**How to Use the Debug Diagnostic Tool v1.1 (DebugDiag) to Debug User Mode Processes**

Mourad Lagdas  
Microsoft Corporation

**CONTENTS**

Introduction

Installation

Components

Usage

Advanced Usage

FAQ

Content Change History

**INTRODUCTION**

When users experience application stability and performance problems such as crashes, hangs, and unexplained high memory usage, the best first step toward a remedy requires looking at the active process at the time the problem occurs. However, server applications like IIS, Exchange, SQL Server, COM+, and Biztalk often provide no user interface information when they fail and subsequently restart, and this complicates this type of troubleshooting.

The right debugging tool can dramatically simplify the isolation of these problem s and the provision of solutions. There are several types of these issues for which the Debug Diagnostic Tool v1.1 (DebugDiag) is a better choice than other debugging tools such as Adplus and UserDump. Following are three types of processing problems that DebugDiag can help identify.

**Crash —** Process crash is defined as an unexpected termination when a process exits abnormally. Typically this is caused by an unhandled exception; however, a crash can also include scenarios where the process detects a problem condition and exits clearly without an exception (for instance, process recycling that is caused by excess memory utilization).

**Hang —** A process hangs when it stops responding completely (deadlock) or takes a long time to respond. Sometimes this slow response is accompanied by high CPU utilization (busy hang).

**High memory usage —** High memory usage or memory leak can cause virtual memory usage in a process to keep growing over time and prevent it from ever returning to normal usage levels. The process can then run out of memory and this can cause it to terminate unexpectedly. During these out-of-memory instances, the virtual memory may fall below 1 Gb, instead of the 2 Gb allowed to Win 32 processes. This problem is sometimes caused by high memory fragmentation.

To determine what causes these issues, you must analyze the process state at the time of the failure. The state of a process could be captured at any time by generating a **userdump**. Userdumps are generated by any Windows debugger and they have the file name extension .dmp, .hdmp or .mdmp. The most frequently used Windows debuggers for user mode processes are Windbg, Cdb, and ntsd. All these programs are built on top of the primary Windows debugger engine, dbgeng.dll, by using the debugger APIs in dbghelp.dll.

Many other debugging tools besides these have been developed to facilitate the debugging process in production environments. The development of these other tools was necessary because it is difficult to use the core Windows debuggers (windbg, cdb, and ntsd) to find the cause of the failure and still make sure that there will be no down time caused by the debugger itself. Some of these debugging tools are: the IIS Exception Monitor, Adplus, IISDebugTools, and IISState.

DebugDiag has combined many of the important features of each of these debugging tools and added a rich UI for ease of use. Some of the best features in DebugDiag include:

* Memory and handle leak tracking
* No Terminal service limitation
* Automatic re-attach to target processes
* Advanced post-mortem analysis of userdumps
* Extensible object model for debugging and analysis
* Userdump generation based on performance counters triggers
* .NET exceptions monitoring

**Debug Diagnostic Tool v1.1 (DebugDiag)**

DebugDiag was designed to help identify common issues with user mode processes such as crashes, hangs, and memory, and to handle leak issues. It was designed and developed by the IIS escalation services team at Microsoft; however, the tool does not target IIS processes only but can be used to debug any user mode process. The tool includes debugging and analysis scripts that can be used to troubleshoot issues in any process, including Web data access components and COM+ components.

DebugDiag provides an extensible object model in the form of COM objects and provides a script host with a built-in reporting framework. Users can enhance the scripts to cover specific areas. It is composed of three main components: the debugging service, the debugger host, and the user interface.  
**Note:** As of the Windows Vista operating system, DebugDiagAnalysisOnly.exe does not require elevation, so the only UI view it contains is the Advanced Analysis view.

DebugDiag has two modes of operation: it can be used as a command-line utility (Dbghost.exe), or it can be used from the user interface (DebugDiag.exe).

**Terms and Definitions**

Within the context of DebugDiag, a **rule** is a set of actions for the debugger to execute against a target process when some conditions are met. These actions are called a control script and are in the form of VBScript statements arranged in a file with a .vbs file name extension. There are three kinds of rules: crash rule, hang rule, and memory and handle leak rule. The crash rule has its own control script (<rule name>.vbs) whereas a hang rule and a memory and handle leak rule update the default service control script (dbgsvc.vbs).

An Analysis Script is a .asp file with VBScript statements that loads the built-in COM objects for analysis and reporting; it runs under the debugger host (Dbghost.exe). There are two default analysis scripts shipped with the tool (CrashHangAnalysis.asp and MemoryAnalysis.asp). These scripts could be modified to enhance analysis of specific areas.

**INSTALLATION**

**Minimum System Requirements**

DebugDiag is a very lightweight tool. It requires less than 19MB of disk space. It installs on Windows NT, Windows 2000, Windows XP, Windows Server 2003, and Windows Vista. It has not been tested on Windows Server 2008.

**Obtaining DebugDiag**

You can download the latest version of DebugDiag (version 1.1) from the Microsoft download center [www.microsoft.com/downloads](http://www.microsoft.com/downloads) at:

<http://www.microsoft.com/downloads/info.aspx?na=22&p=1&SrcDisplayLang=en&SrcCategoryId=&SrcFamilyId=&u=%2fdownloads%2fdetails.aspx%3fFamilyID%3d28bd5941-c458-46f1-b24d-f60151d875a3%26DisplayLang%3den>

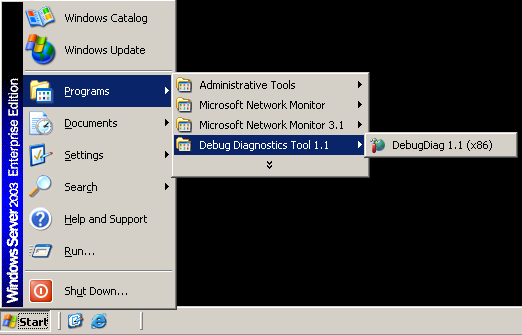
or from [www.iis.net](http://www.iis.net) at: <http://www.iis.net/downloads/default.aspx?tabid=34&g=6&i=1286>

**Installing DebugDiag** by using **the MSI**

Using the DebugDiag installation wizard is straightforward.

* You must be the local administrator or a member of the Administrators group to install DebugDiag.
* You will be presented with two EULAs during the wizard (one for the tool and the other one for the use of the Microsoft Public Symbol Server).
* If DebugDiag 1.1 is installed on a Windows Vista computer, an executable called **DebugDiagAnalysisOnly.exe** is added. This means that the tool can only be used for analysis.

To start DebugDiag, click “***Start 🡪 Programs 🡪 Debug Diagnostic Tool 1.1 🡪 DebugDiag 1.1 (x86)”***



**Installing DebugDiag Manually for Enterprise deployment**

1. Install DebugDiag on a workstation.
2. Copy the \DebugDiag folder to the destination.
3. Run the following commands:
   * Dbgsvc /service
   * Dbghost /regserver
   * Regsvr32.exe complusDDExt.dll
   * Regsvr32.CrashHangExt.dll
   * Regsvr32.exe IISInfo.dll
   * Regsvr32.exe MemoryExt.dll

**Troubleshooting installation**

Most DebugDiag 1.1 installation issues occur because previous versions (1.0 and beta releases) are installed on the target computer.

* Make sure that any previous version of DebugDiag is uninstalled. The MSI for 1.1 halts if it detects a previous version. Also note that DebugDiag 1.0 was released in two different packages: a stand-alone package and as part of the IIS Debug Tools. In both cases, make sure that DebugDiag 1.0 is successfully uninstalled before you install DebugDiag 1.1.
* Make sure that the installation is made under an administrator account if you are using COM/COM+ components. Use [Process Monitor](http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx) to find the root cause of the install failure.
* Use MSI executable (Msiexec.exe) with error logging to find the cause on an install failure. (At the command line, type **msiexec /?** for information about how to use verbose mode and error logging.)

**COMPONENTS**

The Debug Diagnostic Tool is composed of three main components:

***The Debugging Service*** - DbgSvc.exe

The **debugging service**, DbgSvc.exe, interacts with the debugger host by sending it appropriate commands when acting on specific system events. These events—and the actions associated with these events—constitute the default control script (DbgSvc.vbs) for the debugger service.

The debugging service performs the following tasks:

* Collects performance monitor data.
* Attaches/Detaches the debugger host to processes.
* Implements HTTP ping to detect hangs when debugging IIS.
* Injects Leaktrack.dll to monitor for memory leaks.
* Collects debugging session state information and saves it to the file “ServiceState.xml.”
* Shows the state of each rule defined.

DbgSVC\_Date\_\_05\_11\_2008\_\_Time\_10\_54\_54PM\_\_693\_\_Log.txt

Any time the service starts; it will start perfmon logging for the following counters: Memory, Process, Processor, System, ASP.NET, Active Server Pages, Web Service, Internet Information Services Global, TCP, TCPV4, TCPV6 . This feature can be disabled.

[5/12/2008 12:25:15 AM] DbgSVC started

[5/12/2008 12:25:18 AM] New process found: Process Name - System Process Process ID - 0 Process Identity - SYSTEM

[5/12/2008 12:25:18 AM] New process found: Process Name - svchost.exe Process ID - 748 Process Identity - NT AUTHORITY\SYSTEM

[5/12/2008 12:25:18 AM] New process found: Process Name - spoolsv.exe Process ID - 984 Process Identity - NT AUTHORITY\SYSTEM

[5/12/2008 12:25:18 AM] New process found: Process Name - msdtc.exe Process ID - 1008 Process Identity - NT AUTHORITY\NETWORK SERVICE

[5/12/2008 12:25:18 AM] New process found: Process Name - vmsrvc.exe Process ID - 1100 Process Identity - NT AUTHORITY\SYSTEM

…

…

[5/12/2008 9:35:59 PM] Process Exited: Process Name - DbgHost.exe Process ID - 3228

[5/12/2008 10:34:32 PM] New process found: Process Name - wpabaln.exe Process ID - 3988 Process Identity - WEB01-PROD\Administrator

[5/12/2008 10:34:32 PM] Process Exited: Process Name - wpabaln.exe Process ID - 1692

[5/12/2008 11:00:15 PM] New process found: Process Name - msiexec.exe Process ID - 3992 Process Identity - WEB01-PROD\Administrator

[5/12/2008 11:38:09 PM] Service state changed: Service Name - WinHttpAutoProxySvc Process ID - 732 Current State - SERVICE\_RUNNING

[5/12/2008 11:38:18 PM] New process found: Process Name - wmiprvse.exe Process ID - 4016 Process Identity - NT AUTHORITY\NETWORK SERVICE

[5/12/2008 11:49:02 PM] Process Exited: Process Name - wmiprvse.exe Process ID - 4016

[5/12/2008 11:54:36 PM] Service state changed: Service Name - WinHttpAutoProxySvc Process ID - 0 Current State - SERVICE\_STOPPED

[5/13/2008 1:12:49 AM] Process Exited: Process Name - msiexec.exe Process ID - 3992

Request to reload control script received.

Request to reload control script received.

[5/13/2008 1:13:15 AM] Process Exited: Process Name - DebugDiag.exe Process ID - 216

[5/13/2008 1:13:23 AM] DbgSVC stopped

The main class that the service exposes is the “Controller” class.

The “Controller” class exposes 2 interfaces:

IController (Write, HTTPPinger, ReloadScript…)

\_IControllerEvents (OnStart, OnShutdown, OnProcessExited…)

Any time the service starts, it creates the log file. The general syntax of the name of the log file is “DbgSvc \_\_Date\_\_<Date>\_\_Time\_\_<Time>\_\_Log.txt DbgSvc\_xxx\_Log.txt under the “\Program Files\DebugDiag\Logs” folder. The log file contains all events related to processes and services system wide as well debugger events and errors that the service may encounter.

The log file looks like the following:

***The Debugger Host*** – Dbghost.exe

The **debugger host** hosts the Windows Symbolic Debugger Engine (dbgeng.dll) to attach to processes and generate userdumps.

DbgHost.exe also hosts the main analyzer module used to analyze userdumps.

For each process debugged or memory dump analyzed there is an instance of DbgHost.exe running.

Dbghost.exe is spawned by either the user interface or the service or even directly from the command line. (For command line usage, run “**DbgHost /?**”.)

DbgHost.exe exposes three main classes:

**DbgControl**: Attach/Detach from processes or open/analyze a memory dump.

**DbgObj:** Collect process or memory dump information.

**Manager:** Output analysis data to the report file.

Dbghost.exe has no dependency on the DbgSvc.exe service and can be used separately.

***The user interface*** – DebugDiag.exe/DebugDiagAnalysisOnly.exe

The user interface presents an interface to analyze memory dumps, automates the creation of control scripts, and shows the status of running processes and services.

To Start the user interface, go to Start/All Programs/Debug Diagnostics Tool 1.1/DebugDiag 1.1 (x86).

The user interface provides these three views:

* **Rules:** This view helps create and update control script for the debugger host by using a wizard. The script is located under the \scripts folder.
* **Advanced Analysis:** In this view, userdump analysis is performed. Two types of analysis are available to run: Crash/Hang analyzer is implemented by the “CrashHangAnalysis.asp” analysis script, and the Memory Pressure Analyzers are implemented by the “MemoryAnalysis.asp” analysis script.
* **Processes:** Shows status of running processes and services. Similar to task manager with extra features such as attaching the debugger, generating userdumps, or starting tracking for memory leaks.

The user interface is not required to run the debugger host or the debugging service.

**USAGE**

**Adding Rules**

Debugging a high CPU issue (busy hang) is totally different from debugging an unexpected termination of a process, and in each scenario, DebugDiag is used differently.

DebugDiag uses the concept of a “rule” to approach crashes, hangs, and memory and handle leak issues. As explained previously, a rule is a set of actions that the debugger host or the debugging service will execute when certain conditions are met. These actions are called the control script and are in the form of VBScript statements that manipulate the COM objects exposed by the service or the host. The control script is stored in a VBS file.

There are three kinds of rules:

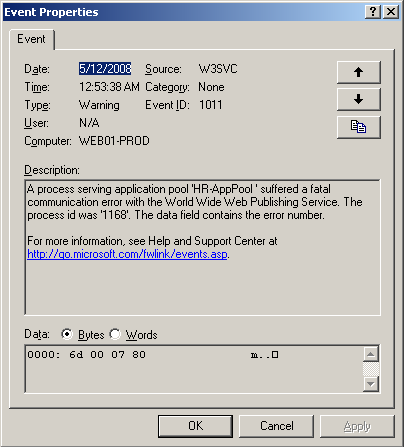
***Crash Rule***

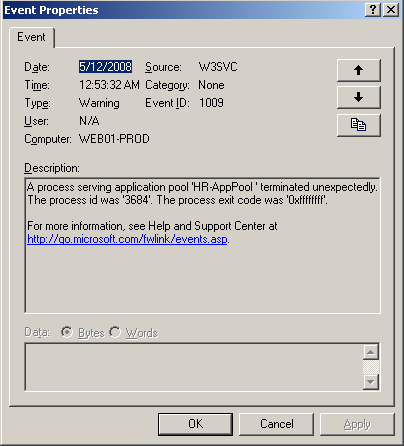
A crash rule is needed when the issue being debugged is an unexpected termination of a process.

When a process encounters an unhandled exception, the OS terminates the process. In general, the goal of using a debugger to resolve such an issue is to let you know where the unhandled exception occurred. This is shown when the debugger first attaches to the target process that experiences the crash. The debugger will then monitor the target process execution for unhandled exceptions, and once the unhandled exception occurs, process execution halts, and the debugger gets control. The debugger can then perform any needed actions, such as displaying the call stack of the faulting thread that caused the unhandled exception or generating a userdump.

So, when a crash rule is created, DebugDiag spawns the debugger host “Dbghost.exe” that attaches to and monitors the target process for unhandled exceptions. When it encounters one, it generates a userdump. This userdump is used later for post-mortem analysis.

For example, when a worker process terminates unexpectedly in IIS, you see events entries such as the following in the event log:





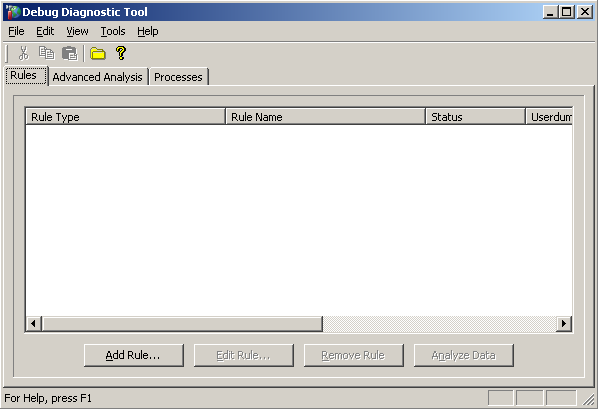
This series of event dialog boxes shows that the W3wp.exe process that serves the HR-AppPool application pool terminated unexpectedly because of an unhandled exception.

To address this problem, you would create a crash rule against the application pool “HR-AppPool” or directly against a newly spawned W3wp.exe that serves the “HR-AppPool” application pool.

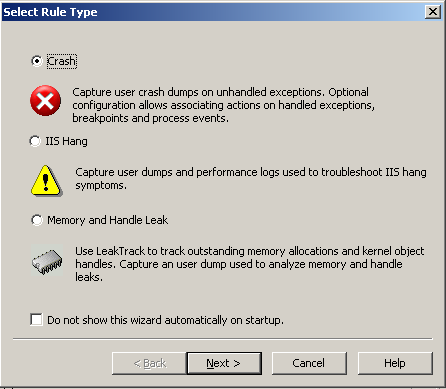
A default crash rule is sufficient in this scenario.

