Driver Verifier in Windows Vista

February 9, 2009

Abstract

Driver Verifier monitors kernel-mode drivers to detect illegal function calls or actions that might corrupt the system. It was first included in Windows® 2000 and is available in all later versions of Windows. Driver Verifier works on both the free and checked builds of Windows operating systems.

For Windows Vista®, Driver Verifier was enhanced with new tests and features that let Driver Verifier expose more classes of typical driver bugs. Also, Driver Verifier is easier to use, beginning with Windows Vista. This paper provides a preview of Windows Driver Kit (WDK) documentation for the Driver Verifier enhancements in Windows Vista.

This paper applies for these operating systems:

 Windows Server® 2008
 Windows Vista

The current version of this paper is maintained on the Web at:
 <http://www.microsoft.com/whdc/DevTools/tools/DrvVerifier.mspx>

References and resources discussed here are listed at the end of this paper.

Disclaimer:

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Change |  |  |  |
| February 9, 2009 | Removed references to Driver Hang Verification |
| October 26, 2006 | Revision |
| 2005 | First published |

Contents

[Introduction 4](#_Toc221702191)

[Enabling Driver Verifier and Changing Settings without Rebooting 4](#_Toc221702192)

[Enhanced Low Resources Simulation 6](#_Toc221702193)

[Simulating STATUS\_ALERTED Return Value from Alertable Waits 7](#_Toc221702194)

[Debugging Driver Crashes 8](#_Toc221702195)

[Force Pending I/O Requests 10](#_Toc221702196)

[Using Force Pending I/O Requests 10](#_Toc221702197)

[Activating the Force Pending I/O Requests Option 11](#_Toc221702198)

[Viewing the Results of the Force Pending I/O Requests Option 12](#_Toc221702199)

[Security Checks 13](#_Toc221702200)

[More Thorough I/O Verification 14](#_Toc221702201)

[Viewing the IoAllocateIrp, IoCompleteRequest, and IoCancelIrp Calls Log 14](#_Toc221702202)

[Activating the I/O Verification Option 17](#_Toc221702203)

[Enhanced IRQL Checking 17](#_Toc221702204)

[Miscellaneous Checks 19](#_Toc221702205)

[Enabling Kernel Handle Tracing for the System Process 19](#_Toc221702206)

[Viewing the Kernel Handle Tracing Log for the System Process 19](#_Toc221702207)

[Activating the Miscellaneous Checks Option 24](#_Toc221702208)

[Viewing the Results of the Miscellaneous Checks Option 25](#_Toc221702209)

[Locked Memory Page Tracking 27](#_Toc221702210)

[Additional Automatic Checks 27](#_Toc221702211)

[Logging Pool Memory Allocate and Free Calls 28](#_Toc221702212)

[Viewing the Log of Pool Memory Allocate and Free Calls 28](#_Toc221702213)

[Resources 31](#_Toc221702214)

# Introduction

Driver Verifier monitors kernel-mode drivers to detect illegal function calls or actions that might corrupt the system. It can subject the drivers to many different stresses and tests to find incorrect behavior.

We strongly encourage hardware manufacturers to test their drivers by using Driver Verifier to ensure that drivers are not making illegal function calls or causing system corruption. Driver Verifier is included in Windows® 2000 and all later versions of Windows. Driver Verifier works on both the free and checked builds of Windows operating systems.

For Windows Vista®, Driver Verifier was enhanced with new tests and features that let Driver Verifier expose more classes of typical driver bugs. Also, Driver Verifier is easier to use, beginning with Windows Vista. This paper provides a preview of Windows Driver Kit (WDK) documentation for the Driver Verifier enhancements in Windows Vista.

The Debugging Tools for Windows are typically used to investigate issues that Driver Verifier exposes, as exemplified in this preview. For availability of Windows Vista and Debugging Tools for Windows, see “Resources” at the end of this paper.

Note  Driver Verifier runs only with elevated credentials in Windows Vista and later versions of Windows.

# Enabling Driver Verifier and Changing Settings without Rebooting

In Windows Vista, Driver Verifier lets you activate most options and to add and remove drivers for verification without rebooting the computer.

Specifically, in the version of Driver Verifier included in Windows Vista:

* You can start verification of any driver without rebooting, even if Driver Verifier is not already running.

This eliminates the requirement in earlier operating systems that Driver Verifier be verifying another driver before you reboot Windows.

* You can start the verification of a driver that is already loaded.

This eliminates the earlier requirement that drivers load after you start Driver Verifier.

* You can activate or deactivate most Driver Verifier options without rebooting.

Again, Driver Verifier need not be running before you can change options.

This new functionality is designed to eliminate the time that you spend on disruptive reboots, which lets you use Driver Verifier on production servers that cannot afford to reboot and to monitor your driver while you attach and remove devices.

Qualifications. There are a few qualifications, however:

* You cannot activate or deactivate the SCSI Verification or Disk Integrity Checking options without rebooting.
* You cannot stop the verification of a driver without rebooting if the driver is currently loaded. However, you can deactivate all the Driver Verifier options without rebooting, which minimizes the overhead until the next reboot.
* Drivers that are verified after they are loaded and without rebooting are not monitored as thoroughly as drivers that are loaded after a reboot. Approximately 90 percent of the typical Driver Verifier tests are active when verification starts after you load the driver. To get the full Driver Verifier power, whenever possible, enable the driver for verification and then reboot the computer.
* The options and drivers that you select without rebooting are effective immediately, but they are lost when you shut down or restart the system.

Using the No-Reboot Feature. You can use the no-reboot feature in Driver Verifier Manager or at the command line. To start or stop verification, or to change options without rebooting, use the procedures that were established for volatile settings.

The following examples show how to use the no-reboot feature at the command line by adding the /volatile parameter.

To start or stop the verification of a driver without rebooting

**verifier /volatile /adddriver** *DriverName.sys*

**verifier /volatile /removedriver** *DriverName.sys*

You can use this command syntax to add any driver (start the verification), even a driver that is currently loaded. Commands to remove a currently loaded driver (stop the verification) fail. As always, the verification of a driver that is not loaded begins as soon as the driver is loaded.

To activate or deactivate options without rebooting

**verifier /volatile /flags** [*Options*]

You can use this command syntax with any Driver Verifier option, except for SCSI Verification and Disk Integrity Checking. For example:

verifier /volatile /flags 0x20

This command activates the deadlock detection option without rebooting.

You cannot stop the verification of a driver that is currently loaded without rebooting. However, you can use the following command syntax to deactivate all Driver Verifier options without rebooting, which minimizes the overhead until the next reboot.

To turn off all Driver Verifier options

* Use the following command to deactivate all Driver Verifier options without rebooting:

verifier /volatile /flags 0

Driver Verifier continues to monitor the driver by using the options in the Automatic Checks feature, which cannot be turned off, but the overhead is reduced to approximately 10 percent of the overhead of a typical verification.

# Enhanced Low Resources Simulation

When the Low Resources Simulation option is active, Driver Verifier fails random instances of the driver’s memory allocations, as might occur if the driver is running on a computer that has insufficient memory. This tests the driver’s ability to respond correctly to low-memory and other low-resource conditions.

The Low Resources Simulation test fails allocations that are requested by calls to several different functions, including ExAllocatePoolXXX, MmProbeAndLockPages, MmMapLockedPagesSpecifyCache, and MmMapIoSpace. Starting with Windows Vista, calls to **MmAllocateContiguousMemory(SpecifyCache)**, **MmAllocatePagesForMdl**, **IoAllocateIrp**, **IoAllocateMdl**, **IoAllocateWorkItem** and **IoAllocateErrorLogEntry** are also injected with faults.

In Windows Vista, you can specify the following custom settings:

* Probability that a given allocation will fail.

The default is 6%.

* Applications affected.

This setting limits the injected failed allocations to the specified applications. By default, all applications are affected.

* Pool tags affected.

This setting limits the injected faults to allocations with the specified pool tags. By default, all allocations are affected.

* Delay (in minutes) before allocations are failed.

This delay lets the system start and stabilize before faults are injected. The default is 7 minutes.

On earlier versions of Windows, you cannot customize these settings; the default values are used. In Windows Vista, you can change these settings at the command line.

The syntax for these settings is as follows.

