Kernel Stack Overflows

ntdebug 1 Feb 2008 12:53 PM | 5

Hello, this is Omer, and today I would like to talk about a common error that we see in a lot of cases reported to us by customers. It involves drivers taking too much space on the kernel stack that results in a kernel stack overflow, which will then crash the system with one of the following bugchecks:

1. **STOP 0x7F: UNEXPECTED_KERNEL_MODE_TRAP** with Parameter 1 set to EXCEPTION_DOUBLE_FAULT, which is caused by running off the end of a kernel stack.

2. **STOP 0x1E: KMODE_EXCEPTION_NOT_HANDLED, 0x7E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED, or 0x8E: KERNEL_MODE_EXCEPTION_NOT_HANDLED**, with an exception code of STATUS_ACCESS_VIOLATION, which indicates a memory access violation.

3. **STOP 0x2B: PANIC_STACK_SWITCH,** which usually occurs when a kernel-mode driver uses too much stack space.

Kernel Stack Overview

Each thread in the system is allocated with a kernel mode stack. Code running on any kernel-mode thread (whether it is a system thread or a thread created by a driver) uses that thread's kernel-mode stack unless the code is a DPC, in which case it uses the processor's DPC stack on certain platforms. Stack grows negatively. This means that the beginning (bottom) of the stack has a higher address than the end (top) of the stack. For example, let's stay the beginning of your stack is 0x80f1000 and this is where your stack pointer (ESP) is pointing. If you push a DWORD value onto the stack, its address would be 0x80f0ffc. The next DWORD value would be stored at 0x80f0ff8 and so on up to the limit (top) of the allocated stack. The top of the stack is bordered by a guard-page to detect overruns.

The size of the kernel-mode stack varies among different hardware platforms. For example:

· On x86-based platforms, the kernel-mode stack is 12K.

· On x64-based platforms, the kernel-mode stack is 24K. (x64-based platforms include systems with processors using the AMD64 architecture and processors using the Intel EM64T architecture).

· On Itanium-based platforms, the kernel-mode stack is 32K with a 32K backing store. (If the processor runs out of registers from its register file, it uses the backing store to hold the contents of registers until the allocating function returns. This doesn't affect stack allocations directly, but the operating system uses more registers on Itanium-based platforms than on other platforms, which makes relatively more stack available to drivers.)

The stack sizes listed above are hard limits that are imposed by the system, and all drivers need to use space conservatively so that they can coexist.

Exception Overview

So, now that we have discussed the kernel stack, let’s dive into how the double fault actually happens.

When we reach the top of the stack, one more push instruction is going to cause an exception. This could be either a simple *push* instruction, or something along the lines of a *call* instruction which also pushes the return address onto the stack, etc.

The push instruction is going to cause the first exception. This will cause the exception handler to kick in, which will then try to allocate the trap frame and other variables on the stack. This causes the second exception.

This time around, the operating system takes advantage a special x86 structure called the Task State Segment(TSS). The OS stores the state of the registers in the TSS and then stops. The TSS can be accessed via an entry in the global descriptor table, and can be used to debug the memory dump that is created.

The Usual Suspects

Rogue drivers are usually guilty of one or more of the following design flaws:

1. Using the stack liberally. Instead of passing large amounts of data on the stack, driver writers should design functions to accept pointers to data structures. These data structures should be allocated out of system space memory(paged or non-paged pool). If you need to pass large number of parameters from one function to another, then group the parameters into a structure and then pass a pointer to that structure.

2. Calling functions recursively. Heavily nested or recursive functions that are passing large amounts of data on the stack will use too much space and will overflow. Try to design drivers that use a minimal number of recursive calls and nested functions.

Since the size of the stack is much smaller on x86 machines, you will run into these problems with x86 machines more frequently than any other platform.

For a more detailed description, please visit

http://www.microsoft.com/whdc/Driver/tips/KMstack.mspx.

Debugging Kernel Stack Overflows

Full kernel dumps are usually enough to find the offending driver. The most common bugcheck code that appears in these dumps is UNEXPECTED_KERNEL_MODE_TRAP (0x7f), with the first argument being EXCEPTION_DOUBLE_FAULT (0x8).