Following is a closer look at the crash rule creation wizard:

1. Start DebugDiag user interface,

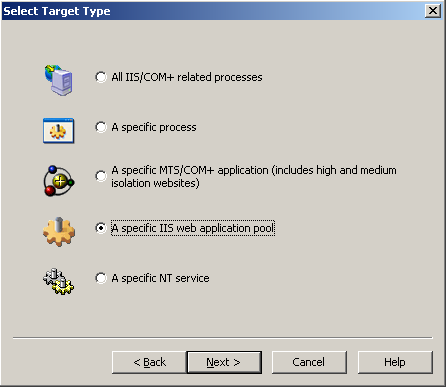


1. The first time DebugDiag UI is opened, it shows the ***Select Rule Type*** dialog box.



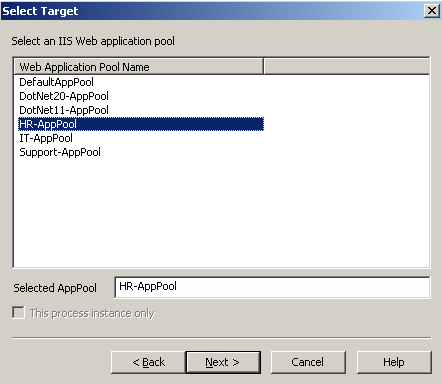
You can choose not to display this dialog box by selecting “Do not show this wizard automatically on startup”. The same dialog box displays when you click **Add Rule** in the main DebugDiag UI.

1. Select **Crash** and click **Next**.

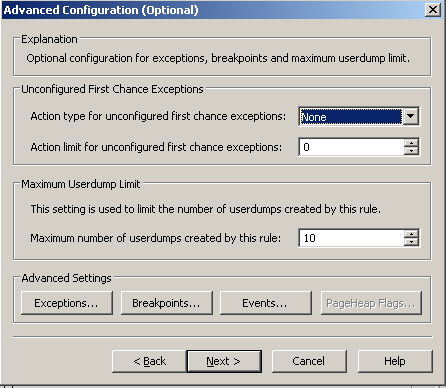


1. In the ***Select Target Type*** dialog box, select **A specific IIS Web application pool** and click **Next**.

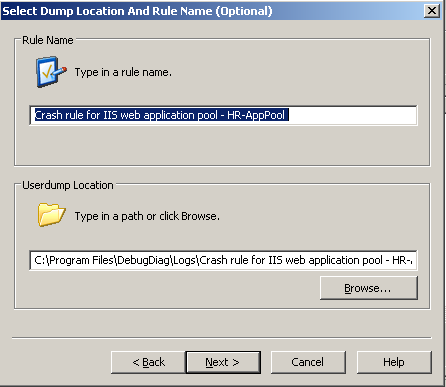
This selection is the appropriate one in this scenario, because it lets you monitor worker processes that serve the “HR-AppPool” only. (More discussion of all available options are in the **Discussion** section.)



1. Select the **HR-AppPool** in the ***Select Target*** dialog box, and click **Next.**



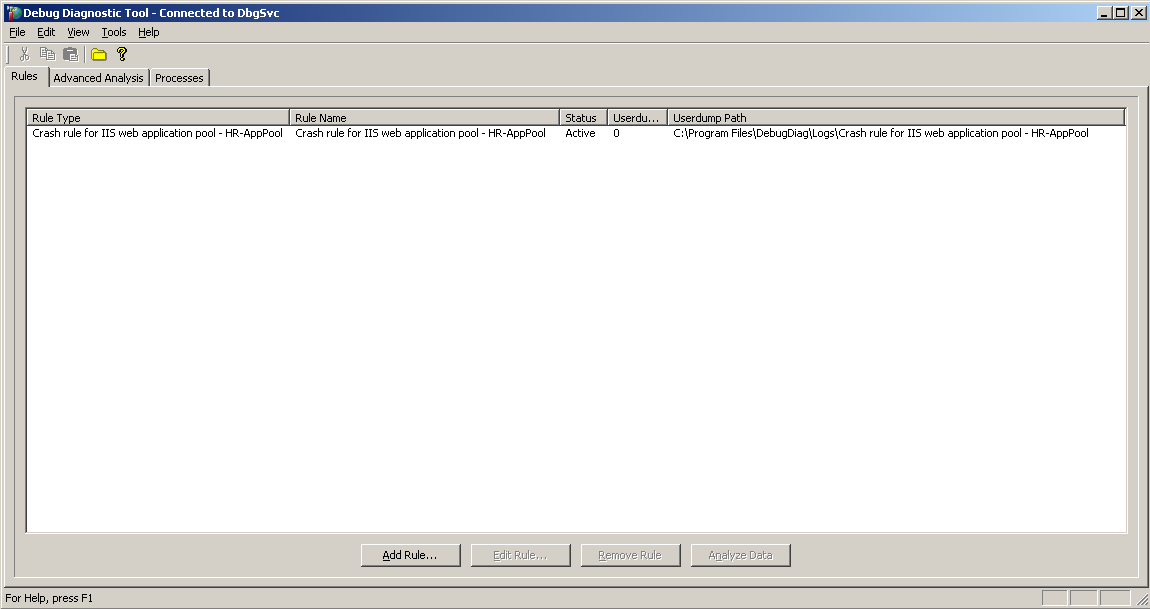
1. Accept all the defaults in the ***Advanced Configuration (Optional)*** dialog box, and click **Next**,



1. Accept the defaults or make the necessary changes in the ***Select Dump Location and Rule Name (Optional)*** dialog box, and click **Next**.



1. In the ***Rule Completed*** dialog box, select **Activate the rule now**, and click **Finish**.



The rule now appears activated in the Rules view in the DebugDiag UI.

Once the rule is activated, the debugging service is started automatically—if it had previously been stopped. A new instance of the debugger host, Dbghost.exe, is spawned and attached to the target process(es). The rule will be active as long as it is not manually deactivated, removed, or failed for some reason. Even if the target processes are restarted or if the server is rebooted, the rule will be still active. So, in the case where you are troubleshooting a random unexpected termination, you do not have to reconfigure DebugDiag with a new rule any time the server is rebooted.

**Discussion**

Using he Add the crash rule Wizard will create a new control script for the debugger. The above crash rule will generate a control script with the name “CrashRule\_WebAppPool\_HR-AppPool.vbs”, which is located under the “\Program Files\DebugDiag\Scripts” folder.

The control script looks like the following:

CrashRule\_WebAppPool\_HR-AppPool.vbs

Ss Debugger.DumpPath = "C:\Program Files\DebugDiag\Logs\Crash rule for IIS Web application pool - HR-AppPool "

RuleKey = "Crash-TT-3-TNAME-HR-APPPOOL "

Set DbgState = Debugger.State

Dim ServiceController

Dim ServiceState

Sub WriteToLog(ByVal Output)

Debugger.Write "[" & Now() & "] "

Debugger.Write Output & vbLf

End Sub

Sub CreateDump(ByVal DumpReason, ByVal bMiniDump)

Dim DumpLimitVarName

Dim DumpCountVarName

Dim DumpLimit

Dim DumpCount

If ServiceState Is Nothing Then

DumpName = Debugger.CreateDump(DumpReason, bMiniDump)

WriteToLog "Created dump file " & DumpName

Else

DumpLimitVarName = RuleKey & "\_DUMP\_LIMIT"

DumpCountVarName = RuleKey & "\_DUMP\_COUNT"

DumpLimit = CInt(ServiceState(DumpLimitVarName))

DumpCount = CInt(ServiceState(DumpCountVarName))

If DumpCount < DumpLimit Then

DumpName = Debugger.CreateDump(DumpReason, bMiniDump)

WriteToLog "Created dump file " & DumpName

DumpCount = DumpCount + 1

ServiceState(DumpCountVarName) = DumpCount

If DumpCount = DumpLimit Then

WriteToLog "Crash rule dump limit of " & DumpLimit & " reached. No more dump files will be created"

End If

End If

End If

End Sub

Sub AdjustDumpCountOnUnhandledException()

Dim DumpCountVarName

Dim DumpCount

If Not ServiceState Is Nothing Then

DumpCountVarName = RuleKey & "\_DUMP\_COUNT"

DumpCount = CInt(ServiceState(DumpCountVarName))

DumpCount = DumpCount + 1

ServiceState(DumpCountVarName) = DumpCount

End If

End Sub

…

….

This crash rule that you have just added is all that you need to monitor the worker process serving the HR-AppPool after an unexpected termination. Once the rule is activated, an instance of the debugger host (Dbghost.exe) is spawned, and it is attached to the W3wp.exe serving HR-AppPool if that W3wp.exe is running. Otherwise, the debugging service (Dbgsvc.exe) will wait until W3wp.exe runs before it spawns the debugger host.

Once attached, Dbghost.exe monitors W3wp.exe serving the HR-AppPool for any unhandled exceptions.

A log file will be created under “\Program Files\DebugDiag\Logs” by the name “w3wp\_\_PID\_\_168\_\_Date\_\_05\_12\_2008\_\_Time\_01\_19\_13AM\_\_404\_\_Log.txt” where the debugger hosts logs events and exceptions of interest that the target process encounters. If the debugger host itself encounters errors while attached to the target process, it will log that into the log file.

A new log file is generated any time the debugger host attaches to the target process. The general syntax of the name of the file is “<ProcessName>\_\_PID\_\_<PID>\_\_Date\_\_<Date>\_\_Time\_\_<Time>\_\_Log.txt”. The log file looks like:

DbgSVC\_Date\_\_05\_11\_2008\_\_Time\_10\_54\_54PM\_\_693\_\_Log.txt

DumpPath set to C:\Program Files\DebugDiag\Logs\Crash rule for IIS Web application pool - HR-AppPool

[5/12/2008 1:19:13 AM] Process created. BaseModule - c:\windows\system32\inetsrv\w3wp.exeBaseThread System ID - 1168

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 1820

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 1968

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 3520

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 2148

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 2492

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 1900

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 4080

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 3940

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 1540

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 772

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\ntdll.dll loaded at 0x7c800000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\kernel32.dll loaded at 0x77e40000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\msvcrt.dll loaded at 0x77ba0000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\ADVAPI32.dll loaded at 0x77f50000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\RPCRT4.dll loaded at 0x77c50000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\Secur32.dll loaded at 0x76f50000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\USER32.dll loaded at 0x77380000

….

….

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 128

[5/12/2008 1:19:13 AM] Initializing control script

[5/12/2008 1:19:13 AM] Clearing any existing breakpoints

[5/12/2008 1:19:13 AM]

[5/12/2008 1:19:13 AM] Current Breakpoint List(BL)

[5/12/2008 1:19:13 AM] Thread exited. Exiting thread system id - 128. Exit code - 0x00000000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\inetsrv\gzip.dll loaded at 0x685b0000

[5/12/2008 1:19:13 AM] C:\Inetpub\wwwroot\DebugDiagLabs\www\App\badfil.dll loaded at 0x01b10000

[5/12/2008 1:19:13 AM] C:\WINDOWS\system32\mpr.dll loaded at 0x71bd0000

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 3716

[5/12/2008 1:19:13 AM] \\?\C:\Inetpub\wwwroot\HR\badEXT.dll loaded at 0x01b20000

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 704

[5/12/2008 1:19:13 AM] Thread created. New thread system id - 3980

[5/12/2008 1:19:20 AM] First chance exception - 0xc0000005 caused by thread with system id 1540

[5/12/2008 1:19:28 AM] Second chance exception - 0xc0000005 caused by thread with system id 1540

[5/12/2008 1:19:28 AM] Thread exited. Exiting thread system id - 1168. Exit code - 0xffffffff

[5/12/2008 1:19:28 AM] Thread exited. Exiting thread system id - 1820. Exit code - 0xffffffff

[5/12/2008 1:19:28 AM] Thread exited. Exiting thread system id - 1968. Exit code - 0xffffffff

[5/12/2008 1:19:28 AM] Thread exited. Exiting thread system id - 3520. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 2148. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 2492. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 1900. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 4080. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 3940. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 772. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 3716. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 704. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Thread exited. Exiting thread system id - 3980. Exit code - 0xffffffff

[5/12/2008 1:19:29 AM] Process exited. Exit code - 0xffffffff

When W3wp.exe encounters an unhandled exception, Dbghost.exe gets control and creates a userdump. The debugger (Dbghost.exe) and the debuggee are both of them exited after that.

The DebugDiag UI keeps a count of the number of userdumps generated. In the default crash rule that was added in the earlier scenario, there is a limit of 10 userdumps that the rule could create. DebugDiag imposes this limit so that the hard disk drive does not run out of space. The size of a full userdump is the size of the virtual memory the target process was consuming at the time the userdump was generated. If your W3wp.exe was consuming 400MB Virtual memory at the time the unhandled exception occurred, then the size of the userdump would have been approximately 400MB.

A rule will stop generating new userdumps when the maximum number is reached, but there still will be an instance of Dbghost.exe attached to the W3wp.exe serving the HR-AppPool as long as the rule is active.

Following is a brief overview of the other options that are available during the crash rule generation and a description of the scenarios in which they might be used.

The ***Select Target Type*** dialog box provides the following choices:

**All IIS/COM+ related processes**

If this option is chosen, all IIS/COM+ processes will be monitored for unhandled exceptions.

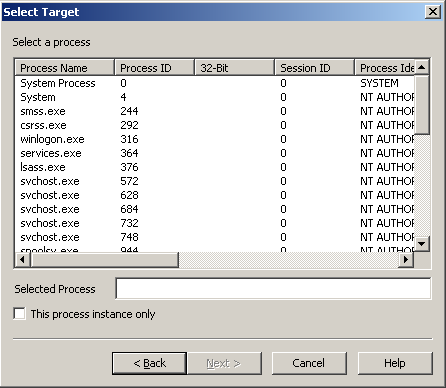
IIS/COM+ processes are: Inetinfo.exe, Dllhost.exe Aspnet\_wp.exe, W3wp.exe , and Svchost.exe (hosting WWW and RpcSS).

Use this option if you are trying to capture failures in multiple processes at the same time, or if you know the failure is IIS related but you are unsure which process to attach the debugger host to.

**A specific process**

If you select this option, it will provide a list of all running processes to choose from. This helps you troubleshoot when the process is known and you need to attach the debugger host to all instances of the process with the same name.

The process you create the rule for does not have to be in the list. You can type the name of the process in the **Selected Process** dialog box. When the rule is activated, and once the process starts, the debugger host will be attached to it. Please note that if the process fails at startup (never finishes the startup routines), a normal crash rule will not help. In this case the Pre-Attach feature should be used.



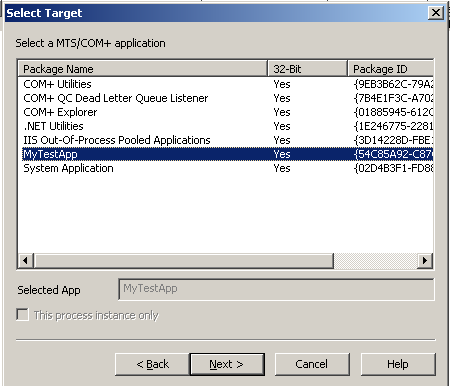
For example, if you had selected the W3wp.exe serving HR-AppPool in the crash rule that was added previously, two options would occur. First, if **This process instance only** is checked, then the target process being debugged would be only the existing W3wp.exe and the rule would be complete once that W3wp.exe is exited or terminated. On the other hand, if this check box is left cleared, the rule would cause all instances of W3wp.exe that were running on the server would be targeted for debugging. The disparity between these options explains why you should choose a specific IIS Web application pool instead of this process.

**Note**:

It is necessary to choose this section along with checking **This process instance only** if there is an indication that the unhandled exception is caused by heap corruption. More details on debugging heap corruption issues can be found in the Advanced Usage section.

**A specific MTS/COM+ application (includes high and medium isolation Web sites)**

Choose this option if you are debugging a specific MTS/COM+ package.

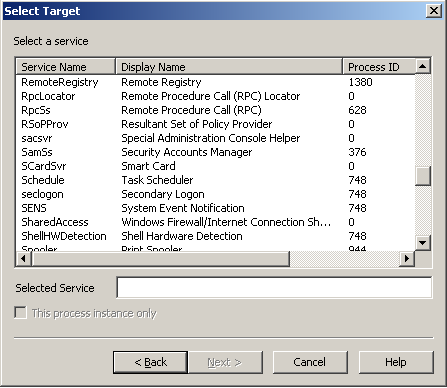


**A specific IIS Web application pool**

This option is available only on Windows Server 2003 and above. It allows you to debug only instances of W3wp.exe that serve a specific application pool.

**A specific NT Service**

This option allows you to pick a specific service to attach to.



Similarly, choose the **A specific IIS Web application Pool** option when you want to debug all process instances that run a specific service.

There are a number of advanced configuration parameters available by using this option. For example, the default setting was used in the crash rule added previously, and that setting is acceptable in most cases. (It generates a userdump when an unhandled exception is encountered.) However, in many situations, you need to configure DebugDiag to collect data in different states of process execution.

Before describing the different available configuration settings, the following DebugDiag terminology will be useful to know:

First Chance Exception and Second Chance Exception.