**verifier /faults** [*Probability PoolTags Applications DelayMins*] **/driver** *DriverList*

**verifier /volatile /faults** [*Probability PoolTags Applications*]

Note The custom settings parameters must appear in the order displayed. If you omit a value, type quotation marks to hold its place. For example:

verifier /volatile /faults 700 "" ""

#### Custom Settings Subparameters

/faults

Enables the Low Resources Simulation option in Driver Verifier.

You cannot use /flags 0x4 with the custom setting subparameters.

*Probability*

Specifies the probability that Driver Verifier will fail a given allocation.

Type a number in decimal or hexadecimal format to represent the number of chances in 10,000 that Driver Verifier will fail the allocation. The default value—600—means 600/10000, or 6 percent.

*PoolTags*

Limits the allocations that Driver Verifier can fail to allocations with the specified pool tags.

You can use a wildcard character (\*) to represent multiple pool tags. To list multiple pool tags, separate the tags with spaces and enclose the list in quotation marks. By default, all allocations can fail. For example:

verifier /faults 1000 "Tag1 Fred" Notepad.exe 5

*Applications*

Limits the allocations that Driver Verifier can fail to allocations for the specified programs.

Type the name of an executable file. To list programs, separate the program names with spaces and enclose the list in quotation marks. By default, allocations in all applications can fail. For example:

verifier /faults 1000 "Tag1 Fred" "Notepad.exe explorer.exe" 5

*DelayMins*

Specifies the number of minutes after booting during which Driver Verifier does not intentionally fail any allocations.

This delay lets the drivers load and the system to stabilize before the test begins. Type a number in decimal or hexadecimal format. The default value is 7 (minutes).

For example, the following command enables Low Resources Simulation with a probability of 10 percent (1000/10000) and a delay of 5 minutes for the pool tags named Tag1 and Fred plus the Notepad.exe application:

verifier /faults 1000 "Tag1 Fred" Notepad.exe 5

The following command enables Low Resources Simulation with the default values, except that it extends the delay to 10 minutes:

verifier /faults "" "" 0xa

## Simulating STATUS\_ALERTED Return Value from Alertable Waits

Typical wait APIs such as KeWaitForSingleObject accept as one of their parameters a Boolean Alertable value. An application that has a handle to the thread that is executing the alertable wait can alert the waiting thread. Under those circumstances, the wait call returns the special value STATUS\_ALERTED before the dispatcher object being waited on is signaled. The driver that is executing the wait should be prepared to deal correctly with this kind of possible unexpected wait result.

When the Low Resources Simulation option is enabled, Driver Verifier randomly forces some alertable waits to return STATUS\_ALERTED. The same probability value that is used for injecting faults in other APIs is also used for injecting STATUS\_ALERTED results.

##  Debugging Driver Crashes

This information applies to any operating system version when it is using Low Resources Simulation, not just to Windows Vista.

The easiest crashes to understand are probably those where a driver is accessing a NULL pointer. Usually a source code inspection around the code path that crashed reveals that the driver called ExAllocatePoolWithTag, that function returned NULL, and the driver did not check the return value, so it crashed when it used the pointer.

Understanding the cause of a driver crash is not always trivial, though. Often you can extract useful information by looking at the stack traces for recently injected faults. You can do this by using !verifier 4 in the kernel debugger. By default, !verifier 4 displays the four most recent injected faults. You can specify an additional parameter for !verifier to display more stack traces. The stack trace for the most recent fault appears first.

An example follows of a hypothetical crash in win32k!GreEnableEUDC:

|  |
| --- |
| kd> !verifier 4Resource fault injection history:Tracker @ 8354A000 (# entries: 80, size: 80, depth: 8)Entry @ 8354B258 (index 75) Thread: C2638220 816760CB nt!VerifierExAllocatePoolWithTag+0x49 A4720443 win32k!bDeleteAllFlEntry+0x15d A4720AB0 win32k!GreEnableEUDC+0x70 A47218FA win32k!CleanUpEUDC+0x37 A473998E win32k!GdiMultiUserFontCleanup+0x5 815AEACC nt!MiDereferenceSession+0x74 8146D3B4 nt!MmCleanProcessAddressSpace+0x112 815DF739 nt!PspExitThread+0x603Entry @ 8354B230 (index 74) Thread: 8436D770 816760CB nt!VerifierExAllocatePoolWithTag+0x49 A462141C win32k!Win32AllocPool+0x13 A4725F94 win32k!StubGdiAlloc+0x10 A4631A93 win32k!ExAllocateFromPagedLookasideList+0x27 A47261A4 win32k!AllocateObject+0x23 A4726F76 win32k!HmgAlloc+0x25 A47509D8 win32k!DCMEMOBJ::DCMEMOBJ+0x3b A4717D61 win32k!GreCreateDisplayDC+0x31Entry @ 8354B208 (index 73) Thread: D6B4B9B8 816760CB nt!VerifierExAllocatePoolWithTag+0x49 A462141C win32k!Win32AllocPool+0x13 A46C2759 win32k!PALLOCMEM+0x17 A477CCF2 win32k!bComputeGISET+0x82 A477D07D win32k!PFEMEMOBJ::bInit+0x248 A475DC18 win32k!PFFMEMOBJ::bAddEntry+0x6c A475E3E6 win32k!PFFMEMOBJ::bLoadFontFileTable+0x81 A475BADE win32k!PUBLIC\_PFTOBJ::bLoadFonts+0x2c4Entry @ 8354B1E0 (index 72) Thread: CCA0A480 816760CB nt!VerifierExAllocatePoolWithTag+0x49 813B8C30 fltmgr!FltpAllocateIrpCtrl+0x122 813CB1C9 fltmgr!FltpCreate+0x28d 81675275 nt!IovCallDriver+0x1b1 8141EDF1 nt!IofCallDriver+0x1f 81566106 nt!IopParseDevice+0xde6 815B9916 nt!ObpLookupObjectName+0x61a 815B72D5 nt!ObOpenObjectByName+0xf7 |

The most recent allocation failure was induced on the GreEnableEUDC code path. Remember that GreEnableEUDC was the function that crashed in the example scenario. Note that the allocation failure happened in the context of thread C2638220. Run !thread -1 and, if the address of the current thread is C2638220, then it is even more probable that the most recent fault was related to the current crash. So you should review the source code around that area, looking for a code path that could result in that kind of crash.

Often, the current crash is related to the most recently injected failure. If looking at the most recent stack trace was not helpful, you can examine the other three stack traces in our example. If these are not helpful either, you can use !verifier 4 80 to display the most recent 0x80 stack traces, and one of them might prove to be useful.

Driver Verifier keeps track of the number of faults injected since the system booted. Also, Driver Verifier keeps track of the number or attempted pool allocations. These two numbers can help understand:

* Whether the Low Resources Simulation is actually injecting faults in the driver being tested.
* Whether too many faults were injected.

For example, if the number of injected faults reported as compared to the number of attempted allocations is too large, you can adjust the fault injection probability to a smaller value for the next test pass.

* Whether too few faults were injected, so that the probability should be increased for future test passes.

These counters can be displayed by using !verifier as follows:

|  |
| --- |
| !verifierVerify Level 5 ... enabled options are: Special pool Inject random low-resource API failuresSummary of All Verifier StatisticsRaiseIrqls 0x2c671fAcquireSpinLocks 0xca1a02Synch Executions 0x10a623Trims 0x0Pool Allocations Attempted 0x862e0ePool Allocations Succeeded 0x8626e3Pool Allocations Succeeded SpecialPool 0x768060Pool Allocations With NO TAG 0x0Pool Allocations Failed 0x34fResource Allocations Failed Deliberately 0x3f5 |

# Force Pending I/O Requests

The Force Pending I/O Requests option randomly returns STATUS\_PENDING in response to a driver’s calls to IoCallDriver. This option tests the driver’s logic in response to STATUS\_PENDING return values from IoCallDriver. This option is new to Windows Vista.

|  |
| --- |
| Caution Do not use the Force Pending I/O Requests option on a driver unless you have detailed knowledge of the operation of the driver and have verified that the driver is designed to handle STATUS\_PENDING return values from all its calls to IoCallDriver. Running this option on a driver that is not designed to handle STATUS\_PENDING from all calls can result in crashes, memory corruptions, and unusual system behavior that can be difficult to debug or correct. |

## Using Force Pending I/O Requests

Higher-level drivers in a driver stack call IoCallDriver to pass an I/O request packet (IRP) to lower-level drivers in the driver stack. The driver dispatch routine in the lower-level driver that receives the IRP can either complete the IRP immediately or return STATUS\_PENDING and complete the IRP later.