When you get this dump, the first command that you should run is *!analyze-v*.

```
0: kd> !analyze -v

*******************************************************************************

* Bugcheck Analysis *

*******************************************************************************

UNEXPECTED_KERNEL_MODE_TRAP (7f)

This means a trap occurred in kernel mode, and it's a trap of a kind that the kernel isn't allowed to have/catch (bound trap) or that

is always instant death (double fault). The first number in the bugcheck params is the number of the trap (8 = double fault, etc)

Consult an Intel x86 family manual to learn more about what these traps are. Here is a *portion* of those codes:

If kv shows a taskGate

use .tss on the part before the colon, then kv.

Else if kv shows a trapframe

use .trap on that value

Else

.trap on the appropriate frame will show where the trap was taken(on x86, this will be the ebp that goes with the procedure KiTrap)

Endif

kb will then show the corrected stack.

Arguments:

Arg1: 00000008, EXCEPTION_DOUBLE_FAULT

Arg2: 80042000

Arg3: 00000000

Arg4: 00000000

Debugging Details:

------------------

BUGCHECK_STR: 0x7f_8

TSS: 00000028 -- (.tss 0x28)

eax=87b90328 ebx=87b90328 ecx=8aa3d8c0 edx=87b90328 esi=b8cb7138 edi=8084266a

eip=f7159c53 esp=b8cb7000 ebp=b8cb7010 iopl=0 nv up ei pl nz na po nc

cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010202

Ntfs!NtfsInitializeIrpContext+0xc:

f7159c53 57 push edi

Resetting default scope

DEFAULT_BUCKET_ID: DRIVER_FAULT

PROCESS_NAME: System

CURRENT_IRQL: 1

TRAP_FRAME: b8cb8620 -- (.trap 0xffffffffb8cb8620)

ErrCode = 00000000

eax=c1587000 ebx=0000000e ecx=0000000f edx=00000000 esi=87dca350 edi=00000000

eip=8093837b esp=b8cb8694 ebp=b8cb86d0 iopl=0 nv up ei ng nz ac po cy

cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010293

nt!CcMapData+0x8c:

8093837b 8a10 mov dl,byte ptr [eax] ds:0023:c1587000=??
```

```
Resetting default scope

LAST_CONTROL_TRANSFER: from f7158867 to f7159c53
```

Let's follow the instructions that the debugger is giving us. Since the debugger gave us a *.tss* command, lets run that. After that, run a *!thread* to get the thread summary:

```
0: kd> .tss 0x28

eax=87b90328 ebx=87b90328 ecx=8aa3d8c0 edx=87b90328 esi=b8cb7138 edi=8084266a

eip=f7159c53 esp=b8cb7000 ebp=b8cb7010 iopl=0 nv up ei pl nz na po nc

cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010202

Ntfs!NtfsInitializeIrpContext+0xc:

f7159c53 57 push edi

0: kd> !thread

THREAD 87dca350 Cid 0420.0990 Teb: 7ffdf000 Win32Thread: efdbe430 RUNNING on processor 0

IRP List:

89cba088: (0006,01fc) Flags: 00000404 Mdl: 00000000

Not impersonating

DeviceMap e10008d8

Owning Process 8ab8e238 Image: System

Wait Start TickCount 7260638 Ticks: 0

Context Switch Count 17 LargeStack

UserTime 00:00:00.000

KernelTime 00:00:00.015

Start Address 0x4a6810ea

Stack Init b8cba000 Current b8cb7c64 Base b8cba000 Limit b8cb7000 Call 0

Priority 14 BasePriority 13 PriorityDecrement 0
```

We are looking for the kernel stack limits(above in red). For this particular stack we see that the stack starts at **b8cba000**, and ends at **b8cb7000**. If you look at the ESP register in the .tss output above, you will see that we have reached the stack limit. The current instruction being attempted is a *push* which overflows the stack and causes the bugcheck.

Now that we have determined we do have a stack overflow, let's find out what caused this, and who the offending driver is.