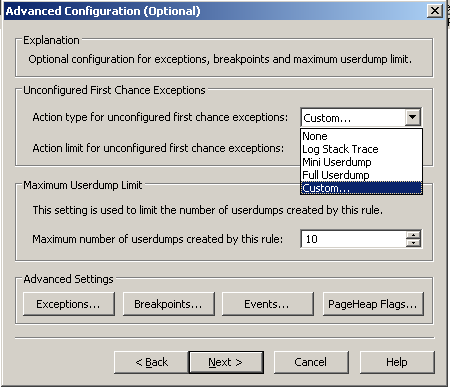
If a debugger is attached to a target process, every time an exception occurs during process execution, the debugger gets a notification before the exception handlers in the process are notified. Since the debugger is notified first to handle the exception, this notification is called as the first chance exception. If the exception handlers in the process are notified and don’t handle the exception, the debugger is notified for the second time and the exception now is a second chance exception or an unhandled exception.

Full Userdump and Mini Userdump

A full userdump contains the entire memory space of a process. It contains the program’s executable image, the handle table, and other necessary information. The size of the full userdump is quite large. A mini userdump or minidump, on the other hand, contains less information and the size is considerably smaller.

**Configuration Settings**

The first available setting is **Unconfigured First Chance Exceptions**; the default is **None,** and the available options appear as follows:



**Log Stack Trace**

This option makes the debugger host write the call stack backtrace of the thread that caused the first chance exception to the rule’s log file. If the debug symbols are not lined up properly, the callstack back-trace is not displayed properly. (Please consult Help for more information about how to use symbols). This option can help you find modules that loaded within the process that is throwing exceptions.

**Mini Userdump**

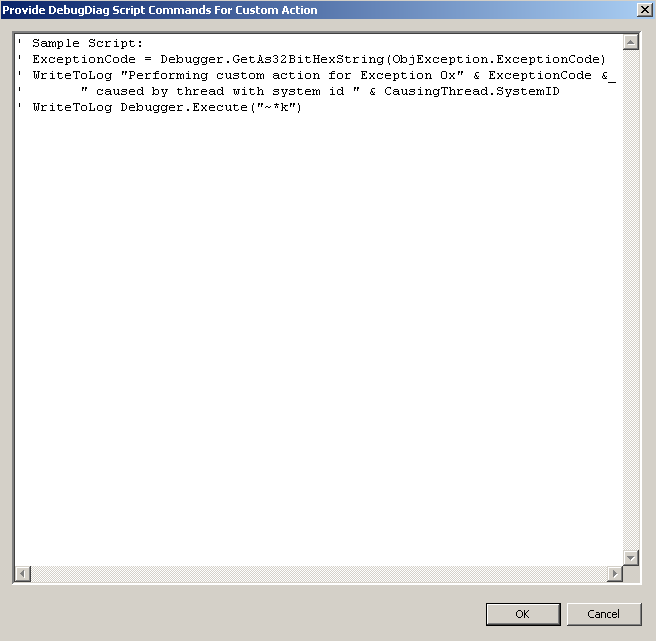
This option instructs the debugger host to generate a minidump any time a first chance exception is encountered. This is useful if you would like to debug the first chance exceptions by doing post-mortem analysis using a Windows core debugger. (DebugDiag does not analyze minidumps.)

**Full Userdump**

This option instructs the debugger host to generate a full userdump any time a first chance exception is encountered. This option is rarely of use, and it is costly in terms of process performance.

**Custom**

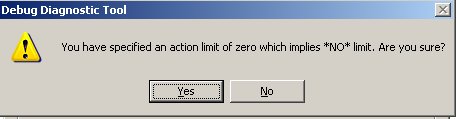
This option instructs the debugger to run custom debugger commands when a first chance exception is encountered. When you choose this option, the following dialog box opens:



**Note:** In most cases, you don’t need to configure first chance exceptions as they are rarely the cause of unexpected process termination.

You can use any supported debugger commands to execute when the first chance exception is encountered. The debugger commands should be passed to the Execute method of the Debugger object—that is, the*WriteToLog Debugger.Execute(“u eip”)—*to display the disassembly code of the instruction loaded in the eip register that caused the first chance exception. The display shows the log file of the crash rule. In addition to debugger commands, you should be able to run in a VBScript statement. Please see Help for all exposed objects and how to use them.

After you configure an action to be performed, set the number of times that action should be executed in **Action limit for unconfigured first chance exceptions**. If you would like the action to be executed as long as the process is alive and the debugger host is attached, choose number 0. You will be then presented with the following warning:



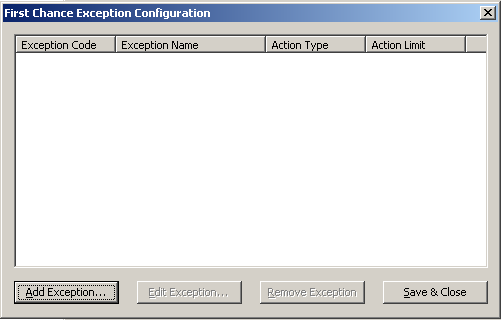
The second available setting is **Maximum Userdump Limit**. This setting governs all userdump generation for the rule. It overrides any other setting and cannot be set to unlimited.

The last setting available in the **Advanced Configuration (Optional)** dialog box is the **Advanced Settings** that let you configure specific exceptions, breakpoints, and events and set appropriate Pageheap flags if needed.

**Exceptions**

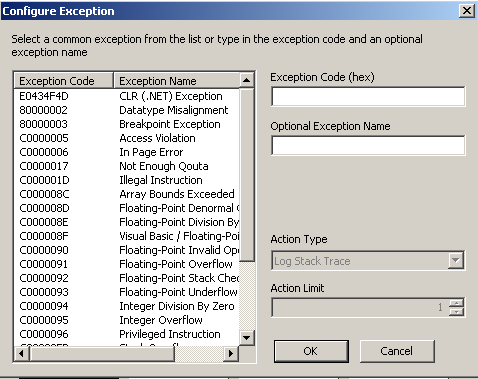
In many scenarios, it is necessary to configure DebugDiag to take actions on specific exceptions. This means that you know in advance the exception code you want the debugger to act on. For example, in the .NET world, assume that you would want to get more information (by looking at a userdump) when a specific .NET exception is thrown by the process. If you are looking for the cause of a System.Threading.ThreadAbortException and the information you received in the event log for that exception was not enough to draw a conclusion, here is how to achieve the configuration that you must have:

After you click **Exceptions**, the following dialog box is displayed:

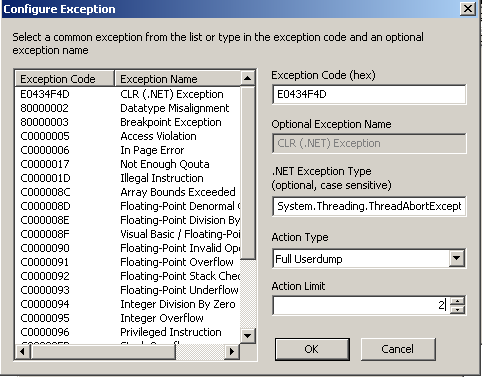


Click **Add Exception**.

The following dialog box is displayed:



Select the first exception in the list of known exceptions, **E0434F4D**, and type the name of the exception (case sensitive) in **.NET Exception Type**. In this case, type **System.Threading.ThreadAbortException**. In the **Action Type**, choose a **Full Userdump** and enter **2** in **Action Limit**.



Once the crash rule is activated, the first two System.Threading.ThreadAbortException exceptions that the process throws will trigger the debugger host to generate full userdumps. You can add as many exceptions as you need to monitor.

**Note:** the Crash Rule Wizard generates the control script for the debugger host.

In, here is how the control script looks for the .NET exception just described:

**Note:** the “additions” are made to the ***Debugger\_OnException()*** function.

CrashRule\_WebAppPool\_HR-AppPool.vbs

Debugger.DumpPath = "C:\Program Files\DebugDiag\Logs\Crash rule for IIS Web application pool - Support-AppPool"

RuleKey = "Crash-TT-3-TNAME-SUPPORT-APPPOOL"

…

…

Sub Debugger\_OnException(ByVal ObjException, ByVal CausingThread, ByVal FirstChance)

ExceptionCode = Debugger.GetAs32BitHexString(ObjException.ExceptionCode)

If FirstChance Then

WriteToLog "First chance exception - " & ExceptionCode & \_

" caused by thread with system id " & CausingThread.SystemID

Else

WriteToLog "Second chance exception - " & ExceptionCode & \_

" caused by thread with system id " & CausingThread.SystemID

AdjustDumpCountOnUnhandledException

Exit Sub

End If

ExceptionCode = UCase(ExceptionCode)

Select Case ExceptionCode

Case "0XE0434F4D"

CLRExceptionType = GetCLRExceptionType(GetCLRExceptionPtr(CausingThread), False)

WriteToLog "CLR Exception Type - '" & CLRExceptionType & "'"

Select Case CLRExceptionType

Case "System.Threading.ThreadAbortException"

If DbgState("Exception\_E0434F4D:SYSTEM.THREADING.THREADABORTEXCEPTION\_ACTION\_COUNT") < 2 Then

CreateDump "First Chance System.Threading.ThreadAbortException", false

DbgState("Exception\_E0434F4D:SYSTEM.THREADING.THREADABORTEXCEPTION\_ACTION\_COUNT") = DbgState("Exception\_E0434F4D:SYSTEM.THREADING.THREADABORTEXCEPTION\_ACTION\_COUNT") + 1

If DbgState("Exception\_E0434F4D:SYSTEM.THREADING.THREADABORTEXCEPTION\_ACTION\_COUNT") >= 2 Then

…

…

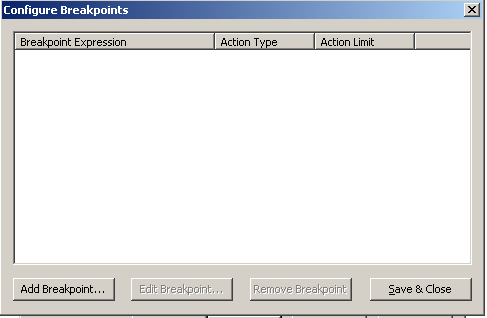
What happens behind the scenes is that the debugger host loads the .NET debugger file name extension “sos.dll” along with the matching mscordacwks.dll from the .NET framework location of the loaded version in the target process. Anytime a .NET exception happens, the debugger host gets the exception address and runs the “!dumpobj” against it. It finally compares the output of “Dumpobj” to the configured .NET exception and takes appropriate actions.

**Breakpoints**

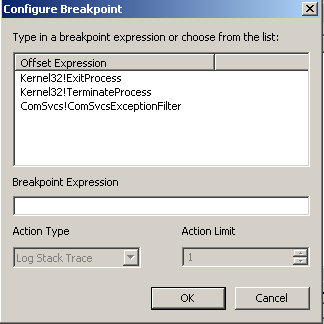
DebugDiag makes it easy to set breakpoints on function calls. Setting breakpoints is helpful when you want the debugger to take action at the time a function is called. To set a breakpoint on a function call inside a module, you need to have the debug symbols of the module unless the function is an export function.

To set breakpoints:

1. Click **Breakpoints**. The following dialog box is displayed:



1. Click **Add Breakpoint**…. The following dialog box is displayed:



As you can see, DebugDiag comes with three breakpoints that you may want to set. These breakpoints are very useful and help track issues when a process was terminated by using a direct call to Kernel32!ExitProcess or Kernel32!TerminateProcess or when a COM+ package issues a “failfast.”

For example, when an ASP.NET application detects that requests are taking too long to execute, it assumes something must be seriously wrong and terminates the process by calling Kernel32!TerminateProcess.

In the case of a COM+ package, the COM+ unhandled exception handler calls the ComSvcs!ComSvcExceptionFilter to failfast the process instead of terminating it.

Because of this, you should set both Kernel32 breakpoints in your crash rule. If the target process is a COM+ package, add the ComSvcs breakpoint as well.

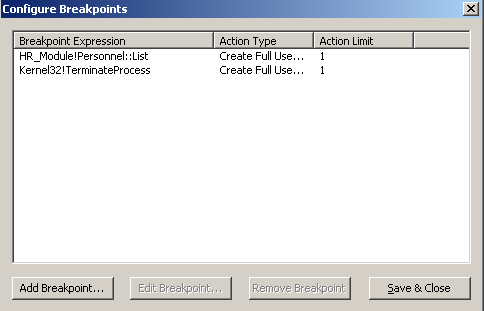
If you are debugging an IIS worker process, make sure that you have set both Kernel32 breakpoints. Likewise, if recycling is turned on for the application pool, any time the recycle happens, the debugger host will execute the action set for the Kernel32!ExitProcess breakpoint (that is, it will generate a userdump). Therefore, any time you are debugging a process, turn off any health monitoring against that process and anything else that may disrupt the debugging process.

For example, set a breakpoint on a custom module called HR\_Module that gets loaded by the worker process that serves the application pool HR-AppPool in the previous crash rule. If the function name is List() and it is inside a class called “Personnel,” then the offset expression of the breakpoint is HR\_Module!Personnel::List.

So in the ***Configure Breakpoint*** dialog box, select **Kerenel32!TerminateProcess,** and then select the **Action Type** to be **Full Userdump** and set the **Action Limit** to **1**.

After you click **OK**, click **Add breakpoint…** in the ***configure Breakpoints*** dialog box, and enter **HR\_Module!Personnel::List** in the **Breakpoint Expression** text box and in the **Action Type** choose **Full Userdump** and leave **1** as the default in **Action Limit**.

The resulting breakpoint configuration looks like the following.



So this configuration tells the debugger host to generate a full userdump any time the function **Kernel32!TerminateProcess** or **HR\_Module!Personnel::List** is called.

**Note:** the debugger host does not need the debug symbols for kernel32.dll and comsvcs.dll in order to resolve the function calls in the predefined breakpoints. However, if the functions are for custom modules and are not export functions, you must supply the debugger host with the location of the symbols from before it will be able to resolve function names in the breakpoints. (Please consult DebugDiag Help for more information about how to use symbols.)

When the breakpoints are set, the control script for the generated crash rule looks like the following:

Note that the breakpoints get set first in the ***Debugger\_OnInitialBreakpoint()*** sub and get evaluated in the ***Debugger\_OnBreakPoint(ByVal BreakPoint, ByVal CausingThread)*** sub.

CrashRule\_WebAppPool\_HR-AppPool.vbs

…

Sub Debugger\_OnInitialBreakpoint()

WriteToLog "Initializing control script"

On Error Resume Next

Set ServiceController = CreateObject("DbgSVC.Controller")

Set ServiceState = ServiceController.ServiceState

On Error Goto 0

WriteToLog "Clearing any existing breakpoints"

WriteToLog Debugger.Execute("bc \*")

WriteToLog "Attempting to set breakpoint at HR\_Module!Personnel::List"

DbgState("BP\_HR\_Module!Personnel::List\_ID") = Debugger.AddCodeBreakpoint("HR\_Module!Personnel::List")

WriteToLog "Attempting to set breakpoint at Kernel32!TerminateProcess"

DbgState("BP\_Kernel32!TerminateProcess\_ID") = Debugger.AddCodeBreakpoint("Kernel32!TerminateProcess")

WriteToLog "Current Breakpoint List(BL)"

Debugger.Write Debugger.Execute("bl")

End Sub

…

Sub Debugger\_OnBreakPoint(ByVal BreakPoint, ByVal CausingThread)

WriteToLog "Breakpoint at " & BreakPoint.OffsetExpression & " caused by " & CausingThread.SystemID

Select Case BreakPoint.ID

Case DbgState("BP\_HR\_Module!Personnel::List\_ID")

If DbgState("BP\_HR\_Module!Personnel::List\_ACTION\_COUNT") < 1 Then

CreateDump Breakpoint.OffsetExpression, false

DbgState("BP\_HR\_Module!Personnel::List\_ACTION\_COUNT") = DbgState("BP\_HR\_Module!Personnel::List\_ACTION\_COUNT") + 1

If DbgState("BP\_HR\_Module!Personnel::List\_ACTION\_COUNT") >= 1 Then

WriteToLog "Action limit of 1 reached for breakpoint HR\_Module!Personnel::List."

End If

End If

Case DbgState("BP\_Kernel32!TerminateProcess\_ID")

If DbgState("BP\_Kernel32!TerminateProcess\_ACTION\_COUNT") < 1 Then

CreateDump Breakpoint.OffsetExpression, false

DbgState("BP\_Kernel32!TerminateProcess\_ACTION\_COUNT") = DbgState("BP\_Kernel32!TerminateProcess\_ACTION\_COUNT") + 1

If DbgState("BP\_Kernel32!TerminateProcess\_ACTION\_COUNT") >= 1 Then

WriteToLog "Action limit of 1 reached for breakpoint Kernel32!TerminateProcess."

End If

End If

End Select

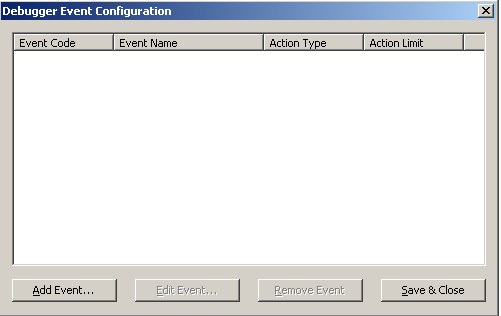
End Sub

…

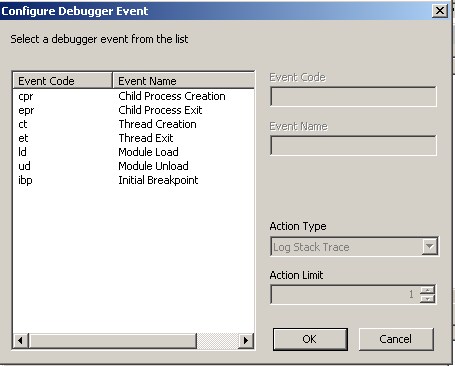
**Events**

Events such as module load and unload or thread creation and exit might be needed in the debugging process. DebugDiag provides a user interface for easy configuration of events.