Typically, the caller must be prepared to handle either outcome. However, because most dispatch routines handle the IRP immediately, the STATUS\_PENDING logic in the caller is not often exercised, so serious logic errors might not be detected during regular testing. The Force Pending I/O Requests option intercepts calls to IoCallDriver and returns STATUS\_PENDING to test the calling driver’s infrequently used logic.

#### When do you use Force Pending I/O Requests?

Before you run this test, review the driver design and source code and confirm that the driver is intended to handle STATUS\_PENDING from all its IoCallDriver calls.

Many drivers are not designed to handle STATUS\_PENDING on all calls to IoCallDriver. They might be sending the IRP to a particular well-known driver that is guaranteed to complete the IRP immediately. Sending STATUS\_PENDING to a driver that does not handle it can cause memory corruption and driver and system crashes.

#### How should drivers handle STATUS\_PENDING?

The higher-level driver that calls IoCallDriver must handle a STATUS\_PENDING return value as follows:

* Before it calls IoCallDriver, the driver must call IoBuildSynchronousFsdRequest to arrange for synchronous processing of the IRP.
* If IoCallDriver returns STATUS\_PENDING, the driver must wait for the completion of the IRP by calling KeWaitForSingleObject on the specified event.
* The driver must anticipate that the IRP might be freed before the I/O manager signals the event.
* After it calls IoCallDriver, the caller cannot reference the IRP.

#### Which errors does Force Pending I/O Request detect?

The Force Pending I/O Request option detects the following errors in the driver that calls IoCallDriver and receives a STATUS\_PENDING return value:

* The driver does not call IoBuildSynchronousFsdRequest to arrange for synchronous processing.
* The driver does not call KeWaitForSingleObject.
* The driver references a value in the IRP structure after it calls IoCallDriver.

After it calls IoCallDriver, the higher-level driver cannot access the IRP unless it has set a completion routine—and then only when all lower-level drivers have completed the IRP. If the IRP is freed, the driver crashes.

* The driver calls a related function incorrectly.

For example, the driver calls KeWaitForSingleObject and passes a handle to the event (as the Object parameter) instead of passing a pointer to an event object.

* The driver waits for the wrong event.

For example, the driver calls IoSetCompletionRoutine, but waits for an internal event that is signaled by its own completion routine, instead of waiting for the IRP event that is signaled by the I/O manager when the IRP is complete.

## Activating the Force Pending I/O Requests Option

To activate Force Pending I/O Requests, you must activate both Enhanced I/O Verification and the Force Pending I/O Requests options. You can activate the Force Pending I/O Requests option for one or more drivers by using Driver Verifier Manager or the command line.

To activate the Force Pending I/O Requests option at the command line

* Use a flag value of 0x240 or add 0x240 to the flag value.

This value activates Enhanced I/O Verification (0x40) and Force Pending I/O Requests (0x200). For example:

verifier /flags 0x240 /driver MyDriver.sys

The option is active after the next boot.

If you try to activate only Force Pending I/O Requests (verifier /flags 0x200), Driver Verifier automatically enables both Force Pending I/O Requests (0x200) and Enhanced I/O Verification (0x40).

You can also activate and deactivate Force Pending I/O Requests without rebooting the computer by adding the /volatile parameter to the command. For example:

verifier /volatile /flags 0x240 /driver MyDriver.sys

This setting is effective immediately, but is lost when you shut down or reboot the computer.

To activate the Force Pending I/O Requests option by using Driver Verifier Manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings from a full list.

3. Select Enhanced I/O Verification and Force Pending I/O Requests.

If you select only Force Pending I/O Requests, Driver Verifier Manager reminds you that Enhanced I/O Verification is required and offers to enable it for you.

## Viewing the Results of the Force Pending I/O Requests Option

The !verifier debugger extension is used to view Driver Verifier results.

To view the results of the Force Pending I/O Requests test

* Use the !verifier debugger extension with a flag value of 0x40.

For information about !verifier, see the !verifier topic in the Debugging Tools for Windows documentation.

If the test machine crashes because of the Force Pending I/O Requests test, you can use the !verifier 40 command to find the cause. In the current stack trace, find the address of the IRP that your driver recently used. For example, if you use the kP command, which displays the stack frame for a thread, you can find the IRP address among the function parameters of the current stack trace. Then, run !verifier 40 and look for the address of the IRP. The most recent force pending stack traces appear at the top of the display.

For example, the following stack trace of Pci.sys shows its response to Force Pending I/O Requests. The test does not reveal any errors in the Pci.sys logic:

|  |
| --- |
| kd> !verifier 40Size of the log is is 0x40========================================================IRP: 8f84ef00 - forced pending from stack trace: 817b21e4 nt!IovpLocalCompletionRoutine+0xb2 81422478 nt!IopfCompleteRequest+0x15c 817b2838 nt!IovCompleteRequest+0x9c 84d747df acpi!ACPIBusIrpDeviceUsageNotification+0xf5 84d2d36c acpi!ACPIDispatchIrp+0xe8 817b258f nt!IovCallDriver+0x19d 8142218e nt!IofCallDriver+0x1c 817c6a9d nt!ViFilterDispatchPnp+0xe9 817b258f nt!IovCallDriver+0x19d 8142218e nt!IofCallDriver+0x1c 84fed489 pci!PciCallDownIrpStack+0xbf 84fde1cb pci!PciDispatchPnpPower+0xdf 817b258f nt!IovCallDriver+0x19d 8142218e nt!IofCallDriver+0x1c 817c6a9d nt!ViFilterDispatchPnp+0xe9 817b258f nt!IovCallDriver+0x19d 8142218e nt!IofCallDriver+0x1c 84ff2ff5 pci!PciSendPnpIrp+0xbd 84fec820 pci!PciDevice\_DeviceUsageNotification+0x6e 84fde1f8 pci!PciDispatchPnpPower+0x10c 817b258f nt!IovCallDriver+0x19d 8142218e nt!IofCallDriver+0x1c 84d76ce2 acpi!ACPIFilterIrpDeviceUsageNotification+0x96 84d2d36c acpi!ACPIDispatchIrp+0xe8 817b258f nt!IovCallDriver+0x19d 8142218e nt!IofCallDriver+0x1c 84f7f16c PCIIDEX!PortWdmForwardIrpSynchronous+0x8e 84f7b2b3 PCIIDEX!GenPnpFdoUsageNotification+0xcb 84f7d301 PCIIDEX!PciIdeDispatchPnp+0x45 817b258f nt!IovCallDriver+0x19d 8142218e nt!IofCallDriver+0x1c |

The stack trace shows that Acpi.sys was trying to complete IRP 8f84ef00. Driver Verifier forced a deferred completion, so Acpi.sys returned STATUS\_PENDING to pci!PciCallDownIrpStack. If this call had caused a crash, the driver owner would need to review the source code for pci!PciCallDownIrpStack and revise it to correctly handle the STATUS\_PENDING.

# Security Checks

The Security Checks option of Driver Verifier monitors the driver for common errors that can result in security vulnerabilities. This option is new in Windows Vista.

Specifically, the Security Checks option looks for the following incorrect driver behavior:

* Calling kernel Zw APIs with user-mode addresses as parameters.

When it calls any Zw API, the current KPROCESSOR\_MODE becomes KernelMode. This means that the Zw API internal code can trust all the parameters of the function. User-mode addresses should not be allowed to cross that boundary of trust because the application can free or change the user-mode address while the kernel code is using it.

* Calling kernel Zw APIs with malformed UNICODE\_STRINGs as parameters.

When it calls any Zw API, the current KPROCESSOR\_MODE becomes KernelMode. This means that the Zw API code can trust all the parameters of the function. Typical bugs detected by Driver Verifier are:

* Strings with the Buffer field pointing to user-mode memory.
* Strings with incorrect Length or MaximumLength fields.

For example, MaximumLength < Length—or any of these two fields—is an odd number. These two fields are the number of bytes that represent a Unicode string, so they should always be even numbers.