The first thing that I do is dump the stack. You might need to increase the number of frames displayed to see the whole stack.

```
0: kd> kb

*** Stack trace for last set context - .thread/.cxr resets it

ChildEBP RetAddr

b8cb7010 f7158867 Ntfs!NtfsInitializeIrpContext+0xc

b8cb71bc 8083f9c0 Ntfs!NtfsFsdRead+0xb7

b8cb71d0 f7212c53 nt!IofCallDriver+0x45

b8cb71f8 8083f9c0 fltmgr!FltpDispatch+0x6f

b8cb720c ba547bcc nt!IofCallDriver+0x45

WARNING: Stack unwind information not available. Following frames may be wrong.

b8cb7214 8083f9c0 tmpreflt!TmpAddRdr+0x7b8

b8cb7228 ba4e08be nt!IofCallDriver+0x45

b8cb7430 ba4e09d3 DRIVER_A+0x28be

b8cb7450 b85fa306 DRIVER_A+0x29d3

b8cb763c b85fa50d DRIVER_B+0x8306

b8cb765c 8082f0d7 DRIVER_B+0x850d

b8cb7674 8082f175 nt!IoPageRead+0x109

b8cb76f8 80849cd5 nt!MiDispatchFault+0xd2a

b8cb7754 80837d0a nt!MmAccessFault+0x64a

b8cb7754 8093837b nt!KiTrap0E+0xdc

b8cb781c f718c0ac nt!CcMapData+0x8c

b8cb783c f718c6e6 Ntfs!NtfsMapStream+0x4b

b8cb78b0 f718c045 Ntfs!NtfsReadMftRecord+0x86

b8cb78e8 f718c0f4 Ntfs!NtfsReadFileRecord+0x7a
```

```
b8cb7920 f7155c3c Ntfs!NtfsLookupInFileRecord+0x37
b8cb7a30 f715746a Ntfs!NtfsLookupAllocation+0xdd
b8cb7bfc f7157655 Ntfs!NtfsPrepareBuffers+0x25d
b8cb7dd8 f715575e Ntfs!NtfsNonCachedIo+0x1ee
b8cb7ec4 f71588de Ntfs!NtfsCommonRead+0xaf5
b8cb8070 8083f9c0 Ntfs!NtfsFsdRead+0x113
b8cb8084 f7212c53 nt!IofCallDriver+0x45
b8cb80ac 8083f9c0 fltmgr!FltpDispatch+0x6f
b8cb80c0 ba547bcc nt!IofCallDriver+0x45
b8cb80c8 8083f9c0 tmpreflt!TmpAddRdr+0x7b8
b8cb80dc ba4e08be nt!IofCallDriver+0x45
b8cb82e4 ba4e09d3 DRIVER_A+0x28be
b8cb8304 b85fa306 DRIVER_A+0x29d3
b8cb84f0 b85fa50d DRIVER_B+0x8306
b8cb8510 8082f0d7 DRIVER_B+0x850d
b8cb8528 8082f175 nt!IoPageRead+0x109
b8cb85ac 80849cd5 nt!MiDispatchFault+0xd2a
b8cb8608 80837d0a nt!MmAccessFault+0x64a
b8cb8608 8093837b nt!KiTrap0E+0xdc
b8cb86d0 f718c0ac nt!CcMapData+0x8c
b8cb86f0 f718ef1b Ntfs!NtfsMapStream+0x4b
b8cb8720 f7186aa7 Ntfs!ReadIndexBuffer+0x8f
b8cb8894 f7187042 Ntfs!NtfsUpdateFileNameInIndex+0x62
b8cb8990 f7186059 Ntfs!NtfsUpdateDuplicateInfo+0x2b0
b8cb8b98 f7186302 Ntfs!NtfsCommonCleanup+0x1e82
b8cb8d08 8083f9c0 Ntfs!NtfsFsdCleanup+0xcf
b8cb8d1c f7212c53 nt!IofCallDriver+0x45
b8cb8d44 8083f9c0 fltmgr!FltpDispatch+0x6f
b8cb8d58 ba54809a nt!IofCallDriver+0x45
b8cb8d80 ba54d01d tmpreflt!TmpQueryFullName+0x454
b8cb8d90 8083f9c0 tmpreflt!TmpQueryFullName+0x53d7
b8cb8da4 ba4e08be nt!IofCallDriver+0x45
b8cb8fac ba4e09d3 DRIVER_A+0x28be
b8cb8fcc b85fa306 DRIVER_A+0x29d3
b8cb91b8 b85fa50d DRIVER_B+0x8306
b8cb91d8 80937f75 DRIVER_B+0x850d
b8cb9208 8092add4 nt!IopCloseFile+0x2ae
b8cb9238 8092af7a nt!ObpDecrementHandleCount+0x10a
b8cb9260 8092ae9e nt!ObpCloseHandleTableEntry+0x131
b8cb92a4 8092aee9 nt!ObpCloseHandle+0x82
b8cb92b4 80834d3f nt!NtClose+0x1b
b8cb92b4 8083c0fc nt!KiFastCallEntry+0xfc
b8cb9330 bf835765 nt!ZwClose+0x11
b8cb9608 bf8aa2dd win32k!bCreateSection+0x2ad
b8cb9660 bf826b45 win32k!EngMapFontFileFDInternal+0xc6
b8cb96c0 bf82784a win32k!PUBLIC_PFTOBJ::bLoadFonts+0x17f
b8cb991c bf9bcb67 win32k!PUBLIC_PFTOBJ::bLoadAFont+0x77
b8cb9af0 bf9bcb16 win32k!bInitOneStockFontInternal+0x42
b8cb9b0c bf9bb0e8 win32k!bInitOneStockFont+0x3f
b8cb9cf4 bf9ba845 win32k!bInitStockFontsInternal+0x12a
```

```
b8cb9cfc bf8246ad win32k!bInitStockFonts+0xa

b8cb9d48 bf8242d5 win32k!InitializeGreCSRSS+0x149

b8cb9d50 80834d3f win32k!NtUserInitialize+0x66

b8cb9d50 7c82ed54 nt!KiFastCallEntry+0xfc

0015fdb0 00000000 0x7c82ed54
```