If you click **Events** in the **Advanced Configuration (Optional)** dialog box, the following dialog box is displayed:



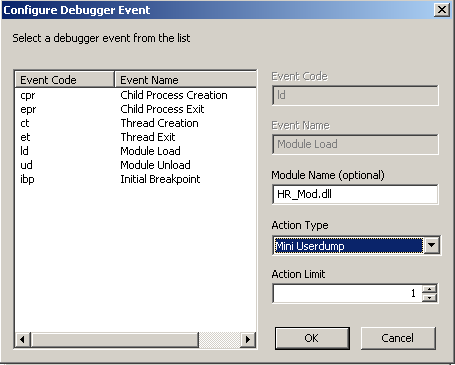
To add an event, click **Add Event**. The following dialog box will be displayed:



You can configure as many events as you need by using the required actions.

In the crash rule against the HR-AppPool, if you would like to know when and why a module called HR\_Mod.dll gets loaded, you need to set the **ld** event against HR\_Mod.dll with an action to log call stack. Another option would be to create and read minidumps generated for this application pool.

The configuration would look like the following:



The resulting control script with the above setting, will show the setting in the Debugger\_OnLoadModule(ByVal NewModule) sub as follows:

CrashRule\_WebAppPool\_HR-AppPool.vbs

…

Sub Debugger\_OnLoadModule(ByVal NewModule)

WriteToLog NewModule.ImageName & " loaded at " & Debugger.GetAs32BitHexString(NewModule.Base)

Select Case UCase(NewModule.ImageName)"

Case "HR\_Mod.dll

If DbgState("Event\_LD:HR\_MOD.DLL\_ACTION\_COUNT") < 1 Then

CreateDump "Module Load - HR\_Mod.dll", true

DbgState("Event\_LD:HR\_MOD.DLL\_ACTION\_COUNT") = DbgState("Event\_LD:HR\_MOD.DLL\_ACTION\_COUNT") + 1

If DbgState("Event\_LD:HR\_MOD.DLL\_ACTION\_COUNT") >= 1 Then

WriteToLog "Action limit of 1 reached for Event 'Module Load - HR\_Mod.dll'."

End If

End If

End Select

End Sub

…

Note that in this case of events, every event has its own VBScript sub.

**Pageheap Flags**

One of the most complicated issues to deal with in production environments is heap corruption. Because heap corruption is such a complex subject, this article will provide only a high level overview of this topic and, subsequently, how to use DebugDiag to approach the performance issues created by heap corruption.

Heap Corruption

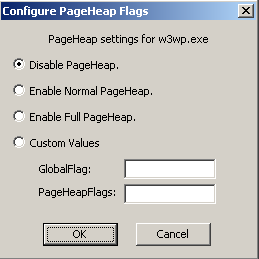
Memory blocks may become corrupted in the heap if dynamic allocation and deallocation of memory is not handled properly by user codes. Some of the common causes of heap corruption are: buffer overrun (writing beyond the allocated memory), double free (freeing a pointer twice), and old pointer reuse (reusing a pointer after it has been released). Troubleshooting heap corruption can be difficult a process does not terminate or throw an error when a thread corrupts the heap. As long as the corrupted heap is not used, the process will not crash; however, once a thread tries to use the corrupted block of memory in the heap, the process crashes. If a crash rule is active and the process crashes because of heap corruption, you will see as a “culprit” thread that seems to have caused the crash but is actually nothing more than a “victim” of heap corruption.

Enable Pagegeap to find the root cause that corrupted the heap. Pageheap is simply an extra layer in Ntdll.dll that controls and validates every heap operation before it is performed. However, this extra layer of heap verification does impact process performance.

**Note:** Pageheap is enabled only if the target type in the rule is “A specific process” and if the rule is applied to all instances of that process (**This process instance only** should be cleared). When Pageheap is enabled for a process, all instances of that process will have Pageheap enabled and the process needs to be restarted before Pageheap will take effect.

For example: if the application pool is crashing because of heap corruption, then you need to enable Pageheap against “A specific Process” and choose or type in W3wp.exe and make sure that **This process instance only** is cleared.

Pageheap dialog box



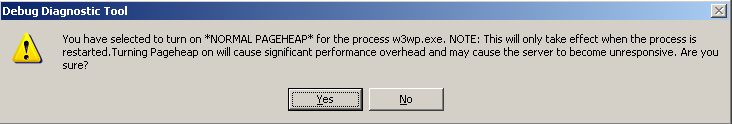
* **Disable PageHeap**

Choose this option if PageHeap was already enabled for this rule and you would like to remove it. Please note that if the rule is disabled or removed, PageHeap is disabled.

* **Normal Pageheap**

Normal Pageheap was introduced to overcome the performance issue caused when Full Pageheap is used on large scale processes. However, there is a tradeoff because this option does not find the root cause of the corruption as reliably as does Full Pageheap. Furthermore, process performance is still impacted, though not as much as with Full Pageheap

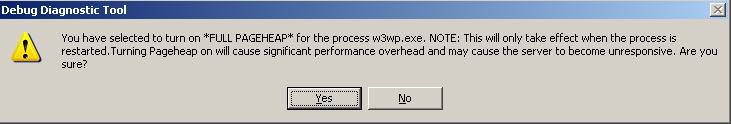
When turned on, Normal Pageheap will issue the following warning:



* **Full Pageheap**

This option turns on the full Pageheap scale. Process performance is impacted more with this option than with the other forms of Pageheap; however, Full Pageheap quickly and reliably detects corruption.

DebugDiag displays the following warning when Full Pageheap is enabled:



* **Custom Values**

This option lets you set the custom Pageheap flags on the Pageheap features that you want to enable.

**Note**: Set exception 80000003 with an action type of Full Userdump when you enable Pageheap. Pageheap may call into this breakpoint exception if it detects some types of corruption.

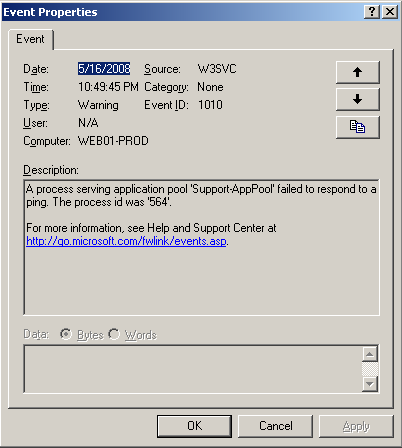
A crash rule is flexible and you can change it at any time. You can add or remove exceptions, breakpoints, and events. You can also change the maximum number of userdumps for the rule, the rule name, and the userdumps location. All these changes can be made without restarting the process. Since the process has to be restarted whenever Pageheap features are changed (except for changes to Pageheap flags), crash rules are often a better process management option.

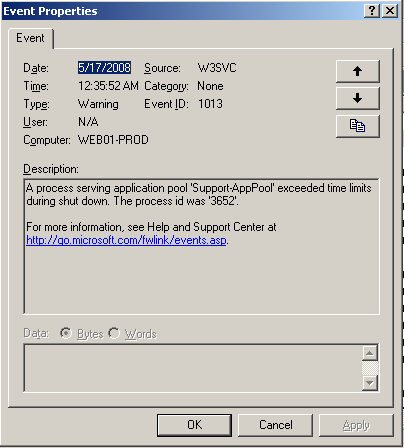
A crash rule can even be deactivated without affecting the target process. This means that you can physically detach the debugger host from the target process without bringing the target down. (You can do this in Windows Server 2003 and in later Microsoft operating systems; however, when a rule is deactivated in Windows 2000, the debugger host remains attached to the target process and in a dormant state).

**Hang Rule**

Please note that the hang rule feature in DebugDiag can only detect hangs in IIS processes. The hang rule builds a ping mechanism against the Web application by configuring a ping interval and a time-out. When the Web application does not respond to the ping request in the specified time-out, DebugDiag executes the configured action (usually it takes a userdump of the affected process).

For example, when a hang occurs you may see system events such as the following in the event log:

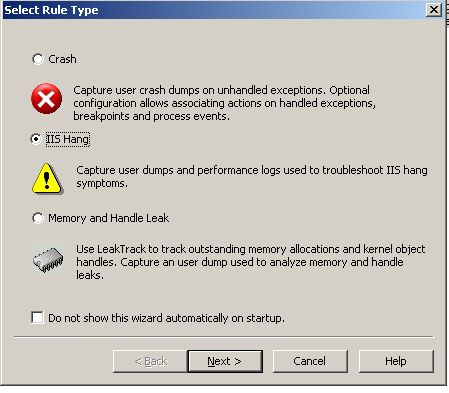




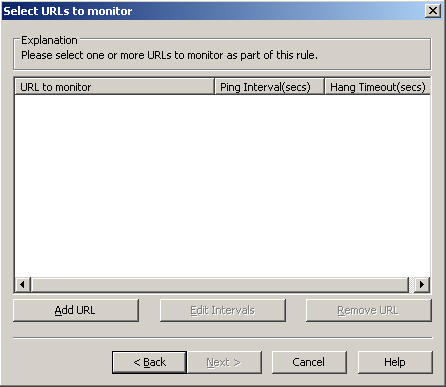
To configure an IIS hang rule:

Add a hang rule against the SupportWeb Web site. This will monitor the page default.asp in the root and will use the default ping interval and time-out.

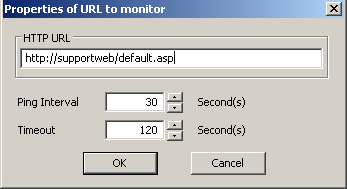
1. In the **Rules view** of DebugDiag UI, click **the Add Rule**.
2. Select **IIS Hang** and click **Next**.



1. In **Select URLs to monitor**, click **Add URL**, and then click **Next**.



1. In **HTTP URL**, type [http://supportWeb/default.asp](http://supportweb/default.asp), and then click **OK**.

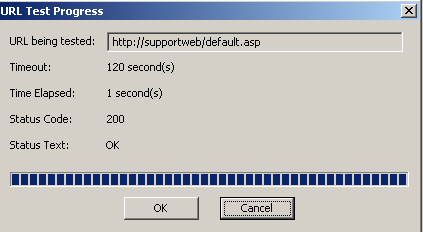


The following dialog box is displayed to make sure that the URL can be reached:



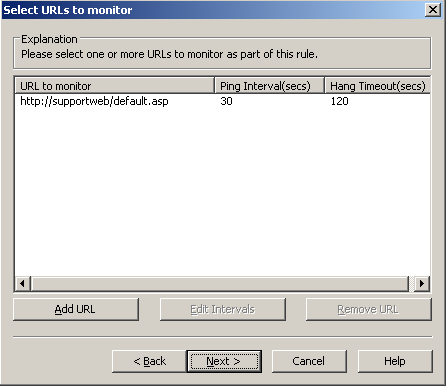
1. Click **Yes** to test the specified URL.

If the URL test is successful, the following dialog box is displayed:



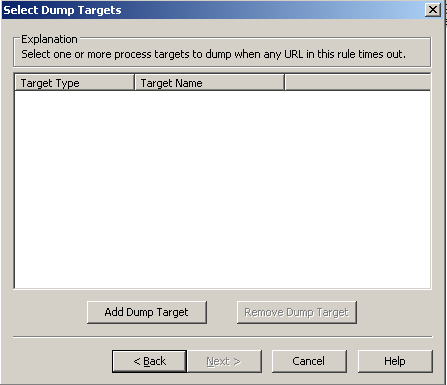
If the test is not successful, The **URL Test Progress** dialog box will show the reason why the test failed.

1. Click **OK** on **URL Test Progress**.



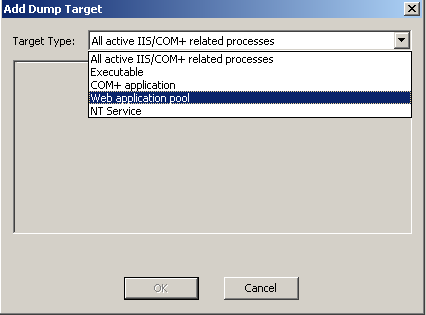
You can add and monitor other URLs at this point..

1. Click **Next** in **Select URLs to monitor**. The following dialog box is displayed.



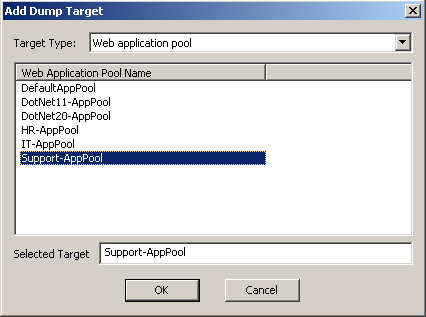
Select the action to be taken when the hang is detected. The action is always set to generate a userdump.

1. Click **Add Dump Target** and the following dialog box is displayed:

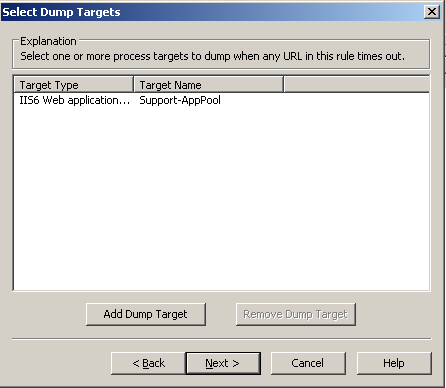


Choose the types of targets to dump. In this case, you want to generate userdumps when the supportWeb Web site is running in the Support-AppPool application pool and it is in a hang state, assuming that you know in advance that it is the only Web site that is hanging.

1. Choose **Web application pool** from the dropdown box and click **Next**.

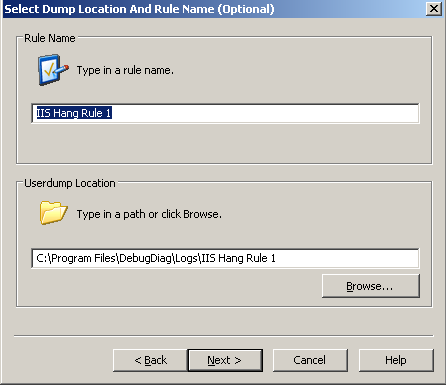


1. Select the **Support-AppPool** and click **OK**.

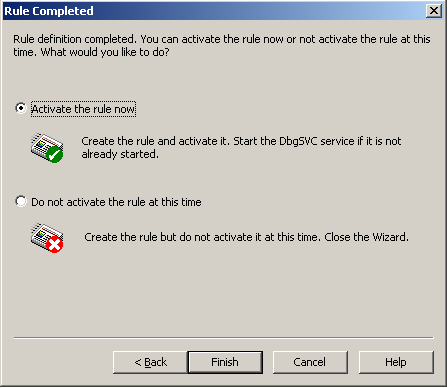


Add as many Dump Targets as needed for the rule.

1. Click **Next** on the **Select Dump Targets** dialog box.

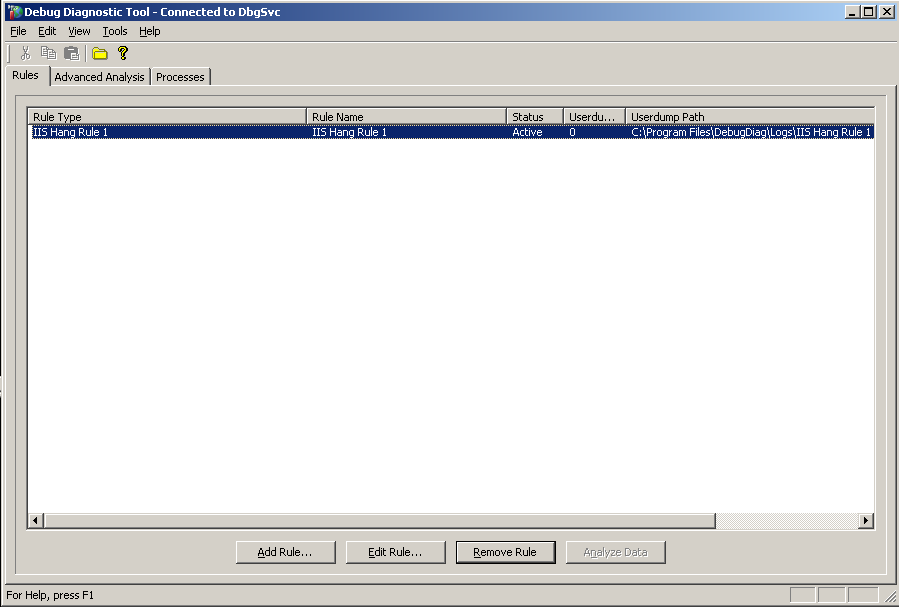


1. Accept the defaults and click **Next**.



1. Click **Activate the rule now**, and click **Finish**.

The Rules view in the DebugDiag UI shows that the rule is active.



Now that the hang rule for the SupportWeb Web site is active and the debugging service starts pinging the resource default.asp every 30 seconds, if no response is received in 120 seconds, a memory dump of the worker process serving the Support-AppPool is generated.