* Calling kernel Zw APIs with an incorrect OBJECT\_ATTRIBUTES structure as parameter.

The checks described earlier for user-mode addresses and UNICODE\_STRINGs are performed for each OBJECT\_ATTRIBUTES structure field of interest.

* Inconsistent Irp->RequestorMode and I/O Request parameters.

If Irp->RequestorMode is KernelMode, then a driver that is processing the IRP can assume that the I/O Request parameters are trusted. So the I/O Request parameters (Irp->AssociatedIrp.SystemBuffer or Irp->UserBuffer) should never be a user-mode address in that case.

You can activate the Security Checks option for one or more drivers by using Driver Verifier Manager or the Verifier.exe command line. At the command line, the Security Checks option is represented by bit 8 (0x100).

To activate the Security Checks option at the command line

* Use a flag value of 0x100 or add 0x100 to the flag value.

For example:

verifier /flags 0x100 /driver MyDriver.sys

The option is active after the next boot.

You can also activate and deactivate Security Checks without rebooting the computer by adding the /volatile parameter to the command. For example:

verifier /volatile /flags 0x100 /driver MyDriver.sys

This setting is effective immediately, but is lost when you shut down or reboot the computer.

The Security Checks option is also included in the /standard settings. For example:

verifier /standard /driver MyDriver.sys

To activate Security Checks by using Driver Verifier Manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings from a full list.

3. Select Security Checks.

# More Thorough I/O Verification

The I/O Verification option of Driver Verifier monitors the driver for common errors while it processes IRPs. The Driver Verifier was enhanced in Windows Vista to check for the following typical incorrect driver behavior:

* Releasing a Remove Lock without acquiring it first.
* Calling IoReleaseRemoveLock or IoReleaseRemoveLockAndWait with a Tag parameter that is different from the Tag parameter for the corresponding IoAcquireRemoveLock call.
* Driver IRP dispatch routines that return after disabling APCs.

The I/O Verification option also records IoAllocateIrp, IoCompleteRequest, and IoCancelIrp calls in an IRP transition log. When I/O Verification is enabled, the stack trace for all calls to IoAllocateIrp, IoCompleteRequest, and IoCancelIrp is recorded in a log. This log contains just the recent history of these function calls. When the log becomes full, older entries from the log are reused to store newer entries.

## Viewing the IoAllocateIrp, IoCompleteRequest, and IoCancelIrp Calls Log

The contents of the IRP transition log can be displayed by using !verifier 0x100 in the kernel debugger.

#### Example 1

Display all the available information about IRP at address 0x8f402d70:

|  |
| --- |
| 0: kd> !verifier 100 8f402d70Parsing 00001000 array entries, searching for IRP address 8f402d70.====================================================================IRP 8f402d70, Thread 855d2cb881f18796 nt!IovAllocateIrp+0xe681827a58 nt!IoPageRead+0x3c81885615 nt!MiDispatchFault+0x207d818d6678 nt!MmAccessFault+0x50fe8193a7e8 nt!\_KiTrap0E+0xe4Parsing entry 00000fff/00001000 |

In this example, you can see that the IRP at address 0x8f402d70 was recently allocated by the thread at address 0x855d2cb8.

#### Example 2

Display the four most recent entries from the log:

|  |
| --- |
| 0: kd> !verifier 100 4Parsing 00000004 array entries.====================================================================IRP 8f402d70, Thread 855d2cb881f18796 nt!IovAllocateIrp+0xe681827a58 nt!IoPageRead+0x3c81885615 nt!MiDispatchFault+0x207d818d6678 nt!MmAccessFault+0x50fe8193a7e8 nt!\_KiTrap0E+0xe4====================================================================IRP b1544d70, Thread 84bf42b881f189f0 nt!IovCompleteRequest+0x2085f82a83 CLASSPNP!ClassCompleteRequest+0x1185f84cc7 CLASSPNP!TransferPktComplete+0x2ab81f18c2b nt!IovpLocalCompletionRoutine+0xfb81826d3f nt!IopfCompleteRequest+0x17b81f18aec nt!IovCompleteRequest+0x11c85e20287 ataport!IdeCompleteScsiIrp+0x3185e20c6a ataport!IdeCommonCrbCompletion+0x4485e1df9a ataport!IdeTranslateCompletedRequest+0x2685e22df9 ataport!IdeProcessCompletedRequests+0x12185e2302f ataport!IdePortCompletionDpc+0xab8184302d nt!KiRetireDpcList+0x167====================================================================IRP a79aaf20, Thread 84bf42b881f189f0 nt!IovCompleteRequest+0x2085e20287 ataport!IdeCompleteScsiIrp+0x3185e20c6a ataport!IdeCommonCrbCompletion+0x4485e1df9a ataport!IdeTranslateCompletedRequest+0x2685e22df9 ataport!IdeProcessCompletedRequests+0x12185e2302f ataport!IdePortCompletionDpc+0xab8184302d nt!KiRetireDpcList+0x167====================================================================IRP 8e94cd70, Thread 84bf42b881f189f0 nt!IovCompleteRequest+0x2085f82a83 CLASSPNP!ClassCompleteRequest+0x1185f84cc7 CLASSPNP!TransferPktComplete+0x2ab81f18c2b nt!IovpLocalCompletionRoutine+0xfb81826d3f nt!IopfCompleteRequest+0x17b81f18aec nt!IovCompleteRequest+0x11c85e20287 ataport!IdeCompleteScsiIrp+0x3185e20c6a ataport!IdeCommonCrbCompletion+0x4485e1df9a ataport!IdeTranslateCompletedRequest+0x2685e22df9 ataport!IdeProcessCompletedRequests+0x12185e2302f ataport!IdePortCompletionDpc+0xab8184302d nt!KiRetireDpcList+0x167 |

#### Example 3

Display all the recent history that is available in the log:

|  |
| --- |
| 0: kd> !verifier 100====================================================================IRP 8f402d70, Thread 855d2cb881f18796 nt!IovAllocateIrp+0xe681827a58 nt!IoPageRead+0x3c81885615 nt!MiDispatchFault+0x207d818d6678 nt!MmAccessFault+0x50fe8193a7e8 nt!\_KiTrap0E+0xe4====================================================================IRP b1544d70, Thread 84bf42b881f189f0 nt!IovCompleteRequest+0x2085f82a83 CLASSPNP!ClassCompleteRequest+0x1185f84cc7 CLASSPNP!TransferPktComplete+0x2ab81f18c2b nt!IovpLocalCompletionRoutine+0xfb81826d3f nt!IopfCompleteRequest+0x17b81f18aec nt!IovCompleteRequest+0x11c85e20287 ataport!IdeCompleteScsiIrp+0x3185e20c6a ataport!IdeCommonCrbCompletion+0x4485e1df9a ataport!IdeTranslateCompletedRequest+0x2685e22df9 ataport!IdeProcessCompletedRequests+0x12185e2302f ataport!IdePortCompletionDpc+0xab8184302d nt!KiRetireDpcList+0x167====================================================================IRP a79aaf20, Thread 84bf42b881f189f0 nt!IovCompleteRequest+0x2085e20287 ataport!IdeCompleteScsiIrp+0x3185e20c6a ataport!IdeCommonCrbCompletion+0x4485e1df9a ataport!IdeTranslateCompletedRequest+0x2685e22df9 ataport!IdeProcessCompletedRequests+0x12185e2302f ataport!IdePortCompletionDpc+0xab8184302d nt!KiRetireDpcList+0x167====================================================================IRP 8e94cd70, Thread 84bf42b881f189f0 nt!IovCompleteRequest+0x2085f82a83 CLASSPNP!ClassCompleteRequest+0x1185f84cc7 CLASSPNP!TransferPktComplete+0x2ab81f18c2b nt!IovpLocalCompletionRoutine+0xfb81826d3f nt!IopfCompleteRequest+0x17b81f18aec nt!IovCompleteRequest+0x11c85e20287 ataport!IdeCompleteScsiIrp+0x3185e20c6a ataport!IdeCommonCrbCompletion+0x4485e1df9a ataport!IdeTranslateCompletedRequest+0x2685e22df9 ataport!IdeProcessCompletedRequests+0x12185e2302f ataport!IdePortCompletionDpc+0xab8184302d nt!KiRetireDpcList+0x167====================================================================IRP b2792f20, Thread 84bf42b881f189f0 nt!IovCompleteRequest+0x2085e20287 ataport!IdeCompleteScsiIrp+0x3185e20c6a ataport!IdeCommonCrbCompletion+0x4485e1df9a ataport!IdeTranslateCompletedRequest+0x2685e22df9 ataport!IdeProcessCompletedRequests+0x12185e2302f ataport!IdePortCompletionDpc+0xab8184302d nt!KiRetireDpcList+0x167... |

## Activating the I/O Verification Option

You can activate the I/O Verification feature for one or more drivers by using Driver Verifier Manager or the Verifier.exe command line. At the command line, the I/O Verification option is represented by Bit 4 (0x10).

