* would be a pointer to a KTHREAD structure, which we verified using the *!pool* command:

```
kd> !pool ff983da0
 ff983000 size:   c0 previous size:    0  (Allocated)  ....
 ff9830c0 size:   c0 previous size:   c0  (Free)       ....
 ff983180 size:   c0 previous size:   c0  (Allocated)  ....
 ff983240 size:   c0 previous size:   c0  (Allocated)  ....
```

```
 ff983300 size:   c0 previous size:   c0  (Allocated)  ....
 ff9833c0 size:   c0 previous size:   c0  (Allocated)  ....
 ff983480 size:   c0 previous size:   c0  (Allocated)  ....
 ff983540 size:  260 previous size:   c0  (Allocated)  LScn
 ff9837a0 size:   20 previous size:  260  (Free)       Qota
 ff9837c0 size:  220 previous size:   20  (Allocated)  SIce
 ff9839e0 size:  140 previous size:  220  (Free)       MmCa
 ff983b20 size:   60 previous size:  140  (Allocated)  MmLd
 ff983b80 size:   60 previous size:   60  (Lookaside)  MmCa
 ff983be0 size:   20 previous size:   60  (Free)       Npfs
 ff983c00 size:   c0 previous size:   20  (Allocated)  ....
 ff983cc0 size:   c0 previous size:   c0  (Allocated)  ....
*ff983d80 size:  280 previous size:   c0  (Allocated) *Thre (Protected)
```

Using this information, we can begin digging in to the KTHREAD structure using the above offsets to determine what pageable field is being accessed at DISPATCH_LEVEL.

First, we must view the field at offset 58 hex:

```
kd> dt nt!_kthread ff983da0
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0xff983db0 - 0xff983db0 ]
   +0x018 InitialStack     : 0xbcebb000
   +0x01c StackLimit       : 0xbceb8000
   +0x020 Teb              : (null)
   +0x024 TlsArray         : (null)
   +0x028 KernelStack      : 0xbcebace8
   +0x02c DebugActive      : 0 ''
   +0x02d State            : 0x5 ''
   +0x02e Alerted          : [2]  ""
   +0x030 Iopl             : 0 ''
   +0x031 NpxState         : 0xa ''
   +0x032 Saturation       : 0 ''
   +0x033 Priority         : 9 ''
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : 0x6e
   +0x050 WaitStatus       : 258
   +0x054 WaitIrql         : 0 ''
   +0x055 WaitMode         : 0 ''
   +0x056 WaitNext         : 0 ''
   +0x057 WaitReason       : 0 ''
   +0x058 WaitBlockList    : 0xff983e0c _KWAIT_BLOCK
   +0x05c WaitListEntry    : _LIST_ENTRY [ 0x8128247c - 0xff98407c ]
     …
```

We see here that offset 58 hex contains a pointer to a KWAIT_BLOCK structure. KWAIT_BLOCKs are the structures used by the dispatcher to keep track of all the objects that a particular thread is waiting on.

In order to find the crashing address, we must view this structure and see what field is at offset 4:

```
kd> dt nt!_KWAIT_BLOCK 0xff983e0c
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0xe1d1d800 - 0xe1d1d800 ]
   +0x008 Thread           : 0xff983da0 _KTHREAD
   +0x00c Object           : 0xe1d1d7f8
   +0x010 NextWaitBlock    : 0xff983e54 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
```

And we finally arrive at our crashing ESI value, it is the Blink field of the LIST_ENTRY at the start of the structure. Note also the dispatcher object address, which is in close proximity to the failing address. This indicates that the thread in this wait block structure is waiting on a dispatcher object in paged pool. If we view the thread indicated here with *!thread* we find your driver is the driver performing the wait operation:

```
kd> !thread 0xff983da0
THREAD ff983da0  Cid 8.528  Teb: 00000000  Win32Thread: 00000000 WAIT: (Executive)
KernelMode Alertable
    e1d1d7f8  unable to get Wait object
Not impersonating
Owning Process 818a5020
Wait Start TickCount    211849        Elapsed Ticks: 201
Context Switch Count    110
UserTime                  0:00:00.0000
KernelTime                0:00:00.0000
Start Address pdfs (0xbc37b140)
Stack Init bcebb000 Current bcebace8 Base bcebb000 Limit bceb8000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0 DecrementCount 0

ChildEBP RetAddr  Args to Child
bcebad00 8042b241 00000000 00000000 00000000 nt!KiSwapThread+0xc5
bcebad28 bc37b224 e1d1d7f8 00000000 bcebad00 nt!KeWaitForSingleObject+0x1a1
bcebad48 bc37b325 00000000 00000001 bcebad74 <CLIENT DRIVER>+0x5224
bcebad9c bc37b157 e1d1d7c8 bcebaddc 80453844 <CLIENT DRIVER>+0x5325
bcebada8 80453844 e1d1d7d0 00000000 00000000 <CLIENT DRIVER>+0x5157
bcebaddc 80468022 bc37b140 e1d1d7d0 00000000 nt!PspSystemThreadStartup+0x54
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16
```

This strongly indicates that *<CLIENT DRIVER>* is allocating dispatcher objects out of paged pool.

RECOMMENDATIONS

Our recommendation is to review all existing driver code and ensure that all dispatcher objects are non-pageable. Failure to do so will continue to lead to similar crashes.

Also, this crash makes it clear that this driver has never been tested under Driver Verifier. Driver Verifier would have forced this condition during routine testing, instead of having to wait for a system sufficiently low on memory to force the Memory Manager to begin paging data out to disk. Thus, we would strongly recommend performing a series of stress tests on the driver with Driver Verifier, to help identify this and other similar problems.

FURTHER SERVICES

Based on the analysis of this crash, *<CLIENT>* may be interested in working with OSR to perform any of the following services.

Training

It is unclear from the crash if this is a result of a lack of understanding of the IRQL rules or a simple coding error. If the reason for the crash is unclear in any way, it may be beneficial to *<CLIENT>* to attend OSR's *Writing WDM Kernel Mode Drivers* seminar. This course covers the topic of IRQL in great detail.

In addition to or instead of WDM training, OSR also offers a Kernel Debugging seminar, which covers many of the techniques described in this analysis.

Design/Code Review

These types of issues could also be addressed via a thorough review of the existing driver code. This could also discover any fundamental design issues that are lurking in the code base.