The hang rule does not create a new control script, instead it modifies the default service control script (dbgsvc.vbs). This is what the modification looks like:

Dbgsvc.vbs

…

Sub CreatePingURLs()

If ServiceState("IIS\_HANG\_RULE1\_HIT") = 0 Then

Set PingURL = HTTPPinger.AddPingURL("http://supportWeb/default.asp", 30, 120)

PingURL.GroupID = 1

End If

End Sub

Sub HandleIISHangRule(ByVal TimedOutURL)

Select Case TimedoutURL.GroupID

Case 1

WriteToLog "IIS Hang Rule - 'IIS Hang Rule 1' activated."

If ServiceState("IIS\_HANG\_RULE1\_HIT") = 0 Then

DumpWebAppPool "C:\Program Files\DebugDiag\Logs\IIS Hang Rule 1", "Support-AppPool"

ServiceState("IIS\_HANG\_RULE1\_HIT") = 1

HTTPPinger.RemovePingURLGroup 1

End If

End Select

End Sub

**Discussion**:

The hang detection mechanism in DebugDiag for IIS Web applications relies on the DebugDiag service to send HTTP requests by using WinHTTP. A request is sent in a specified ping interval and if no response is received by the debugging service in the configured time-out, further action is taken.

However, this method does not detect all Web server hangs. For instance, ping requests might respond with no problem in a Web application that is experiencing severe queuing of dynamic requests. This could occur if the ping requests were addressing a different part of the application than the part that was causing the queue backup. This problem highlights the importance of choosing the correct URL to monitor. For example, if a Web administrator received complaints that requests to “page123.aspx” take more than a minute to load, the best first response would be to review the IIS log files to verify the problem, and then to set the time-out for this URL to 60 seconds.

However, there are restrictions that apply to the choice of the URL to be monitored. If the resource to be monitored must use NTLM authentication, then you need to specify “localhost” as the server name. If you can’t do that, then you may need to specify a different resource that will allow you to use Anonymous authentication, since there is no limitation on anonymous requests. Resources protected by Basic, Digest, or Advanced Digest authentication cannot be monitored.

Userdumps are another important tool that can be used to help solve server hangs. If you are using a monitoring tool on your server that can detect a hang and that can run commands on the Web server, you can configure such a tool to execute Dbghost.exe and generate a userdump. (For more information on userdumps, see Advanced command-line usage of DebugDiag.)

Hang rules are flexible and can be modified any time after they are created. Logging for a hang rule is kept in the debugging service log file. When a hang is detected, the following information is added to the log file:

DbgSVC\_Date\_\_05\_16\_2008\_\_Time\_10\_38\_36PM\_\_578\_\_Log.txt

…

Request to reload control script recieved.

[5/16/2008 10:39:27 PM] Process Exited: Process Name - notepad.exe Process ID - 3984

[5/16/2008 10:40:06 PM] New process found: Process Name - notepad.exe Process ID - 3080 Process Identity - WEB01-PROD\Administrator

[5/16/2008 10:41:16 PM] New process found: Process Name - dllhost.exe Process ID - 868 Process Identity - WEB01-PROD\Administrator

[5/16/2008 10:50:25 PM] HTTP Ping Timeout: Ping URL - http://supportweb/default.asp Timeout(secs) - 120

[5/16/2008 10:50:25 PM] IIS Hang Rule - 'IIS Hang Rule 1' activated.

[5/16/2008 10:50:34 PM] Process Dump Created: Process ID - 564 Dump Path - C:\Program Files\DebugDiag\Logs\IIS Hang Rule 1\w3wp.exe\_\_Support-AppPool\_\_PID\_\_564\_\_Date\_\_05\_16\_2008\_\_Time\_10\_50\_25PM\_\_958\_\_IIS\_COM+ Hang Dump.dmp

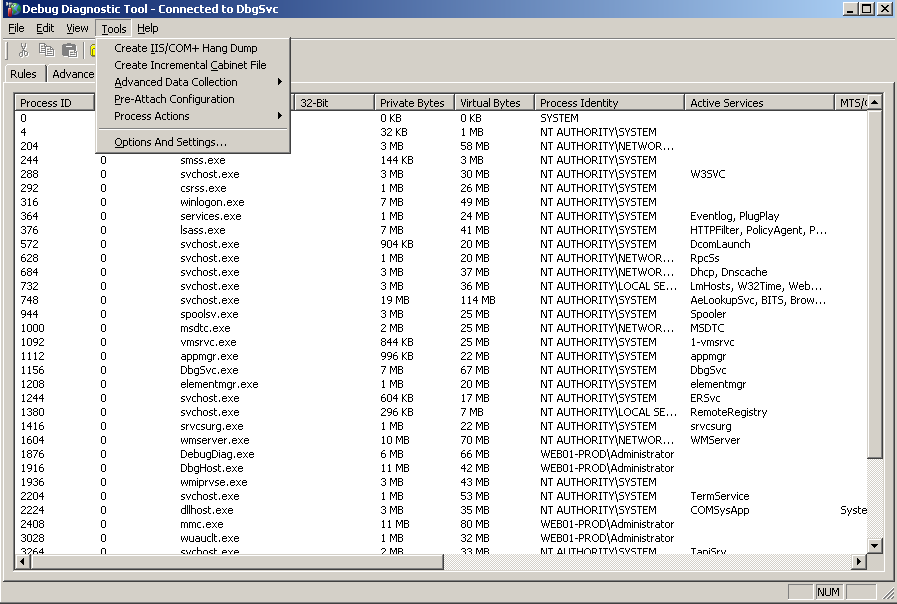
[5/16/2008 10:54:06 PM] Process Exited: Process Name - notepad.exe Process ID - 2488

…

…

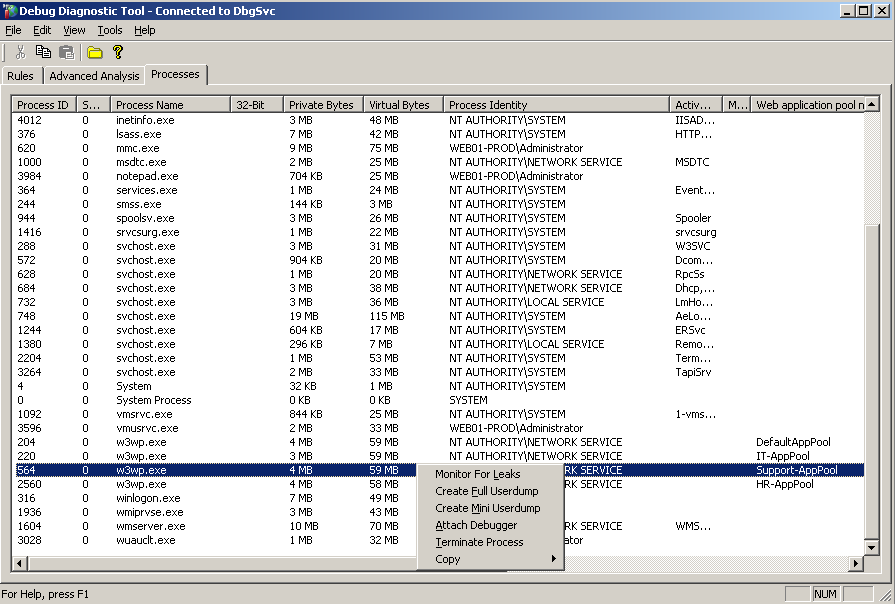
If a process hang occurs while an administrator is monitoring a server, the administrator can manually initiate a userdump.

The DebugDiag UI provides three ways of creating manual userdumps:  
1. If you want to take userdumps of all IIS/COM+ processes at once at the time of the hang (that is, you know IIS/COM+ is hanging but you don’t know which application pool or COM+ package is causing the problem), then use this shortcut: **Tools 🡪 Create IIS/COM+ Hang Dumps**.

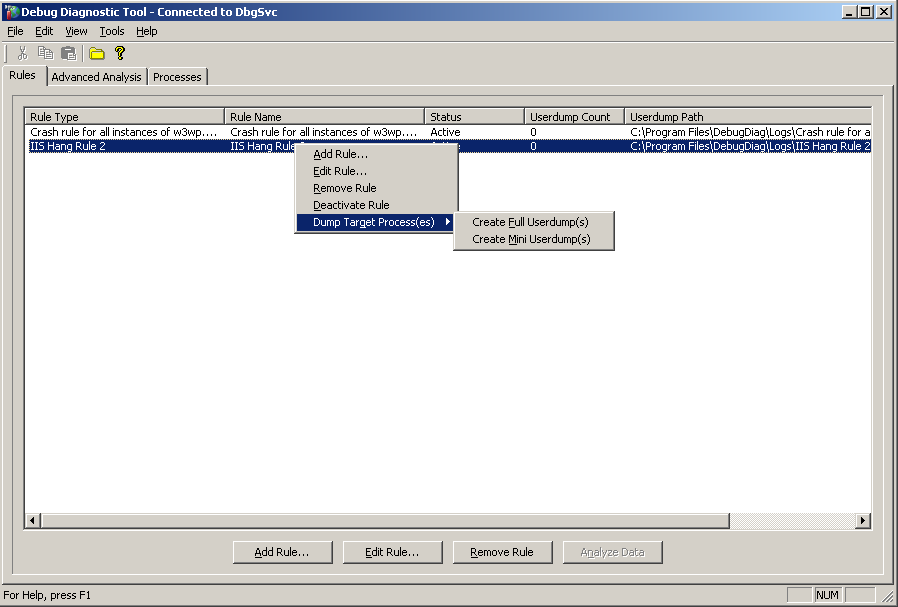


2. If you know the specific application pool, COM+ package, or user mode process that you want to take a memory dump of, you can use the context menu provided in the processes view in the DebugDiag UI.

Right-click the process in question and choose **Create Full Userdump**.



3. If you already have a rule set, you can right-click the rule in the Rules view and choose **Dump Target Process(es) 🡪 Create Full Userdump(s)**.



By default, userdumps created by using the shortcut or the context menu will be saved in the folder \Program Files\DebugDiag\Logs\Misc.

***Memory and Handle Leak Rule***

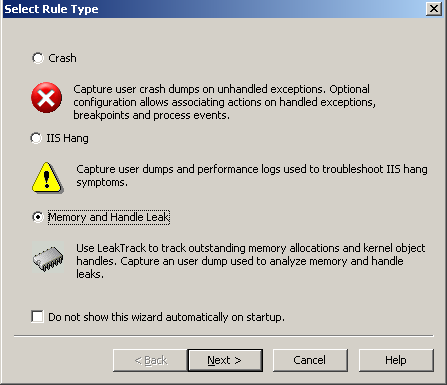
One of DebugDiag’s most powerful features is the ability to track memory and handle leaks.

A memory leak occurs when memory is allocated in a process and not released. When this occurs it is useful to know what allocated the memory that has not been released. DebugDiag's memory tracking feature is designed to do just that. DebugDiag has a memory leak monitoring feature designed to track memory allocations for a process. DebugDiag will inject a DLL into the specified process and monitor memory allocations over time. When it’s time to analyze a process for leaks, a dump is generated, and the dump is analyzed to determine which allocations are not being released and which are most likely causing memory leaks.

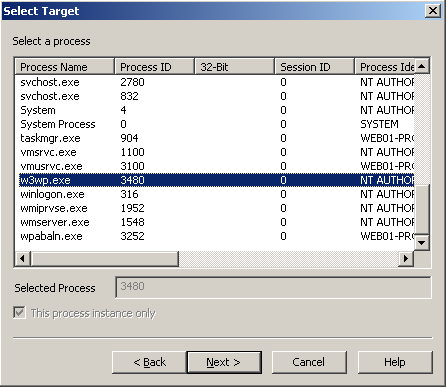
Most memory allocations belong in one of three groups:  caching, short term allocations that will be released later, and memory leaks. All three allocation methods have very distinct allocation patterns when measured over time. The leak-tracking feature calculates a leak probability using a formula that is based on these allocation patterns as measured over a specific time period. More precisely, leak probability is a number between 0 and 100 that measures how allocations are spread over time. Empirical studies show that leak allocations tend to be evenly spread over time. When allocations are evenly spread over time, leak probability equals 100. If allocations are bunched either at the beginning or at the end of the tracking duration, this usually indicates either caching allocations or short term allocations, respectively. If all allocations occurred at the beginning or at the end of a process, then leak probability will equal zero. Additional studies show that a high allocation count accompanied by leak probability higher than 75 percent indicates memory leaks. A properly functioning process could be experiencing heavy caching or short-term allocations and these phenomena could mask other behaviors, so it is important to use this synthetic time distribution calculation to get stack samples for those functions.

**Example**: You want to add a memory and handle leak rule against the W3wp.exe serving the Support-AppPool application pool. Assume that you also would like to track memory consumption for four hours and automatically generate a userdump at the end of the tracking period. To do this, follow these steps:

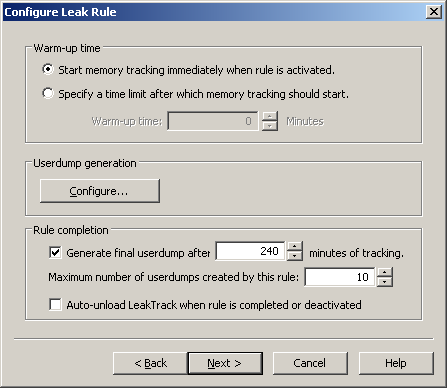
1. In the DebugDiag UI, click **Add Rule**.
2. In the ***Select Rule Type***, select **Memory and Handle Leak**, and click **Next**.



1. Choose the process to monitor for leaks, and click **Next**. Note here that the process should be already running.



1. In the ***Configure Leak*** Window, check **Generate final userdump after \_ minutes of** tracking and type **240**. (4 hours = 240 minutes).



These defaults settings would have been enough to track for memory consumption, but would not generate userdumps automatically unless the process crashes, so with all the default settings userdumps should be taken manually. For automatic userdump generation, it should be either based on tracking period (as set above) or based on memory consumption (private or virtual). Here are the available options and how to generate userdumps automatically:

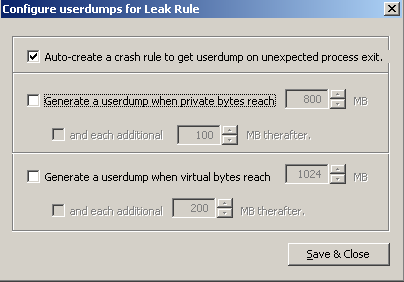
**Warm-up time**

**Start memory tracking immediately when rule is activated**, as the text indicates, tracks for memory consumption starts once the rule is activated. Choose this rule when you do not know exactly when the memory is leaked.

To use this rule, you must specify a time limit after which memory tracking should start. This means that after the rule is activated, the debugging service will wait for the specified time before it injects leaktrack and starts tracking memory consumption. If you believe that the initial memory allocations in the process might be caused by caching, you could choose this option and ignore those initial allocations.

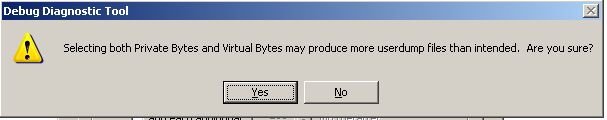
**Userdump generation**

DebugDiag provides a flexible way to generate userdumps. When you click **Configure**, the following dialog box is displayed:



The first option in the **Configure userdumps for Leak Rule** dialog box, **Auto-create a crash rule to get userdump on unexpected process exit**, is checked by default. If this remains selected, it adds a second rule—a crash rule, along with the memory and handle leak rule. This rule is added because troubleshooting memory leaks causes most processes to run out of virtual memory and a subsequent call to allocation memory will fail with an Access Violation. This, in turn, will cause the process to terminate unexpectedly. Since you do not want to lose tracking session data, you must generate a userdump even if the process crashes.

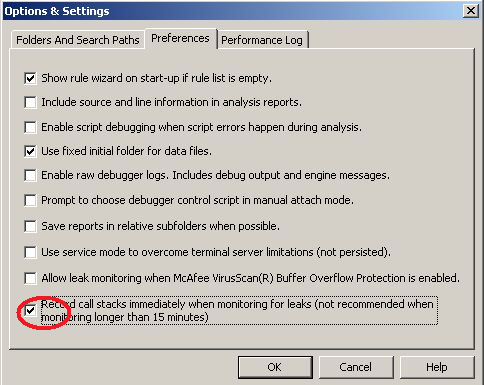
There are two other options for generating userdumps, both are based on the amount of memory consumption of the process: physical or virtual. As DebugDiag can generate userdumps based on the amount of memory the process consumes, it is a good idea to configure either the virtual or the physical memory allocation. Default values should be fine; however, if you configure both virtual and physical memory, the following message will be displayed to warn that this choice may produce many undesired userdumps.



**Rule Completion**

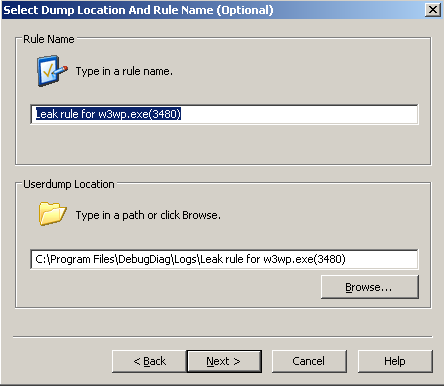
This section shows how to configure the tracking period, the maximum number of dumps, and how to cause leaktrack.dll to be unloaded once the rule is deactivated or completed.