To activate I/O Verification at the command line

* Use a flag value of 0x10 or add 0x10 to the flag value.

For example:

verifier /flags 0x10 /driver MyDriver.sys

The feature is active after the next boot.

In Windows Vista, you can also activate and deactivate I/O Verification without rebooting the computer by adding the /volatile parameter to the command. For example:

verifier /volatile /flags 0x10 /driver MyDriver.sys

This setting is effective immediately, but is lost when you shut down or reboot the computer.

The I/O Verification feature is also included in the /standard settings. For example:

verifier /standard /driver MyDriver.sys

To activate I/O Verification by using Driver Verifier Manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings from a full list.

3. Select I/O verification.

To use the I/O Verification feature in the standard settings, in Driver Verifier Manager, click Create Standard Settings.

# Enhanced IRQL Checking

Although kernel-mode drivers are forbidden to access pageable memory at a high IRQL or while holding a spin lock, such an action might not be noticed if the memory page has not actually been trimmed from the working set and paged out to disk.

When the Force IRQL Checking option is enabled, Driver Verifier provides extreme memory pressure on the selected driver. Whenever a driver that is being verified requests a spin lock, calls KeSynchronizeExecution, or raises the IRQL to DISPATCH\_LEVEL or higher, all the driver’s pageable code and data are trimmed from the working set—including system pageable pool, code, and data. If the driver tries to access any of this memory, Driver Verifier issues a bug check.

Driver Verifier was improved in Windows Vista by adding tests that detect when certain synchronization objects are included in pageable memory. These synchronization objects cannot be paged because the operating system kernel is accessing them at elevated IRQL. Driver Verifier can detect pageable:

KTIMER
KMUTEX
KSPIN\_LOCK
KEVENT
KSEMAPHORE
ERESOURCE
FAST\_MUTEX

You can activate the Force IRQL Checking feature for one or more drivers by using Driver Verifier Manager or the Verifier.exe command line. At the command line, the Force IRQL Checking option is represented by Bit 1 (0x2).

To activate Force IRQL Checking at the command line

* Use a flag value of 0x2 or add 0x2 to the flag value.

For example:

verifier /flags 0x2 /driver MyDriver.sys

The feature is active after the next boot.

In Windows 2000 and later versions of Windows, you can also activate and deactivate Force IRQL Checking without rebooting by adding the /volatile parameter to the command. For example:

verifier /volatile /flags 0x2 /driver MyDriver.sys

This setting is effective immediately, but is lost when you shut down or reboot the computer.

The Force IRQL Checking feature is also included in the /standard settings. For example:

verifier /standard /driver MyDriver.sys

To activate Force IRQL Checking by using Driver Verifier Manager

1. Select Create custom settings (for code developers) and then click Next.

2. Select Select individual settings from a full list.

3. Select Force IRQL Checking.

To use this feature in the standard settings, in Driver Verifier Manager, click Create Standard Settings.

# Miscellaneous Checks

The Miscellaneous Checks option of Driver Verifier monitors the driver for common errors that cause the driver or system to crash, such as freeing memory that still contains active kernel objects.

Specifically, the Miscellaneous Checks option looks for the following incorrect driver behavior:

* Active work items in freed memory.

The driver calls ExFreePool to free a pool block that contains work items that were queued by using IoQueueWorkItem. This check is enabled by default in the checked build of Windows Server® 2003.

* Active resources in freed memory.

The driver calls ExFreePool to free a pool block that contains active ERESOURCE structures. The driver should call ExDeleteResource to delete ERESOURCE objects before it calls ExFreePool.

* Active lookaside lists in freed memory.

The driver calls ExFreePool to free a pool block that still contains active lookaside list structures (NPAGED\_LOOKASIDE\_LIST or PAGED\_LOOKASIDE\_LIST). The driver should call ExDeleteNPagedLookasideList or ExDeletePagedLookasideList to delete the lookaside lists before it calls ExFreePool.

* Windows Management Instrumentation (WMI) and Event Tracing for Windows (ETW) registration issues:
* Drivers that are unloaded but did not unregister their WMI callback.
* Drivers deleting device objects that were not unregistered yet from WMI.
* Drivers that are unloaded but did not unregister their ETW kernel-mode provider.
* Drivers trying to unregister providers that were unregistered already.

## Enabling Kernel Handle Tracing for the System Process

When the Miscellaneous Checks option is enabled, Kernel Handle Tracing is enabled for the System process. Handle Tracing saves stack traces for all recent handle OPEN, CLOSE, and BAD REFERENCE operations in the System process handle table. This can be helpful for debugging typical driver bugs that are related to kernel handles. This stack trace log contains just the recent history. When the log becomes full, older entries from the log are reused to store newer log entries.

## Viewing the Kernel Handle Tracing Log for the System Process

The contents of the System Process Kernel Handle Tracing log can be displayed by using !htrace in the kernel debugger.

#### Example 1

Display information that can be useful for understanding an INVALID\_KERNEL\_HANDLE (0x93) bug check:

|  |
| --- |
| \*\*\* Fatal System Error: 0x00000093 (0x00000**C7C**,0x00000001,0x00000000,0x00000000)0: kd> kbChildEBP RetAddr Args to Child994c56c4 81833464 00000003 994cec54 00000000 nt!RtlpBreakWithStatusInstruction994c5714 81833e9d 00000003 8193d980 846f5760 nt!KiBugCheckDebugBreak+0x1c994c5ac0 8183330e 00000093 00000c7c 00000001 nt!KeBugCheck2+0x5f3994c5ae4 81dd9275 00000093 00000c7c 00000001 nt!KeBugCheckEx+0x1e994c5b18 81dd9324 00000c7c 00000000 994c5bbc nt!ObpCloseHandle+0x1ff994c5b34 819373f3 80000c7c 994c5bdc 81928e31 nt!NtClose+0x50994c5b34 81928e31 80000c7c 994c5bdc 81928e31 nt!KiFastCallEntry+0x163994c5bb0 8712b5d7 80000**c7c** 871314a4 00000018 nt!ZwClose+0x11994c5bdc 871280f6 afadcf68 857b9190 afadcf68 buggy!VrftBadKernelHandle+0x57994c5bf4 81f18647 8492fd88 afadcf68 8492fd88 buggy!TdDeviceControl+0xc4994c5c18 818269f3 afadcfd8 81940950 84b0bf80 nt!IovCallDriver+0x277994c5c34 81c8e303 84b0bf80 afadcf68 afadcfd8 nt!IofCallDriver+0x1d994c5c58 81c8f398 8492fd88 afadcf68 84b0bf80 nt!IopSynchronousServiceTail+0x331994c5cf8 81c952bc 8492fd88 00000004 00000000 nt!IopXxxControlFile+0x72e994c5d34 819373f3 0000004c 00000000 00000000 nt!NtDeviceIoControlFile+0x4c994c5d34 7724d364 0000004c 00000000 00000000 nt!KiFastCallEntry+0x1630012fd5c 007a30e1 0000004c 00222124 00000000 ntdll+0x2d3640012fd84 007a363a 0000004c 00222124 00000000 ctlbuggy!TdSendIoctl+0x2a0: kd> !process 4 0Searching for Process with Cid == 4PROCESS **84073510** SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000 DirBase: 00122000 ObjectTable: 87401118 HandleCount: 582. Image: System0: kd> !htrace **c7c** **8407351**0Process 0x84073510ObjectTable 0x87401118--------------------------------------Handle 0xC7C - \*\*\* BAD REFERENCE \*\*\*:Thread ID = 0x000009c8, Process ID = 0x00000a440x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x1630x7722dfb8: ntdll+0xDFB80x7702f4ad: kernel32!DeviceIoControl+0x1420x007a30e1: ctlbuggy!TdSendIoctl+0x2A0x007a363a: ctlbuggy!ProcessCommandLine+0x6E0x007a38c7: ctlbuggy!main+0x2390x007a3aba: ctlbuggy!\_\_mainCRTStartup+0x102--------------------------------------Handle 0xC7C - CLOSEThread ID = 0x000009c8, Process ID = 0x00000a440x81dd9185: nt!ObpCloseHandle+0x10F0x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x1630x7722dfb8: ntdll+0xDFB80x7702f4ad: kernel32!DeviceIoControl+0x1420x007a30e1: ctlbuggy!TdSendIoctl+0x2A0x007a363a: ctlbuggy!ProcessCommandLine+0x6E0x007a38c7: ctlbuggy!main+0x239--------------------------------------Handle 0xC7C - OPENThread ID = 0x000009c8, Process ID = 0x00000a440x81de96ec: nt!ObInsertObject+0x1760x81e9ec1d: nt!NtCreateEvent+0xD10x819373f3: nt!KiFastCallEntry+0x1630x81928ef9: nt!ZwCreateEvent+0x110x8712b5ba: buggy!VrftBadKernelHandle+0x3A0x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x163 |