The next step is to calculate how much space each frame is taking up. This can be done by walking the stack manually. Just subtract the subsequent EBP from the current EBP for each frame and add up the space used by all the modules.

Module Stack Usage Percentage
Ntfs 4152 36%
DRIVER_A 1572 14%
win32k 2592 22%
DRIVER_B 1656 14%
tmpreflt 72 1%
fltmgr 120 1%
nt 1420 12%

It would be easy to blame NTFS since it is the top stack consumer, but look closer. Even though NTFS is using the most of space in our example, this is due to both DRIVER_A and DRIVER_B making repeated calls into NTFS to access data. Alone, it is likely that neither driver would have caused a problem, but both drivers combined resulted in a bugcheck. Conscientious driver writing and efficient stack usage would have prevented this problem. Both drivers need to optimize the number of calls they make to NTFS.

Further reading

http://www.microsoft.com/whdc/driver/kernel/mem-mgmt.mspx

http://www.microsoft.com/whdc/Driver/tips/KMstack.mspx

http://support.microsoft.com/kb/186775

For more information on Task State Segments, please see the Intel and AMD Processor manuals.

Comments

Dominik Rappaport
1 Feb 2008 8:16 PM

Again a very interesting article with much background information. Keep on blogging. :-)

Skywing
2 Feb 2008 11:52 AM

Use 'kf' and the debugger will count stack variable usage per frame for you (see **http://www.nynaeve.net/?p=60** for an example).

Domnet uk
21 Mar 2008 8:33 PM

This article helped me sort out a tricky problem; just a bit of homebrew code messing with my hardware. Keep up the great work guys.

Steve Liu
12 Nov 2008 3:38 AM

This help me a lot on driver stack using, great article.

Sean
6 Apr 2014 10:02 PM

Nice article!

But I am kind of lost in how to calculate the space each frame is taking up.

Can someone explain and take tmpreflt as an example?

[Run the command "kcf". On the left hand side is a column of numbers representing the size of each frame. Add up the numbers next to the module to determine how much it is using.

In the below example clusdisk is using 0x318 bytes (0x310+0x8) and fltmgr is using 0x120 bytes (0x90+0x90).

```
8: kd> kcf
  Memory  Call Site
          nt!KiSwapContext
      140 nt!KiCommitThreadWait
       90 nt!KeWaitForSingleObject
       a0 nt!IoReportTargetDeviceChange
```

```
 70 nt!FsRtlNotifyVolumeEventEx
 30 nt!FsRtlNotifyVolumeEvent
 60 Ntfs!NtfsLockVolume
 d0 Ntfs!NtfsUserFsRequest
 40 Ntfs!NtfsCommonFileSystemControl
 b0 Ntfs!NtfsFsdFileSystemControl
 90 fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted
 90 fltmgr!FltpFsControl
 60 nt!IopXxxControlFile
130 nt!NtFsControlFile
 70 nt!KiSystemServiceCopyEnd
208 nt!KiServiceLinkage
  8 ClusDisk!ClusDskpOfflineVolume
310 ClusDisk!ClusDskpHaltProcessignWorker
 70 nt!IopProcessWorkItem
 30 nt!ExpWorkerThread
 90 nt!PspSystemThreadStartup
 40 nt!KxStartSystemThread
```

]