Set the tracking period to 15 minutes or longer to get usable data. If you are debugging a process that leaks memory in less than 15 minutes, you need to enable the FastTrack option in the DebugDiag UI **(Tools 🡪 Options and Settings 🡪 Preferences 🡪 Record Call stacks immediately… ).**

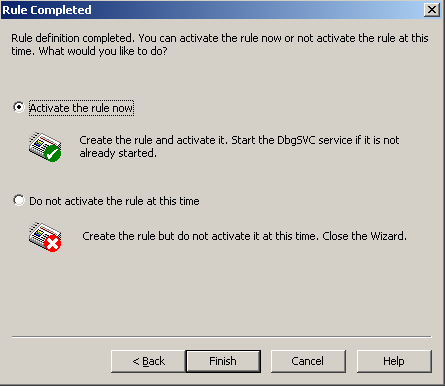


The **Auto-unload LeakTrack when rule is completed or deactivated** option will unload the injected dll, leaktrack.dll, once the rule is either completed or deactivated.

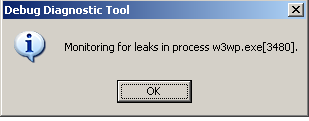
1. Click **Next** in the **Configure Leak Rule** dialog box.



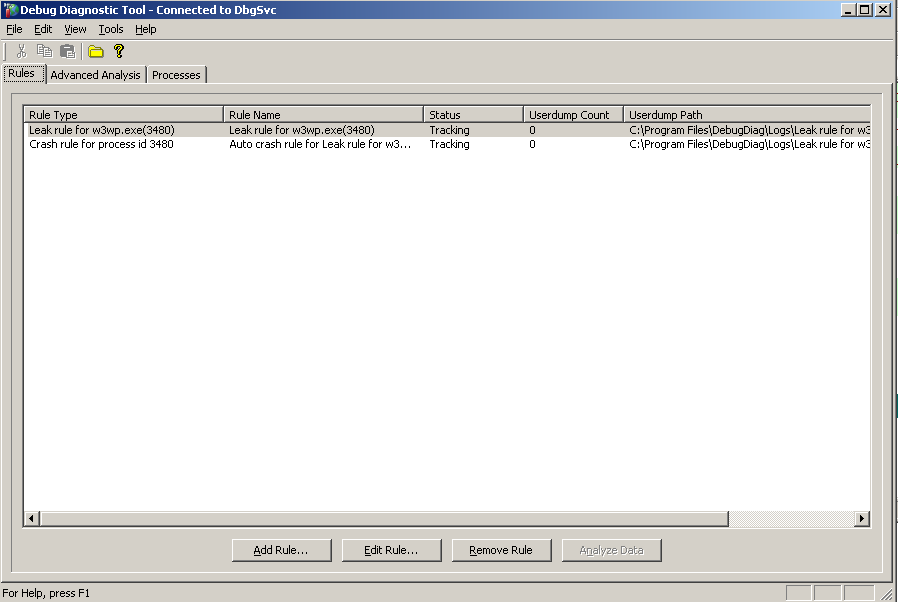
1. Accept the defaults or make desired changes and click **Next**.



1. Click **Finish**, and the following information dialog box is displayed:



The **Rules** view in the DebugDiag UI will look like the following:



Notice the status of the Rule set to Tracking. This means that leaktrack.dll is collecting data now.

When the rule is activated, the debugging service control script is modified as follows:

DbgSVC.vbs

…

Sub StartLeakRule(ByVal RuleKey, ByVal ProcessID, ByVal DumpLimit, ByVal DumpPrivateBytes, ByVal DumpPrivateBytesInterval, ByVal DumpVirtualBytes, ByVal DumpVirtualBytesInterval)

…

…

End Sub

Function GetControlScriptForProcess(ByVal Process)

GetControlScriptForProcess = ""

End Function

Sub Controller\_onTimeTriggerHit(ByVal ActiveTrigger)

Select Case ActiveTrigger.TimeTriggerID

Case ServiceState("LEAK-TPID-4080\_WARMUP\_TRIGGER\_ID")

StartLeakRule "LEAK-TPID-3480", 3480, 10, 0, 0, 0, 0

Case 2

DumpLeakTarget "LEAK-TPID-3480", 4080, "C:\Program Files\DebugDiag\Logs\Leak rule for w3wp.exe(3480)", "Leak Rule Expired"

StopLeakRule "LEAK-TPID-3480", True

End Select

End Sub

The tracking time for the leak rule is stored in the ServiceState.xml file:

ServiceState.xml

…

<StateVariable Name="CRASH-TT-3-TNAME-HR-APPPOOL\_RULEFAILED" Value="0" />

<StateVariable Name="CRASH-TT-3-TNAME-SUPPORT-APPPOOL\_RULEFAILED" Value="0" />

<StateVariable Name="DISABLE\_PERFLOG" Value="1" />

<StateVariable Name="ENABLE\_RAWLOGGING" Value="0" />

<StateVariable Name="LEAK-TPID-2168\_TRACKING\_TIME" Value="600" />

<StateVariable Name="LEAK-TPID-2168\_WARMUP\_TRIGGER\_ID" Value="5" />

<StateVariable Name="LEAK-TPID-2552\_VIRTUAL\_BYTES\_INITIAL\_VALUE" Value="1073741824" />

<StateVariable Name="LEAK-TPID-2552\_WARMUP\_TRIGGER\_ID" Value="3" />

<StateVariable Name="LEAK-TPID-3480\_DUMP\_LIMIT" Value="10" />

<StateVariable Name="LEAK-TPID-3480\_TRACKING\_TIME" Value="240" />

<StateVariable Name="LEAK-TPID-3480\_WARMUP\_TRIGGER\_ID" Value="1" />

<StateVariable Name="LEAK-TPID-3236\_TRACKING\_TIME" Value="15" />

…

…

**Discussion**

How does DebugDiag work to track for memory and handle leak?

LeakTrack uses all the Handle allocation functions, and LeakTrack also works by using the following family of API functions that allocate memory and handles.

Memory allocation API families

NT Heap allocator (HeapAlloc, LocalAlloc, GlobalAlloc)

Virtual Memory allocator (VirtualAlloc)

CRT Memory allocator (malloc, calloc, operator new)

BSTR allocator (SysAllocString, SysAllocStringLen)

OLE/COM allocator (CoTaskMemAlloc, IMalloc interface)

Additionally, LeakTrack uses any call that would open a handle to the following functions:

Event

Section

File

Port

Directory

Mutant

WindowStation

Semaphore

Key

Token

Process

Thread

Desktop

IoCompletion

Timer

KeyedEvent

As soon as LeakTrack is injected into a process, it starts tracking and recording every allocation. Memory allocations are stored separately from handle allocations. LeakTrack also tracks released allocations. If a specific allocation is cleared, its corresponding record is removed. Over time, allocation records that have not been cleared accumulate (Outstanding allocations).

The following information is recorded for each memory allocation.

1) Address of the allocation.

2) Size of the allocation.

3) Time of the allocation.

4) Return address from the API tracked.

5) Type of memory allocation based on the API family.

Each handle allocation contains very similar information, except for size.

As they accumulate, these allocations are sorted based on the following filters:

1) Top 10 functions sorted by allocation count.

2) Top 10 functions sorted by total allocation size.

3) Top 10 functions sorted by leak probability.

Likewise, handle allocations are sorted based on the top 10 functions sorted by allocation count and by the top 10 functions sorted by leak probability (since size information is not kept for handles

The final sorted combinations determines which functions must have stack samples associated with them.

A stack sample is a heuristic record based on the X86 op codes of possible return addresses that are found on the stack at execution time. In almost all cases, these samples will contain spurious addresses. MemoryExt.dll, the analysis module extension, uses the symbol information found in these samples to reconstruct a stack that will resemble the stack at the time of failure as closely as possible. This method is used because debug symbols are not used at run time, and reading and accessing the stack at every allocation would cause significant performance overhead.

The debugging service log file will begin to fill with information or errors once the leak rule is initiated. It will display something like this after the rule is initiated and after a userdump is generated.

DbgSVC\_Date\_\_05\_20\_2008\_\_Time\_10\_58\_08PM\_\_445\_\_Log.txt

…

[5/21/2008 12:52:30 AM] Process Exited: Process Name - notepad.exe Process ID - 620

Request to reload control script recieved.

Request to reload control script recieved.

[5/21/2008 12:52:56 AM] New process found: Process Name - DbgHost.exe Process ID - 3768 Process Identity - NT AUTHORITY\SYSTEM

[5/21/2008 12:53:05 AM] Leak Rule Started: Process ID - 3480

[5/21/2008 12:53:19 AM] New process found: Process Name - notepad.exe Process ID - 3832 Process Identity - WEB01-PROD\Administrator

[5/21/2008 1:02:53 AM] Process Exited: Process Name - notepad.exe Process ID - 3832

[5/21/2008 1:08:12 AM] Leak Dump Created: Process ID - 3480 Rule Key - LEAK-TPID-3480 Dump Path - C:\Program Files\DebugDiag\Logs\Leak rule for w3wp.exe(3480)\w3wp.exe\_\_Support-AppPool\_\_PID\_\_3480\_\_Date\_\_05\_21\_2008\_\_Time\_01\_08\_05AM\_\_687\_\_Leak Dump - Leak Rule Expired.dmp…

…

**Analyzing Userdumps**

Using the Windows core debuggers (Windbg.exe or Cdb.exe) for post-mortem analysis is a time consuming process and requires many debugging skills.

Automated post-mortem analysis of usedumps is one of the main goals of DebugDiag. It is delivered by using the analysis module of the tool and promises to give an accurate solution by:

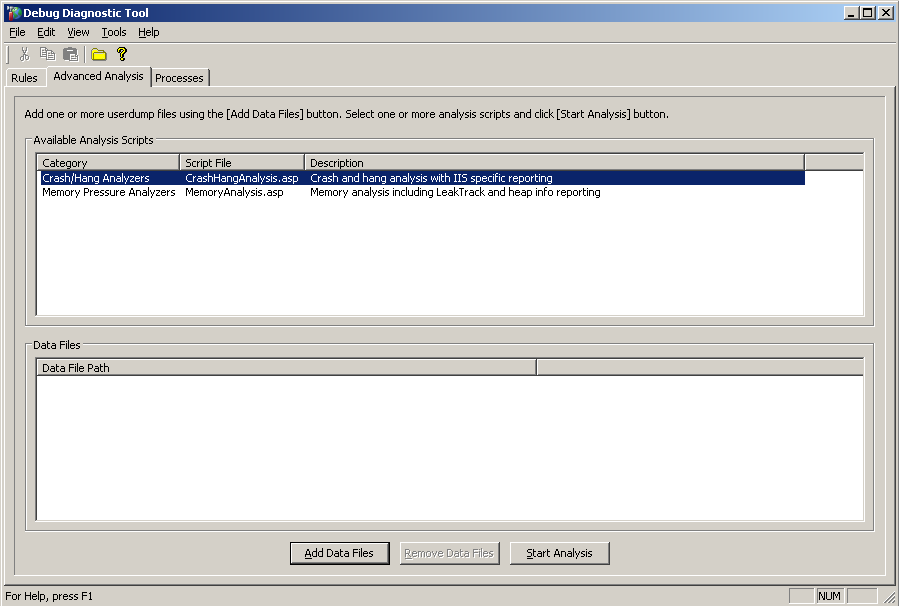
* Separating the raw data extraction from the analysis algorithms.
* Providing a script-based solution for building analysis algorithms, thus reducing the debugging skills necessary for implementing such analysis scripts.
* Providing an extensible object model solution to meet the demands of future unidentified requirements.
* Providing a built-in HTML-based report generation and formatting solution similar to ASP pages.

DebugDiag is shipped with two main analysis scripts: CrashHangAnalysis.asp and MemoryAnalysis.asp. The first of these is used to analyze crash and hang userdumps, and the second is used for memory and handle leak analysis.

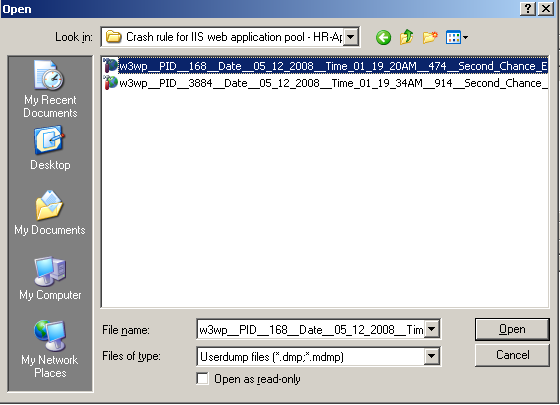
Before you use these scripts to start the analysis, make sure that you have lined up the debug symbols properly. By default, DebugDiag accesses the Microsoft public symbol server, but you must have an Internet connection to access that server. To add or modify the symbol path, go to *Tools🡪 Options and Settings… 🡪 Symbol Search Path for Analysis.*

To analyze a userdump:

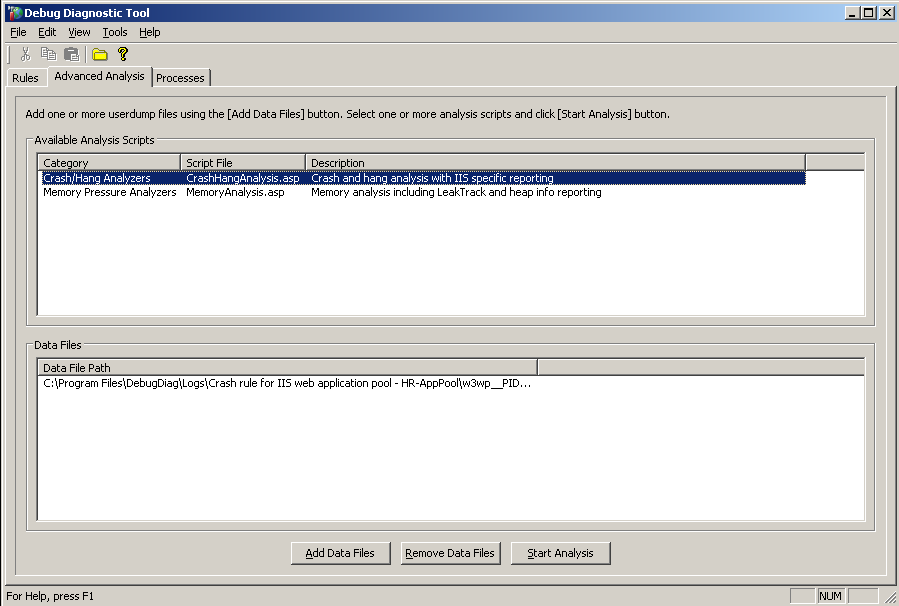
1. Open the DebugDiag UI and click the Advanced Analysis view. The following dialog box is displayed:



1. Add the userdumps to be analyzed by clicking **Add data Files**.

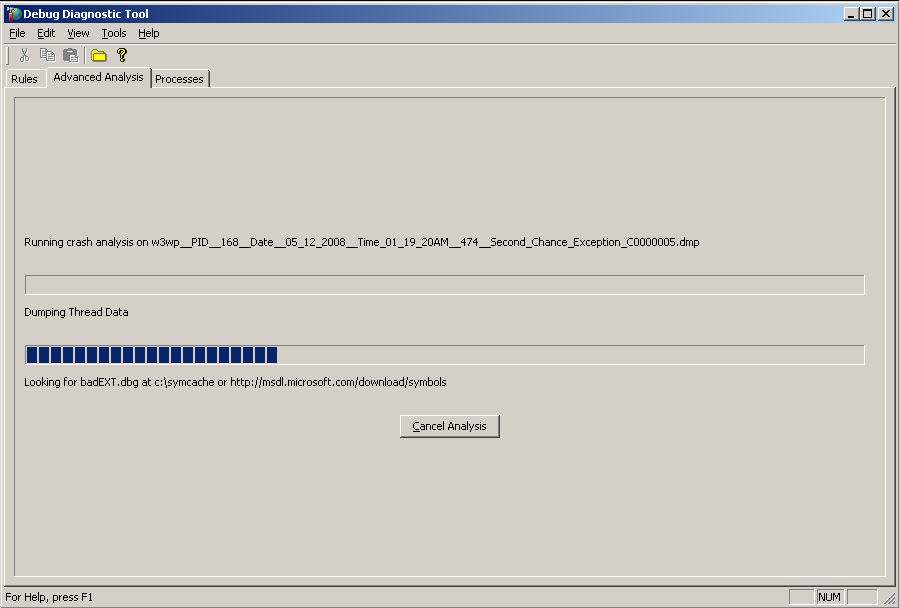


1. You can select multiple userdumps for analysis. Once you have made your selection, click **Open**.



Choose the **analysis category** (Crash/Hang or Memory Pressure) or choose both analysis categories to run against the userdump(s).

1. Once the selection is made, click **Start Analysis**. DebugDiag will show the analysis progress as follows.



You can also start the analysis from the Rules view if the rule has already generated dumps. If you want to do this, right-click the rule and select **Analyze data**.

You can also start the analysis of a user dump by going to the userdump in Windows Explorer and right-clicking the userdump file, then choosing the type of analysis required.

Once the analysis is complete, DebugDiag will automatically save and open the analysis report. The report will be saved in the DebugDiag\Reports folder and will open automatically in Internet Explorer.

Every analysis report is composed of three main sections:

**Analysis Summary**

The analysis summary is an *event viewer* type of message that records errors, warnings, and information that is relevant to the userdump analysis. It also includes descriptions of the userdump and recommendations about how to resolve the problems it shows.

**Analysis Details**

The analysis details section begins with a table of contents that lists all the memory dumps that are analyzed. For each memory dump, there is a listing of report titles indicating the type of analysis that was performed.