In this example, buggy.sys crashed because it called ZwClose with kernel handle 0x80000c7c as a parameter. The operating system kernel did not find handle 0xc7c in the process handle table, so another driver must have already closed the handle. Sometimes it is difficult to determine which driver might have closed the handle. In this case, the Miscellaneous Checks option was enabled and Handle Tracing information was available for handle 0xc7c in the system process (process ID = 4). That information shows that buggy.sys closed that handle once and now it is trying to close it again. This is a potentially troublesome driver bug because the handle value might have been reused by another driver. In that case, buggy.sys would close the other driver's handle, with unpredictable results.

#### Example 2

Display the five most recent operations available in the System process Handle Tracing log:

|  |
| --- |
| 0: kd> !htrace 0 84073510 5Process 0x84073510ObjectTable 0x87401118--------------------------------------Handle 0xC7C - \*\*\* BAD REFERENCE \*\*\*:Thread ID = 0x000009c8, Process ID = 0x00000a440x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x1630x7722dfb8: ntdll+0xDFB80x7702f4ad: kernel32!DeviceIoControl+0x1420x007a30e1: ctlbuggy!TdSendIoctl+0x2A0x007a363a: ctlbuggy!ProcessCommandLine+0x6E0x007a38c7: ctlbuggy!main+0x2390x007a3aba: ctlbuggy!\_\_mainCRTStartup+0x102--------------------------------------Handle 0xC7C - CLOSEThread ID = 0x000009c8, Process ID = 0x00000a440x81dd9185: nt!ObpCloseHandle+0x10F0x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x1630x7722dfb8: ntdll+0xDFB80x7702f4ad: kernel32!DeviceIoControl+0x1420x007a30e1: ctlbuggy!TdSendIoctl+0x2A0x007a363a: ctlbuggy!ProcessCommandLine+0x6E0x007a38c7: ctlbuggy!main+0x239--------------------------------------Handle 0xC7C - OPENThread ID = 0x000009c8, Process ID = 0x00000a440x81de96ec: nt!ObInsertObject+0x1760x81e9ec1d: nt!NtCreateEvent+0xD10x819373f3: nt!KiFastCallEntry+0x1630x81928ef9: nt!ZwCreateEvent+0x110x8712b5ba: buggy!VrftBadKernelHandle+0x3A0x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x163--------------------------------------Handle 0xFAC - CLOSEThread ID = 0x000006c0, Process ID = 0x0000022c0x81dd9185: nt!ObpCloseHandle+0x10F0x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x819373f3: nt!KiFastCallEntry+0x1630x7722da48: ntdll+0xDA480x75a6a644: +0x75A6A6440x75a6b6be: +0x75A6B6BE0x75a6bbd1: +0x75A6BBD10x75a6518e: +0x75A6518E0x75a859f0: +0x75A859F00x77252523: ntdll+0x32523--------------------------------------Handle 0xFAC - OPENThread ID = 0x000006c0, Process ID = 0x0000022c0x81ddb052: nt!ObOpenObjectByPointer+0x1000x81eba7ef: nt!ApphelpCacheControlValidateParameters+0x860x81ebaacb: nt!NtApphelpCacheControl+0x6D0x819373f3: nt!KiFastCallEntry+0x1630x7722da48: ntdll+0xDA480x75a6a644: +0x75A6A6440x75a6b6be: +0x75A6B6BE0x75a6bbd1: +0x75A6BBD10x75a6518e: +0x75A6518E0x75a859f0: +0x75A859F00x77252523: ntdll+0x32523--------------------------------------Parsed 0x5 stack traces.Dumped 0x5 stack traces. |

#### Example 3

Display all the recent Handle Tracing history for the System process:

|  |
| --- |
| 0: kd> !htrace 0 84073510Process 0x84073510ObjectTable 0x87401118--------------------------------------Handle 0xC7C - \*\*\* BAD REFERENCE \*\*\*:Thread ID = 0x000009c8, Process ID = 0x00000a440x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x1630x7722dfb8: ntdll+0xDFB80x7702f4ad: kernel32!DeviceIoControl+0x1420x007a30e1: ctlbuggy!TdSendIoctl+0x2A0x007a363a: ctlbuggy!ProcessCommandLine+0x6E0x007a38c7: ctlbuggy!main+0x2390x007a3aba: ctlbuggy!\_\_mainCRTStartup+0x102--------------------------------------Handle 0xC7C - CLOSEThread ID = 0x000009c8, Process ID = 0x00000a440x81dd9185: nt!ObpCloseHandle+0x10F0x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x1630x7722dfb8: ntdll+0xDFB80x7702f4ad: kernel32!DeviceIoControl+0x1420x007a30e1: ctlbuggy!TdSendIoctl+0x2A0x007a363a: ctlbuggy!ProcessCommandLine+0x6E0x007a38c7: ctlbuggy!main+0x239--------------------------------------Handle 0xC7C - OPENThread ID = 0x000009c8, Process ID = 0x00000a440x81de96ec: nt!ObInsertObject+0x1760x81e9ec1d: nt!NtCreateEvent+0xD10x819373f3: nt!KiFastCallEntry+0x1630x81928ef9: nt!ZwCreateEvent+0x110x8712b5ba: buggy!VrftBadKernelHandle+0x3A0x871280f6: buggy!TdDeviceControl+0xC40x81f18647: nt!IovCallDriver+0x2770x818269f3: nt!IofCallDriver+0x1D0x81c8e303: nt!IopSynchronousServiceTail+0x3310x81c8f398: nt!IopXxxControlFile+0x72E0x81c952bc: nt!NtDeviceIoControlFile+0x4C0x819373f3: nt!KiFastCallEntry+0x163--------------------------------------Handle 0xFAC - CLOSEThread ID = 0x000006c0, Process ID = 0x0000022c0x81dd9185: nt!ObpCloseHandle+0x10F0x81dd9324: nt!NtClose+0x500x819373f3: nt!KiFastCallEntry+0x1630x81928e31: nt!ZwClose+0x110x819373f3: nt!KiFastCallEntry+0x1630x7722da48: ntdll+0xDA480x75a6a644: +0x75A6A6440x75a6b6be: +0x75A6B6BE0x75a6bbd1: +0x75A6BBD10x75a6518e: +0x75A6518E0x75a859f0: +0x75A859F00x77252523: ntdll+0x32523… |

## Activating the Miscellaneous Checks Option

You can activate the Miscellaneous Checks option for one or more drivers by using Driver Verifier Manager or the Verifier.exe command line. At the command line, the Miscellaneous Checks option is represented by bit 11 (0x800).

To activate Miscellaneous Checks at the command line

* Use a flag value of 0x800 or add 0x800 to the flag value.

For example:

verifier /flags 0x800 /driver MyDriver.sys

The option is active after the next boot.

In Windows Vista, you can also activate and deactivate Miscellaneous Checks without rebooting by adding the /volatile parameter to the command. For example:

verifier /volatile /flags 0x800 /driver MyDriver.sys

This setting is effective immediately, but is lost when you shut down or reboot the computer.

The Miscellaneous Checks option is also included in the /standard settings. For example:

verifier /standard /driver MyDriver.sys

To activate Miscellaneous Checks by using Driver Verifier Manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings from a full list.

3. Select Miscellaneous Checks.

## Viewing the Results of the Miscellaneous Checks Option

To view the results of the Miscellaneous Checks option

* Use the !verifier extension in the kernel debugger.

For information about !verifier, see the Debugging Tools for Windows documentation.

In the following example, the Miscellaneous Checks option detected an active ERESOURCE structure in memory that the driver was trying to free, which resulted in Bug Check 0xC4. The Bug Check 0xC4 display includes the address of the ERESOURCE and the affected memory:

|  |
| --- |
| 1: kd> !verifier 1Verify Level 800 ... enabled options are: Miscellaneous checks enabledSummary of All Verifier StatisticsRaiseIrqls 0x0AcquireSpinLocks 0x0Synch Executions 0x0Trims 0x0Pool Allocations Attempted 0x1Pool Allocations Succeeded 0x1Pool Allocations Succeeded SpecialPool 0x0Pool Allocations With NO TAG 0x0Pool Allocations Failed 0x0Resource Allocations Failed Deliberately 0x0Current paged pool allocations 0x0 for 00000000 bytesPeak paged pool allocations 0x0 for 00000000 bytesCurrent nonpaged pool allocations 0x0 for 00000000 bytesPeak nonpaged pool allocations 0x0 for 00000000 bytesDriver Verification ListEntry State NonPagedPool PagedPool Module8459ca50 Loaded 00000000 00000000 buggy.sys\*\*\* Fatal System Error: 0x000000c4 (0x000000D2,0x9655D4A8,0x9655D468,0x000000B0) 0xD2 : Freeing pool allocation that contains active ERESOURCE. 2 - ERESOURCE address. 3 - Pool allocation start address. 4 - Pool allocation size. |

To investigate the pool allocation, use the !pool debugger extension with the starting address of the pool allocation, 9655D468. For example:

|  |
| --- |
| 1: kd> !pool 9655d468 2Pool page 9655d468 region is Paged pool\*9655d468 size: b0 previous size: 8 (Allocated) \*Bug\_ |

To find information about the ERESOURCE, use the !locks debugger extension with the address of the structure. For example:

|  |
| --- |
| 1: kd> !locks 0x9655D4A8Resource @ 0x9655d4a8 Available1 total locks |

In the previous example, ERESOURCE at 0x9655D4A8 lives inside the pool block that is being freed. You can also use the kb debugger command to display a stack trace of the calls that led to the failure. The following example shows the stack, including the call to ExFreePoolWithTag that Driver Verifier intercepted.

|  |
| --- |
| 1: kd> kbChildEBP RetAddr Args to Child92f6374c 82c2c95a 00000003 92f68cdc 00000000 nt!RtlpBreakWithStatusInstruction92f6379c 82c2d345 00000003 9655d468 000000c4 nt!KiBugCheckDebugBreak+0x1c92f63b48 82c2c804 000000c4 000000d2 9655d4a8 nt!KeBugCheck2+0x5a992f63b6c 82e73bae 000000c4 000000d2 9655d4a8 nt!KeBugCheckEx+0x1e92f63b88 82e78c32 9655d4a8 9655d468 000000b0 nt!VerifierBugCheckIfAppropriate+0x3c92f63ba4 82ca7dcb 9655d468 000000b0 00000000 nt!VfCheckForResource+0x5292f63bc8 82e7fb2d 000000b0 00000190 9655d470 nt!ExpCheckForResource+0x2192f63be4 82e6dc6c 9655d470 92f63c18 89b6c58c nt!ExFreePoolSanityChecks+0x1fb92f63bf0 89b6c58c 9655d470 00000000 89b74194 nt!VerifierExFreePoolWithTag+0x2892f63c00 89b6c0f6 846550c8 846550c8 846e2200 buggy!MmTestProbeLockForEverStress+0x2e92f63c18 82e6c5f1 846e2200 846550c8 85362e30 buggy!TdDeviceControl+0xc492f63c38 82c1fd81 82d4d148 846550c8 846e2200 nt!IovCallDriver+0x25192f63c4c 82d4d148 85362e30 846550c8 84655138 nt!IofCallDriver+0x1b92f63c6c 82d4df9e 846e2200 85362e30 00000000 nt!IopSynchronousServiceTail+0x1e692f63d00 82d527be 00000001 846550c8 00000000 nt!IopXxxControlFile+0x68492f63d34 82cb9efc 0000004c 00000000 00000000 nt!NtDeviceIoControlFile+0x2a |

# Locked Memory Page Tracking

Driver Verifier enables the TrackLockedPages option for the operating system when the Pool Tracking option is enabled. For more information about TrackLockedPages, see the Windows Debuggers documentation for bug check 0x76 (PROCESS\_HAS\_LOCKED\_PAGES) and 0xCB (DRIVER\_LEFT\_LOCKED\_PAGES\_IN\_PROCESS).

You can activate the Pool Tracking feature for one or more drivers by using Driver Verifier Manager or the Verifier.exe command line.

To activate Pool Tracking at the command line

* Use a flag value of 0x8 or add 0x8 to the flag value.

For example:

verifier /flags 0x8 /driver MyDriver.sys

The option is active after the next boot.

In Windows Vista, you can also activate and deactivate Pool Tracking without rebooting by adding the /volatile parameter to the command. For example:

verifier /volatile /flags 0x8 /driver MyDriver.sys

This setting is effective immediately, but is lost when you shut down or reboot the computer.

The Pool Tracking feature is also included in the /standard settings. For example:

verifier /standard /driver MyDriver.sys

To activate Pool Tracking by using Driver Verifier Manager

1. Select Create custom settings (for code developers), and then click Next.

2. Select Select individual settings from a full list.

3. Select Pool Tracking.

# Additional Automatic Checks

Driver Verifier performs certain checks whenever it is verifying one or more drivers. You cannot activate or deactivate these checks. In Windows Vista, New Automatic Checks that monitor these incorrect driver actions include the following:

* Calling IoCallDriver with interrupts disabled.
* Calling IoCallDriver above DISPATCH\_LEVEL.
* Returning from driver dispatch routine with interrupts disabled.
* Returning from driver dispatch routine with a changed IRQL.
* Returning from driver dispatch routine with APCs disabled.

The driver might have called KeEnterCriticalRegion more times than KeLeaveCriticalRegion. This is a leading cause of bug check 0x20 (KERNEL\_APC\_PENDING\_DURING\_EXIT).

* MmProbeAndLockPages or MmProbeAndLockProcessPages called on a memory descriptor list (MDL) having incorrect flags.

For example, it is incorrect to call MmProbeAndLockPages for an MDL setup by calling MmBuildMdlForNonPagedPool.

* MmMapLockedPages called on an MDL having incorrect flags.

For example, it is incorrect to call MmMapLockedPages for an MDL that is already mapped to a system address. Another example of incorrect driver behavior is calling MmMapLockedPages for an MDL that was not locked.

* MmUnlockPages or MmUnmapLockedPages called on a partial MDL (created by using IoBuildPartialMdl).
* MmUnmapLockedPages called on an MDL that is not mapped to a system address.

If Driver Verifier is not active, these violations might not cause an immediate system crash in every case. Driver Verifier monitors the driver’s behavior and issues bug check 0xC4 if any of these violations occur. For a list of the bug check parameters, see Bug Check 0xC4 (DRIVER\_VERIFIER\_DETECTED\_VIOLATION).

## Logging Pool Memory Allocate and Free Calls

Automatic Checks also includes a log of kernel pool memory Allocate and Free operations. Stack traces of ExAllocatePoolWithTag calls from the driver that is being verified are recorded in this log. Also, ExFreePool calls from any driver (including those drivers that are not marked to be verified) are recorded in the same log.

This log contains just the recent history of Allocate and Free calls. When the log becomes full, older entries from the log are reused to store newer log entries.

The pool memory log also contains entries for calls of IoAllocateMdl , IoAllocateWorkItem, MmMapLockedPagesSpecifyCache, MmMapIoSpace, MmAllocateContiguousMemorySpecifyCache, MmAllocatePagesForMdl, and MmCreateMdl from verified drivers.