**Script Summary**

In this section, the analysis reports the status of the script that was used to analyze the userdump. If there were any errors encountered while running the script, this section will show the error codes, sources, descriptions, and the lines that caused the errors.

The following section more closely examines the two built-in analysis scripts in DebugDiag: CrashHangAnalysis.asp and MemoryAnalysis.asp.

**CrashHangAnalysis.asp**

As the name indicates, the CrashHangAnalysis.asp performs both crash analysis and hang analysis.

In the case of crash analysis, the script starts by using base exception analysis and then determines what kind of exception the process has encountered. The extracted exception will fall in one of the following three categories:

* **Heap Corruption**

If the exception has been caused by heap corruption, DebugDiag will either recommend to enable Pageheap or it will perform root cause analysis if Pageheap is enabled. If Pageheap has not been enabled, then it will also get another crash userdump after it enables Pageheap.

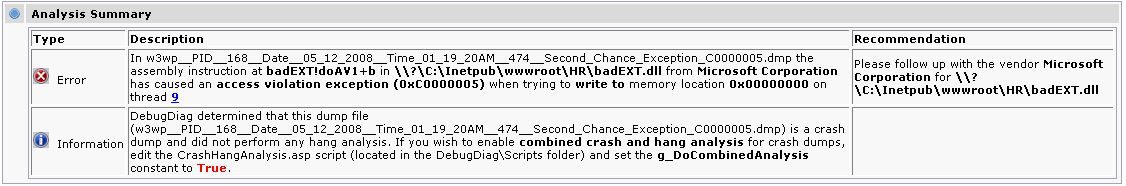
* **Nested Exception**

When DebugDiag determines that the exception it is analyzing was caused by another exception, it will unwind the call stack, and then use root exception analysis to find the cause of the exception.

* **Access violation and other Unhandled Exceptions**

If the script determines that this was the cause of the exception, then the exception is straightforward and DebugDiag will perform root cause analysis.

Following is an example of analysis summary in a crash userdump analysis report:



The analysis details of a crash userdump analysis report will contain the following information:

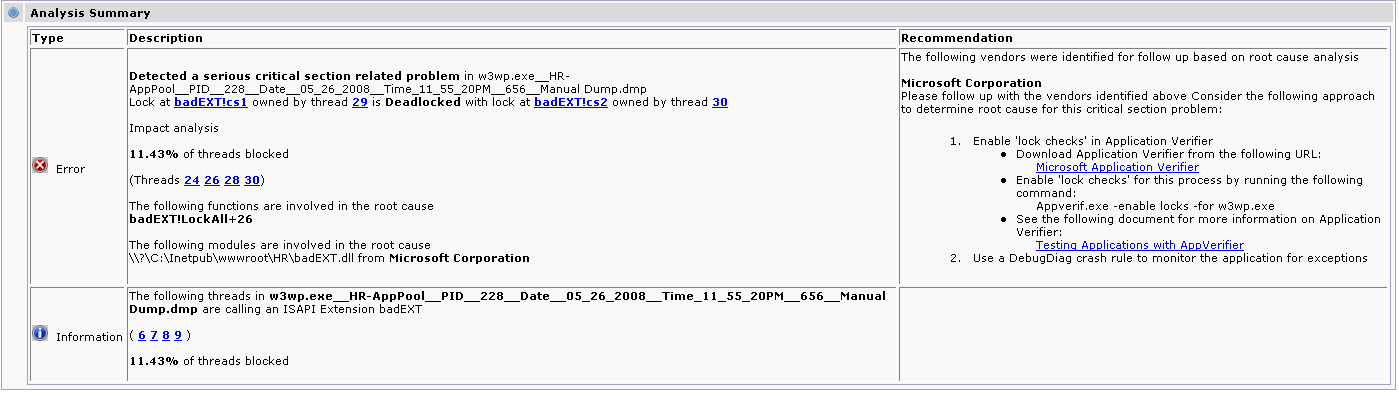
* **Faulting Thread:** This displays the call stack of the thread that caused the unhandled exception. The resulting display is similar to the results you receive when you run the debugger command **kv**, but it also supplies the thread's entry point, the create time, the time spent in user mode, the time spent in kernel mode, and a comment explaining the kind of work the thread is performing.
* **Faulting Module Information:** This shows all information concerning the module that caused the crash. The information displays file attributes, helps double-check the versioning of the file, and shows whether there was any matching symbol file used during the analysis.

Following is an example of the analysis details from a crash userdump analysis report:



If there is a hang, CrashHangAnalysis.asp analysis determines whether the userdump being analyzed has loaded the main HTTP module. If such a userdump is available, it can be used by either W3wp.exe in Windows Server 2003 or by Inetinfo.exe in Windows 2000. If analysis determines that the HTTP module is present, it continues by scanning for locks (including possible deadlocks) on critical sections, and then follows this with a full scan for common hang causes, such as COM calls, database calls, and socket calls. Finally, if the userdump is of a COM+ component or ASP is loaded, the analysis adds a detailed COM+ and a detailed ASP report, respectively.

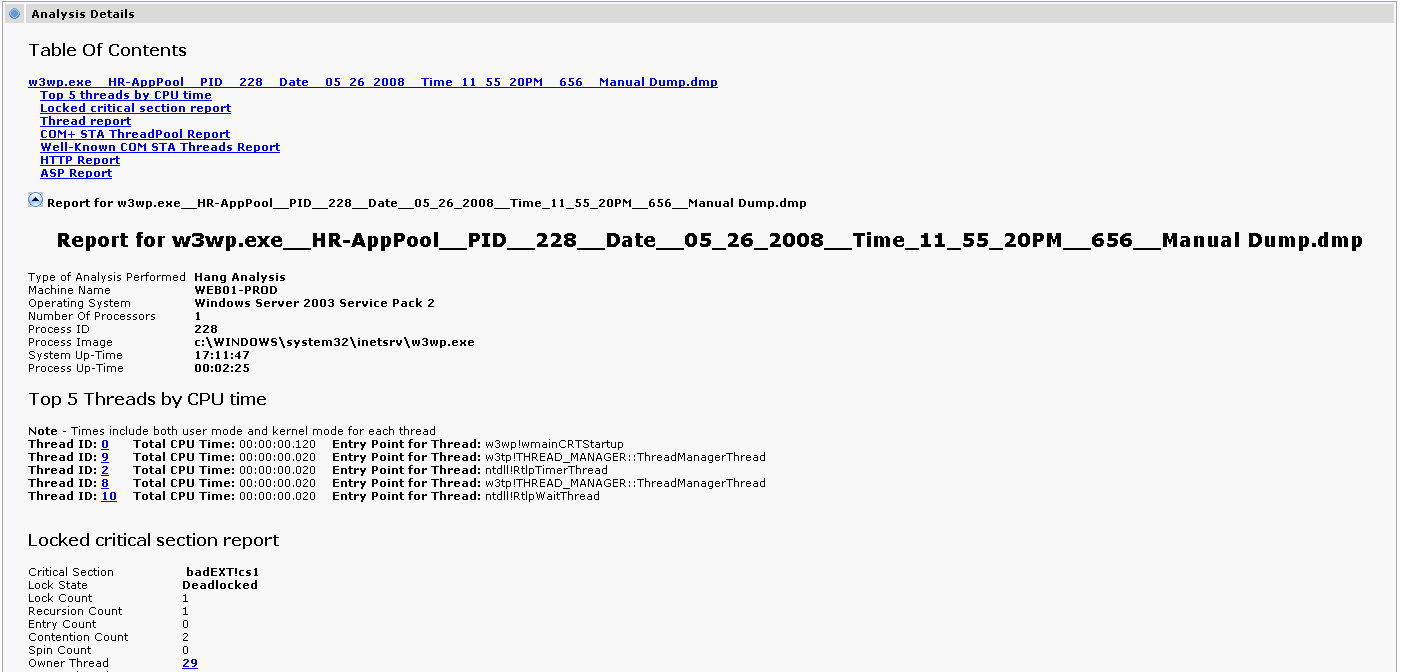
Following is an example of the analysis summary in a hang userdump analysis report:



Following is an example of the kind of detail contained in a hang userdump analysis report:

* **Top five threads by CPU time:** This shows CPU time consumption per thread. The time includes user mode and kernel mode. This information is important when dealing with busy hangs issues (high CPU usage).
* **Locked critical section report:** This report shows all the critical sections that are locked and whether they are deadlocked. (Critical sections with 0 lock count are ignored.)
* **Thread report:** This report shows all threads in the process, it is similar to executing the debugger command **~\*kv**, but for each thread displayed the report adds the thread's entry point, the create time, the time spent in user mode, the time spent in kernel mode, and a comment explaining what kind of work the thread is performing. If the thread is making a database call or a socket call, there will be detailed information about calls such as socket properties and ADO properties. Similar information will be available if the thread is running an ASP page. The page name and the line of code being executed will be added as a comment below the thread.
* **COM+ STA ThreadPool Report:** If required, the report adds a thread report for COM+ STA. This report contains Max STA Threads, Min STA Threads, Current STA Threads, g\_activitiesPerThrea, and other thread-related details.
* **Well-Known COM STA Threads Report:** This report adds the COM Sta thread report, which includes information about the Main STA and the apartment-threaded host for MTA clients. This report is only generated if it is required.
* **HTTP Report:** This report contains the current client connection, the maximum number of ATQ worker threads, and similar information. This report is only generated if it is required.
* **ASP Report:** This report contains information such as the number of ASP application loaded, the total number of ASP templates, and similar information. This report is only generated if it is required.

Following is an example of the analysis details in a hang userdump analysis report:

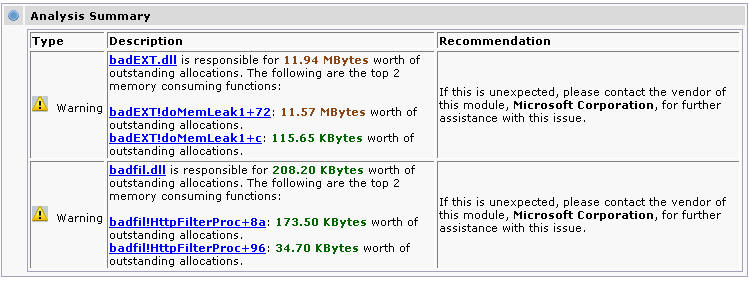


**MemoryAnalysis.asp**

Memory pressure analysis starts by looking for leaktrack.dll. If this dll is loaded, analysis will provide a detailed leak analysis, heap analysis, and virtual memory Analysis reports. If the dll is not loaded, analysis will provide only detailed heap and virtual memory analyses.

When leaktrack.dll is loaded, the analysis summary section will show the suspected leaking modules and functions; otherwise, it will just provide a warning that no leak analysis is provided in the report.

Following is an example of the analysis summary in a memory pressure analysis report:



There are two sections of analysis details in a memory pressure analysis report:

**Leak Analysis Report**  
  
This section contains a summary of outstanding allocations and a detailed report of suspected leaking modules (memory and handle).

The outstanding allocation summary contains the following items:

* **Number of allocations:** Total number of allocations counts that have been found during the tracking time and that have not been cleared.
* **Total outstanding handle count:** Total number of handles that have been acquired during the tracking period but that have not been released yet.
* **Total size of allocations:** Total memory size that has been allocated during tracking time but that has not been released yet.
* **Tracking duration:** Period of time leaktrack.dll was tracking memory usage.

This report then sorts the modules decreasingly per allocation count and per allocation size, and displays the top 10 modules.  
  
   
LeakTrack has the capability to track all types of memory allocators in one single tracking session. This report sorts allocators by allocation size and allocation count, and then provides the following memory manager statistics:

* **Heap memory manager:** Memory that has been allocated by using the HeapAlloc, LocalAlloc, and GlobalAlloc APIs.
* **Virtual memory manager:** Memory that has been allocated by using the VirtualAlloc API.
* **C/C++ runtime memory manager:** Memory that has been allocated by using the malloc, calloc, and operator new APIs.
* **OLE/COM memory manager:** Memory that has been allocated by using the CoTaskMemAlloc and IMalloc APIs.
* **OLE Automation BSTR memory manager:** Memory that has been allocated by using the SysAllocString and SysAllocStringLen APIs.

For each module displayed in both lists (Top 10 modules by allocation count and by allocation size) the report provides the following details:

* **Module:** Name of the module.
* **Allocation Count:** Number of allocations made during the tracking time.
* **Allocation Size:** The total allocations size made by the module during the tracking time.
* **Module Information:** Relevant information about the module such as type, vendor, and version.
* **Top five functions by allocation count:** List of five functions, reverse-sorted by allocation count.
* **Top five functions by allocation size:** List of five functions, reverse-sorted by allocation size.

For each function displayed in both lists (Top five functions by allocation count and by allocation size) the report provides the following details:

* **Function:** Name of the function and a description of the offset (symbols required).
* **Source Line:** Source file name and the code line number where the function is found (Private symbols required).
* **Allocation type:** Kind of allocation the function made (Heap allocation, C-Runtime, etc.).
* **Heap handle:** Handle to the heap where the allocation is made.
* **Allocation Count:** Number of allocations made by the function during the tracking time.
* **Allocation Size** Total allocations size made by the function during the tracking time.
* **Leak Probability:** Leak probability of the function.
* **Top 10 allocation sizes by allocation count:** List of 10 allocation sizes, reverse-sorted by allocation count.
* **Top 10 allocation sizes by total size:** List of 10 allocation sizes, reverse-sorted by allocation size.
* **Call stack samples:** Up to 12 call stack samples for each function; addresses where each allocation is made, times since tracking started, and sizes of each allocation.

**Heap Analysis Report**  
  
This section contains detailed statistics on heap usage:

* **Heap Summary:** Summary of the heap usage, includes the following information:
  + Number of heaps
  + Total reserved memory
  + Total committed memory
* **Top 10 heaps by reserved memory:** List of heaps, reverse-sorted by reserved memory.
* **Top 10 heaps by committed memory:** List of heaps, reverse-sorted by committed memory.
* **Heap Details:** Details for all heaps found in the process, each heap contains the following information:
  + Heap Name
  + Heap Description
  + Reserved memory
  + Committed memory
  + Uncommitted memory
  + Number of heap segments
  + Number of uncommitted ranges
  + Size of largest uncommitted range
  + Calculated heap fragmentation
  + Segment Information
  + Top five allocations by size
  + Top five allocations by count

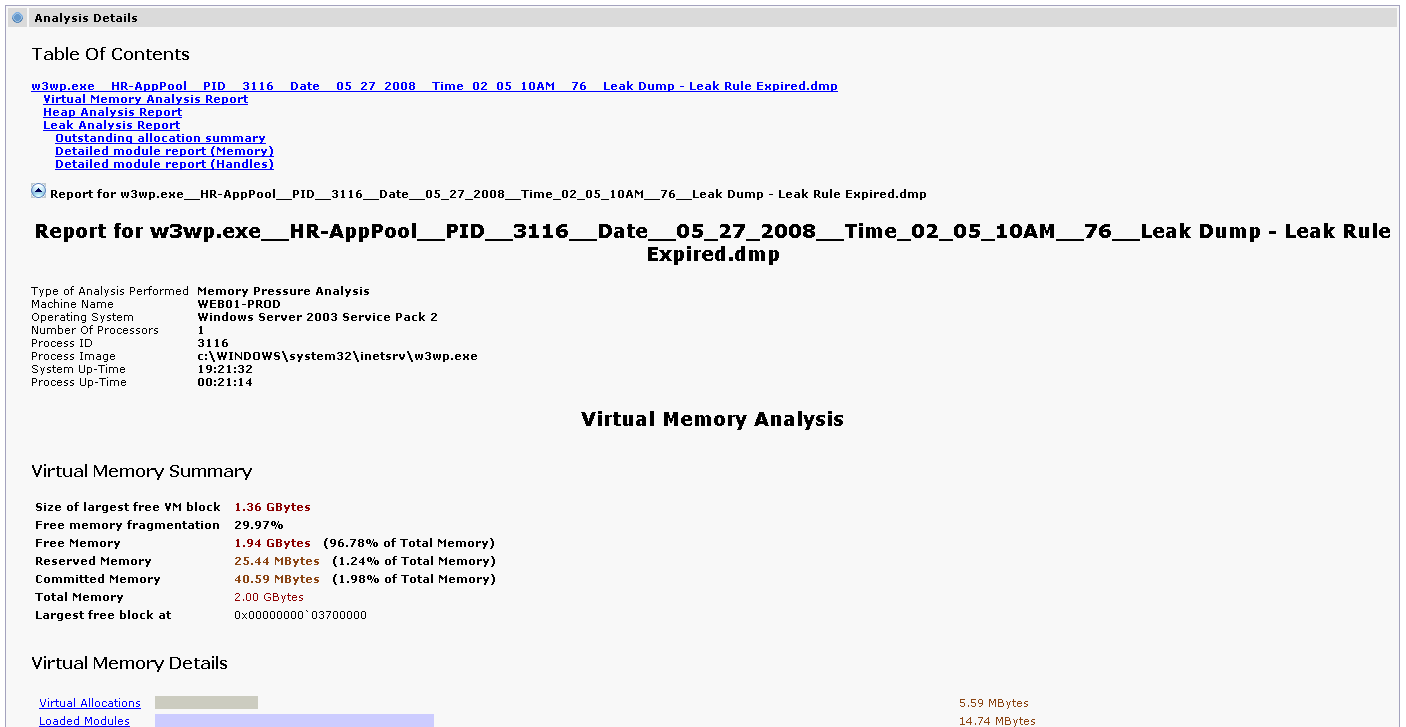
It is important to look at heap statistics while analyzing a memory leak, especially **heap fragmentation**, because sometimes the leak may be small but the heap may still be so badly fragmented that the memory manager cannot make any more large contiguous memory allocations.