## Viewing the Log of Pool Memory Allocate and Free Calls

The contents of the pool Allocate and Free log can be displayed by using !verifier 0x80 in the kernel debugger.

#### Example 1

Display all the available information about pool address 0xa474eff0:

|  |
| --- |
| 0: kd> !verifier 80 a474eff0Parsing 00010000 array entries, searching for address a474eff0.…Finished reading all pool tracking information.====================================================================Pool block a474efd8, Size 00000028, Thread a6d3ad78819551e2 nt!ExFreePoolWithTag+0xb281f1a385 nt!VerifierExFreePool+0x1f865bf056 volsnap!ExFreeToNPagedLookasideList+0x1e865c910a volsnap!VspFreeWriteContext+0x50865c0681 volsnap!VspReadCompletionRoutine+0x3981f18c2b nt!IovpLocalCompletionRoutine+0xfb81826d3f nt!IopfCompleteRequest+0x17b81f18aec nt!IovCompleteRequest+0x11c85f82a83 CLASSPNP!ClassCompleteRequest+0x1185f84cc7 CLASSPNP!TransferPktComplete+0x2ab81f18c2b nt!IovpLocalCompletionRoutine+0xfb81826d3f nt!IopfCompleteRequest+0x17b81f18aec nt!IovCompleteRequest+0x11c====================================================================Pool block a474efd8, Size 00000028, Thread 8558e92881f1a1c9 nt!VeAllocatePoolWithTagPriority+0x21b81f19c89 nt!VerifierExAllocatePoolWithTag+0x59865bf02d volsnap!ExAllocateFromNPagedLookasideList+0x27865c90b1 volsnap!VspAllocateWriteContext+0x13865c0cf9 volsnap!VolSnapRead+0x7b81f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d81f29dbe nt!ViFilterDispatchGeneric+0x5e81f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d865f9822 Ntfs!NtfsStorageDriverCallout+0x148192b9e9 nt!KiSwapKernelStackAndExit+0x11d865f985d Ntfs!NtfsCallStorageDriver+0x2dParsed entry 00010000/00010000...Finished parsing all pool tracking information. |

In this example, you can see that address 0xa474eff0 was part of a volsnap.sys pool allocation that started at address 0xa474efd8, 0x28 bytes long. The thread at address 0x8558e928 allocated that pool block, and thread 0xa6d3ad78 freed it.

#### Example 2

Display the three most recent operations available in the log:

|  |
| --- |
| 0: kd> !verifier 80 3Parsing 00000003 array entries.Reading pool tracking information - 000000c0 bytes left...Finished reading all pool tracking information.====================================================================Pool block 986ff490, Size 00000020, Thread 855d2cb881955965 nt!ExFreePoolWithTag+0x8358182e588 nt!IoReleaseRemoveLockEx+0xb681f29e09 nt!ViFilterReleaseRemoveLock+0x981f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d865f9822 Ntfs!NtfsStorageDriverCallout+0x148192b9e9 nt!KiSwapKernelStackAndExit+0x11d865f985d Ntfs!NtfsCallStorageDriver+0x2d865fbb82 Ntfs!NtfsPagingFileIo+0x30086608280 Ntfs!NtfsFsdRead+0xac81f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d86114a5c fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x22a====================================================================Pool block 98637390, Size 00000020, Thread 855d2cb881955965 nt!ExFreePoolWithTag+0x8358182e588 nt!IoReleaseRemoveLockEx+0xb681f29e09 nt!ViFilterReleaseRemoveLock+0x981f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x79865c0dcf volsnap!VolSnapRead+0x15181f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d81f29dbe nt!ViFilterDispatchGeneric+0x5e81f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d865f9822 Ntfs!NtfsStorageDriverCallout+0x148192b9e9 nt!KiSwapKernelStackAndExit+0x11d====================================================================Pool block a0cce660, Size 00000020, Thread 855d2cb881955965 nt!ExFreePoolWithTag+0x8358182e588 nt!IoReleaseRemoveLockEx+0xb681f29e09 nt!ViFilterReleaseRemoveLock+0x981f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x7986583ac2 ecache!EcDeviceIoCallLowerDriver+0x1d6865846e8 ecache!EcDispatchReadWrite+0xbe481f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d81f29dbe nt!ViFilterDispatchGeneric+0x5e81f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x79865c0dcf volsnap!VolSnapRead+0x151Parsed entry 00000003/00000003...Finished parsing all pool tracking information. |

#### Example 3

Display all the recent history in the log:

|  |
| --- |
| 0: kd> !verifier 80…====================================================================Pool block 986ff490, Size 00000020, Thread 855d2cb881955965 nt!ExFreePoolWithTag+0x8358182e588 nt!IoReleaseRemoveLockEx+0xb681f29e09 nt!ViFilterReleaseRemoveLock+0x981f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d865f9822 Ntfs!NtfsStorageDriverCallout+0x148192b9e9 nt!KiSwapKernelStackAndExit+0x11d865f985d Ntfs!NtfsCallStorageDriver+0x2d865fbb82 Ntfs!NtfsPagingFileIo+0x30086608280 Ntfs!NtfsFsdRead+0xac81f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d86114a5c fltmgr!FltpLegacyProcessingAfterPreCallbacksCompleted+0x22a====================================================================Pool block 98637390, Size 00000020, Thread 855d2cb881955965 nt!ExFreePoolWithTag+0x8358182e588 nt!IoReleaseRemoveLockEx+0xb681f29e09 nt!ViFilterReleaseRemoveLock+0x981f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x79865c0dcf volsnap!VolSnapRead+0x15181f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d81f29dbe nt!ViFilterDispatchGeneric+0x5e81f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d865f9822 Ntfs!NtfsStorageDriverCallout+0x148192b9e9 nt!KiSwapKernelStackAndExit+0x11d====================================================================Pool block a0cce660, Size 00000020, Thread 855d2cb881955965 nt!ExFreePoolWithTag+0x8358182e588 nt!IoReleaseRemoveLockEx+0xb681f29e09 nt!ViFilterReleaseRemoveLock+0x981f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x7986583ac2 ecache!EcDeviceIoCallLowerDriver+0x1d6865846e8 ecache!EcDispatchReadWrite+0xbe481f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d81f29dbe nt!ViFilterDispatchGeneric+0x5e81f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x79865c0dcf volsnap!VolSnapRead+0x151====================================================================Pool block 98545ed0, Size 00000020, Thread 855d2cb881955965 nt!ExFreePoolWithTag+0x8358182e588 nt!IoReleaseRemoveLockEx+0xb681f29e09 nt!ViFilterReleaseRemoveLock+0x981f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x798043b6f5 volmgr!VmReadWrite+0x1b781f18647 nt!IovCallDriver+0x277818269f3 nt!IofCallDriver+0x1d81f29dbe nt!ViFilterDispatchGeneric+0x5e81f18647 nt!IovCallDriver+0x27781f22409 nt!VerifierIoCallDriverStackSafe+0x7986583ac2 ecache!EcDeviceIoCallLowerDriver+0x1d6865846e8 ecache!EcDispatchReadWrite+0xbe4… |

# Resources

#### Windows Logo Program:

Requirements for Hardware (System and Devices):
<http://www.microsoft.com/whdc/winlogo/hwrequirements.mspx>

#### WHDC Web site:

Driver Verifier

<http://www.microsoft.com/whdc/DevTools/tools/DrvVerifier.mspx>

Debugging Tools for Windows

<http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx>

Windows Driver Foundation

<http://www.microsoft.com/whdc/driver/wdf/default.mspx>

#### Windows Driver Kit:

<http://www.microsoft.com/whdc/driver/WDK/aboutWDK.mspx>

Canceling IRPs

<http://msdn.microsoft.com/en-us/library/ms795761.aspx>

Completing IRPs

<http://msdn.microsoft.com/en-us/library/ms795821.aspx>

Using Volatile Settings

<http://msdn.microsoft.com/en-us/library/ms792869.aspx>

#### White Papers:

I/O Completion/Cancellation Guidelines

[http://www.microsoft.com/whdc/driver/kernel/Iocancel.mspx](http://www.microsoft.com/whdc/driver/kernel/IOcancel.mspx)

Cancel Logic in Windows Drivers

<http://www.microsoft.com/whdc/driver/kernel/cancel_logic.mspx>