**Virtual Memory Analysis Report**  
  
This section contains detailed statistics on virtual memory usage:

* **Virtual Memory Summary:** Summary of virtual memory usage, containing this information:
  + Size of largest free VM block
  + Free memory fragmentation
  + Free Memory
  + Reserved Memory
  + Committed Memory
  + Total Memory
  + Location of largest free block

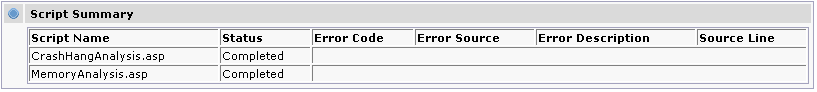
* **Virtual Memory Details:** Detailed information about virtual memory usage:
  + Virtual Allocations
  + Loaded Modules
  + Threads
  + System
  + Page Heaps
  + Native Heaps
* **Virtual Allocation Summary:** Summary of all virtual allocations, including the following information:
  + Reserved memory
  + Committed memory
  + Mapped memory
  + Reserved block count
  + Committed block count
  + Mapped block count

Example of the Details in a memory pressure analysis report:



In both analysis scripts, the analysis report contains a script summary section that shows the status of the script that was run.

**Example of the script summary section**



**Advanced Usage**

**Command line**

DebugDiag includes command-line support for the debugger host.

In the debugger, you can launch a program, invasively attach to a process, or generate hang userdumps. You can also use the command line to configure Dbghost.exe to be the Just In Time (JIT) debugger in the system.

Generating Manual userdumps.

Dbghost –dump [-IIS] [-p PID1] [-p PID2] … [-pn PName1] [-pn PName2]…

Examples

Dbghost –dump –IIS

This command generates userdumps of all IIS processes. The userdumps location is in \DebugDiag\Logs\Misc

Dbghost –pn Iexplore.exe –p 1234

This command generates userdumps of all instances of Internet Explorer along with a userdump of a process with PID 1234. The userdumps location is in \DebugDiag\Logs\Misc

Launching a program under the debugger host

Dbghost.exe –launch <command line>

dbghost -launch "C:\Program Files\Internet Explorer\Iexplore.exe"

This command launches Internet Explorer in the debugger and monitors for unhandled exceptions. Once it encounters an exception, a userdump is generated in the folder \DebugDiag\Logs\Misc

Attaching invasively to a running process

Dbghost.exe [options] –attach <PID>

Examples:

dbghost -attach 3708

This command attaches to the process by using ID 3708, and monitors for unhandled exceptions (second chance exception). Once it encounters one, a userdump is generated in the folder \DebugDiag\Logs\Misc

dbghost -script C:\Program Files\DebugDiag\Scripts\CrashRule\_custom.vbs -attach 3544

This command attaches to a process by using ID 3544. The debugger host uses the control script “CrashRule\_custom.vbs” to monitor the process.

Installing the debugger host as a JIT debugger

Dbghost –JITInstall

This command makes Dbghost.exe the JIT debugger in the system. The JIT debugger is usually DrWatson32.exe.

Register and Unregister type library

Dbghost –RegServer

Dbghost –UnRegServer

**Using Custom Control Scripts**

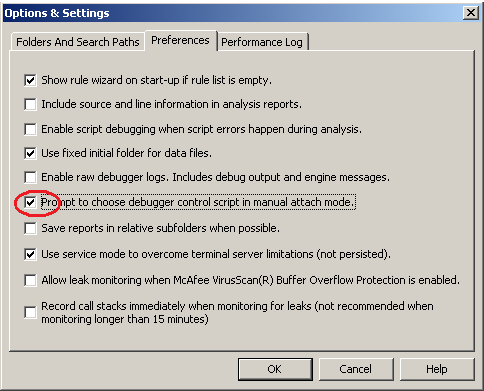
* Custom Crash Rule Control script

You can use custom crash rule control scripts to do more advance debugging, or to port a control script from one computer to another. If you generate a control script by using a rule, you guarantee no scripting errors. This lets you later modify the script to include other tasks that you want the debugger to perform.

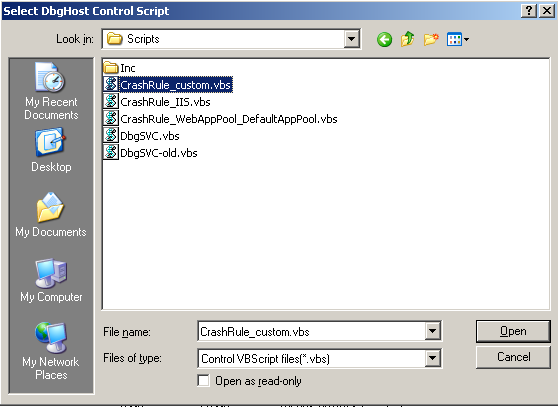
Once the control script is ready for use, there are two ways to load it:

1. Click **Preferences**. You can then use the Processes view to manually attach by using a control script to a specific process.

Tools 🡪 Options and Settings 🡪 Preferences



**Processes View 🡪 Right-Click a process🡪 Attach Debugger**



1. Create a deactivated crash rule. Overwrite the generated control script with your own version, preserving the original name. Use the context menu to activate the crash rule. The crash rule will now use your custom control script.

**Note:** The second option provides more flexibility. Remember not to use rule wizard for the same process again since it will regenerate the control script, overwriting your custom changes.

* Custom Debugging Service Control Script

In addition to using custom crash rule control scripts, DebugDiag lets you customize the debugging service main control script (dbgscv.vbs). Using a custom dbgsvc.vbs gives you more flexibility to control process performance.

As mentioned before, DebugDiag is shipped with different samples of the debugging service control script. These are located at: “DebugDiag\Samples\ControlScripts”

Example: Assume that you are debugging an ASP.NET application running under W3wp.exe and you want to know why some of the .aspx pages are taking longer than 80 seconds to render. To find this information, generate a userdump when the Request Execution Time counter for the ASP.NET performance monitor object exceeds 80 seconds. When the userdump is generated, a post-mortem analysis should reveal the root cause of this performance problem.

You can use the debugging service control script at “\DebugDiag\Samples\ControlScripts\PerfTriggers\ASPNETExecutionTime” to set this trigger.

To load this control script, follow these steps:

1. Stop the debugging service (net stop dbgsvc).
2. Rename the existing service control script at “\DebugDiag\Scripts”.
3. Copy the customized service control script from “\DebugDiag\Samples\ControlScripts\PerfTriggers\ASPNETExecutionTime” to “\DebugDiag\Scripts”.
4. Start the debugging service (net start dbgsvc).

**Note:** The customized debugging service control script is NOT persistent! It will be overwritten by a newly generated service control script anytime a new rule is added or when an existing rule is activated, deactivated, or removed!

For more information about other available service control scripts designed for performance triggers or on how to develop your own control scripts, please visit the DebugDiag help.

**Using Custom Analysis Scripts**

The analysis module in DebugDiag is extensible. You can also use existing analysis scripts to extract more data or format reports by using the objects exposed by Dbghost.exe. Additionally, you can create new analysis scripts to address specific areas.

The starter analysis script that is shipped with DebugDiag will help you build your own analysis script.

DebugDiag help contains the Developers guide that will assist you in building both analysis and control scripts.

To use a newly built analysis script, add the script to the “\DebugDiag\Scripts” folder. The script will appear in the Advanced Analysis view in the DebugDiag UI, and then you can run it—separately or in conjunction with other analysis scripts—against one or many userdumps.

Example: Assume that you would like to automate the analysis of .NET userdumps in a new analysis script. For the sake of this example, consider how you might get the size of managed memory and the number of objects in the Finalize queue by running SOS.DLL commands in DebugDiag.

To run such a script, edit the StarterScript.asp located in “\DebugDiag\Samples\AnalysisScripts” and add the following changes:

SomeCLRInfo.asp

…

Manager.Write "<b>" & g\_ShortDumpFileName & "</b><BR>"

Manager.Write DebuggerExecuteReplaceLF(".time", "<BR>")

Manager.Write "<BR>"

‘Load the .NET debugger extension sos.dll (2.0 version)

CmdOutput = g\_Debugger.Execute("!load C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\SOS.dll")

‘Get the size of the GC

CmdOutput = g\_Debugger.Execute("!eeheap -gc")

Manager.Write "<b> CLR Memory Usage </b> <BR>"

‘Format the output

If Len(CmdOutput) > 0 Then

li = Split(CmdOutput, vbLf)

If UBound(li) >= 1 Then

For I = LBound(li) To UBound(li)

Manager.Write li(I) & "</b><BR>"

Next

End If

End If

li1 = Split(li(UBound(li)-1), "(")

li2 = Split(li1(1), ")")

CLRMemoryUsed = li2(0)

Manager.Write "CLRMemoryUsed " & CLRMemoryUsed & "</b><BR>"

‘Report the size as information in the analysis summary

Manager.ReportInformation "The Amount of managed memory used in this process is <b>" & CLRMemoryUsed & "</b> Bytes"

Manager.Write "<BR>"

Manager.Write "<b> Finalize Queue </b> <BR>"

‘Get the number of objects in the Finaluze queue

CmdOutput = g\_Debugger.Execute("!FinalizeQueue")

‘Format the output

If Len(CmdOutput) > 0 Then

li = Split(CmdOutput, vbLf)

If UBound(li) >= 1 Then

For I = LBound(li) To UBound(li)

Manager.Write li(I) & "</b><BR>"

Next

End If

End If

ObjFQ = Split(li(UBound(li)-1))

‘Report the size as information in the analysis summary

Manager.ReportInformation "There are <b>" & ObjFQ(1) & "</b> objects in the Finalize queue "

End If

Manager.CloseDebugger g\_DataFile

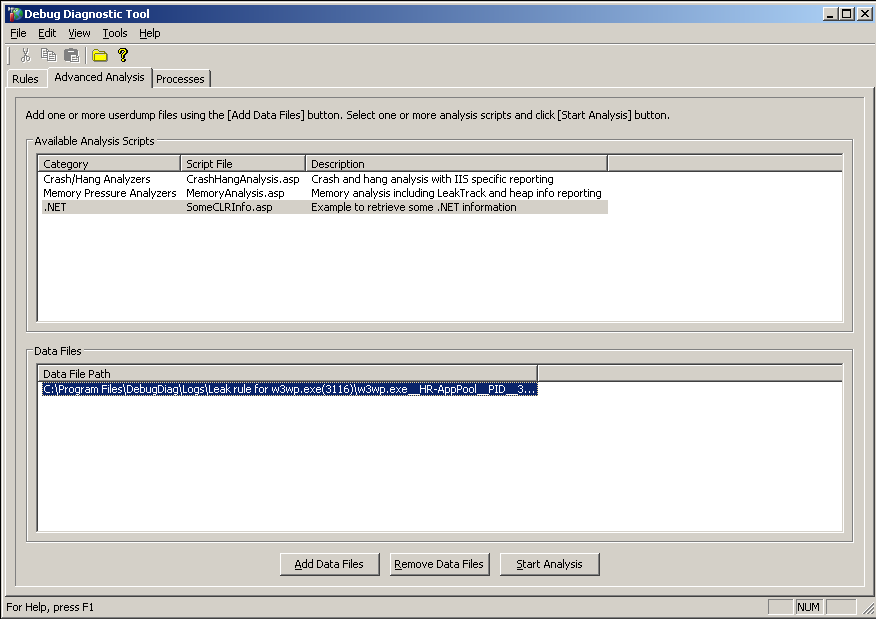
End If

UpdateOverallProgress

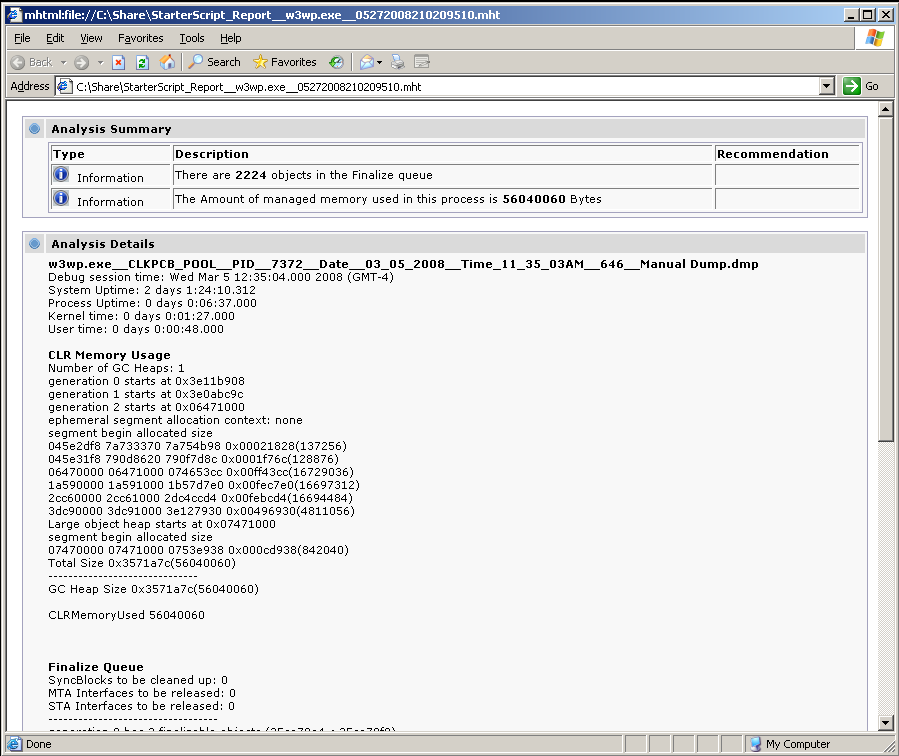
Next

Rename the script file to “SomeCLRInfo.asp” and copy it to the “\DebugDiag\Scripts” folder.

The script now looks like this in the DebugDiag UI:



The example above shows how to use the .NET debugger extension in an analysis script to extract and report .NET information. When you run this script against a userdump, the report shows the size of managed memory as well as the number of objects in the Finalize queue. This information will display in the following way:



**Note**: When you build a custom analysis script and you receive script errors, you can debug this problem by enabling script debugging.

**FAQ**

Q: Can I use DebugDiag to debug 64–bit processes?

A: Yes, use the 64–bit build of DebugDiag available at the same download center location.

Q: Can I install Debugdiag 32 and Debugdiag 64 on the same 64 bit OS.

A: Debugdiag 64 blocks the install when the 32 bit is present. But Debugdiag 32 will still install on a 64 bit OS even if Debugdiag64 is installed. It is highly recommended to not have both builds on the same 64 bit OS.

Q: How can I debug 32 bit processes on a 64 bit OS?

A: Install the 32 bit Debugdiag, but make sure the 64 bit Debugdiag is removed!

Q: Can I debug Vista/Windows 2008 and Windows7/Windows2008 R2 target processes with Debugdiag 1.1?

A: Debugdiag 1.1 was not tested ton these platforms for debugging target processes, but you still can install analysis modules to analyze userdumps.

Q: Can I generate userdumps on specific .NET exceptions?

A: Yes, you can (see **Exceptions** section above).

Q: Can DebugDiag perform .NET memory analysis?

A: DebugDiag 1.1 does not include .NET memory analysis, but you can develop an analysis script to perform such an analysis.

Q: I am debugging a process that crashes at startup. Can I use DebugDiag for that?

A” Yes, you can. Use the pre-attach option.

Q: Does a memory leak and handle leak rule track .NET memory usage?

A: No, Leak rules track native memory usage only. In managed applications, DebugDiag leak analysis can report false positives since it tracks allocations that cause the managed heap to grow, and those allocations may or may not be related to the real memory problem. However, post mortem analysis of a .NET userdump can reveal the root cause of a managed memory leak by using debugger commands available in the debugger extension sos.dll. (For additional information, please visit <http://blogs.msdn.com/debugdiag/>.)

Q: I am debugging a high CPU issue, and I can’t use a hang rule because the process is not an IIS process, and I can’t take a manual dump because the problem is random. What can I do?

A: Modify the service script that is included with the tool to set a trigger on process CPU usage. Service scripts (dbgsvc.vbs) are located in the \Samples folder.

Q: I am debugging a memory leak in an ActiveX control that loads in Internet Explorer. The leak happens very quickly after the control is loaded. I injected LeakTrack and generated a userdump a few minutes later. When I ran the memory pressure analysis, I saw no call stacks for the leaked allocations. Why?

A: LeakTrack can wait 15 minutes before starting to collect accurate call stacks for leaked allocation. In this case, you need to use the FastTrack feature (Option and Settings 🡪 Preferences 🡪 Record call Stacks Immediately…).

Q: I ran the memory pressure analysis against a userdump (size: 650 MB), and it showed that mscorsrv.dll is responsible for allocating 460 MB. Doesn’t this mean that mscorsrv.dll is leaking memory?

A: No, DebugDiag tracks only native memory. When managed allocations are made from your .NET code, they will translate to native allocations made by the .NET core dlls. So, this means that further inspection of .NET objects is required to determine which objects are leaked.

Content Change History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comment |
| 1.0 | April 2009 | Mourad Lagdas | Initial publish. |
| 1.1 | October 2009 | Mourad Lagdas | Updated for 64 bit release of Debugdiag 1.1 |
|  |  |  |  |
|  |  |  |  |